Monkey business: reinforcement learning meets neighborhood search for virtual network embedding.

Maxime Elkael, Massinissa Ait Aba, Andrea Araldo, Hind Castel-Taleb, Badii Jouaber

Monkey Business: Reinforcement learning meets neighborhood search for Virtual Network Embedding

Maxime Elkael[a], Massinissa Ait aba[b], Andrea Araldo[a], Hind Castel-Taleb[a], Badii Jouaber[a]

[a]*Telecom SudParis, SAMOVAR, IP-Paris, {Maxime.Elkael, Hind.Castel, Badii.Jouaber, Andrea.Araldo}@telecom.sudparis.eu*
[b]*Davidson consulting, Massinissa.ait-aba@davidson.fr*

Abstract

In this article, we consider the Virtual Network Embedding (VNE) problem for 5G networks slicing. This problem requires to allocate multiple Virtual Networks (VN) on a substrate virtualized physical network while maximizing among others, resource utilization, maximum number of placed VNs and network operator's benefit. We solve the online version of the problem where slices arrive over time. Inspired by the Nested Rollout Policy Adaptation (NRPA) algorithm, a variant of the well known Monte Carlo Tree Search (MCTS) that learns how to perform good simulations over time, we propose a new algorithm that we call Neighborhood Enhanced Policy Adaptation (NEPA). The key feature of our algorithm is to observe NRPA cannot exploit knowledge acquired in one branch of the state tree for another one which starts differently. NEPA learns by combining NRPA with Neighbordhood Search in a frugal manner which improves only promising solutions while keeping the running time low. We call this technique a monkey business because it comes down to jumping from one interesting branch to the other, similar to how monkeys jump from tree to tree instead of going down everytime. NEPA achieves better results in terms of acceptance ratio and revenue-to-cost ratio compared to other state-of-the-art algorithms, both on real and synthetic topologies.

1. Introduction

The fifth-generation (5G) communications system is envisioned to serve a variety of novel services and industries, such as, autonomous vehicles, Virtual Reality (VR), Augmented Reality (AR) and remote healthcare, each requiring different Quality-of-Service (QoS). In this context, network slicing is a new way to manage telecommunication networks in a similar manner to what is done with cloud computing, relying on virtualization. The idea is that an operator owns a physical network (analogous to a data center for cloud computing) that can host multiple virtual networks (or slices). The operator can instantiate slices on the fly to a third party service provider. Each slice provides resources in an isolated, adaptable and dynamic manner. Service providers can have demands for slices with specific QoS/security constraints, topologies and resource requirements. Slices would be implemented using Network Function Virtualization (NFV) and software defined networking (SDN). The former allows to instantiate Network Functions (NFs - entities managing networking blocks such as authentication, gateways, etc). The latter enables packet routing to be modified by a centralized controller, enabling efficient and adaptable configuration of links ("virtual links") of the slice. The interest of such an approach mainly lies in its flexibility: one could envision supporting various use cases such as autonomous vehicles (requiring ultra low latency and ultra high reliability), virtual reality (requiring high throughput) or sensor networks (requiring an enormous amount of connections on a small area)[1]. In the current "one size fits all" paradigm, supporting such use cases would imply building a new physical network for each one, which would hardly be economically viable.

In this context, an important question is how to place slices on such a network: clients give the operator slice (virtual network) requests in the form of interconnected NFs (e.g. a graph), and the operator tries to embed them onto the physical infrastructure (*e.g.* to provide enough CPU for each virtual node and enough bandwidth for each virtual link between those nodes), by accepting as many slices as possible, so as to maximize the operator's gain. This problem is known as Virtual Network Embedding[2] (VNE) problem which has been extensively studied in the recent years[3][4][5]. The VNE being NP-complete and inapproximable [6], running an exact algorithm is not an option in most cases. Various methods have been studied for this problem, among which many are heuristic algorithms based on Linear programming[3][7], ranking algorithms[4] or reinforcement learning (RL) methods [8][9]. We are particularly interested in the latter ones, as these approaches enable to construct heuristics in an autonomous manner, based on experience and learning while solving the on-line version of the problem, where slices arrive and leave the system over time. Research on RL methods is still lacking, as current neural networks based methods either have hard constraints on the network topologies [10] or require very large amount of computing resources for training [9]. On the other hand, Monte Carlo based methods such as [8] still have large rooms for improvement, as we show in this work. These shortcomings of other RL approaches are further developped in section 2.

In this paper, our main contribution is to provide a new state-of-the-art algorithm called Neighborhood Enhanced Policy Adaptation for this problem which combines reinforcement learning techniques with neighborhood search. This article builds on our

previous work from [11], since it improves the proposed Nested Rollout Policy Adaptation (NRPA)[12] algorithm with a neighborhood search technique. To our knowledge this is the first time the Monte-Carlo Tree Search (MCTS) based NRPA algorithm is complemented with such neighborhood search technique for any problem, enabling it to beat several state-of-the-art algorithms. The MCTS approach (called Maven-S) from [8] uses UCT (Upper Confidence bound for Trees)[13], which is adapted to stochastic problems. On the other hand, NRPA is specifically adapted to deterministic optimization problems. In our case, we do not know the slices in advance making the arrival process stochastic; however, once a slice arrives it is fully observable, and so is the physical network, making NRPA more adapted to tackle the placement of slices, given the current state of the network. NRPA learns through exploring the NF placement possibilities of the slice at random while learning weights for biasing future explorations, which enables it to focus on regions of the search space that have been the most rewarding so far while still maintaining a good level of exploration. These characteristics make it a very efficient algorithm for solving the VNE. However we show it can be further enhanced when combining it with neighborhood search. They key idea is that NRPA bases its search on the tree structure of the search space, which is good for quickly finding good solutions. However it can limit exploration of new, better branch once the algorithm has converged. Neighborhood search enables us to exploit knowledge of those good solutions for jumping to better branches of the search tree (similar to how monkeys jump from branch to branch) and continue the search from there, which enables better future exploration. We also propose a heuristic for initialization of the weights, and we show our numerical results, showing an improvement in the slice acceptance probability on real and synthetic networks compared to other methods, and therefore an increase in financial gains for an operator. Our contributions in this paper are then the following :

- We combine NRPA with neighborhood search and our heuristic weight initialization, deriving the Neighborhood Enhanced Policy Adaptation (NEPA) algorithm for the virtual network embedding problem which outperforms state-of-the-art methods in both acceptance and revenue-to-cost ratio on all tested instances, including both synthetic and real topologies. Our approach is particularly effective on real topologies, on which it can even triple the number of accepted slices compared to some of the previous algorithms. We also investigate the topological features of those real topologies and explain how our algorithm can exploit them. Note that NRPA had never been used for the VNE problem.
- We publish a large set of testing scenarios for the community to experiment with, patching a lack of publicly available instances for quicker experimentation and comparisons (126 instances).
- We publish our implementations of several algorithms publicly (including NRPA, NEPA and algorithms from [14][8]), since during this work, we found most algorithms lacked a well-documented implementation.
- To our knowledge, we are the first to explore the combination of NRPA with neighborhood search for any problem. We believe the idea can be exploited in other application where NRPA has been successful and where good neighborhood search algorithms are known such as the Travelling Salesman Problem (TSP)[15] or the Vehicle Routing Problem (VRP)[16]
- Finally, we assess wether the results of NEPA for the VNE can be improved by utilizing the reward function described in [17] (see Appendix B).

The paper is organized as follows : Section I introduced our work, section II presents our litterature review of the VNE, then section III presents our model. We describe NEPA in section IV, section V presents our numerical experiments, and in section VI we summarize our work and we propose extensions and future perspectives

2. Litterature review

Several methods have already been proposed for the VNE problem.

2.1. Mathematical programming

Notoriously, some work has been done for exact VNE using Mathematical programming. In [18], the authors propose an ILP formulation. This has the advantage to give guaranteed optimal solutions. However, since the VNE is NP-hard[19], such an approach would not be able to cope with even medium slices with a reasonable execution time. Hence, a lot of work in the literature focus on heuristic algorithms. In [3], two heuristics based on linear programming and rounding (either randomized or deterministic) are derived. These give good results in terms of acceptance and revenue-to-cost ratio, although most other approaches manage to beat them ([8][14][9]). These two rounding heuristics also sometimes suffer from relatively high runtimes, as [9] shows they run up to 13 times slower than the approach from [8] for worst results, and that for some cases the approaches are even unable to run due to a lack of computational resources. In [7], an ILP heuristic is derived by reducing the number of candidate paths to a small amount, which enables the solver to find a solution quicker. However since it is ILP-based the algorithm is still non-polynomial. Our approach addresses these issues by proposing a solution which both runs quickly (sub-second runtime) and provides high quality (state-of-the art) embeddings.

2.2. Graph neural networks

Some recent papers [20][21] process the problem with a deep neural network for performing the embedding (note that in this section we do not consider approaches using neural networks in conjunction with RL). In [20], the graph is clustered

with a graph neural network, which then helps guide the embedding procedure. On the other hand [21] pre-processes the network in order to reduce the state-space, making the problem more manageable for other algorithms. Overall, [21] addresses a slightly different problem than we do, since the paper is concerned with feeding a VNE algorithm (such as ours) with hints for solving the VNE, and both could be used in conjunction. On the other hand, [20] is concerned with the VNE, and although it has good results, the runtime is a significant problem as it is exponential in the number of nodes. The authors patch this issue with the use of a GPU. However, our experiments show that although the runtime is manageable it is still higher than all other algorithms we tested (in the order of ...).

2.3. Heuristics and meta-heuristic techniques

There is also a wealth of meta-heuristic algorithms for the VNE. This includes genetic algorithms [22] and ant colony optimization [23]. However the most popular class of meta-heuristic approach for the VNE is particle swarm optimization (PSO), with several well performing algorithms such as [14], [24] or [25]. These PSO approaches work by initializing "particles" as a swarm of random solutions which move in the space of candidate solutions. They find new solutions by sarcastically combining the best solutions found so far with current solutions.

