IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 21 June 2023, accepted 14 August 2023, date of publication 24 August 2023, date of current version 30 August 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3308492

RESEARCH ARTICLE

Unsupervised Deep Learning for Distributed Service Function Chain Embedding

PANTELEIMON RODIS AND PANAGIOTIS PAPADIMITRIOU, (Senior Member, IEEE)
Department of Applied Informatics, University of Macedonia, 546 36 Thessaloniki, Greece

Corresponding author: Panteleimon Rodis (rodis@uom.edu.gr)

This research was funded by the European Union's Horizon Europe research and innovation program under grant agreement No. 101070487 (NEPHELE). The publication of the article in OA mode was financially supported by Hellenic Academic Libraries Link (HEAL-Link).

ABSTRACT Network Function Virtualization (NFV) has paved the way for the migration of Virtual Network Functions (VNFs) into multi-tenant datacenters, lowering the barrier for the introduction of new processing functionality into the network. Recent trends for resource orchestration across the entire compute continuum raise the need for decision making at low timescales, a requirement which can be hardly met by centralized resource optimizers that rely either on Linear Programming or Machine Learning (ML). In this respect, we present a distributed approach tailored to a crucial resource orchestration aspect, *i.e.,* the embedding of Service Function Chains (SFCs) onto large-scale virtualized network infrastructures. In order to confront the computational hardness of the SFC embedding problem, we utilize a clustering method for the partitioning of the solution space, empowering the search for efficient solutions in parallel across all clusters. Another salient feature of our approach is the use of unsupervised deep learning for the computation of embeddings within each cluster. Our distributed SFC embedding framework is benchmarked against a state-of-the-art heuristic and a distributed greedy algorithm. Our evaluation results uncover notable gains in terms of resource efficiency, combined with solver runtimes in the order of milliseconds with thousands of substrate nodes.

INDEX TERMS Network function virtualization, resource orchestration, deep learning, distributed computation.

I. INTRODUCTION

Network Function Virtualization (NFV) has evolved over the years as a major enabler for network service deployment with higher flexibility and resource efficiency [1], [2], [3], [4], [5]. Besides traditional telco-oriented network functions, such as security appliances, network address translation or proxies, NFV has also found traction across Radio Access Network (RAN) infrastructures [2]. In this respect, NFV orchestration has attracted significant attention, with regards to aspects such as service chaining, service function chain (SFC) embedding, as well as the scaling of running network service instances. Although various solutions exist for these problems [1], [2], [3], [4], [5], [6], [7], [8], handling these orchestration operations at large scale still poses significant challenges and opens up opportunities for new approaches.

The associate editor coordinating the review of this manuscript and approving it for publication was Wenbing Zhao.

To this end, we focus on the problem of SFC-graph embedding, which entails the assignment of Virtualized Network Functions (VNFs) and the corresponding SFC-graph edges onto the respective NFV infrastructure counterparts (*i.e.,* servers and network paths). This particular problem is known to be NP-Hard [9] and has been mainly tackled by heuristics [6], [7], [10], [11], [12] and exact methods, [1], [2], as well as (deep) learning-based techniques [8], [13], [14], [15].

Despite these advancements, the massive scale of modern core cloud infrastructures and the ever-increasing computing and network capacities inevitably increase the complexity of SFC embedding. Taking this complexity into consideration, exact methods turn out to be inefficient due to the prohibitive solver runtime, heuristics commonly yield notable suboptimality and may fall short of computing embeddings at the timescales required by network operators, whereas the training stage of (deep) learning-based methods also introduces significant scalability limitations.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

In order to address SFC embedding at large scale and confront its computational hardness, we employ a distributed embedding computation approach. Fully decentralized solutions enable adaptability to different problem configurations and scalability variations of the substrate network. Relying on substrate nodes for the computation of the mapping (instead of a centralized embedder) empowers us to detect unreachable nodes or nodes/links with saturated capacity, thereby eliminating them from candidates for hosting VNFs. Such a distributed approach facilitates the parallel embedding computation across the whole range of the substrate network. This yields a significant advantage over centralized solutions, since the solver runtime can be less dependable on the substrate network size.

Our distributed SFC embedding framework couples node clustering with deep learning. To decentralize the embedding computation, we utilize agents deployed within each server and a controller which is responsible for the agent coordination. The main role of these agents is to create clusters for the partitioning of the solution space and the parallel computation of efficient SFC mappings. Embeddings are computed within each cluster using unsupervised deep learning. To this end, we utilize deep neural network (NN) models, trained using genetic algorithms. We perform a comparison against (i) a state-of-the-art centralized embedding method (*i.e.,* BACON [3]) in order to quantify any potential gains of distributed embedding computation, and (ii) a distributed greedy algorithm to gain insights into the benefits stemming from deep learning. For these evaluations, we conduct simulations on fat-tree network topologies of two sizes and up to 2662 nodes.

The paper is organized as follows. In Section II, we lay out a Mixed Linear Integer Programming (MILP) formulation of the SFC embedding (SFC-E) problem. In Section III, we argue for the advantages of deep learning in comparison with heuristic and greedy algorithms for the solution of NP-Hard problems, such as SFC-E. Section IV presents an overview of our distributed SFC embedding framework. Section V introduces our node clustering method, followed by the description of the distributed unsupervised deep learning method in Section VI. In Section VII, we discuss the computational complexity of our solution. Section VIII presents our evaluation results. Section IX provides an overview of related work. Finally, Section X highlights our conclusions.

II. SFC EMBEDDING FORMULATION

We hereby present a formulation for the SFC-E problem. Let graph $G_s(V_s, E_s)$ model the substrate network, where r_z denotes the residual computing capacity of substrate node $z \in V_s$, and b_p expresses the available bandwidth of substrate path p in G_s. Furthermore, $G_v(V_v, E_v)$ models the SFC request, where d_n, d_e denote the CPU demand of virtual node $n \in V_v$ and the bandwidth demand of virtual edge $e \in E_v$, respectively. By $x_{n,z}$, we express whether virtual node n is mapped onto the substrate node z (*i.e.,* $x_{n,z} = 1$ implies an assignment), whereas $f(e, p)$ indicates the amount of traffic flow (*i.e.,* bandwidth) of virtual edge e that is assigned to the substrate network path p.

TABLE 1. Notation table.

Symbol	Description
G_s	substrate network graph representation
V_s	substrate nodes representing servers
E_s	substrate network links
$p_{u,z}$	path in E_s connecting substrate nodes u, z
G_v	SFC-graph
V_v	virtual nodes
E_v	virtual links
$e_{k,m}$	virtual edge $\in E_v$ connecting nodes $k, m \in V_v$
$x_{n,z}$	assignment of virtual node n on substrate node z
$f(e,p)$	traffic flow on virtual edge e assigned to substrate path p
$l_{e,p}$	length of mapped virtual link e on path p
r_z	residual computing capacity of substrate node z
d_n	CPU demand of virtual node n
b_p	available bandwidth of substrate path p
d_e	bandwidth demand of virtual edge e
$\#n$	agent deployed in substrate node n
demands $\#d$	set of clustering demands d

The SFC-E problem is formulated as an (embedding) cost minimization problem, *i.e.,* a SFC mapping is sought that minimizes the CPU and bandwidth consumption in the substrate network. Thereby, the SFC-E problem is formulated as:

$$\text{minimize} \sum_{n \in V_v} \sum_{z \in V_s} x_{n,z} d_n + a \sum_{e \in E_v} \sum_{p \in E_s} f(e,p)$$

where a is a normalization weight that acts as a balancing factor between the CPU and bandwidth cost.

The embedding cost minimization is subject to a set of capacity and variable domain constraints:

$$\sum_{z \in V_s} x_{n,z} = 1 \qquad \forall n \in V_v \tag{1}$$

$$\sum_{k,m \in V_v} f(e_{k,m}, p_{u,z}) \leq b_p \qquad \forall u, z \in V_s, \forall p \in E_s \tag{2}$$

$$\sum f(e_{k,m}, p_{u,z}) - \sum f(e_{k,m}, p_{z,u}) = d_e(x_{k,u} - x_{m,z})$$
$$k \neq m, \forall k, m \in V_v, u \neq z, \forall u, z \in V_s, \forall e \in E_v \tag{3}$$

$$\sum_{n \in V_v} x_{n,z} d_n \leq r_z \qquad \forall z \in V_s \tag{4}$$

$$x_{n,z} \in \{0, 1\} \qquad \forall n \in V_v, \forall z \in V_s \tag{5}$$

$$f(e_{k,m}, p_{u,z}) \geq 0 \qquad \forall k, m \in V_v, \forall u, z \in V_s \tag{6}$$

Constraint (1) ensures that each VNF is assigned exactly to one substrate node. Constraint (2) enforces a capacity limit on substrate links. Constraint (3) implies flow conservation, *i.e.,* the summation of incoming and outgoing flows of a substrate node must be zero. Condition (4) ensures that the CPU demands of the assigned VNFs do not exceed the residual CPU capacity of the corresponding substrate nodes. Lastly, constraint (5) enforces the binary domain constraints for variable $x_{n,z}$, whereas condition (6) enforces the causality of the flows $f(e, p)$. Table (1) provides a list of all notations.

