On the Use of Graph Neural Networks for Virtual Network Embedding

Anouar Rkhami, Tran Anh Quang Pham, Yassine Hadjadj-Aoul, Abdelkader Outtagarts, Gerardo Rubino

On the Use of Graph Neural Networks for Virtual Network Embedding

Anouar Rkhami[1], Tran Anh Quang Pham[1], Yassine Hadjadj-Aoul[2], Abdelkader Outtagarts[3], and Gerardo Rubino[1]

[1]Inria, Univ. Rennes, CNRS, IRISA
[2]Univ. Rennes, Inria, CNRS, IRISA
[3]Nokia-Bell Labs, France

***Abstract*—Resource allocation of 5G network slices is one of the most important challenges for network operators. It can be formulated using the Virtual Network Embedding (VNE) problem, which was and remains an active field of studies, also known because of its NP-hardness. Owing to its complexity, several heuristics, meta-heuristics and Deep Learning-based solutions have been proposed. However, these solutions are inefficient either due to their slowness or to not taking into account the structure of data which results in an inefficient exploration of the solutions space. To overcome these issues, in this work we unveil the potential of Graph Convolutional Neural (GCN) networks and Deep Reinforcement Learning techniques in solving the VNE problem. The key point of our approach is modeling of the VNE problem as an episodic Markov Decision Process which is solved in a Reinforcement Learning fashion using a GCN-based neural architecture. The simulation results highlight the efficiency of our approach through an increased performance over time, while outperforming state-of-art solutions in terms of the services' acceptance ratio.**

***Index Terms*—Virtual Network Embedding, Deep Reinforcement Learning, Graph Neural Networks, 5G.**

I. Introduction

With the recent emergence of the fifth generation of mobile networks (5G) and the diversification of the services to be supported, Infrastructure Providers (InPs) are compelled to adapt to these new changes, requiring in particular the deployment of constrained services almost instantaneously [1].

As hardware solutions are too rigid and unsuitable for real-time deployment, InPs are moving towards new scalable software solutions available with the advent of the Network Function Virtualization (NFV) paradigm [2]. NFV enables the co-existence of several virtual network requests (VNRs) on top of the same substrate network (SN). The problem of embedding a VNR into the SN, which is known as the Virtual Network Embedding (VNE) problem, is considered as the main resource allocation challenge in network virtualization. Indeed, it has been qualified as a very high computational complexity problem [3].

One of the most important challenges in VNE is the placement of network services efficiently and automatically, which would make it possible to converge to a fully automated network, also referred to as a zero-touch network [4].

Several solutions have already been proposed in the literature to address these types of problems [5]. Most existing techniques solve them using classical combinatorial optimization methods. These tools, although effective for very small network instances, reach their limits even for fairly small-scale networks. Moreover, it is difficult to apply them in a real context because of the lack of accurate models [6] for several metrics, such as latency and loss, which actually result from the placement. Other approaches propose heuristics that can be executed in near real time, at the cost of a very partial exploration of the space of solutions [7]. Metaheuristics, and in particular evolutionary algorithms, have also been proposed to attack these difficulties [8]. These techniques, although efficient, can present some convergence issues and therefore be slow. On the other hand, with these techniques you can solve problems without learning how to solve them, so you don't benefit from past experiences. More recently, techniques based on Deep Reinforcement Learning (DRL) have been used [9]. These techniques show some effectiveness. However, in some cases they rely on a manual feature extraction or they use neural networks which are not suitable to graph-structured data. Moreover, they put particular assumptions on the shape of the substrate and virtual networks. Besides, these techniques are very slow, which could be an obstacle to their adoption to control real networks.

In this work, we propose an end-to-end solution based on DRL and Graph Convolutional Neural networks (GCNs) [10] to train an agent to solve the VNE problem.

The contributions of this paper can be summarized as follows:

- formalization of the VNE problem as an MDP,
- features' extraction automation using GCN-based neural architecture,
- proposition of a more efficient placement strategy.

The rest of this paper is structured as follows. The formulation of VNR embedding problem is presented in Sec. III. The proposed framework is presented in Sec. IV. The comparison of the proposed framework and other approaches is presented in Sec. V. Finally, we discuss the advantages and disadvantages of the proposed framework in Sec. VI.