Regarding heuristics, in [26], authors propose metric for evaluating a nodes' resource capacity/demand and then match highly demanding virtual nodes to highly available physical nodes. A similar idea is used in [4] where it is combined with the Pagerank algorithm for ranking nodes.

These heuristic and meta-heuristic approaches show relatively good performances that we aim at beating in this article. Especially, to our knowledge, none of them exploits the fact that solutions can be improved by keeping virtual nodes close to one another. In that regard, our work could inspire enhanced versions of the cited algorithms.

2.4. Reinforcement learning approaches

The family of approaches that interests us the most is reinforcement learning. First of all [8] showed how to use the Monte Carlo Tree Search algorithm (MCTS) [13] for the VNE problem. MCTS intelligently explores the space of possible placement solutions in order to find the best, but its exploration is based on multi-armed bandit theory, which assumes stochastic rewards. Instead, the outcome of a given embedding is deterministic and our method more efficaciously exploits determinism. Both can be considered *online* methods, since they can immediately take decisions on any slice arrivals.

By contrast, *offline methods* accumulate knowledge during an extensive learning (training) stage, which is then reused for a near-instantaneous high-quality embedding. Recently, DeepVine [10] used a deep neural network in order to learn embedding. This approach learns from graphs that are turned into images, enabling easy use of convolutional neural network (CNN) architectures. Although successful, this method makes strong assumptions about the input graphs: CNNs rely on the networks to be grid-shaped. Another method is [9] where the neural neural network is fed directly with graphs. In this article, they use the A3C (Asynchronous Advantage Actor-Critic) algorithm for learning, which has been successful for other RL tasks. These approaches rely on function approximators (namely, neural networks) coupled with model-free RL techniques. This use of neural networks enables them to deal with big state-spaces, but comes at the cost of having no convergence guarantees to an optimal embedding or even an approximation. On the other hand, online methods like ours can be tweaked to guarantee that given enough time, they could converge to the optimal solution. They are also able to handle similar state-spaces compared to neural-network based methods.

The huge computation needed to perform a very costly a-priori training (*e.g.*, training for 72h on 24 for parallel instances of the problem [9]) may make these offline methods [9][10] infeasible in practical situations. In particular when applying embedding on different scenarios (or with different conditions or constraints), the huge offline learning phase must start from scratch. It is also an open question whether or not in a real world scenario we will have enough samples in order to enable such algorithms to learn. The advantage of online methods is instead their ability to immediately adapt and take decisions on new instances of the problem.

For these reasons we improve upon the state-of-the-art online methods [8] [11], providing convergence at regime toward the optimal embedding, sample efficiency and better empirical performance.

Notation	Description
VNE	Virtual Network Embedding
VNR	Virtual Network Request/Slice
$G(V, E)$	Physical network with nodes V and links E
$\bar{H}^x(V^x, E^x, t_a^x, t_d^x)$	x^{th} slice with nodes V^x, links E^x, arrival and departure dates t_a^x and t_d^x
CPU_{v_i}	CPU capacity of physical node v_i
BW_{v_i,v_j}	BW capacity of physical link (v_i, v_j)
$CPU^o_{v_i}$	Occupied CPU of physical node v_i
$BW^o_{v_i,v_j}$	Occupied BW of physical link (v_i, v_j)
$CPU^d_{v_j^x}$	CPU demanded by virtual node v_j^x
$BW^d_{v_i^x,v_j^x}$	BW demanded by virtual link (v_i^x, v_j^x)
$\bar{BW}^x_{v_i,v_j}$	Bandwidth used by slice x on physical link (v_i, v_j)
$\bar{CPU}^x_{v_j}$	CPU used by slice x on physical node v_j
MDP	Markov Decision Process
$\mathcal{A}$	Set of possible actions in MDP
$s(k)$	State of MDP at step k
a_k	Action chosen in MDP at step k
NRPA	Nested rollout policy adaptation
MCTS	Monte Carlo Tree Search
P	Policy function (associates a State-action couple with its weight
$\mathcal{P}$	Link-mapping function (associates virtual links with physical paths)

Table 1: Notation and Acronyms

3. VNE with a MDP approach

The physical network belongs to an operator. At any point in time, the operator has a full knowledge of the state of the network, e.g. the amount of resources available, the slices it hosts and the resources they use. The operator receives slice requests from its clients over time. These requests are descriptions of a virtual network they would like to embed on the network, including resources required and topology. The goal of the oper-

ator is to place the incoming slices on its network in order to maximize a given objective (slice acceptance rate in our case).

3.1. Graph theoretic notation

The VNE problem can be formally described as a graph embedding problem:

- the physical network is represented as an undirected graph $G(V, E)$, where V is the set of n physical nodes, $v_1, \ldots, v_i, \ldots, v_n$, that represent several physical machines where virtual network functions can be hosted, and E is the set of the physical edges between the nodes. So we have:
  - Each physical node v_i is characterized by a CPU capacity, CPU_{v_i} and an occupied CPU quantity, $CPU^o_{v_i}$. One could extend this model with other resource types (RAM, HDD, ...) without loss of generality.
  - On the other hand, each physical edge $(v_i, v_j) \in E$ is weighted by a maximum bandwidth amount, BW_{v_i,v_j} and an occupied bandwidth amount $BW^o_{v_i,v_j}$. In case $BW_{v_i,v_j} = 0$, then we consider that there is no edge between v_i and v_j.
- We denote by $H^x(V^x, E^x, t^x_a, t^x_d)$ the undirected graph describing the x^{th} slice with the resources needed: each virtual node of the slice, $v^x_i \in V^x$ carries a CPU demand, $CPU^d_{v^x_i}$ and each virtual link $(v^x_i, v^x_j) \in E^x$ carries a bandwidth demand, $BW^d_{v^x_i,v^x_j}$. Since we are in a dynamical system, each slice also has a time of arrival t^x_a and a time of departure t^x_d. Observe that as slices are placed or leaving, the physical occupied resources, $CPU^o_{v_i}$ and $BW^o_{v_i,v_j}$ change over time. The problem is to map each virtual node on a physical node and each virtual link on a physical path between the two host of its extremities, taking into account the available resources.

3.2. Problem constraints

If at a certain instant time instant the x^{th} slice request arrives, placement decisions must satisfy the following constraints:

- Each placed virtual node should have enough CPU, e.g. if we choose v_i hosts virtual node v^x_j we should have $CPU^d_{v^x_j} \leq CPU_{v_i} - CPU^o_{v_i}$ (where $CPU_{v_i} - CPU^o_{v_i}$ represents available CPU on node v_i)
- For virtual link (v^x_m, v^x_p) all physical links (v_i, v_j) it uses should be chosen so $BW^d_{v^x_m,v^x_p} \leq BW_{v_i,v_j} - BW^o_{v_i,v_j}$ (where $BW_{v_i,v_j} - BW^o_{v_i,v_j}$ represents the available bandwidth between nodes v_i and v_j) such that these links form a path between the physical nodes hosting v^x_m and v^x_p).
- If two virtual nodes belong to the same slice, they can't be placed on the same physical node. This constraint is present in most previous works on the VNE[8, 3, 4]. It ensures reliability by preventing a significant portion of a slice from going off if a single physical node is down.

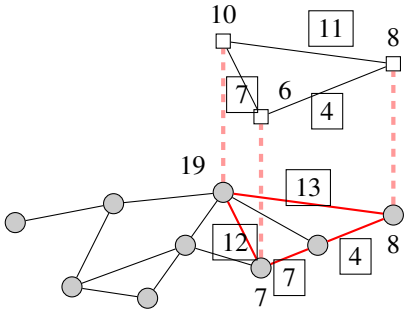


Figure 1: Slice (white nodes) embedded on physical network (gray nodes). Link demands and remaining capacities are boxed, used physical links are in red. CPU demands and capacities are non-boxed.

To our knowledge, the optimal trade-off between sharing physical nodes (thus economizing bandwidth) and redundancy has not been well studied. Our approach, as most of the others cited, could easily be adapted to a relaxation of this constraint.

3.3. Online VNE description

An example of slice is shown in Figure 1. We solve the VNE online :

- when a slice x arrives at time t^x_a, we directly try to embed it. If a feasible solution is found, the slice is placed on the physical network, consuming the corresponding CPU and bandwidth resources, *i.e.* updating the corresponding $CPU^o_{v_i}$ and $BW^o_{v_i,v_j}$. If no solution is found, the slice leaves the system and is dropped.
- When time t^x_d is reached, the slice leaves the physical network and resources are freed.

The full system time is continuous and gives us the arrival and departure dates for slices (t^x_a and t^x_d refer to this scale) we assume the VNE is instantaneous : in the same instant the slice request arrives, the corresponding MDP (described in the next subsection) is solved, instantaneously, and the slice is either placed or discarded. For each virtual node to be placed, we select a physical node via RL (Reinforcement learning). In order to learn an optimized sequence of decisions for virtual resource placement via RL, one needs to frame the VNE problem as a Markov Decision Process (MDP)[27].

3.4. MDP description

A MDP is a system made up of two elements: the agent (the network operator in our case) and the environment (the description of the slice to place and of the state of the physical network in our case *i.e.* the amount of resources available and occupied).