III. DEEP LEARNING VS. HEURISTICS

SFC orchestration aspects are often tackled using heuristic and greedy algorithms, whereas lately AI is also in the spotlight for this class of problems. Hereby, we reason in favor of the use of deep learning for solving NP-Hard problems, such as SFC-E, over heuristic and greedy algorithms and justify our choice of moving towards this direction in our work. In this respect, we refer to the classical modeling of heuristic search as state space search, provide a similar modeling for the greedy algorithms, and subsequently examine the potential of using each method for the computation of problem state space.

State space (or *solution space*) S consists of all the possible solutions of a problem; or else all the possible configurations of the problem. An algorithm is modeled as state space search, if it gradually computes the whole or a part of S so as to find a goal state that comprises an acceptable solution for the problem under study. As an example, let us refer to SFC-E. The state space of each instance of the problem consists of all the possible mappings of the incoming SFC-graph to the given substrate network. Every algorithm that solves it, computes an element of S that represents a (near-)optimal mapping of the SFC-graph onto the substrate network.

Heuristic search is modeled as graph $G(V, E)$, where nodes in V model the states of S and edges in E model the possible actions that can be applied to the states [16]. A heuristic search algorithm is a sequence of actions that generates a state trajectory leading from an initial state to a goal state. This procedure generates a path in G that connects the initial and the goal state [17].

Greedy algorithms perform locally optimal choices in each step of their function aiming at reaching globally optimal solutions. Let us formulate this as a state space search and apply it to the SFC-E in order to gain intuition on this formulation.

Initially, all the states in S are possible solutions of the problem at hand. Greedy algorithm A in step n performs a locally optimal choice that promotes the states in $S_n \subseteq S$ as possible solutions of the problem; in step $n+1$ promotes the states of $S_{n+1} \subseteq S_n$ as possible solutions, and so on, until a goal state has been reached. In each step n, algorithm A applies function $f(S_n) \rightarrow S_{n+1}$, until in step q the algorithm has formed $S_{n+q} \subseteq T$, where T is the set of goal states in S.

This formulation is highly relevant to greedy algorithms for the SFC-E problem. The most common pattern that these algorithms follow is that based on some criteria, they sequentially map the VNFs of an incoming request to the substrate network. When a greedy algorithm maps VNF a onto substrate node z, the states in S that include this mapping are promoted as solutions of the problem, while the states that map a onto any other node are rejected. This procedure gradually generates the mapping that the greedy algorithm will produce, while rejecting subsets of S as possible solutions. This is the case for the greedy algorithm that we present in Section VIII-B, where the VNFs are sequentially mapped to the cluster nodes reducing the possible mappings in each step.

The state space of NP-Hard problems grows exponentially, so assuming that NP $\neq$ P holds, there is no efficient method that can compute the whole state space efficiently (that would incur polynomial time). This implies that every efficient algorithm computes only a subset of S of polynomial size, aspiring to identify a near-optimal solution. As such, in order to compute the whole S using heuristics, an exponential number of state trajectories is required. This implies using an exponential number of efficient heuristics. On a similar fashion, computing S using greedy algorithms requires an exponential number of steps, or an exponential number of efficient greedy algorithms.

It is widely accepted that deep neural networks approximate efficiently linear and non-linear functions and express probability distributions [18], [19]. There are families of functions that can be efficiently approximated by a single model [20]. It is reasonable to assume then that we can develop deep neural network models of appropriate architecture and training that achieve an effective level of generalization so as to include the functionality of families of heuristics and greedy algorithms. Along these lines, the use of deep learning outperforms other solutions. Based on this observation, we leverage on deep learning for tackling the problem of SFC embedding.

The complexity of the heuristics is determined by the length of the state trajectories, while for greedy algorithms the number of steps determines their asymptotic behavior. The efficiency of the learning models is determined by the training procedure which needs to be competitive against other solutions.

IV. SFC EMBEDDING FRAMEWORK OVERVIEW

Our SFC embedding approach relies on a multi-agent framework that consists of:

- Agents hosted in each substrate node, where each agent computes a candidate mapping for every request using unsupervised learning.
- A controller that coordinates the distributed agents and selects the most efficient mapping computed by the agents.

An indicative example is illustrated in Fig. 1. Upon the arrival of a SFC request, each agent forms a cluster of nodes that are capable of hosting the SFC and are also proximate to the agent's hosting node. Subsequently, each agent computes a mapping of the SFC on the nodes of its cluster. Eventually, the controller returns the most efficient mapping generated by the agents or alternatively declares a rejection, in case a feasible mapping has not been identified.

Initially, the controller generates a sorted list of all the paths among the substrate nodes in ascending order based on their length. Upon the arrival of a new request, the controller traverses the sorted list from the beginning, and if a path and the nodes it connects meet the SFC request demands, the nodes are used for the formation of clusters by the agents.

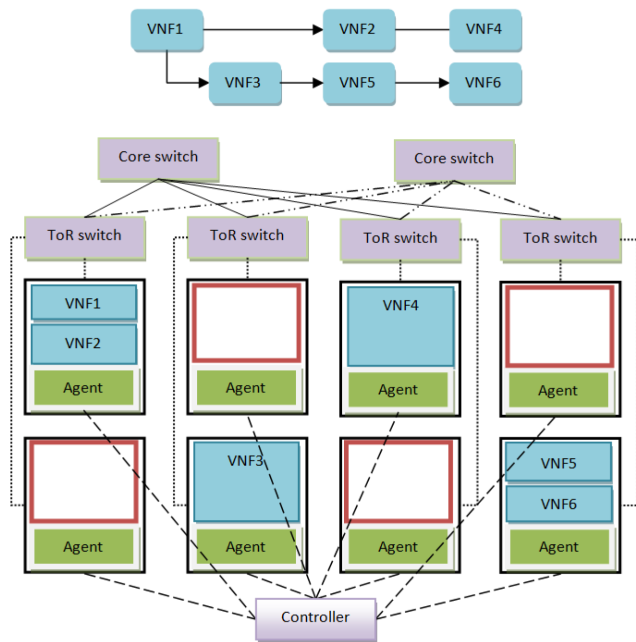


FIGURE 1. Example of the distributed SFC embedding framework in a two-layer fat-tree substrate network, with four racks and one agent within each server. The controller interacts with the agents coordinating the SFC embedding on the servers.

The maximum size of a cluster is bounded in order to maintain a low complexity for the algorithm.

Subsequently, a candidate mapping is computed by every agent using unsupervised deep learning. If a trained model suitable for the computation of the mapping is identified by the controller, the agents utilize it for the mapping output. In case there is no valid mapping, the agents train deep NN models in order to generate valid mappings. The NN is trained by a genetic algorithm, which comprises a salient feature of our framework, given that genetic algorithms are commonly utilized for the training of supervised learning models. The newly trained models that generate efficient solutions are stored in the controller for future use by the agents. In case valid mappings are computed at any stage of the procedure, the controller returns the mapping with the best fitness as the algorithm output.

The source code of a Java implementation of the algorithms and our simulation environment are available at [21]. In the following, we elaborate further on the clustering procedure (Section V) and the unsupervised deep learning method for the placement of VNFs (Section VI).

V. NODE CLUSTERING

Clustering is carried out in dynamic fashion, favoring adaptability in the variations of CPU and network load on the infrastructure. Initially, the substrate paths that connect the nodes (*e.g.,* servers) are ranked in ascending order based on their hop count. This procedure takes place only once at the beginning, whereas the next steps are executed at the arrival of each request. Starting from the top of the sorted list, if the residual bandwidth of each path and the residual capacities of the nodes along this path meet the demands of the SFC-graph, these nodes are used for the formation of the clusters. For instance, for a substrate path $p_{u,z}$ that meets the request demands, node u is inserted into the cluster of agent #z and node z is inserted into the cluster of agent #u.