II. Background

Graph Neural Networks (GNNs), are a type of neural networks dedicated to graph structured data. They were in-

Notation	Description
$\mathcal{G}^s, \mathcal{G}^v$	The substrate network and VNR graphs
$\mathbb{N}^s, \mathbb{N}^v$	Set of substrate nodes and VNFs
$\mathbb{L}^s, \mathbb{L}^v$	Set of substrate links and VLs
$\mathbb{P}^s$	Set of substrate paths
c_{n^s}	Available CPU at substrate node n^s
b_{l^s}	Available bandwidth of substrate link l_s
c_{n^v}	CPU request of VNF n^v
b_{l^v}	Bandwidth request of VL l_v

TABLE I
NOTATIONS

troduced in [11] and several variants of it were proposed [10], [12], [13]. The most known one is the Graph Convolutional Networks (GCN), which generalizes the convolution operation from euclidean data (images and grid shaped data) to non-euclidean data (graphs). The key objective of GCNs is to learn a function that generates for each node in a graph a vector representation by aggregating its own features and its neighbors features.

Let $\mathcal{G} = (V, E)$ be a graph, where V is the set of its nodes and E the set of edges. We assume that each node $n \in V$ is characterized by an input feature vector in_n. The graph convolution operation takes as input the feature vector in_n and outputs a new vector out_n according to the following equation:

$$out_n = f\Big(\sum_{m \in \mathcal{N}(n)} \frac{1}{\sqrt{d_m d_n}} in_m W^l\Big) \tag{1}$$

where $\mathcal{N}(n)$ represents the set composed of node n and its neighbours, d_m is the degree of node m plus one, W^l is the weight matrix of the l^{th} GCN layer, and f is an activation function.

III. Problem Definition

In this section, we describe the model of the SN, the VNR and their resources. Then, we define formally the VNE problem. The notations used in this section are shown in Table I.

A. Networks and resources Models

The susbtrate network is defined by an undirected graph $\mathcal{G}^s = (\mathbb{N}^s, \mathbb{L}^s)$, where $\mathbb{N}^s$ represents the set of substrate nodes and $\mathbb{L}^s$ the set of substrate links. The number of the substrate nodes and substrate links are $|\mathbb{N}^s|$ and $|\mathbb{L}^s|$ respectively. Each substrate node n^s has a computing capacity c_{n^s} and each substrate link l^s has a bandwidth capacity b_{l^s}. Similarly, the VNR is represented by a directed graph $\mathcal{G}^v = (\mathbb{N}^v, \mathbb{L}^v)$, where $\mathbb{N}^v$ represents the set of VNFs and $\mathbb{L}^v$ represents the set of virtual links. The numbers of VNFs and VLs are $|\mathbb{N}^v|$ and $|\mathbb{L}^v|$ respectively. Each VNF n^v has a computing demand denoted as c_{n^v}, and each VL l^v has a bandwidth demand b_{l^v}.

B. VNE Problem

The VNE problem can be divided into two stages: the virtual node mapping VNM and the virtual link mapping VLM. The former is defined as a function $f_{\text{VNM}}\colon\ N^v \rightarrow N^s$ that maps a virtual node to a substrate node. It must be injective, that is, two VNFs of the same VNR can not be hosted by the same substrate network. On the other hand, VLM can be modeled as a function $f_{\text{VLM}}\colon\ \mathbb{L}^v \rightarrow \mathbb{P}^s$ that maps a virtual link to a physical path (set of physical links).

At the node mapping level, a virtual node is successfully deployed if the substrate node that hosts it has sufficient CPU resources, i.e. if

$$c_{n^v} \leq c_{f_{\text{VNM}}(n^v)}, \quad \forall n^v \in \mathbb{N}^v. \tag{2}$$

At the link mapping stage, a VL is successfully deployed if its virtual nodes are deployed and if each physical link belonging to the physical path on which it is mapped has sufficient bandwidth, i.e, if

$$b_{l^v} \leq \min_{l^s \in f_{vlm(l^v)}} b_{f_{\text{VLM}(l^v))}}, \quad \forall l^v \in \mathbb{L}^v. \tag{3}$$

In this work, the aim is to maximize the number of accepted VNRs. A VNR is accepted if and only if the placement of its VNFs and VLs satisfies, respectively, the constraints (2) and (3).

IV. Proposed Method

We formulate the VNE problem as a Reinforcement Learning problem, in which an agent finds iteratively a sub-optimal placement of a VNR on top of the SN.