Observe that our MDP works as a sequence of steps, each step corresponding to the decision of placing a virtual node onto a physical node. Note that these steps do not have any time-dimension, they can be considered to be all taken instantaneously. Also observe that our MDP is fully deterministic:

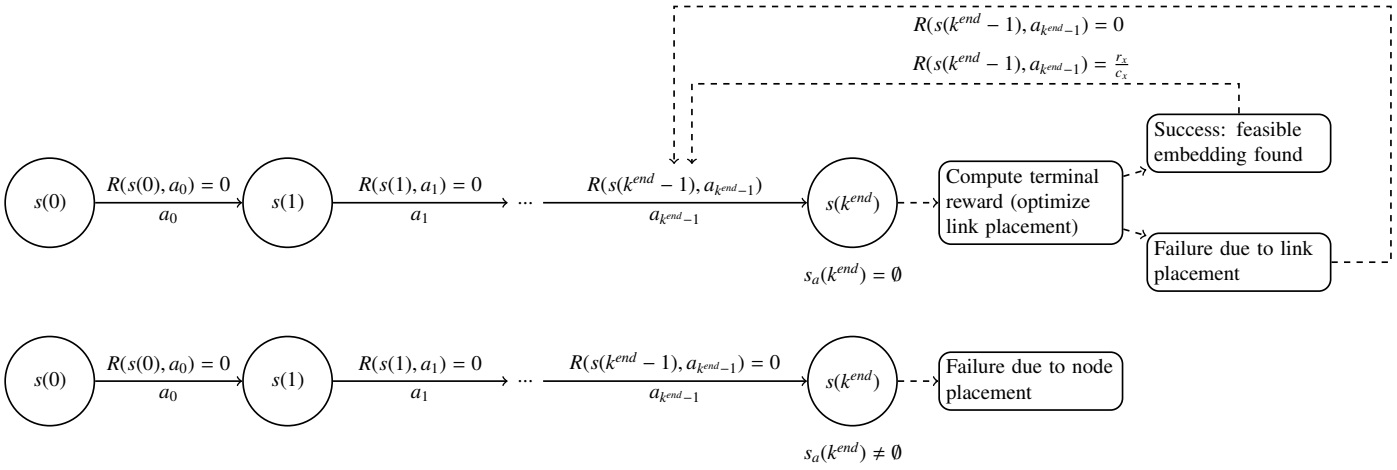


Figure 2: Example sequences of actions in the MDP. Dashedy arrows are transitions not occurring in the MDP (no action choice).

all transitions and all rewards (which we will define later) are deterministic and computable in advance.

In our particular setting, we consider the optimization problem where we have to place a single slice at a time. This means that as soon as one slice requests arrives, an MDP is initialized in order to decide the embedding of each virtual node and link it demands.

We assume that the agent only decides where to place each virtual node. After all virtual nodes of a certain slice have been placed, we calculate Link placement with a shortest path heuristic (see algorithm 5). Therefore, we adopt MDP only for virtual node placement.

3.4.1. Elements of the MDP

Let $s(k) = (s_a(k), s_b(k))$ be a state of the MDP, it is composed of two components :

- $s_a(k)$ is the set of virtual nodes yet to be embedded at step k.
- $s_b(k)$ represents, at step k, the occupation of the physical nodes by the virtual nodes. It is a vector with $|V|$ elements where $s_b(k)[i] = j$ if virtual node v_j^x from slice x is hosted on physical node v_i. If v_i hosts no node from the current slice, $s_b(k)[i] = 0$ (we assume indexes of virtual nodes are strictly positive integers).

For the incoming slice x, we consider the virtual nodes $v_j^x \in V^x$ one by one[1] and we take an action i which corresponds to placing it on a physical node v_i. Therefore, the set of possible actions $\mathcal{A} = \{1, \ldots, n\}$ corresponds to the physical nodes of V. Choosing action i would mean placing the current virtual node on v_i. We also consider $\mathcal{A}(s(k)) \subseteq \mathcal{A}$ the set of legal actions from state $s(k)$, which will be specified later.

[1]The order in which we iterate through virtual nodes can be chosen arbitrarily

3.5. System's evolution

The main steps of the system evolution are described as follows:

a. At step 0, $s(0) = (V^x, u)$, where u is a vector of $|V|$ components all equal to 0.
b. At step $k \geq 0$, from the state $s(k)$, let v_l^x be the first virtual node of $s_a(k)$. Then $\mathcal{A}(s(k))$ is the set of actions $j \in \mathcal{A}$ such that $CPU_{v_l^x}^d \leq CPU_{v_j} - CPU_{v_j}^o$ and $s_b(k)[j] = 0$. Assume the chosen action from $\mathcal{A}(s(k))$ is $a_k = i$. Then the virtual node v_l^x is embedded on physical node v_i and we have a transition to the state $s(k+1) = (s_a(k) - \{v_l^x\}, s_b(k) + b_i)$ where b_i is a vector with the i^{th} component equal to index 1 of virtual node v_l^x and all other components equal to 0.

The embedding process continues at each step until we reach the final state at a certain step k^{end}, where $\mathcal{A}(s(k^{end})) = \emptyset$. At this point, two situations can occur:

- Either the node embedding is a success, so the set of virtual nodes is $s_a(k^{\text{end}}) = \emptyset$. The second part of the state holds a vector $s_b(k^{\text{end}})$ indicating which physical nodes are used by each virtual node of the slice. So the final state is $(\emptyset, u')$, where $u'[i] = l$ if virtual node v_l^x is hosted by physical node v_i .
- Or the embedding fails, which means that for a virtual node, there is no suitable physical node to host it e.g. $s_a(k^{end}) \neq \emptyset$. In this case, the entire slice is rejected.

If node embedding is successful, the Link embedding is calculated using algorithm 5 which is a shortest path heuristic. Then, if link embedding is successful too, we need to update the physical network to acknowledge for the used resources, *e.g.* update $CPU_{v_i}^o$ and BW_{v_i,v_j}^o for all physical nodes v_i and physical links v_i, v_j used by the slice. On the other hand, if one of the two phases fails, the slice is discarded.

3.5.1. Reward Function

We now define the reward obtained by the agent over the course of its actions. Let us first define the revenue of the operator r^x (representing the revenue gained thanks to a client paying for slice x) and the cost c^x (the cost induced by operating the physical resources allocated to host the slice) for a successfully placed slice x as:

$$r^x = \sum_{\forall v_i^x, v_j^x \in V^x} BW^d_{v_i^x, v_j^x} + \sum_{\forall v_m^x \in V^x} CPU^d_{v_m^x} \tag{1}$$

$$c^x = \sum_{\forall (v_i, v_j) \in E} B\bar{W}^x_{v_i, v_j} + \sum_{\forall v_i \in V} C\bar{P}U^x_{v_i}, \tag{2}$$

where for slice x, $B\bar{W}^x_{v_i,v_j}$ is the bandwidth used on physical link (v_i, v_j) and $C\bar{P}U^x_{v_i}$ the CPU used on physical node v_i. In other words, service providers pay proportionally to the resource demands by their slices. The cost of operation of a slice is proportional to the physical resources consumed. We define the immediate reward function of our MDP as:

$$R(s(k), a_k) = \begin{cases} \frac{r^x}{c^x} & \text{if } s_a(k+1) = \emptyset \text{ and node and link} \\ & \text{mapping are successful} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Examples of sequences of actions in the MDP are shown in figure 2, in which the circular states correspond to the states of the MDP, where the decisions are taken by the agent. The sequence at the top diagram corresponds to a successful embedding (after node and link placement), while the bottom one returns a failure. Note that since rewards happens during transitions, the last reward is $R(s(k^{end}-1), a_{k^{end}-1})$ as it happens during the last transition, from $s(k^{end}-1)$ to $s(k^{end})$.

3.5.2. Objective function

From the initial state $s(0)$, we consider a sequence of k^{end} actions : $seq = a_0, a_1, \ldots, a_{k^{end}-1}$. We define the total reward from the state $s(0)$ for seq as follows :

$$R^{seq}(s(0)) = \sum_{k=0}^{k^{end}-1} R(s(k), a_k) \tag{4}$$

Then the objective function is:

$$\max_{seq} R^{seq}(s(0)) \tag{5}$$

And the agent seeks to find the best sequence of actions:

$$seq^* = \arg\max_{seq} R^{seq}(s(0)) \tag{6}$$

and the corresponding reward:

$$R^*(s(0)) = R^{seq^*}(s(0)) \tag{7}$$

Notice that in practice, all rewards except the last one are equal to 0 due to equation (3). With this definition of reward, the agent, i.e., the network operator always tries to choose valid embeddings, since any valid embedding has a non-zero revenue-to-cost ratio. It also favors embeddings that use the least possible amount of resources, since the reward increases as $\sum_{\forall (v_i,v_j)\in E} B\bar{W}^x_{v_i,v_j}$ decreases. An intuitive way to frame this is that the reward encourages the choice of embeddings that lead to placing virtual links on short physical paths, effectively trying to place the slice on a cluster of physical nodes. We do this based on the idea that if a slice uses the least possible amount of resources, then it will leave more resources available for future slices, thus enabling us to improve the acceptance ratio on the full scenario. Note that, at best, each virtual link is mapped on a physical link of length 1. Note also that for a successfully embedded slice, $\sum_{v_i \in V} C\bar{P}U^x_{v_i} = \sum_{v_m^x \in V^x} CPU^d_{v_m^x}$, hence the best achievable reward is 1 and, the closer the reward is to 0, the worst the embedding is in terms of resource usage (with 0 being the worst reward, reserved for failed embeddings). Therefore, this reward function quantifies the quality of an embedding regardless of the size of the slice. This has clear advantages over the reward function used in [8] which is $r'_x - c_x$ with $r'_x = \alpha \sum_{\forall v_i^x, v_j^x \in V^x} BW^d_{v_i^x, v_j^x} + \beta \sum_{\forall v_m^x \in V^x} CPU^d_{v_m^x}$, where α, β are weight parameters which have to be tweaked. In [8] they use parameters of 1 which provides an upper bound of 0 and no lower bound, making it harder to compare the quality of embeddings for different slice sizes. In the general case they do not provide any bound. This is particularly unfortunate for the MCTS algorithm they use, as it is based on the upper confidence bounds algorithm UCB-1, which provides its theoretical guarantees only for a reward bounded between 0 and 1.