Each cluster has a predetermined maximum size equal to four times the size of the SFC-graph. The effectiveness of this adjustment has been determined experimentally. More specifically, forming clusters of smaller size reduces the efficiency of the generated mappings, while clusters of larger size increase the complexity of the algorithm without improvements on its efficiency. Each cluster is formed by the candidate nodes (for hosting VNFs), which are more proximate to the agent.

We apply four sets of demands in order to determine if path p and its connected nodes u, z meet the demands of the SFC-graph:

demands #1:

$r_u \leq max(d_n) \wedge r_z \leq max(d_n) \wedge b_p \geq min(d_e)$

demands #2:

$r_u \geq min(d_n) \wedge r_z \geq min(d_n) \wedge b_p \geq min(d_e)$

demands #3:

$r_u \leq max(d_n) \wedge r_z \leq max(d_n) \wedge b_p \geq max(d_e)$

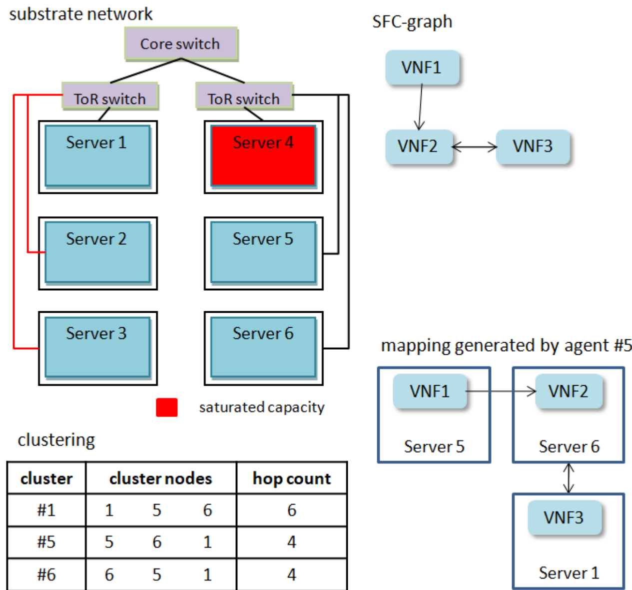
demands #4:

$r_u > min(d_n) \wedge r_z > min(d_n) \wedge b_p \geq max(d_e)$

where $max(d_n)$ and $min(d_n)$ denote the maximum and minimum CPU demands, respectively, while $max(d_e)$ and $min(d_e)$ express the maximum and the minimum bandwidth demands in the SFC graph. The use of only four sets of demands retains low complexity on the algorithm, which is corroborated by our evaluation results.

In the case where *demands* #1 fails to form clusters that generate valid mappings, the cluster formation procedure and the mapping computations are repeated using the *demands #2* and so forth. Each set of demands is used for the computation of mappings that minimize the residual capacity of the substrate nodes. In case of failure, the next set of demands relaxes the clustering formation criteria in order to generate an efficient mapping, inline with the previous objective. After the formation of the clusters, the cluster nodes meet the bandwidth demands of the request. Subsequently, the agents identify the most efficient placement of the VNFs onto the cluster nodes.

At the end of this procedure, each mapping generated by the agents is conveyed to the controller and the one with the best fitness eventually becomes the output of the algorithm. In case more than one agents have computed mappings with the same fitness, then the mapping of the first agent in the list is chosen. If the agents have not generated any valid mapping after all sets of demands, the request is rejected. These procedures are described in Algorithm 1. A simple example of the clustering procedure and the produced mapping is illustrated in Fig. 2.



cluster	cluster nodes			hop count
#1	1	5	6	6
#5	5	6	1	4
#6	6	5	1	4

FIGURE 2. Clustering example, where each cluster consists of 3 nodes, equal to the size of the SFC-graph. Agents #2, #3 and #4 do not form clusters due to the saturation in the capacity of server 4 and the links in servers 2 and 3. The clusters in the rest of the agents are formed by the nodes of closer proximity with sufficient capacity. The nodes in clusters #5 and #6 are connected with shorter paths than the nodes of cluster #1. The SFC is then mapped to the first shortest-path cluster of the list, *i.e.,* agent #5.

VI. DEEP LEARNING FOR VNF PLACEMENT

A. MODEL DESCRIPTION

After the clustering procedure, the agents compute the mapping of the cluster nodes onto the VNFs. The clusters are formed considering the bandwidth constraints; therefore, at this point, the agents only take capacity demands into account. The VNF placement is computed using an artificial feedforward NN in every agent. This approach is based on the concept of pattern recognition, *i.e.,* the NN is trained to identify within the cluster nodes the pattern that forms an efficient mapping.

The NN receives as input the cluster of nodes and the VNF demands, and subsequently generates the mapping. Our proposed solution satisfies the demand for efficient and low-complexity computation of the mapping in the agents. Unlike other solutions, for every problem configuration our NN is trained effectively with only one input. The leads to a satisfactory degree of generalization, obviating the need for a training dataset of many labeled examples (as in most cases of supervised learning) or the need to explore many states of the state space of the problem (as in most cases of reinforcement learning).

As illustrated in Fig. 3, the NN encompasses two hidden layers. The activation function for all nodes except the output layer is the bipolar sigmoid function:

$$f(x) = (1 - \exp(x))/(1 + \exp(x))$$

whereas the activation function of the output layer is the linear function $f(x) = x$. The output x_i of output layer node i is further filtered producing the final output x'_i, where x'_i is the integer part of x_i. Bias is constant and set equal to 10 for all nodes.

Algorithm 1 Controller Node

Input: graphs $G_s(V_s, E_s)$, SFC-graph, *path list*, *cluster size*
Output: mapping

1: run **initialization**()
{on new SFC-graph input}
2: run **clustering**(*demands* 1)
{after agents compute candidate mappings}
3: run **output**()
4: **Procedure initialization**()
5: **for** $u = 0$ to substrate nodes **do**
6: **for** $z = 0$ to substrate nodes **do**
7: store hop count of path $p_{u,z}$ in *path list*
8: **end for**
9: **end for**
10: bucket sort *path list* in ascending order of hop counts
11: **Procedure clustering**(*demands*)
12: **for** $z = 0$ to substrate nodes **do**
13: delete stored data from *agent #z*
14: send SFC capacity demands
15: add node z to cluster of *agent #z*
16: **end for**
17: *clustersize* $= 4\times$SFC-graph size
18: **for** $p = 0$ to *path list* **do**
19: **if** path $p_{u,z}$ meets *demands* **then**
20: **if** size of cluster in *agent #z* $<$ *cluster size* **then**
21: add u to cluster of *agent #z*
22: **end if**
23: **if** size of cluster in *agent #u* $<$ *cluster size* **then**
24: add z to cluster of *agent #u*
25: **end if**
26: **end if**
27: **end for**
{compute VNF-placement}
28: **for** $j = 0$ to substrate nodes **do**
29: compute mapping in *agent #j*
30: **end for**
31: **Procedure output**()
32: **if** mappings are produced **then**
33: m = mapping of *agent #0*
34: **for** $j = 1$ to substrate nodes **do**
35: **if** mapping of *agent #j* has better fitness than m **then**
36: m = mapping of *agent #j*
37: **end if**
38: **end for**
39: return m
40: **else if** *demands #d* $<$ 4 were used **then**
41: run clustering(*demands #d* + 1)
42: **else**
43: return rejection
44: **end if**

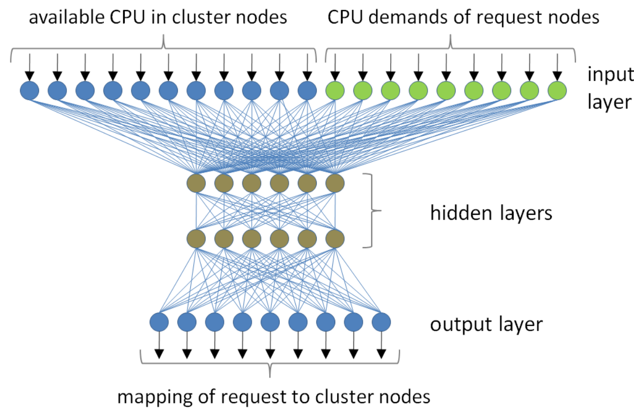


FIGURE 3. The deep neural network model.