In what follows we present an overview of the Markov Decision Process (MDP) formulation of the VNE problem, the neural architecture of the RL agent we used to solve it and its training process.

A. Deep Reinforcement Learning overview

Reinforcement Learning (RL) is the area of machine learning that deals with sequential decision-making tasks. It is formulated as the interaction of an agent with an environment to optimize a given total amount of return. This interaction can be modelled as a Markov Decision Process (MDP) M = ($\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$, $\mathcal{R}$) with a state space $\mathcal{S}$, an action space $\mathcal{A}$, transition dynamics $\mathcal{P}$ and a reward function $\mathcal{R}$. At each time step t, the agent observes a state s_t and performs an action a_t, then it receives a reward r_t as well as the next state of the environment s_{t+1}. If the agent receives a special state called *terminal*, the interaction stops. The objective of the agent is to maximize the expected cumulative discounted return:

$$R_t = \mathbb{E}[\sum_t \gamma^t r_t], \quad \gamma \in [0, 1]. \tag{4}$$

The parameter γ is the discount factor, by which we ensure the convergence of the cumulative return in the infinite horizon setups. The behavior of the agent is defined by a policy π which maps a state s to a distribution over the actions A. Each state s is associated with a value function $V^\pi(s)$ that maps it to a scalar corresponding to the expected reward the agent will receive in this state and acting according to the policy π:

$$V^\pi(s) = \mathbb{E}_\pi[R_t | s_t = s]. \tag{5}$$

In traditional RL, both the policy and the value functions are modeled by tables. However, in most real scenarios, the state and action spaces are large, therefore instead of the tabular representation, the policy and the value functions are approximated by Deep Neural Networks (DNNs). DNNs are proposed for two main reasons: to overcome the slowness of the tabular-based solutions and to extract automatically useful features from the state representation. The combination of RL with DNNs is known as Deep Reinforcement Learning (DRL). To learn the policy function, several DRL algorithms were introduced. They can be divided into three main categories: value-based methods, policy-based methods and actor-critic methods. The policy-based solutions parameterize the policy function directly by a neural architecture and try to find the optimal policy using gradient ascent, while the value-based solutions estimate the value of each state using a DNN and then infer the optimal policy using this estimation. Finally, the actor-critic methods are a combination of the above methods: the agent learns both functions to find the optimal policy. One of the most widely used actor-critic algorithms is A2C [14].

B. VNE as MDP

In this section, we describe our MDP formulation of the VNE problem and its main components, namely: the state representation, the actions, the rewards and the transition dynamics.

State Representation: The state s is represented by the two graphs $(\mathcal{G}^s, \mathcal{G}^v)$. Each node $n^s \in \mathbb{N}^s$ has two features: (1) available cpu c_{n^s}, (2) sum of bandwidth defined as the total available bandwidth of links to which belongs n^s. On the other hand each VNF $n^v \in \mathbb{N}^v$ is characterized by four features: (1) cpu required by the VNF, (2) total bandwidth required by the links to which belongs the VNF, (3) a flag indicating whether this node is the current node to place, (4) a flag indicating whether the VNF was placed. At the initial state s_0, the current flag of the first VNF is assigned to 1 and the others to -1, and the placement flag is set to -1 to all VNFs.

Action Space: At the time step t in the MDP, the agent has to select a substrate node to host the positive-flagged VNF (+1). Note that the VNFs of the same VNR should not be placed at the same substrate node. Hence the action space at time step t is defined as follows:

$$\mathcal{A}_t = \mathbb{N}^s \backslash \mathbb{S}_t \tag{6}$$

where $\mathbb{S}_t$ represents the set of substrate nodes selected before the time step t. Initially, the action space $\mathcal{A}_0$ is equal to $\mathbb{N}^s$.

State Transition Dynamics: At time step t, the agent observes the state s_t and selects an action $a_t = n^s$ to place the current VNF n^v. The MDP then transits to a new state s_{t+1} in which the node features of the two graphs experience the following updates:

- If the selected substrate node has sufficient CPU resource, the VNF n^v will be placed at the substrate node n^s. Then, the CPU resource of n^s is updated by $c_{n^s} := c_{n^s} - c_{n^v}$
- After each successful VNF placement, the VLs related to the VNF and having both head and tail VNFs successfully deployed are considered. The shortest path $l^s \in \mathbb{P}^s$ between the head VNF and the tail VNF is computed. If the bandwidth required by this VL is less than the one offered by all physical links belonging to the obtained physical path, then the available bandwidth of each physical link in this physical path is updated as follows:

$$b_{l^s} = b_{l^s} - b_{l^v}, \quad \forall l^s \in \mathbb{P}^s. \tag{7}$$

- If the previous conditions are satisfied, the placement flag of the current node is set to 1, and another non visited VNF is chosen as the current node.