3.6. Characteristics of the MDP and implications on resolution

Since the MDP transition model for a given slice is completely known in advance and deterministic, one could be tempted to use a method such as dynamic programming to solve the problem. However, it would be unrealistic due to the number of states: there are $\frac{|V|!}{(|V|-|V^x|)!}$ final states (which corresponds to the number of possible repetition-free permutations of $|V|$ physical nodes of size $|V^x|$), each requiring to calculate link placement. For a slice of size 12 placed on a 50 nodes network , we have 5×10^{19} possible terminal states.

Also note virtual nodes are taken in an arbitrary order, hence a given final placement is reachable only using a single sequence of actions. This implies the MDP has a tree topology (see Figure 3 which illustrates the full tree of states for a toy example placement). We argue our algorithm should take this structure into account for exploration and exploitation. Particularly, we will see that existing MCTS methods (MaVEN-S from [8] and NRPA) are interesting since they take advantage of the MDP's tree structure for finding good solutions. However this can lead to local optima once the algorithm has converged. The main motivation of our work is to escape these optima by "jumping" to unexplored branches of the tree that we can guarantee are better than the best solutions found. We will show this can be done by getting around the tree topology and sometimes exploring the solution space in a different manner.

Next, we present our online learning algorithm (NEPA) which

improves slice acceptance ratio, with reduced computation time by implementing this idea. As our result section will show, we only need to explore a few hundred complete sequences of actions for our algorithm.

4. Proposed algorithm

The algorithm we propose in this paper is called NEPA. It is based on NRPA, adding weight initialization and neighborhood-search based refinement. For the sake of clarity, instead of directly presenting NEPA, we first present NRPA and weight initialization.

4.1. Review of Nested Rollout Policy Adaptation (NRPA)

The NRPA[12] algorithm is a Monte Carlo Search algorithm that aims at finding near-optimal solutions in deterministic environments. It is perfectly suited for our problem as in our model a given action from a certain state always leads deterministically to the same state. This setup is similar to the puzzle games which NRPA solves remarkably well (with a world record for Morpion Solitaire) [12]. We describe NRPA in **Algorithm 2**. The idea of this algorithm is to consider the MDP as a tree that we have to explore ("search tree"). This is coherent with our model because we treat virtual nodes to place in an ordered manner, hence there is only a single way to reach a given final or partial state. NRPA explores the search tree with recursive calls to the search function. This search function is defined as such:

- At level 0 a search call does a random simulation of legal actions in the MDP. It returns the reward obtained during that run of the MDP along with the sequence of actions used and the virtual link placement solution $\mathcal{P}$, calculated using **Algorithm 5**. If we refer to figure 3, we can see a random simulation as a complete descent from the root of the tree to a leaf (*e.g.* final) state. The return values from that descent are the corresponding seq and R^{seq}.
- At level $l \neq 0$ the algorithm makes N NRPA calls of level $l-1$. It then returns the best sequence returned by these "children" calls to its caller function, which is either a level $l+1$ NRPA call or the main function. In the latter case the returned sequence is the best sequence found over every simulations tried so far (called seq^{best}) and the NRPA algorithm terminates.
- Then, the control flow returns to the main function (algorithm 1), which updates the resources.

The NRPA search function is combined with a policy learning procedure (see **Algorithm 3** : Adapt procedure for NRPA).

Policy improvement. The principle is that during each simulation, we choose the sequence of actions seq in a biased manner that leads to final states close to the best final state found so far, $R^{seq^{best}}$, which has been reached through sequence of actions seq^{best}. This random sampling enables us to focus on sequences of actions that resemble seq^{best}.

We shall now give the details through which we learn and then bias the simulations :

- We define a policy matrix, P which associates each possible tuple $(s(k), a_k)$ with a real weight $P[s(k), a_k]$ from which probabilities are calculated during simulation.
- Given a certain initial state $s(0) = (s_a(0), s_b(0))$ and a policy matrix P, the algorithm will try a sequence of random actions dictated by the probabilities calculated from P.

After each NRPA call, the weights of actions of the best sequence found seq^{best} are incremented with respect to the state where they should be chosen, *e.g.* $P[s(i), a_i]$ is incremented for all $a_i \in seq^{best}$ (see **Algorithm 3**). Then, during the simulation, when we are in state $s(k)$ and need to select the action $a_k = i$ randomly, we draw using Gibbs sampling, e.g. with probability $\frac{\exp P[s(k),i]}{\exp \sum_{1 \leq j \leq |\mathcal{A}|} P[s(k),j]}$. A visualization of those steps is depicted in figure 4, where the recursive nature of the algorithm is particularly noticeable.

4.2. Virtual links placement

Algorithm 5, is used for placing virtual links after the node placement is decided. It is used during each call to the simulation procedure (**Algorithm 4**). The idea is to treat virtual links one by one by descending bandwidth demands, embedding them on the shortest path (in terms of hops) that has enough bandwidth. Note this is not an exact algorithm and it could replaced with other methods of link embedding. We do not use an exact method because the underlying problem of placing virtual links is an instance of the unsplittable flow problem which is itself NP-Hard [28]. One alternative could be to relax the problem and allow "path-splitting", making the problem solvable by linear programming [29]. It might be of interest and has been used for the VNE (see for example [8]), with the relaxed version consistently improving performance metrics at the cost of a larger computation time (in the order of 40 times for their small cases). However it is unclear whether such an algorithm would be implementable in practice, due to scalability issues as well as the need to reorder packets on arrival, incurring potential additional delay and CPU processing times. For these reasons, the case of path-splitting is outside the scope of this article. To conclude with *NRPA*, we give in **Algorithm.1** the main procedure which describes the calls of the different algorithms related to NRPA for a slice placement.

4.3. Heuristic weight initialization

In standard NRPA, when one encounters an unseen state $s(k)$, all its potential following states $s'(k+1)$, reached from $s(k)$ by choosing action a_k are initialized with a weight $P[s(k), a_k] = 0$. However, this leads to exploring completely at random without exploiting knowledge of the problem. We propose to bias the weight initialization towards more interesting actions, drawing inspiration from [16]. Our heuristic for weight initialization assumes that good embeddings tend to cluster virtual nodes, *i.e.* to place virtual nodes of the same slice in close-by physical nodes, which reduces the mean length of the virtual links.

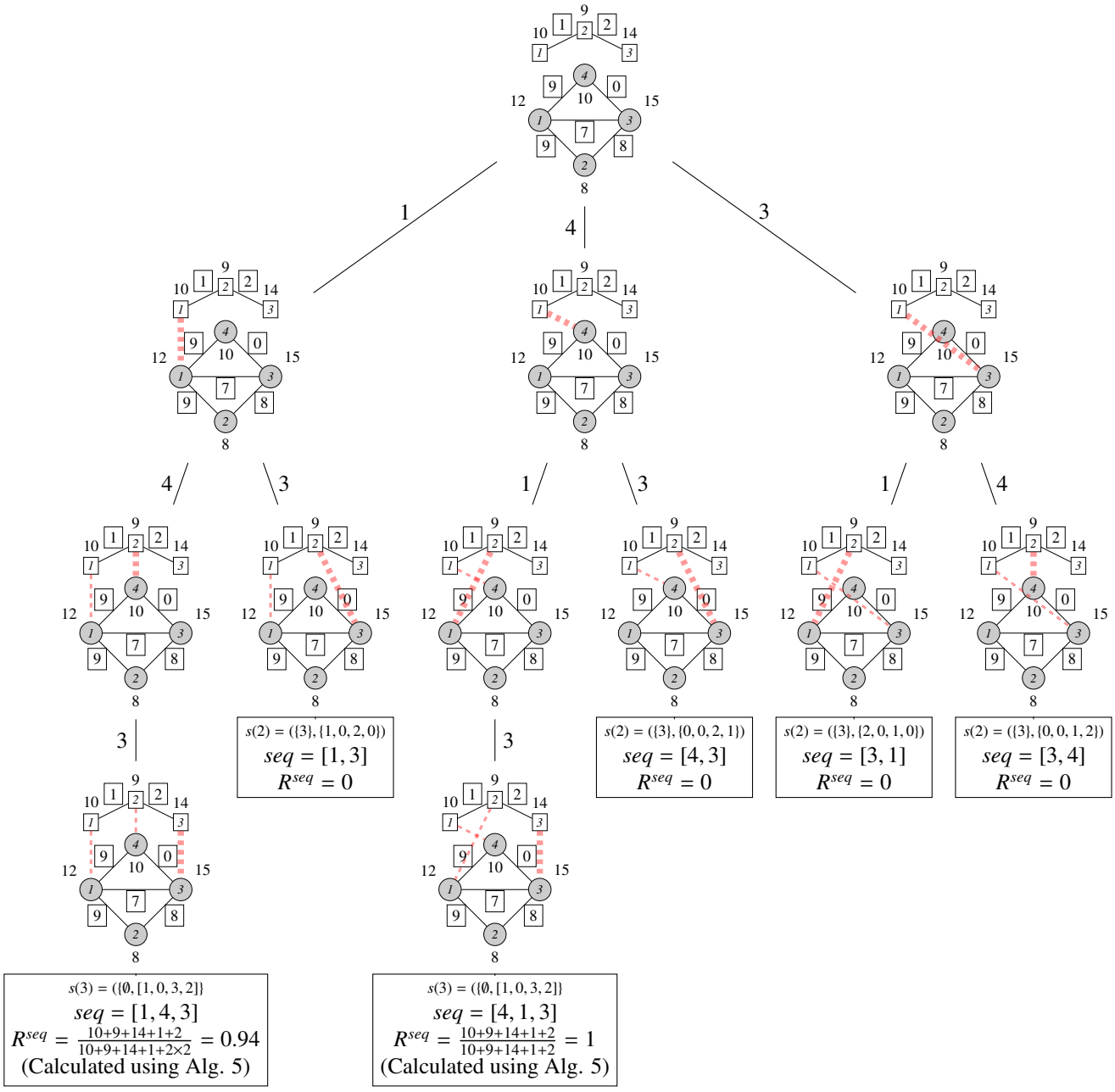


Figure 3: Example MDP for a toy example. Observe transitions are deterministic and MDP has a tree topology.