In order to reduce the complexity of the training procedure, the training algorithm assigns only integer values from the range [-10, 10] onto the NN links. As such, the possible configurations of the NN are limited to a finite and easily computable set.

The application of integer values in NN weights is an effective strategy which leads to low-complexity solutions [22]. It can be combined with training methods based on evolutionary strategies and produce effective solutions for computationally-hard problems [23]. Furthermore, using integer values in a limited range generates very cost-effective implementations of NN that satisfy the demands of our system [24]. In any case, the choice of the range of values and the appropriate architecture of the NN depends on the problem at hand and, in many cases, deviates significantly among problems of different complexity.

In our simulation environment (Section VIII-A), the input layer of the NN receives the residual capacities of the first 11 nodes of the cluster and the demands of the VNFs (which are at most 9 in our simulations). Each of the two hidden layers consists of 6 nodes, whereas the output layer encompasses 9 nodes which indicate the mapping of the cluster nodes to the VNFs. These choices stem from empirical evidence, based on which, decreasing the number of nodes or hidden layers generates undertrained models, while increasing the number of nodes or layers leads to overtrained models.

Any input or output node that is not applicable is assigned with a negative value. The NN finally produces a vector m in the form: $m = \langle n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9 \rangle$, where n_i is the cluster node mapped onto VNF i or a negative value. The mapping generated by the algorithm is formed by replacing in m the cluster nodes with the corresponding substrate node IDs.

B. TRAINING USING GENETIC ALGORITHMS

For the NN training, we rely on a genetic algorithm (GA). Although the use of GAs for NN training is known to exhibit certain advantages that include efficiency and reliability [23], [25], [26], [27], yet GAs are mainly applied to supervised learning as an alternative to the Back Propagation technique. To the best of our knowledge, there are no studies that employ GAs for training unsupervised learning models as we do. As such, we consider this an innovative feature of our proposed solution.

The nature of NN training is rather complex and usually exhibits very complex error surfaces where traditional techniques, such as Back Propagation, often converge in local optima [28]. The GAs presented in the aforementioned works deviate from the simple GA design providing sophisticated training methods that avoid the premature convergence in local optima. Our solution is designed in the same spirit.

Our training algorithm is based on the Parameter Adjustment GA in [8] and is described in Algorithm 2. Its operation is oriented towards the optimization of the operation of other applications, which, in our case is the NN. This GA enriches the design of the simple GA by running multiple instances of the simple GA and then combining the generated results. This operation is facilitated by the *supergenerations* parameter, which determines the number of GA instances that will run in the first stage of the algorithm on randomly generated populations. The models that they return will constitute the populations for another round of GA instances at stage 2. Subsequently, at stage 3, the final solution will be computed by one last GA instance that will run on the population of models produced at stage 2. The generated models during these stages approach local optima, but the final solution manages to avoid convergence in any of them and approaches the global optimum.

The setup of the GA consists of a population of 20 chromosomes, 40 generations, and 4 supergenerations. Furthermore, the crossover probability is set to 1.0, whereas the mutation probability is adjusted to 0.5. The GA is efficient and converges fast to an near-optimal model.

The chromosomes of the GA consist of the weights of the links that connect the NN nodes. The GA produces optimized weights for the links that enable the NN to generate the desired mapping. The objective of the trained NN is the computation of a mapping that minimizes the residual server capacities. In order to compute the fitness of each chromosome of the GA population, we run the network model to be encoded and we feed it with the SFC-graph. The generated mapping is evaluated based on the following fitness function:

$$\text{minimize} \sum_{z \in V_s, n \in V_v} (r_z - d_n) \tag{7}$$

for any substrate node z mapped to virtual node n, as defined by the mapping computed by the NN.

Fig. 4 illustrates how the models produced by the GA in the various stages of its operation gradually converge in a model of minimum fitness, as fitness is described by equation (7). The plot describes the training that takes place in one agent that produces the solution, which is finally used for the embedding of an incoming request of 9 VNFs in the 12-pod network utilized in our simulations (Section VIII). This exhibits how the best solution (the most efficient model) is

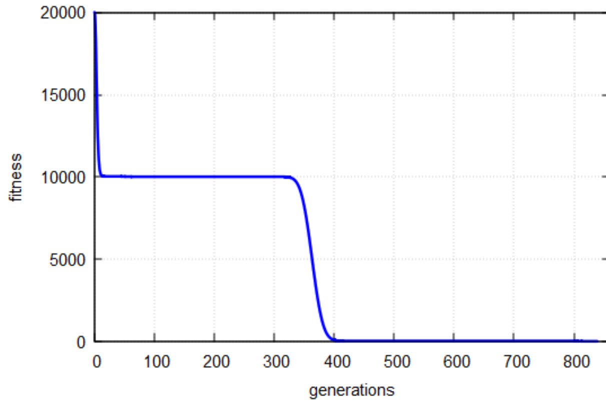


FIGURE 4. The fitness of the most efficient model that is produced during the training operation gradually converges to a global minimum.

improved during the generations of the various GA instances that run.

If a produced model successfully generates a valid mapping, it is stored in a common database where the agents search for appropriate models on any future request. In our simulations, the search criterion is the size of the SFC-graph. Upon a new request, each agent runs the models found in the database until a valid mapping is computed. In case that no model that generates a valid mapping has been found by any of the agents, the training procedure is initiated.

The GA that trains the models uses the current request as input in order to compute the fitness. As such, a model does not generalize on any given input; it is thereby necessary to train new models, as requests of different parameters arrive and network conditions change (these are the factors that define each problem configuration). The flow chart in Fig. 5 describes the procedures that the controller executes. This includes searching for a model and initiating the training procedure, in case no model has been found by the agents.

VII. COMPLEXITY ANALYSIS

In this section, we discuss the complexity of our proposed solution, as well as the communication overhead due to the clustering. Let $S = n^q$ be the solution space of the SFC-E problem. In essence, this corresponds to the number of permutations with repetition of q over a set of n elements, where $q = |V_v|$ and $n = |V_s|$. The proposed algorithm reduces the computation of this large solution space to a small subset of S.

The size of the cluster l that every agent computes is at most equal to $4 \times q$. An agent computes a mapping over a solution space $S_l \leq (4 \times q)^q$. Cumulatively all the agents examine a space of $S_r \leq n \times (4 \times q)^q$ solutions, which is substantially smaller than S. The clustering procedure excludes (from computation) infeasible solutions that do not meet bandwidth constraints, as well as inefficient mappings where the hop-count among the communicating VNFs is large.

The Algorithm 1 in controller node yields a complexity $O(n^2 + q)$. For the computational complexity of the learning algorithm, we take into consideration the operation of the NN

Algorithm 2 Training Genetic Algorithm

Input: neural network model, *population size*, *generations*, *supergenerations*
Output: weight assignment on neurons (best chromosome)

```
1:  run init1()
2:  Procedure init1()
3:  for t = 0 to supergenerations do
4:      for s = 0 to supergenerations do
            {stage 1}
5:          generate population1
6:          run init2(population1)
7:          store best chromosome in population2
8:      end for
        {stage 2}
9:      run init2(population2)
10:     store best chromosome in population3
11: end for
    {stage 3}
12: run init2(population3)
13: return best chromosome
14: Procedure init2(population)
15: compute deviation in population
16: if deviation > 0 then
17:     for i = 0 to generations do
18:         run crossover, mutation, selection
19:     end for
20: end if
21: return best chromosome
```

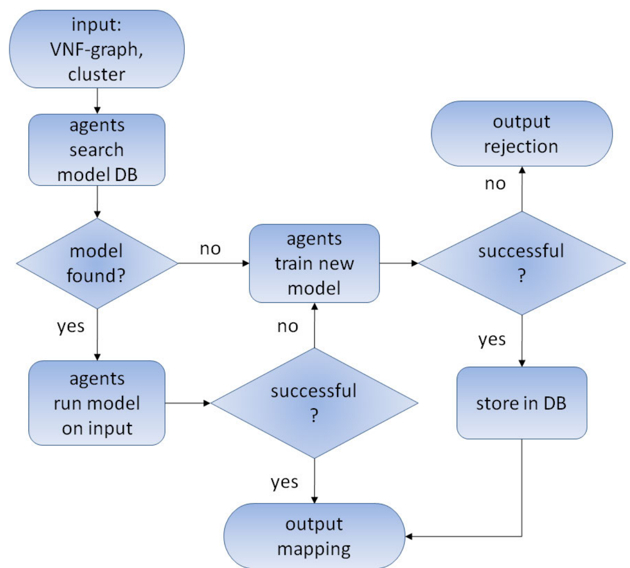


FIGURE 5. The flow of procedures that the controller executes. It requests the agents to search the model database and, in case no valid model has been found, they are requested to train a new model. These procedures either produce a valid mapping or designate a rejection in the case of failure.

that affects the execution and training of the model. Most implementations of artificial NNs exhibit quadratic complexity, as they are mainly based on matrix multiplication [29].