The placement process of the VNR will at most ends in $|\mathbb{N}^v|$ steps.

Reward design: The objective is to maximize the number of accepted requests. To guide the agent to this objective we introduce intermediate and final rewards. When a VNF is placed successfully, it receives an intermediate reward of 0. Final rewards quantify the quality of the final decision made by the agent. Thus, the agent obtains a unit of reward when it successfully places a VNR. Otherwise, it gets a reward of -1 if the VNR is not placed successfully, i.e. if there exists a resource violation at some step in the process.

C. The agent architecture

To solve the MDP defined above, we adopted the actor-critic framework [15]. The DRL agent has to approximate the policy and the state value functions. To implement the actor-critic framewrok, we can either use two distinct neural network architectures or one neural network architecture with two heads, one representing the policy function and the other one the value function. We adopt the last approach, because it lets the actor and critic share the same low level features and use them in different ways.

The two functions are parameterized by neural networks. Their inputs must be then a real-valued vector. However, as we stated earlier the state is represented by two graphs (SN and VNR), hence we must encode the graph-structured information of the state into a real-valued vector.

To achieve this, we proposed an end-to-end architecture represented in Fig. 1. It takes the state of the environment as an input and encodes it to a real valued vector which is fed to the two heads described above. Finally, the output corresponds to the prediction of the policy and the value functions. The key idea is that learning a good representation of the state will led to a fast and accurate optimal policy prediction.

Y. Bai [16] proposed a GCN-based architecture to solve the graph similarity computation task using supervised learning. We adapt their architecture to our problem for the state encoding steps. The encoding of the state into a real valued vector involves three main steps:

- **Node-level encoding using GCNs.** This stage transforms each node of the graph into a vector encoding its features and structural properties. Based on the raw features on

each node of the two graphs $\mathcal{G}^s$ and $\mathcal{G}^v$, we use GCNs to aggregate neighborhood information of each node (Eq. (1)). After multiple layers of GCNs, we obtain two matrices $U^s \in \mathbb{R}^{|N^s| \times D}$ and $U^v \in \mathbb{R}^{|N^v| \times D}$ that represent, respectively, the node-level encoding of SN and VNR. Each row u_n represents the encoding of node n and D is an hyperparemeter representing the dimension of u_n. Note that we used two different GCNs modules to encode SN and VNR.

- **Graph level encoding using attention layers.** After getting the node level encoding matrices U^s and U^v, we can directly use them to encode the graph structure. However, this can be computationally expensive. To overcome this issue and make the representation of the graph independent of its size, we can perform a weighted sum of node representations. The choice of nodes that get higher weights depends on the current node we want to place. Thus, we use an attention mechanism to learn these weights instead of considering fixed ones. For each graph, we first compute a graph context *ct*, then we compute the attention weight for each node in the graph.
  For the VNR we define the context ct_v as the node representation of the current VNF followed by a nonlinear transformation, $ct_v = g_v(u_c \times W_c)$, where $u_c \in \mathbb{R}^D$ is the node representation of the current VNF, $W_c \in \mathbb{R}^{D \times D}$ is a learnable weight matrix and g_v is an activation function. Based on ct_v we compute one attention weight a_{n^v} for each VNF n^v as the inner product between the context ct_v and its encoding u_{n^v}: $a_{n^v} = u_{n_v}^T ct_v$. Finally the VNR encoding h_v is given by

$$h_v = \sum_{n^v \in \mathbb{N}^v} a_{n^v} u_{n^v}. \tag{8}$$

  Regarding the SN we define its context ct_s by a simple average of nodes encoding followed by a non linear transformation $ct_s = g_s((N^{-1} \sum_{n=1}^{|N^s|} u_n) W_s)$, where W_s is a learnable weight matrix and u_n is the encoding of the n^{th} susbtrate node. Similarly we define the encoding of the SN as a weighted sum of the substrate nodes encodings. The weight of each node equals the inner product between its encoding and the context,