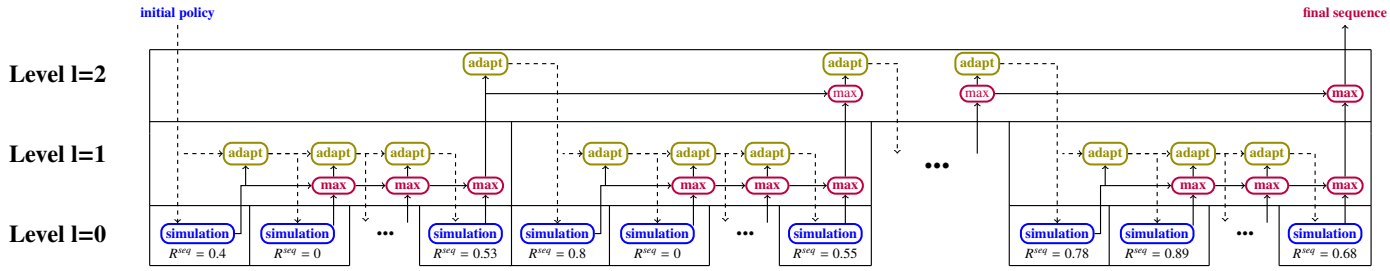
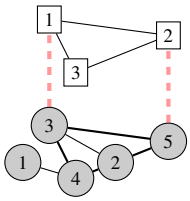


Figure 4: Example execution of NRPA for l=2. Dashed arrows represent the policy going from one function to the other, while plain arrows represent sequences returned between functions. A function needs to get values from all its predecessor before executing.



Next action	Potential next state	Initial weight value
$a_k = 4$	({}, [0, 0, 1, **3**, 2])	$-\frac{d(3,4)+d(5,4)}{2} = -1.5$
$a_k = 2$	({}, [0, **3**, 1, 0, 2])	$-\frac{d(3,2)+d(5,2)}{2} = -1$
$a_k = 1$	({}, [**3**, 0, 1, 0, 2])	$-\frac{d(3,1)+d(5,1)}{2} = -2.5$

Figure 5: Example of state weight heuristic initialization when first choosing action from state ({3}, [0, 0, 1, 0, 2])

When a new state-action couple $(s(k), a_k)$ is encountered, we initialize its weight with:

$$P[s(k), a_k] = \begin{cases} -\sum\limits_{1\le i\le n} \frac{d(i,a_k)\times \mathbb{1}(s_b(k)[i])}{\sum\limits_{1\le j\le n} \mathbb{1}(s_b(k)[j])}, & if\, s_b(k) \neq \vec{0} \\ \frac{1}{n} & otherwise \end{cases} \quad (8)$$

where $\mathbb{1}(s_b(k)[i])$ is equal to 1 if $s_b(k)[i]$ is non-zero (meaning that some virtual node from current slice is associated to physical node v_i) and zero otherwise. The function $d(i, j)$ returns the distance (in terms of hops) between physical node i and j. Note this distance does not take bandwidth into account, making the function computable in advance before starting NRPA, which makes the complexity of weight initialization negligible. In other words we penalise the physical nodes that are far from the ones used up to the current state to embed the current slice. Figure 5 gives an example of such an initialization when the NRPA algorithm first encounters the state ({3}, [0, 0, 1, 0, 2])}. Note that candidate physical nodes that are close to previously placed virtual nodes get a higher weight, since we assume they tend to be more interesting choices. We show an example of such a weight initialization in Figure 5. Notice that the highest weight corresponds to placing virtual node 3 on physical node 2, which leads to using the least resources.

Algorithm 1 MAIN placement procedure

Input: $G(V, E)$: Physical network, $H^x(V^x, E^x)$: slice to place
Output: $G(V, E)$: Physical network, seq^{best} : best node placement, $\mathcal{P}^{best}$: link mapping corresponding to seq^{best}

Choose level parameter l (and level l' for NEPA) and number of iterations per level N
Initialize policy P as all 0s
Derive initial state s(0) from G and H^x
if we call NRPA **then**
    $R^{seq^{best}}, seq^{best}, \mathcal{P}^{best} \leftarrow$ **NRPA**(l, N, P, s(0), E^x, G)
end if
if we call NEPA **then**
    $R^{seq^{best}}, seq^{best}, \mathcal{P}^{best} \leftarrow$ **NEPA**(l, N, P, s(0), E^x, G, l')
end if
if $R^{seq^{best}} \geq 0$ **then**
    Update occupied resources of G with the placed slice
end if
return G, $seq^{best}, \mathcal{P}^{best}$

4.4. Neighborhood Enhanced Policy Adaptation (NEPA)

Observe that once NRPA has found a reasonably good seq^{best}, the weights of the partial state leading to it (*e.g.* the

Algorithm 2 NRPA Algorithm

Input: l: Search level , N: max iterations, P : Policy, $s(0)$: Initial state, E^x: Virtual links, $G(V, E)$: Physical network
Output: $R^{seq^{best}}$: Best score, seq^{best}: best sequence of actions to achieve it, $\mathcal{P}^{best}$: link mapping corresponding to seq

if $l = 0$ **then**
    return SIMULATION($s(0)$, P, E^x, G)
end if
$R^{seq^{best}} \leftarrow -\infty$
$seq^{best} \leftarrow \emptyset$
$\mathcal{P}^{best} \leftarrow \emptyset$
for N iterations **do**
    $R^{seq}, seq, \mathcal{P} \leftarrow$ **NRPA**($l-1, N, P, s(0), E^x, G$)
    if $R^{seq^{best}} \leq R^{seq}$ **then**
        $R^{seq^{best}} \leftarrow R^{seq}$
        $seq^{best} \leftarrow seq$
        $\mathcal{P}^{best} \leftarrow \mathcal{P}$
    end if
    $P \leftarrow$ **ADAPT**(P, seq^{best})
end for
return $R^{seq^{best}}, seq^{best}, \mathcal{P}^{best}$

Algorithm 3 ADAPT procedure for NRPA

Input: P: Policy matrix, seq : sequence of actions
Output: Update of P biased towards drawing actions from seq

$P_{new} \leftarrow P$
for $k = \{0, ..., |seq|\}$ **do**
    $v_l^x \leftarrow$ first node of $s_a(k)$
    $P_{new}[s(k), a_k]$ += 1 // a_k is the k^{th} action of seq
    for $m \in \mathcal{A}(s(k))$ **do**
        P_{new}[s(k), m] -= $\exp(\frac{P[s(k),m]}{\sum_{j\in\mathcal{A}(s(k))}\exp(P[s(k),j])})$
    end for
    s(k+1) $\leftarrow (s_a(k) - \{v_l^x\}, s_b(k) + b_{a_k})$
end for
return P_{new}

Algorithm 4 SIMULATION procedure

Input: $s(0)$: Initial State, P: Policy matrix, E^x: Set of virtual links, G: Physical network
Output: seq: sequence of actions, R^{seq}: the reward it yielded, $\mathcal{P}$: path mapping found by *Alg.* 5 for seq

$seq \leftarrow \emptyset$, $k \leftarrow -1$
while $\mathcal{A}(s(k)) \neq \emptyset$ **do**
    k++
    $v_l^x \leftarrow$ first node of $s_a(k)$
    Deduce $\mathcal{A}(s(k))$ for v_l^x
    $a_k \leftarrow$ random-choice($\mathcal{A}(s(k))$)
    // *draws action from* $\mathcal{A}(s(k))$ *with probability* $\frac{\exp(P[s(k),a_k])}{\sum_{j\in\mathcal{A}(s(k))}\exp(P[s(k),j])}$
    $b \leftarrow \vec{0}$
    $b[a_k] \leftarrow l$
    $s(k+1) \leftarrow (s_a(k) - \{v_l^x\}, s_b(k) + b)$
    $seq \leftarrow seq \bigcup a_k$
end while
$R^{seq}, \mathcal{P} \leftarrow$ **VLINK**(E^x, seq, G)
return R^{seq}, seq, $\mathcal{P}$

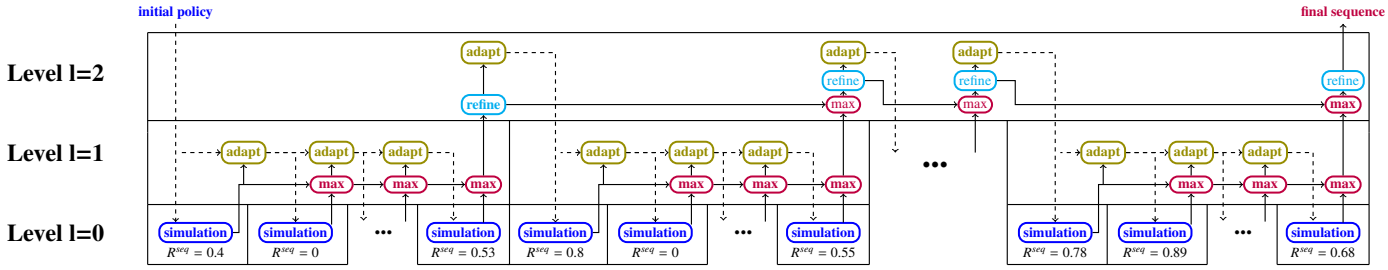


Figure 6: Example execution of NEPA for l=2, l'=2. Dashed arrows represent the policy going from one function to the other, while plain arrows represent sequences returned between functions. A function needs to get values from all its predecessor before executing.