In our NN implementation, we employ an object-oriented approach in order to reduce complexity. Every NN node corresponds to a Java object that includes information about connections with other nodes and the adjacent links, which are also objects. Running the NN consists in merely passing the weight values through all its nodes and, as such, the complexity is linear to the NN size which is constant. As a result, the complexity of the proposed solution is polynomial and the computational burden imposed to the nodes is low.

During clustering, the messages exchanged represent at most q numerical values. The controller sends $n \times q$ messages to the agents (each message contains one substrate node during the cluster formation) and receives at most one message from each agent containing a set of n nodes or less and the fitness of the mapping. As such, the clustering procedure yields an insignificant communication overhead. NN models that are exchanged between the agents and the controller are represented by arrays of size equal to the number of edges of the NN (in our case, 210 numerical values). Therefore, in this case, the communication overhead remains low as well.

VIII. EVALUATION

In this section, we discuss our evaluation results. Initially, we present our evaluation environment (Section VIII-A) and then we elaborate on our comparison methods (Section VIII-B). The efficiency of our DL-based SFC embedding framework is assessed in Section VIII-C.

A. EVALUATION ENVIRONMENT

For our evaluations, we have implemented a simulation environment for SFC embedding onto 3-layer fat-tree datacenter (DC) network topologies, which comprise a representative NFV Point-of-Presence (PoP) [30].

1) NFV INFRASTRUCTURE

The simulations are conducted on a medium and a large-scale DC network. The former consists of 12 pods, containing 72 racks and 432 servers in total. The large-scale DC consists of 22 pods, 242 racks and 2662 server in total. The capacity of each server is 20 GHz, *i.e.,* 8 CPU cores at 2.5 GHz. The bandwidth of the links that connect the servers with their Top-of-the-Rack (ToR) switch is set to 1 Gbps. The links at the upper layers of the DC network are configured with a capacity of 10 Gbps.

2) SFC REQUESTS

Each SFC-graph request consists of a diverse number of VNFs, picked randomly within the range of 5 to 9. The computing demand for each VNF in the request varies between 2 and 6 GHz. The rest of the simulation parameters (related to the SFC requests) vary between the medium-scale and the large-scale topology. For the former, the bandwidth demands in the SFC-graph vary between 20 and 100 Mbps and, in each simulation, we generate and compute mappings for 6000 requests. In the large-scale topology, bandwidth demands range between 40 and 200 Mbps, while mappings are computed for 14000 requests. For each embedded SFC, a lifetime is picked randomly in the range $[0, t]$, where t corresponds to the number of requests that have to be processed before the SFC expires. In particular, we set $t = 1620$ and $t = 10000$ for the medium and large-scale topology, respectively,

The algorithms and the DC network simulator are implemented in Java 8. All distributed computing algorithms, in particular, are implemented with multiple threads in order to enable the parallel operation of the agents. The source code and compiled binaries of the simulator and algorithms are available in [21]. Our simulations are carried out on a workstation equipped with a 16-core Intel Xeon CPU at 2.1 GHz and 8 GB RAM.

B. COMPARISON METHODS

We compare our proposed solution against a state-of-the-art heuristic (*i.e.,* BACON) and a greedy algorithm.

1) BACON

BACON [3] is a centralized approach that shares common objectives with our method. More specifically, BACON is based on the criticality ranking of the VNFs, which is defined as proportional to the interconnections that it has with other VNFs of the SFC, *i.e.,* the degree of the corresponding node in the SFC-graph. We further consider the latency between any two communicating VNFs k, m mapped respectively to substrate nodes u, z in proportion to the length of the physical path $p_{u,z}$. BACON applies a server ranking criterion, namely *Betweeness Centrality* (BC), for choosing the most efficient substrate node for hosting each VNF. In structured topologies, such as fat-trees, the substrate nodes exhibit homogeneous properties, which results in the same ranking for substrate nodes. In order to alleviate this, in each iteration BACON considers only the servers that have sufficient capacity to host the VNF with the minimum demand. BACON ranks the servers in descending order based on BC, and, in every step, it searches for the server with the highest BC that generates an efficient mapping.

2) DISTRIBUTED GREEDY

Furthermore, our deep learning method is compared against a greedy algorithm (Algorithm 3), running on the distributed agents, that accepts the same input as our learning method and solves the VNF placement problem. The greedy algorithm sorts the cluster nodes in ascending order based on their residual capacity and then strives to place as many VNFs as possible onto the sorted nodes, starting from the beginning of the list. In essence, the greedy exercises VNF consolidation. However, it does not conduct an exhaustive search of the possible mappings and does not guarantee an optimal solution for each case.

C. EVALUATION RESULTS

We hereby present our evaluation results, comparing our Distributed Deep Learning method (DistrDL) against

Algorithm 3 Distributed Greedy Algorithm (DistrGr)

Input: cluster, SFC-graph
Output: *mapping, fitness*
1: sort cluster nodes in ascending order based on residual capacities
2: **for** $i = 0$ to cluster nodes **do**
3: **while** i is not mapped **do**
4: initialize variables k, m
5: **for** $v = 0$ to SFC-graph nodes **do**
6: **if** $m > r_i - d_v$ **then**
7: $m = r_i - d_v$
8: $k = v$
9: **end if**
10: **end for**
11: **if** a is null **then**
12: **return** rejection
13: **else**
14: map i to k
15: add m to *fitness*
16: **end if**
17: **end while**
18: **end for**
19: **if** not rejected **then**
20: **return** *mapping, fitness*
21: **end if**

BACON and the distributed greedy algorithm (DistrGr). Since BACON exhibits high solver runtime, we utilize it only in our simulations on the medium-scale DC network.

We initially assess the SFC embedding efficiency in terms of SFC request acceptance ratio. According to Fig. 6, DistrDL exhibits a small gain in comparison with the two other methods. Next, we measure the amount of traffic generated in the network at the inter-rack and intra-rack level. In terms of inter-rack traffic, DistrDL outperforms both BACON and DistrGR, conserving the largest amount of bandwidth at the upper layers of the fat-tree (Fig. 7). This essentially stems from our clustering approach, which inhibits the partitioning of SFCs among multiple racks. More significant gains are observed for DistrDL with respect to intra-rack traffic, as illustrated in Fig. 8. The reason behind this bandwidth saving within each rack is the higher degree of VNF consolidation achieved by DistrDL. Although BACON strives to place interacting VNFs on the same rack based on the BC criterion, in fact it does not necessarily co-locate them on the same server, thereby generating more intra-rack traffic than DistrDL.

We also employ the *Cost-to-Revenue Ratio (CRR)* in order to quantify the efficiency of the SFC embeddings, inline with [11]. To this end, we define the *Revenue* ($\mathbb{R}$) of a SFC request:

$$\mathbb{R} = \sum_{n \in N_v} d_n + a \sum_{e \in E_v} d_e$$

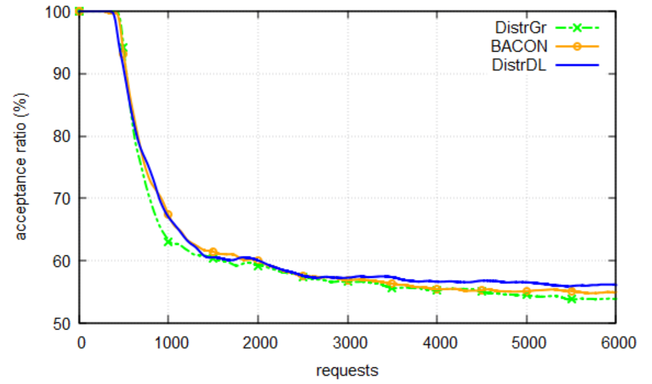


FIGURE 6. Request acceptance ratio on the medium-scale topology.