$$h_s = \sum_{n^s \in \mathbb{N}^s} a_{n^s} u_{n^s}. \tag{9}$$

- **State level encoding using neural tensor networks (NTN).** The previous stages let the agent learn how to model the two graphs of the state. The last step consists in learning how to model their relation. Following [17] and based on the vector representation of SN and VNR, we use Neural Tensor Networks (NTN) to achieve this:

$$g(h_s, h_v) = f_3(h_v^T W_3^{[1:K]} h_v + V[^{h_s}_{h_v}] + b_3), \tag{10}$$

  where $W_3^{[1:K]}$ is a weight tensor, [] denotes the concatenation operation, V is a weight vector, b_3 is a bias, and f_3 is an activation function; K is a hyperparameter.

D. Training process

As stated before, we adopt the actor-critic setup. To update the parameters of the architecture we presented above, we used the Advantage Actor Critic (A2C) algorithm.

V. Simulation Results

In this section, we present the simulation setup as well as an analysis of the obtained results.

A. Simulation Setup

To evaluate the performance of the proposed approach, we consider the following setup:

- The substrate network is modeled by the network topology of BtEurope[1]. It contains 24 nodes and 37 links. The capacity of the substrate nodes and links is drawn uniformly from the interval $[50, 100]$.
- To generate the virtual requests, we used the Erdős–Rényi model [18]. In this model the generated graph is defined by the number of nodes n and the probability p of creating an edge between the nodes. With, for instance, $p = 2 \ln n/n$, the generated graph is in general connected (more precisely, the probability that it is the case goes to one as $n \to \infty$). The requested resources (CPU and BW) of VNRs are drawn randomly following a uniform distribution from the interval $[1, 10]$. The system operates in an episodic manner; during each episode, VNRs come to the system sequentially: once a VNR is processed, the next one arrives. There are 3000 episodes and during each one 30 VNRs come to the system. At the end of each episode VNRs leave the system. We execute 5 runs with different seeds.
- For the model architecture, it is written in `Python` with the `Pytorch` library[2], and the DGL library[3]. The neural architecture is constructed with the following hyperparameters. We set the numbers of GCN layers to 3, and we use $tanh$ as the activation function. For the NTN layer we set K to 8. We use then 2 fully connected layers and finally we have two heads, each one modeled by a fully connected layer that represent, respectively, the policy network and the value network. The learning rate is set to 10^{-3} and the discount factor γ is set to 0.99. To train the model, the Adam optimizer was used [19].

B. Impact of the hidden units

The number of hidden units used in each layer of the agent's neural architecture impacts both the quality of the solution obtained and the computational cost to get it.

In this section we study the effect of the number of hidden units on the performance of the DRL agent in order to find a trade-off between these two metrics. We consider four different agents: agent16, agent32, agent64 and agent128 which have, respectively, 16, 32, 64 and 128 hidden units in all their hidden layers.

[1] http://www.topology-zoo.org/dataset.html
[2] https://pytorch.org/
[3] https://www.dgl.ai/

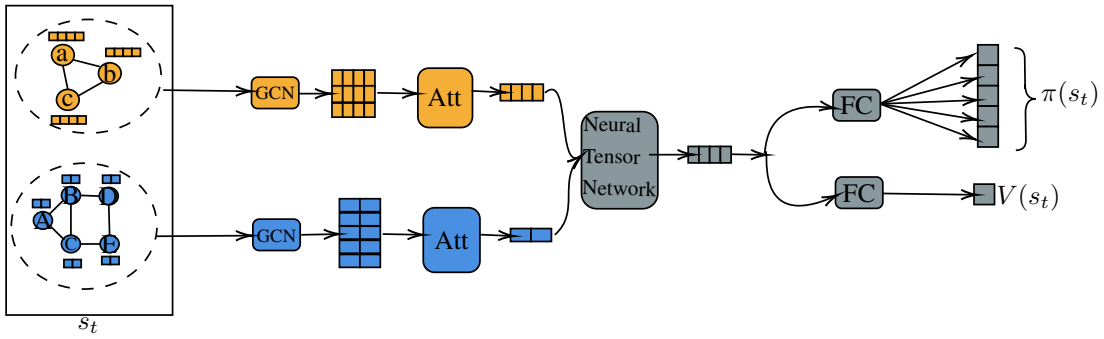

Fig. 1. An overview illustration of the proposed end-to-end agent neural architecture. Given the state of the environment,the agent has to approximate two functions: the policy function π and the value function V.