Algorithm 5 VLINK (virtual link placement) procedure

Input: Set of virtual links E^x, sequence of actions seq, Physical network G

Output: R^{seq} : Reward yielded by seq, $\mathcal{P} = \{\mathcal{P}_{v_i^x,v_j^x}, \forall (v_i^x, v_j^x) \in E^x\}$: set of physical path used by each virtual link

while $E^x \neq \emptyset$ **do**
  Pick $(v_i^x, v_j^x) \in E^x$, the most demanding link.
  Find the shortest path $\mathcal{P}_{v_i^x,v_j^x}$ between the physical nodes hosting v_i^x and v_j^x, minding only physical links with available bandwidth (at least equal to $BW^d_{v_i^x,v_j^x}$)
  if $\mathcal{P} \neq \emptyset$ **then**
    Update the physical links occupied by $\mathcal{P}_{v_i^x,v_j^x}$
    $\mathcal{P} \leftarrow \mathcal{P} \bigcup \mathcal{P}_{v_i^x,v_j^x}$
  else
    return $0, \emptyset$
  end if
end while
compute R^{seq}
return $R^{seq}, \mathcal{P}$

weights on the path from the root of the tree to the best final state found) will start increasing. This means if a random simulation deviates from seq^{best} early on, it will then draw actions from a state which is considered almost unexplored, and where the knowledge of seq^{best} is not used. For example, in Fig. 3, if the best sequence found so far is [1, 4, 3] and at first step the chosen action is 4, it would be desirable to exploit the knowledge that in the best sequence found, virtual node 3 goes on physical node 3, as it is true in [1, 4, 3]. In our toy example, this would imply that the algorithm would have a higher chance to find the optimal sequence, [4, 1, 3]. However, with NRPA this is not how things go: if we descend to an unexplored part of the tree, there is no way to reuse the information gained from the known best sequence. This is the reason we introduce NEPA, which we call a *monkey business* algorithm: our goal is to improve NRPA by enabling it to use its knowledge of seq^{best} for finding better branches, similar to how monkeys jump from one branch to another. When monkeys explore the jungle, they swing from branch to branch, they go faster than if they went back down each time they want to move. Similarly, NEPA swings from branch to branch to explore the MDP while NRPA has to go down every time it wants to explore a new zone of the search space.

One possible solution for this could be to increase weights of all states resembling those found by executing seq^{best}. However, this would incur significant cost due to the factorial number of total states, potentially requiring to design approximate methods or workarounds that are outside of the scope of this paper.

Instead, we observe NRPA often finds good embeddings that would be very easy to improve upon by changing the placement of only a small subset of nodes. We also observe that improving such embeddings could be interesting for discovering better sequences of actions that resemble seq^{best} but would not necessarily be discoverable through NRPA's weight mechanism (such as in our example above). Our key idea is that we can improve upon seq^{best} through neighborhood search, finding a new, better sequence without taking the tree structure of the MDP into account. Then, once this improved sequence has been found, we reinject it into NRPA to use as its new seq^{best}, which it can further improve.

In this section, we devise our method for discovering such sequences while keeping the computational complexity reasonable. We call the resulting algorithm Neighborhood Enhanced Policy Adaptation (NEPA) as it combines NRPA with neighborhood search for improving good solutions. The idea of NEPA is to choose a level l' of search at which the solutions should be improved. Then, when the NEPA search reaches level l', each solution found (which correspond to the best solution of each level $l'-1$ call) is refined through the neighborhood search procedure described in Algorithm 6.

We define the neighborhood of a final state $s(k^{end})$ (corresponding to an embedding solution of the virtual nodes on the physical network), as the set of final obtained by moving a virtual node to another physical nodes.

4.4.1. Main steps of NEPA

We describe in **Algorithm 7** the main steps of NEPA algorithm. It is similar to the NRPA algorithm except that if we reach a level $l = l'$, then we choose to refine the solution by searching a neighboring solution as described in the following algorithm:

a. **Algorithm 6** first finds the nodes with the largest potential improvement among the already placed virtual nodes (*e.g.* we choose a single node v_B^x to move). We find it by

Algorithm 6 REFINE procedure

Input: K: Max number of physical nodes candidates, X:Number of iterations, G: Physical network, H^x: Slice, seq: node placement sequence, $\mathcal{P}$: link mapping for seq, R^{seq}: reward yielded by seq and $\mathcal{P}$

Output: $R^{seq^{ref}}$: Refined solution reward, seq^{ref}: refined node mapping, $\mathcal{P}^{ref}$: refined mapping

$seq^{ref} \leftarrow seq$
$R^{seq^{ref}} \leftarrow R^{seq}$
$\mathcal{P}^{ref} \leftarrow \mathcal{P}$
for X iterations **do**
  $previous\_R \leftarrow R^{seq^{ref}}$
  Compute most promising virtual node to move in placement given by $seq^{ref}, \mathcal{P}^{ref}$ using eq.(9)
  Build set of K best physical nodes (ranked using eq.(8)) $V_r \subset V$ suitable for hosting the virtual node
  for $v \in V_r$ **do**
    Put virtual node on physical node v, *i.e.*:
    • Compute updated version of seq^{ref} seq
    • Compute new shortest paths $\mathcal{P}$ considering the new position of the virtual node
    • Update resources used on the physical node and links newly used.
    Compute R^{seq} using seq and $\mathcal{P}$
    if $R^{seq} > R^{seq^{ref}}$ **then**
      $R^{seq^{ref}} \leftarrow R^{seq}$
      $seq^{ref} \leftarrow seq$
      $\mathcal{P}^{ref} \leftarrow \mathcal{P}$
    else
      Restore resources used on physical node and link to values matching $seq^{ref}, \mathcal{P}^{ref}$
    end if
  end for
  if previous_R = $R^{seq^{ref}}$ **then**
    break
  end if
end for
return $R^{seq^{ref}}, seq^{ref}, \mathcal{P}^{ref}$

calculating:

$$score(v_m^x) = \frac{\sum_{v_p \in V^x} BW^d_{v_m^x, v_p^x} \cdot d(v_m^x, v_p^x)}{deg(v_m^x)} \quad (9)$$

for each virtual node v_m^x, where $l(v_m^x, v_p^x)$ is a function returning the length of the physical path used by virtual link (v_m^x, v_p^x) and $deg(v_m^x)$ is the degree of virtual node v_m^x. The virtual node which maximizes this metric is considered as the most promising for improvement, since it is the one which consumes the most bandwidth compared to its number of neighbors.

b. The refining procedure is then to try several candidate physical nodes that could be better suited to host the selected virtual node v_B^x, in terms of reducing resource consumption. For each candidate, we remap the virtual node on them (which corresponds to flipping values in the state vector), then remap its adjacent virtual links. After all candidate physical nodes have been tried, the new placement of v_B^x is then the one that leads to a maximum reward (see eq (3)).

To control the execution time of the algorithm, we introduce two parameters:

- X : is the number of times that the process is repeated, note that the process is also stopped if a full trial does not lead to any improvement. Typically a criterion can be to do no more than $|V^x|$ tries. This ensures the runtime is reasonable while spending more time on larger slices, since they tend to be harder to place.
- K is the number of candidate physical nodes. For choosing candidates, the simplest thing would be to try all possible physical nodes. However this would lead to poor scalability. Instead, we use our weight initialisation function and define our K candidates as the K nodes with the highest distance score. For example, in Fig. 7, which shows a refinement iteration with $K = 2$, the two candidates for hosting virtual node 3 are physical nodes 4 and 2 because they are the two nodes that are the closest to physical nodes 3 and 5, which host the other virtual nodes.

After the refinement, the resulting placement is treated like a normal state by NEPA, *e.g.* if it is the best found so far, its weight is incremented. In this sense, NEPA (Alg. 7) maintains the structure of NRPA (**Algorithm 2**). Note that in practice, NRPA could potentially have found any sequence of actions (*e.g.* node placement) found by NEPA. However, the virtual link embedding corresponding could be different since NRPA places only using **Algorithm 5** while in the case of refinements, NEPA uses **Algorithm 6** which can find a different link embedding for the sequence than what **Algorithm 5** could have found. This is particularly important since in practice, we observe that some sequences found by **Algorithm 6** have no valid solution if using only **Algorithm 5**. Hence, when using NEPA, it is necessary to save not only the best sequence of actions, but also the link embedding result in case it needs to be restored after execution for future use. This is typically done at the end of **Algorithm 6** by saving the link embedding solution in the global data structure S_{emb}

NEPA requires only very little modifications to NRPA (see algorithm 7), which is particularly noticeable in figure 6, as it illustrates the way NEPA makes its function calls recursively. Also note how few refine calls there are compared to the number of max operations, since those we chose $l' = 2$.

Our method enables us to discover better solutions that would not be easy to find once standard NRPA has converged: once NRPA has found a local optimum, the probabilities of choosing the states of the best sequence found in NRPA will go towards 1, meaning exploration could become poor while there is no point exploiting the same region anymore. With NEPA, since we change the best sequence found so far, we open the opportunity of exploring completely new, but better parts of the

search space. A single change in the first few actions can lead to discovering a whole new part of the state-space where most states are undiscovered, leading to a highly explorative phase with a very good sequence as a starting point (which we newly found through the refinement procedure). Hence NEPA exploits its neighborhood search mechanism in order to help NRPA escape local optima.