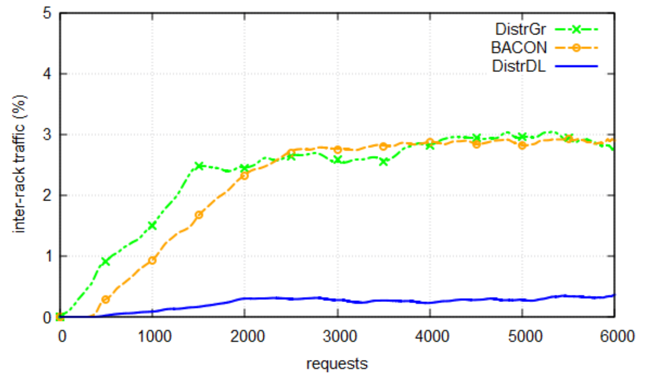


FIGURE 7. Inter-rack traffic generated by the embedded SFCs on the medium-scale topology.

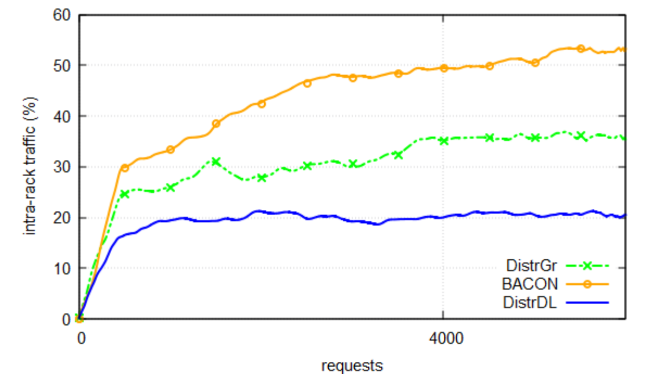


FIGURE 8. Intra-rack traffic generated by embedded SFCs on the medium-scale topology.

In essence, revenue accumulates all the node and link capacity demands of the SFC request. Furthermore, we define the *Embedding Cost* ($\mathbb{C}$) that essentially accumulates all node and link embedding costs, as follows:

$$\mathbb{C} = \sum_{n \in N_v} d_n + a \sum_{e \in E_v} d_e l_{e,p}$$

where $a = 0.5$ and substrate path p is assigned to virtual link e. Based on the definitions above, CRR is computed as: $CRR = \mathbb{C} / \mathbb{R}$. Note that the lower the CRR the better. Practically, CRR is mainly affected by the second term of $\mathbb{C}$, *i.e.,* paths with longer hop-count increase the embedding cost, and, thereby, the CRR. Note that in case of SFCs mapped on the same server, $CRR < 1$ as $l_{e,p} = 0$. Hence, high CRR

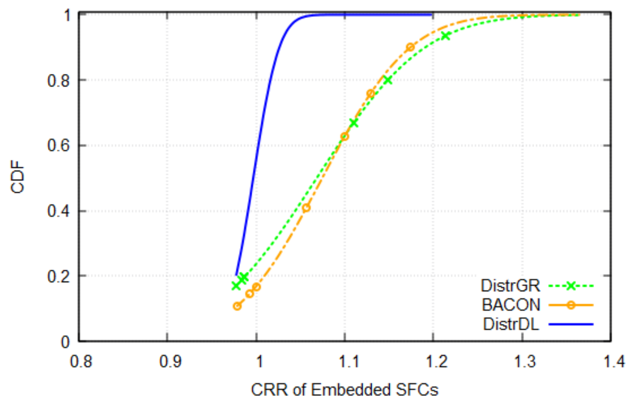


FIGURE 9. CDF of Cost-to-Revenue Ratio (CRR) on the medium-scale topology.

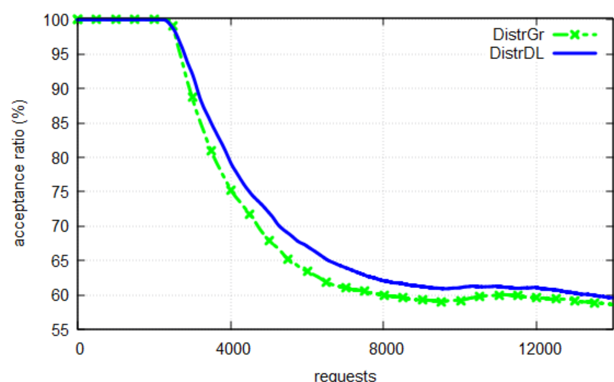


FIGURE 10. Request acceptance ratio on the large-scale topology.

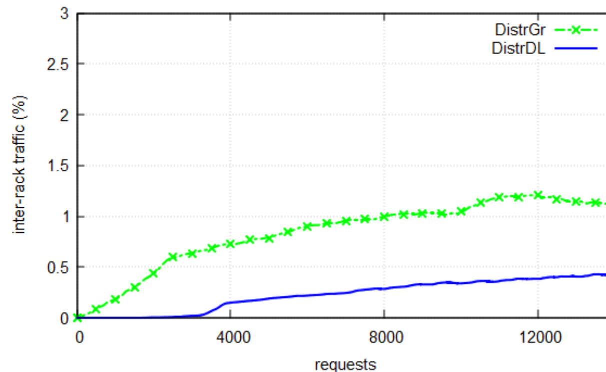


FIGURE 11. Inter-rack traffic generated by embedded SFCs on the large-scale topology.

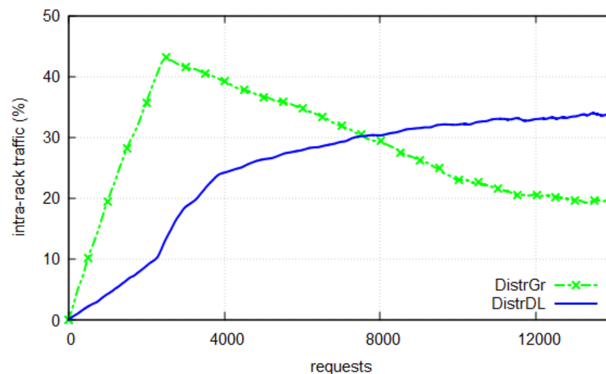


FIGURE 12. Intra-rack traffic generated by embedded SFCs on the large-scale topology.

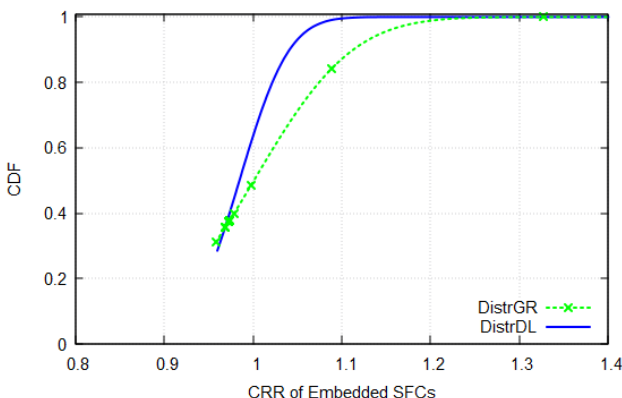


FIGURE 13. CDF of Cost-to-Revenue Ratio (CRR) on the large-scale topology.

values imply a high degree of SFC partitioning among servers and racks.

Fig. 9 depicts the CRR for all three methods. For DistrDL more than 60% of the embedded SFCs exhibit $CRR < 1$, implying that their interacting VNFs are co-located on the same server. In contrast, a lower degree of VNF co-location is indicated by the higher CRR values of BACON and DistrGr.

In the following, we perform a comparison among the distributed methods (*i.e.,* DistDL and DistrGr) on the large-scale DC network topology. Fig. 10 illustrates that both methods converge to the same level of request acceptance ratio. A notable margin between DistDL and DistrGr is observed in terms of the traffic that is generated in the network after the embedding of SFCs. Bandwidth conservation is achieved by DistDL at both inter-rack (Fig. 11) and intra-rack traffic (Fig. 12), with the gain being more notable in the latter due to the increased level of VNF consolidation by DistrDL. This is also corroborated by the CRR values, as shown in Fig. 13. Both CRR plots (Figs. 9 and 13) indicate an adaptability of DistrDL on networks of different scales.