The results illustrated in Figures 2, 3 and 4 show, respectively, the performance in terms of number of accepted requests, number of VNFs deployed and number of VLs deployed. As we see in these figures, the agents with 64 and 128 hidden units are better than the other ones and their performance is more stable.

Figure 5 shows the required time to finish one episode for each configuration. The more hidden units we have the more time we need to finish the episode. The time required to finish one episode with the agent with 128 hidden units is greater than the one with 64 hidden units, while the performance is quite the same. Consequently we use the agent with 64 hidden units to compare its performance against other solutions.

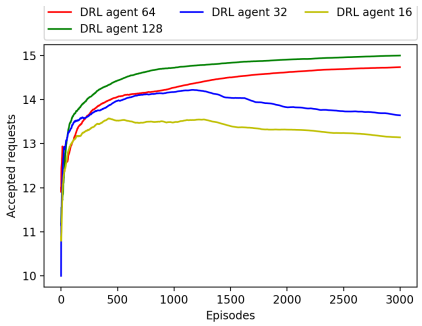

Fig. 2. Number of accepted requests vs Number of hidden units.

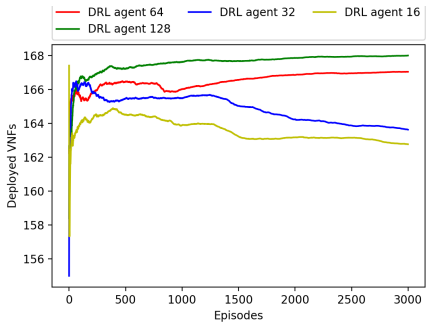

Fig. 3. Number of accepted VNFs vs Number of hidden units.

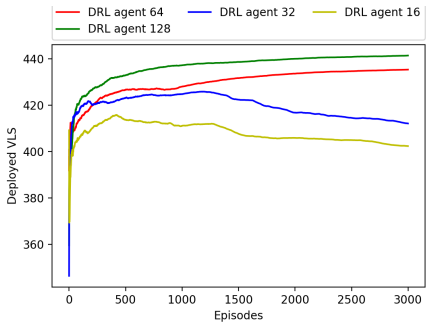

Fig. 4. Number of accepted VLs vs Number of hidden units.

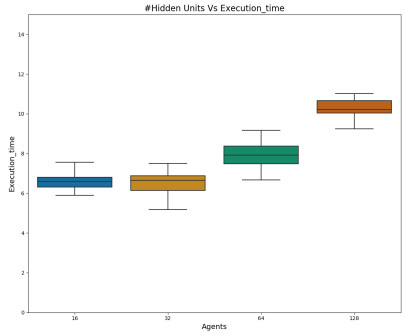

Fig. 5. Execution time VS Number of hidden units.

C. Performance Benchmarking

To benchmark the performance of our DRL-based solution, we compare it against the following approaches:

- **Random agent:** It preforms randomly to choose the substrate nodes that will host the VNFs. Based on the obtained VNF mapping, the agent uses the shortest path algorithm to find the link mapping of the Vls,
- **First Fit:** It adopts the First-Fit (FF) algorithm to allocate the VNFs and runs the shortest path algorithm to find the physical path mapping for each VL. For each VNF, FF selects the first substrate node with enough resources.

Figures 6, 7 and 8 show, respectively, the number of

accepted requests as well as the number of deployed VLs and VNFs. With respect to the three metrics, our approach outperforms the random agent and the First Fit procedures. Unlike the latter algorithm, our approach explores efficiently the possible actions until it finds the optimal ones, while First Fit tries to place the VNFs on the first substrate nodes that have enough resources. Therefore it explores just a part of the possible solutions, resulting in a huge consumption of the residual bandwidth of the physical links, and then in an increased probability of rejection of new arriving requests.

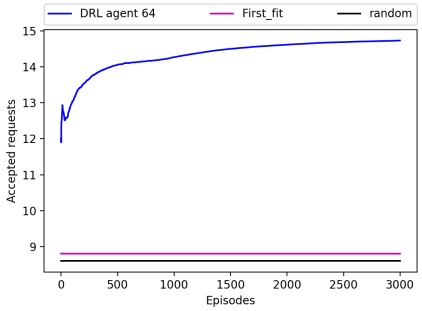

Fig. 6. Number of accepted requests.