Algorithm 7 NEPA Algorithm

Input: Search level l, Number of iterations N, Policy matrix P, Initial state $s(0)$, Set of virtual links E^x, Physical network G, Refinement level l'
Output: Best score achieved and best sequence of actions to achieve it

if $l = 0$ **then**
  return SIMULATION($s(0)$, P, E^x, G)
end if
$R^{seq^{best}} \leftarrow -\infty$
$seq^{best} \leftarrow \emptyset$
$\mathcal{P}^{best} \leftarrow \emptyset$
for N iterations **do**
  $R^{seq}, seq, \mathcal{P} \leftarrow$ **NEPA**$(l-1, N, P, s(0), E^x, G, l')$
  if $R^{seq^{best}} \leq R^{seq}$ **then**
    $R^{seq^{best}} \leftarrow R^{seq}$
    $seq^{best} \leftarrow seq$
    $\mathcal{P}^{best} \leftarrow \mathcal{P}$
  end if
  if l = l' and $R^{seq^{best}} \neq 0$ **then**
    $R^{seq^{best}}$, seq^{best}, $\mathcal{P}^{best}$ = **REFINE**(K, X, G, H^x, seq^{best}, $\mathcal{P}^{best}$, $R^{seq^{best}}$)
  end if
  $P \leftarrow$ **ADAPT**(P, seq)
end for
return $R^{seq^{best}}$, seq^{best}, $\mathcal{P}^{best}$

4.5. Complexity

In this section, we outline complexity results for both NRPA and NEPA. We first start by calculating their computational complexity, then memory complexity is discussed.

4.5.1. Computational Complexity

Proposition 1. *The NRPA algorithm has a computational complexity of $O(|V| \times N^l)$ for sparse physical and virtual graphs.*

Let $T(N, l)$ be the function associating the algorithms' parameters (number of iterations per level N and search level l) with the number of simulations executed (*e.g.* the number of calls to Alg. 3). We shall prove $T(N, l) = N^l$ by induction on l: For $l = 0$ the relationship is verified. We now assume that our hypothesis is verified, *e.g.* $T(N, l) = N^l$. We will now show that this implies $T(N, l+1) = N^{l+1}$.

$$T(N, l+1) = N \times T(N, l) = N \times N^l = N^{l+1} \tag{10}$$

This proves $T(N, l) = N^l$. By the same argument, one could show that the same NRPA search would perform N^l adaptations of its policy (*e.g.* N^l calls to Alg. 2).

Algorithms 3 and 4 are really similar and treat a sequence of length $|V^x|$. For each element of the sequence, both algorithms loop through the list of legal moves. At worst, at each step, all physical nodes that have not already been chosen are legal. In such a case, the complexity of both nested loops would be $O(|V^x| \times |V|)$. In order to compute the rewards in the simulation procedure, we place virtual links of the embedding found. This is done using algorithm 5. A breadth-first search (BFS), used in Alg. 4 for finding shortest path, has a complexity of $O(|V| + |E|)$ which we perform E^x times in Alg. 4. The complexity of the simulation procedure (Alg. 3) is then $O((|V| + |E|) \times |E^x|)$. Furthermore, at worst we have $|E| = \frac{|V|(|V|-1)}{2}$ and $|E^x| = \frac{|V^x|(|V^x|-1)}{2}$, so the complexity of the link embedding phase is $O(|V|^2 \times |V^x|^2)$.

It can then be concluded that the complexity of the NRPA algorithm (Alg. 1) is $O(|E^x| \times |E| \times N^l) = O(|V^x|^2 \times |V|^2 \times N^l)$. Note that we used the simplest possible embedding function for links. If one swaps it out for a more elaborate function, such as an exact method[30], one taking congestion[31], delays[32] or survivability[33] into account, the complexity would typically increase. We expect such a costlier function to be used in a more realistic setting. Also note that in a typical scenario, both physical network and virtual networks are sparse, *e.g.* $|V|^2 >> |E|$ and $|V^x|^2 >> |E^x|$ hence the complexity of BFS can be assumed to be reduced to $O(|V|)$ and the complexity of NRPA to be $O(|V^x| \times |V| * N^l)$. Furthermore in most cases $|V| >> |V^x|$ and it further reduces to $O(|V| \times N^l)$.

Proposition 2. *The NEPA algorithm has a computational complexity of $O(|V| \times (N^l + N^{l'} \times K \times X))$ when physical and virtual graphs are sparse, where K is the number of candidates per refinement, X is the maximum number of times we try to refine the solution and l' is the level where refinements are performed.*

First, note that the number of simulations does not change compared to NRPA and is still N^l. It follows that the total number of operations performed by the simulation part of the algorithm is $O(|V| \times N^l$ as in NRPA.
The complexity of NEPA is then $O(|V| \times N^l + Z)$ where Z is the number of operations incurred by all the refinement steps. At each refinement step, we perform $X \times K$ BFS searches of complexity $|E|$. The total number of operations performed by refinements is then $Z = O(N^{l'} \times K \times X \times |E|)$. In the case of a sparse graph, this number is $Z = O(N^{l'} \times K \times X \times |V|)$. The total computational cost of NEPA is then

$$O(|V| \times N^l + N^{l'} \times K \times X \times |V|) = O(|V| \times (N^l + N^{l'} \times K \times X)) \tag{11}$$

Overall, NEPA has a greater theoretical complexity than NRPA. However, numerical results from Appendix B show that NEPA is far more effective than NRPA when they are given equal time.

4.5.2. Memory Complexity

Proposition 3. *The NRPA and NEPA algorithms have a memory complexity of $O(|V^x| \times N^l)$*

We will now prove proposition 3. First, in the worst case, each simulation procedure call can lead to finding $|V^x|$ new unexplored states, each of which requires to store a float representing its weight in the policy. If every state found in every simulation call is seen only once, we have to store $N^l \times |V^x|$ floats since

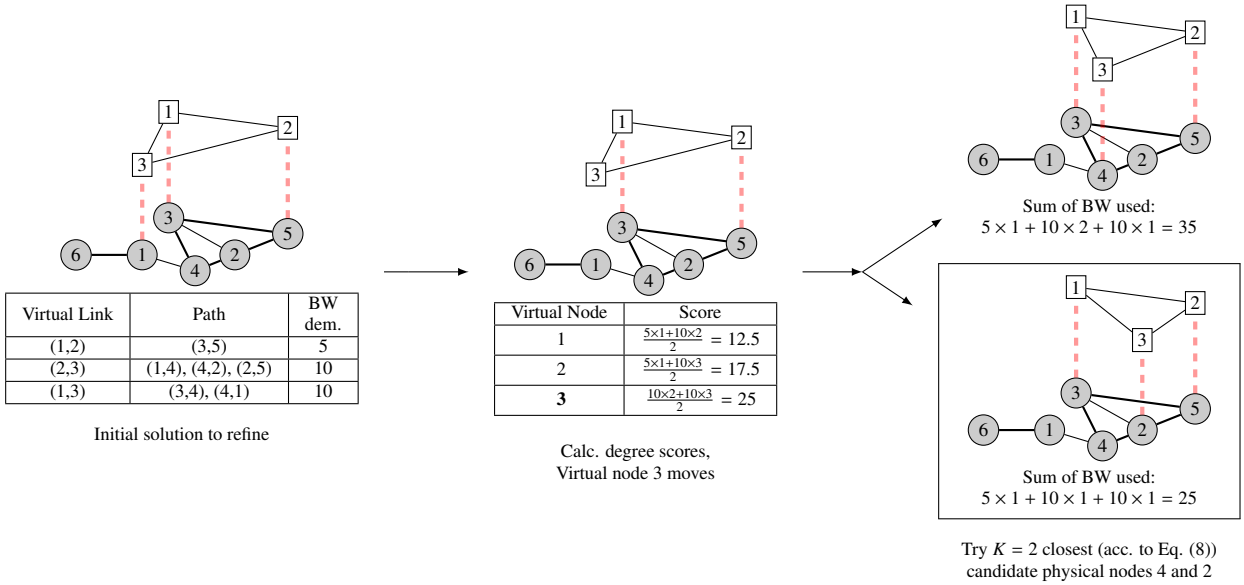


Figure 7: Example iteration of Refine with K=2. The placement chosen after the iteration is in the box. We assume all nodes have enough CPU and all links have enough BW. The chosen solution has the lowest amount of bandwidth used as reward only depends on it (CPU demands and uses are the same for a given slice)

as seen in the previous proofs, we call the simulation procedure N^l times. The other source of memory consumption in NRPA is the storage of the sequences, *seq* and seq^{best}. Those are both of length $|V^x|$. Since NRPA calls itself recursively, we also have to count the sequences stored by its infant calls. There are at worst l such infants since the recursive call depth is of l and there is only one call of a given level active at the same time, and once a call returns it frees the memory. Hence the memory consumption of the stored sequences is $O(l \times |V^x|)$. The total memory complexity of NRPA is $O(|V^x| \times N^l + l \times |V^x|) = O(|V^x| \times N^l)$. For NEPA the memory complexity remains the same as NRPA because the refinement procedure does not incur a significant memory usage, as it only requires memory to store the best solution (a virtual network which requires $O(|V^x|$ memory in case of a sparse graph).

4.6. Dimensionality reduction and pre-treatment

In practice, we make a slight modification to the MDP model in order to make the NEPA search more effective. First, we note that for a given couple of virtual node v_i^x and physical node v_j, if the maximum amount of bandwidth required by links adjacent to v_i^x exceeds the maximum available bandwidth of links adjacent to v_j, then we know one of the adjacent links of v_i^x would be impossible to embed if v_i^x was placed on v_j. Hence, we reduce the size of the action space by removing such actions before running NRPA. Similarly, if the sum of the bandwidth adjacent to v_i^x exceeds the sum of bandwidths available on links adjacent to v_j, then we know it would not be possible to place all virtual links if v_i^x was placed on v_j, hence we remove this action from the set of possible actions for placing v_i^x. Finally, before the placement, we sort the nodes according to the number of physical nodes that could host them. This draws on the idea that if a node has only few possibilities for placement, we should treat it first, otherwise there would be a high chance of blocking the possible host with another virtual node placed before. By doing this, we avoid exploring some unfeasible placements.