Last, we compare the three methods in terms of solver runtime. According to Table 2, BACON yields a runtime of approximately 14 sec in the medium-scale topology, while its runtime explodes in the large-scale DC (>3500 sec). As such, we focus on the distributed methods that exhibit more competitive runtimes (Table 2). Both methods can compute SFC mappings in the order of msec for DC networks of more than 4000 nodes. Recall that the largest part of computation of any of the two distributed algorithms is executed in parallel on the substrate nodes. As such, the runtime is affected by the clustering procedure and the time required by the agents to generate a mapping. In DistrDL, we further consider the delay incurred for training a model, if a suitable one is not found. This delay is approximately 0.5 sec on average for an agent. Note that in our simulations, additional training was required only for 1% and 2.6% of the embedded SFCs in the medium-scale and large-scale topology, respectively. Consequently, this latency for training is rarely incurred, and thereby, its impact on the DistrDL's runtime is minimal.

TABLE 2. Solver runtime.

servers	DistrGR	DistrDL	BACON
432	0.05 ms	0.67 ms	14 sec
1024	0.13 ms	1.36 ms	188 sec
1458	0.22 ms	1.40 ms	522 sec
2662	0.32 ms	2.33 ms	3574 sec
4394	1.99 ms	5.11 ms	6899 sec

IX. RELATED WORK

Our distributed multi-agent framework couples the clustering of substrate nodes with distributed machine learning for the mapping of SFC graphs. In the following, we discuss related work on these topics.

A. DISTRIBUTED EMBEDDING AND NODE CLUSTERING

Numerous distributed frameworks have been proposed for virtual network embedding and provisioning [7], [10], [12]. One drawback of these solutions is the substantial communication overhead that they introduce. In particular, these methods establish communication among the substrate nodes, generating a significant amount of exchanged messages. This downside is also exhibited by methods based on the Pregel framework [31], such as [6].

Methods of static formation of the clusters lack efficiency and flexibility. A static clustering method is not able to adapt to different problem configurations, as by definition it follows a predetermined method of cluster formation, such as graph partitioning [32], [33], [34]. For instance, solutions that combine nodes from different clusters cannot be computed. In works such as [32] and [33], additional computation and memory consumption is required in order to examine such solutions. Furthermore, algorithms that support static formation consider clusters that do not meet the constraints for embedding. These clusters can be rejected by dynamic formation methods and, thereby, will not be considered for the computation of embedding solutions.

The clustering procedures partition the problem in smaller instances that are easier to solve. Then the agents and the distributed procedures compute instances of the embedding problem in their assigned clusters. As the problem is computationally hard, this approach increases the computational burden on each node. In our approach, each agent computes a restricted version of the problem. The algorithm applied by each agent yields low complexity based on unsupervised deep learning.

B. MULTI-AGENT ARCHITECTURES

The aforementioned methods are built on distributed procedures running on the substrate nodes. An alternative and more flexible approach is the development of multi-agent architectures that are mainly used in reinforcement learning methods.

Each agent explores its environment and gradually builds an action table for treading the incoming requests. The environment of an agent defines a part of the state space of the problem where a set of possible mapping is valid. Multi-agent architectures achieve a partitioning of the state space of the problem. On this design, the agents will collectively produce a near optimal solution.

The agents are not enforced to operate in certain elements of the network. This provides flexibility enabling the development of centralized methods [35], [36] where the agents function on a centralised fashion; decentralised methods [37] at which the agents are distributed across the network; hybrid methods combining both approaches [38]. In the aforementioned studies on centralised and decentralised methods, the functionality of the agents is synergistic. The agents share the produced knowledge in order to achieve a more effective exploration of their environment and reduce training time. In other studies the agents are non-cooperative [38], [39] and act independently competing to produce the best solution.

Each approach has certain advantages and limitations. Our approach relies on a distributed architecture where the agents operate at the network nodes. They achieve an efficient state space partitioning as each agent operates in a different cluster of nodes to which a different set of possible mappings corresponds. Although they do not implement a synergistic schema, the effective models generated by each agent during training are stored and remain available for future use by all agents. As such, the agents take advantage of previously generated knowledge, obviating the need for an entire training procedure.

C. DISTRIBUTED MACHINE LEARNING

The main objective of distributed machine learning applications on resource allocation problems is the development of a learning mechanism across the resources of a network, taking advantage of the local resources and data for each distributed procedure (*e.g.,* agent) [40]. The most common approach is that each agent applies some learning method, usually artificial NN training combined with reinforcement learning on a set of local data in order to generate a trained model. The trained models are then shared among the agents or combined with the models generated by the other agents, according to the sharing policy of the system in order to generate a global model or a set of models suitable for general use on demand by all agents.

Currently, the use of distributed ML in NFV orchestration comprises an emerging field of study. Distributed methods are used for the orchestration of SFCs [5], [15], [41], SFC elastic management [13], SFC-graph embedding with distributed versions of the Q-learning reinforcement algorithm [14], and for predicting VNF autoscaling on federated learning systems [4], [42]. However, these solutions are computationally expensive and are commonly evaluated in small-scale network topologies. As such, their efficiency at large scale remains questionable.

Instead, our proposed solution comprises an unsupervised NN training method of low complexity that does not stress

the operation of the agents. As indicated by our evaluation results, our approach yields high efficiency for large-scale networks.

X. CONCLUSION

In this paper, we elaborated on a decentralized approach in order to alleviate the computationally hardness of the SFC embedding problem, especially on large-scale NFV infrastructures. To this end, we utilized a multi-agent based clustering method that facilitates the search for efficient embeddings across a range of dynamically formed clusters. For the computation of the final SFC mapping, we leverage on unsupervised deep learning coupled with a novel training method based on genetic algorithms.

Our evaluation results indicate that our distributed embedding framework yields higher request acceptance rates and bandwidth conservation, both at intra- and inter-rack level, compared to BACON and a decentralized greedy algorithm. These gains of DistrDL are not achieved at the expense of solver runtime, which exhibits nice scalability properties in relation to the network size.

REFERENCES

[1] D. Dietrich, A. Abujoda, A. Rizk, and P. Papadimitriou, "Multi-provider service chain embedding with Nestor," *IEEE Trans. Netw. Service Manage.*, vol. 14, no. 1, pp. 91–105, Mar. 2017.

[2] C. Papagianni, P. Papadimitriou, and J. S. Baras, "Rethinking service chain embedding for cellular network slicing," in *Proc. IFIP Netw. Conf. Workshops*, May 2018, pp. 1–9.

[3] H. Hawilo, M. Jammal, and A. Shami, "Network function virtualization-aware orchestrator for service function chaining placement in the cloud," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 643–655, Mar. 2019.

[4] T. Subramanya and R. Riggio, "Centralized and federated learning for predictive VNF autoscaling in multi-domain 5G networks and beyond," *IEEE Trans. Netw. Service Manage.*, vol. 18, no. 1, pp. 63–78, Mar. 2021.

[5] H. Chen, S. Wang, G. Li, L. Nie, X. Wang, and Z. Ning, "Distributed orchestration of service function chains for edge intelligence in the industrial Internet of Things," *IEEE Trans. Ind. Informat.*, vol. 18, no. 9, pp. 6244–6254, Sep. 2022.

[6] Q. Zhang, X. Wang, I. Kim, P. Palacharla, and T. Ikeuchi, "Vertex-centric computation of service function chains in multi-domain networks," in *Proc. IEEE NetSoft Conf. Workshops (NetSoft)*, Jun. 2016, pp. 211–218.

[7] F. Esposito, D. Di Paola, and I. Matta, "On distributed virtual network embedding with guarantees," *IEEE/ACM Trans. Netw.*, vol. 24, no. 1, pp. 569–582, Feb. 2016.

[8] P. Rodis and P. Papadimitriou, "Intelligent network service embedding using genetic algorithms," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Sep. 2021, pp. 1–7.

[9] E. Amaldi, S. Coniglio, A. M. C. A. Koster, and M. Tieves, "On the computational complexity of the virtual network embedding problem," *Electron. Notes Discrete Math.*, vol. 52, pp. 213–220, Jun. 2016.

[10] I. Houidi, W. Louati, D. Zeghlache, P. Papadimitriou, and L. Mathy, "Adaptive virtual network provisioning," in *Proc. 2nd ACM SIGCOMM Workshop Virtualized Infrastructure Syst. Architectures*, Sep. 2010, pp. 41–48.

[11] D. Dietrich and P. Papadimitriou, "Policy-compliant virtual network embedding," in *Proc. IFIP Netw. Conf.*, Jun. 2014, pp. 1–9.