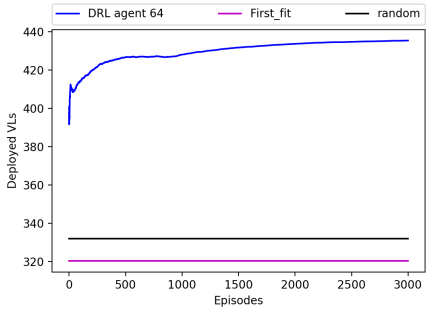

Fig. 7. Number of deploed VLS.

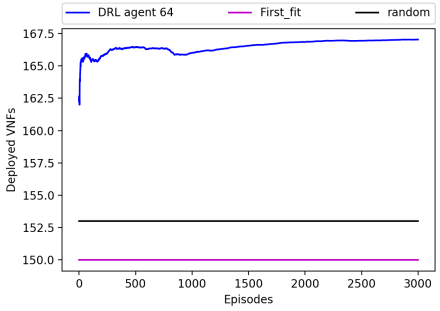

Fig. 8. Number of deployed VNFs.

VI. Conclusions

In this paper, we have proposed a methodology to address the VNE issue. The proposed approach is based on the use of a Deep Reinforcement Learning technique. In particular, we have proposed the use of GCNs, which allow us to take into account the structuring of the data of the problem, and showed how to adapt the tool to the considered VNE problem. The paper also exhibits a new strategy of placement, significantly improving the performance of this type of task. The proposed solution has proven to be very effective compared to existing conventional approaches.

References

[1] "Network Functions Virtualisation (NFV): Architectural framework," *ETSI GS NFV*, vol. 2, no. 2, p. V1, 2013.

[2] B. Yi, X. Wang, K. Li, S. k. Das, and M. Huang, "A comprehensive survey of Network Function Virtualization," *Computer Networks*, vol. 133, pp. 212 – 262, 2018.

[3] A. Gupta, B. Jaumard, M. Tornatore, and B. Mukherjee, "A Scalable Approach for Service Chain Mapping With Multiple SC Instances in a Wide-Area Network," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 3, pp. 529–541, March 2018.

[4] B. Koley, "The zero touch network," 2016.

[5] I. Afolabi, T. Taleb, K. Samdanis, A. Ksentini, and H. Flinck, "Network Slicing and Softwarization: A Survey on Principles, Enabling Technologies, and Solutions," *IEEE Communications Surveys Tutorials*, vol. 20, no. 3, pp. 2429–2453, thirdquarter 2018.

[6] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu, and D. Yang, "Experience-driven Networking: A Deep Reinforcement Learning based Approach," in *IEEE INFOCOM*, April 2018, pp. 1871–1879.

[7] M. Mechtri, C. Ghribi, and D. Zeghlache, "A Scalable Algorithm for the Placement of Service Function Chains," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 533–546, Sep. 2016.

[8] S. Khebbache, M. Hadji, and D. Zeghlache, "A multi-objective non-dominated sorting genetic algorithm for VNF chains placement," in *Proc. IEEE CCNC*, Jan 2018, pp. 1–4.

[9] S. Haeri and L. Trajković, "Virtual network embedding via Monte Carlo tree search," *IEEE Transactions on Cybernetics*, vol. 48, no. 2, pp. 510–521, 2018.

[10] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[11] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2008.

[12] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[13] J. You, R. Ying, X. Ren, W. L. Hamilton, and J. Leskovec, "Graphrnn: Generating realistic graphs with deep auto-regressive models," *arXiv preprint arXiv:1802.08773*, 2018.

[14] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016.

[15] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *Advances in neural information processing systems*, 2000, pp. 1008–1014.

[16] Y. Bai, H. Ding, S. Bian, T. Chen, Y. Sun, and W. Wang, "Simgnn: A neural network approach to fast graph similarity computation," in *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*. ACM, 2019, pp. 384–392.

[17] R. Socher, D. Chen, C. D. Manning, and A. Ng, "Reasoning with neural tensor networks for knowledge base completion," in *Advances in neural information processing systems*, 2013, pp. 926–934.

[18] P. Erdős, "On random graphs," *Publicationes Mathematicae*, vol. 6, pp. 290–297, 1959. [Online]. Available: https://ci.nii.ac.jp/naid/10018689248/en/

[19] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.