[12] I. Houidi, W. Louati, and D. Zeghlache, "A distributed virtual network mapping algorithm," in *Proc. IEEE Int. Conf. Commun.*, 2008, pp. 5634–5640.

[13] A. Dalgkitsis, L. A. Garrido, P.-V. Mekikis, K. Ramantas, L. Alonso, and C. Verikoukis, "SCHEMA: Service chain elastic management with distributed reinforcement learning," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2021, pp. 1–6.

[14] P. T. A. Quang, Y. Hadjadj-Aoul, and A. Outtagarts, "A deep reinforcement learning approach for VNF forwarding graph embedding," *IEEE Trans. Netw. Service Manage.*, vol. 16, no. 4, pp. 1318–1331, Dec. 2019.

[15] H. Huang, C. Zeng, Y. Zhao, G. Min, Y. Zhu, W. Miao, and J. Hu, "Scalable orchestration of service function chains in NFV-enabled networks: A federated reinforcement learning approach," *IEEE J. Sel. Areas Commun.*, vol. 39, no. 8, pp. 2558–2571, Aug. 2021.

[16] B. Bonet and H. Geffner, "Planning as heuristic search," *Artif. Intell.*, vol. 129, nos. 1–2, pp. 5–33, Jun. 2001.

[17] I. Pohl, "Heuristic search viewed as path finding in a graph," *Artif. Intell.*, vol. 1, nos. 3–4, pp. 193–204, Jan. 1970.

[18] Y. Lu and J. Lu, "A universal approximation theorem of deep neural networks for expressing probability distributions," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 3094–3105.

[19] L. Lu, P. Jin, G. Pang, Z. Zhang, and G. E. Karniadakis, "Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators," *Nature Mach. Intell.*, vol. 3, no. 3, pp. 218–229, Mar. 2021.

[20] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[21] (2023). *SFC Embedding Simulator and Algorithms*. [Online]. Available: https://rodispantelis.github.io/SFC-Embedding/

[22] S. Wu, G. Li, F. Chen, and L. Shi, "Training and inference with integers in deep neural networks," in *Proc. Int. Conf. Learn. Represent.*, 2018, pp. 1–14.

[23] V. Plagianakos and M. Vrahatis, "Neural network training with constrained integer weights," in *Proc. Congr. Evol. Comput. (CEC)*, vol. 3, 1999, pp. 2007–2013.

[24] S. Draghici, "On the capabilities of neural networks using limited precision weights," *Neural Netw.*, vol. 15, no. 3, pp. 395–414, Apr. 2002.

[25] S. Ding, C. Su, and J. Yu, "An optimizing BP neural network algorithm based on genetic algorithm," *Artif. Intell. Rev.*, vol. 36, no. 2, pp. 153–162, Aug. 2011.

[26] F. H. F. Leung, H. K. Lam, S. H. Ling, and P. K. S. Tam, "Tuning of the structure and parameters of a neural network using an improved genetic algorithm," *IEEE Trans. Neural Netw.*, vol. 14, no. 1, pp. 79–88, Jan. 2003.

[27] G. F. Miller, P. M. Todd, and S. U. Hegde, "Designing neural networks using genetic algorithms," in *Proc. ICGA*, vol. 89, 1989, pp. 379–384.

[28] J. N. D. Gupta and R. S. Sexton, "Comparing backpropagation with a genetic algorithm for neural network training," *Omega*, vol. 27, no. 6, pp. 679–684, Dec. 1999.

[29] P. Orponen, "Computational complexity of neural networks: A survey," *Nordic J. Comput.*, vol. 1994, no. 1, pp. 94–110, 1994.

[30] T. Wang, Z. Su, Y. Xia, and M. Hamdi, "Rethinking the data center networking: Architecture, network protocols, and resource sharing," *IEEE Access*, vol. 2, pp. 1481–1496, 2014.

[31] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.

[32] W. Lin, "Large-scale network embedding in apache spark," in *Proc. 27th ACM SIGKDD Conf. Knowl. Discovery Data Mining*, Aug. 2021, pp. 3271–3279.

[33] A. A. Nasiri, F. Derakhshan, and S. S. Heydari, "Distributed virtual network embedding for software-defined networks using multiagent systems," *IEEE Access*, vol. 9, pp. 12027–12043, 2021.

[34] A. Song, W.-N. Chen, T. Gu, H. Yuan, S. Kwong, and J. Zhang, "Distributed virtual network embedding system with historical archives and set-based particle swarm optimization," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 51, no. 2, pp. 927–942, Feb. 2021.

[35] A. Pentelas, D. D. Vleeschauwer, C.-Y. Chang, K. D. Schepper, and P. Papadimitriou, "Deep multi-agent reinforcement learning with minimal cross-agent communication for SFC partitioning," *IEEE Access*, vol. 11, pp. 40384–40398, 2023.

[36] S. Wang, C. Yuen, W. Ni, Y. L. Guan, and T. Lv, "Multiagent deep reinforcement learning for cost- and delay-sensitive virtual network function placement and routing," *IEEE Trans. Commun.*, vol. 70, no. 8, pp. 5208–5224, Aug. 2022.

[37] T. Catena, V. Eramo, M. Panella, and A. Rosato, "Distributed LSTM-based cloud resource allocation in network function virtualization architectures," *Comput. Netw.*, vol. 213, Aug. 2022, Art. no. 109111.

[38] Y. Zhu, H. Yao, T. Mai, W. He, N. Zhang, and M. Guizani, "Multiagent reinforcement-learning-aided service function chain deployment for Internet of Things," *IEEE Internet Things J.*, vol. 9, no. 17, pp. 15674–15684, Sep. 2022.

[39] P. T. A. Quang, A. Bradai, K. D. Singh, and Y. Hadjadj-Aoul, "Multi-domain non-cooperative VNF-FG embedding: A deep reinforcement learning approach," in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, Apr. 2019, pp. 886–891.

[40] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, ''A survey on distributed machine learning,'' *ACM Comput. Surv.*, vol. 53, no. 2, pp. 1–33, 2020.

[41] P. Zhang, Y. Zhang, N. Kumar, and M. Guizani, ''Dynamic SFC embedding algorithm assisted by federated learning in space–air–ground integrated network resource allocation scenario,'' *IEEE Internet Things J.*, vol. 10, no. 11, pp. 9308–9318, Jun. 2022.

[42] R. Verma and K. M. Sivalingam, ''Federated learning approach for auto-scaling of virtual network function resource allocation in 5G-and-beyond networks,'' in *Proc. IEEE 11th Int. Conf. Cloud Netw. (CloudNet)*, Nov. 2022, pp. 242–246.

PANTELEIMON RODIS received the B.Sc. degree in computer science and the M.Sc. degree in engineering of pervasive computing systems from Hellenic Open University, in 2011 and 2020, respectively. He is currently pursuing the Ph.D. degree with the Department of Applied Informatics, University of Macedonia, Greece. His research interests include the applications of artificial intelligence in virtual network embedding and VNF orchestration.

PANAGIOTIS PAPADIMITRIOU (Senior Member, IEEE) received the B.Sc. degree in computer science from the University of Crete, Greece, in 2000, the M.Sc. degree in information technology from the University of Nottingham, U.K., in 2001, and the Ph.D. degree in electrical and computer engineering from the Democritus University of Thrace, Greece, in 2008. He is currently an Associate Professor with the Department of Applied Informatics, University of Macedonia, Greece. Before that, he was an Assistant Professor with the Communications Technology Institute, Leibniz Universität Hannover, Germany, and a member of the L3S Research Center, Hanover. He has been a (co-)PI in several EU-funded (e.g., NEPHELE, NECOS, T-NOVA, and CONFINE) and nationally-funded projects (e.g., G-Lab VirtuRAMA and MESON). He was a recipient of Best Paper Awards at IFIP WWIC 2012 and IFIP WWIC 2016, and the runner-up Poster Award at ACM SIGCOMM 2009. He has co-chaired several international conferences and workshops, such as IFIP/IEEE CNSM 2022, IFIP Networking TENSOR 2020–2023, IEEE NetSoft S4SI 2020, IEEE CNSM SR+SFC 2018–2019, IFIP WWIC 2017–2016, and INFOCOM SWFAN 2016. He is also an Associate Editor of IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT. His research interests include (next-generation) internet architectures, network processing, programmable dataplanes, time-sensitive networking (TSN), and edge computing.