5. Numerical Results

In this section we extensively compare NEPA with several other methods from the state-of-the-art, demonstrating the superiority of of its performance consistently on various scenarios. We first compare on synthetic physical networks generated randomly. Then, algorithms are tested with real physical networks from the topologyZoo dataset. Finally, in order to assess the performance of each tested algorithm on large problems against the theoretical optimum, we compare on a set of Perfectly Solvable Scenarios [34], which are constructed so the optimal is known but is very hard to achieve. This step is often overlooked in the literature but we argue it is of key importance in order to assess the quality of each algorithm. Note we make sure the range of CPU and Bandwidth capacities fit reality: for CPUs, a typical server CPU (such as intel Xeon) would have between 8 and 56 cores (for example Xeon Platinum 9282). Also note that some server motherboards can host 2 CPUs (for example ASUS WS C621E). For ethernet links, it is common to find bandwidths in the order of 50-100 Gbps, see for example [35].

5.1. Compared methods

All our experiments are ran with an Ubuntu machine with a 16-core Intel Xeon Gold 5222s machine with 32 GB RAM, excepted for GraphVine which requires to be ran on another machine equipped with a GPU (see below). We compare our proposed method NEPA with the following methods:

- MaVEN-S [8] is a Monte Carlo Tree Search based algorithm which uses a model equivalent to ours for modeling the VNE and the same shortest path algorithm for final reward calculation. It makes sense to compare it with our method as it is similarly based on randomly simulating node placements but uses a different exploration strategy. This strategy, called Upper Confidence Bound for Trees explores the MDP as a tree of states (rooted in the initial state). It chooses where to descend in the tree by balancing exploration of new states and exploitation of known states, with the objective to minimize the regret of exploring new states given the expectated reward yielded by known states.
- UEPSO [14] is a particle swarm optimization (PSO) based meta-heuristic algorithm that shows good performance for the VNE.
- GraphVine [20] is a recently proposed method that exploits graph neural networks for selecting the physical nodes on which to place the virtual nodes. Note that GraphVine, like us, learns online, different from other neural network approaches such as [9], which requires an extensive offline training first, tied to the physical network. For this reason, a comparison with these other approaches would not be fair. This is why we prefer to compare with [20] instead.

For the sake of clarity, we keep in this section only the comparison with the state-of-the-art methods (mentioned above). We postpone the ablation study of NEPA (and its improvement over NRPA) to Appendix B, which shows the benefits brought to NEPA by weight initialization and neighborhood-based refinements. We implemented all these methods in the Julia programming language and made the code available as open source[36], except for GraphVine for which we use the publicly available Python/Pytorch implementation. In order to compare in the fairest manner possible, we run the following experiments:

- We run NEPA with parameters $N = 5$ and $l = 3$.
- MaVEN-S is ran with a computational budget (*e.g.* the total number of link placement attemps it executes per slices) of 445 link placements per slice. Note we tried to run it for longer times (up to 670 iterations per slice) without a significant improvement of results.
- Since UEPSO is a non-recursive algorithm, it is easier to stop it at any moment and get a valid placement. Hence here, we simply stop UEPSO after a certain amount of time equal to the mean time taken by NEPA.
- Finally we run GraphVine with the default implementation, as it is a quite different algorithm which does not rely on repeated simulations and since it can exploit a GPU. However with our original machine, we note that it is the slowest to run. As shown in [20], the algorithm is better suited for using a GPU. For that reason, we run it on another computer which has a GPU (as it gave the best runtime). This machine uses an nvidia A3000 and an Intel i7-11850H CPU.

Runtimes are depicted in Figure 8. We run each of the described experiments 10 times with different random seeds, excepted for GraphVine for which we run it only once due to the high computational cost. Note that we compute 99% confidence intervals of acceptance and revenue-to-cost ratio for MaVEN-S, UEPSO and NEPA. Some figures do not display them because they are too narrow to be visible on figures. (*e.g.* confidence interval in the order of less than $\pm$ 0.01 for acceptance and revenue-to-cost ratio)

5.2. Results on synthetic physical topologies

We start our experiments with a sensitivity analysis. For this part, we generate scenarios with default parameters and we vary each of these one by one in order to assess the results on a representative set of cases. Default parameters are reported in Table 2. We choose to generate our slices and virtual networks with the Waxman generation algorithm as it is commonly used in the VNE litterature [8][5]. We choose to generate 500 slices per scenario as we validated experimentally this gave enough time for the system to stabilize in terms of acceptance ratio. We then vary parameters in the following ways :

- We generate slices with varying Poisson arrival rates between $\lambda = 0.02$ and $\lambda = 0.08$ arrivals per second. (results in Fig. 9.1/9.5)
- We generate slices with sizes (number of virtual nodes) with minimum size $7 + i$ and maximum size $13 + i$ for $i \in [0, 9]$. (results in Fig. 9.2/9.6)
- We modify the physical network from the default scenario by removing bandwidth and CPU capacities in increments of 5 from links and nodes of the physical network, making resources scarcer. Since initially the resource capacities (CPU and BW) are chosen uniformly at random between 50 and 100, their mean value is about 75. Since we remove from all nodes and links, the mean number of resources for the different scenarios is 70, 65, 60, down to 45. (results in Fig. 9.3/9.7)
- We generate 10 different physical networks and slice sets for each physical network size of 50, 60, 70, 80, 90, 100 nodes. In those scenarios, we use the default parameters, but with $\lambda = 0.04$ and slice sizes as specified in Table 3. We scale the size of slices with respect to the physical network size since our early experiments showed that if the slice sizes were the same for all physical networks tested, it resulted in too easy scenarios for larger physical networks, where most algorithms reached performances close to 100% acceptance rate, making the comparison pointless. (results in Fig. 9.4/9.8)

Figure 9 shows that on every tested scenario, NEPA beats all other algorithms by a large margin, consistently beating MCTS of around 50% of acceptance and the best of other contenders

Parameter	Default Value
Slice arrival rate λ	0.02
Slice departure rate μ	0.005
Slice generator	Waxman ($\alpha = 0.5, \beta = 0.2$)
Number of slices	500
Min $\|V^s\|$	7
Max $\|V^s\|$	13
$\|V\|$	75
$\|E\|$	273
CPU demands (number of cores)	1 - 50
BW demands (Gbps)	1 - 50
Physical CPU capacities (number of cores per node)	50 - 100
Physical BW capacities (Gbps per link)	50 - 100

Table 2: Default scenario generation parameters

Number of physical nodes of	Number of virtual nodes
50	7-13
60	8-14
70	9-15
80	10-16
90	11-17
100	12-18

Table 3: Mean number of nodes of virtual networks for each size of physical network tested

(which are close to each other, above MCTS) by 15%, regardless of the case. In terms of revenue-to-cost ratio, it is striking to note that NEPA beats other algorithms by an even larger margin than for acceptance. This means NEPA tends to use less physical resources, which is the reason why it achieves a better acceptance. This suggests that reducing the overall consumption of each slice enables us to leave more resources for future incoming slices, making it possible to place them.
For variable size physical networks (figure 9.4), we observe again that NEPA beats other contenders by a large margin, since it accepts up to 60% more than MaVEN-S, and consistently beats it by 20 points of acceptance. It is remarkable to note how regular the patterns are in the acceptance plots, especially given that results are averaged for different topologies (recall that in the variable size experiment, for each seed, we generate a different random topology). The difference between algorithms is almost always the same regardless of sizes and difficulty of the instance, with NEPA as a clear winner. Regarding revenue-to-cost ratios, we note that all algorithms excepted NEPA have average ratios between 0.5-0.6. NEPA beats them by a large margin, since it is the only one to consistently reach 0.7 to 0.75 of revenue-to-cost ratio, showing again the effectiveness of the neighborhood based refinement in increasing the quality of the solutions found.

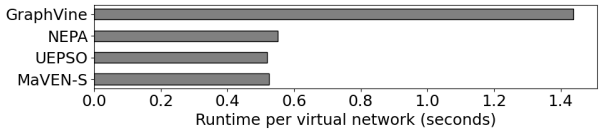


Figure 8: Mean runtime per slice for each algorithm (calulated by averaging runtime per slice on all runs of varying λ scenarios)

5.3. Real Topologies

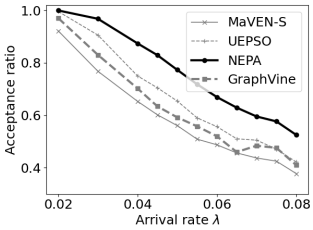
We try all algorithms with real topologies from the TopologyZoo [37] dataset as physical networks. We choose to use topologies that have between 60 and 200 nodes and are connected. This leaves us with 26 topologies with $|V|$ between 60 and 197. We use bandwidth capacities chosen randomly between 250 and 300 Gbps and CPU capacities between 50 and 100 cores. Slices are generated with our standard scheme but with $\lambda = 0.04$ arrivals per second.

The results depicted in Figures 10 and 11 show that NEPA is a lot more effective than UEPSO and MCTS. We achieve improvements of at least one order of magnitude in terms of acceptance compared to these algorithms with our best result being to more-than-triple their acceptance ratio on the Syrin topology by using NEPA. GraphVine interestingly performs much better on those topologies than on random ones, however NEPA still is the best in terms of acceptance rate with only few experiments where GraphVine manages to reach a similar acceptance as NEPA, an only 2 where it beats our algorithm by a thin margin. In terms of revenue-to-cost ratios, results are on par with acceptance, since again NEPA beats other algorithms (GraphVine aside) by an order of magnitude. We note that on some instances, GraphVine has a worst revenue-to-cost ratio than NEPA but still matches it in terms of acceptance (CogentCo, GtsCe, Pern, ...). This observation implies that although improving revenue-to-cost ratio is a key factor in order to reach a higher acceptance, it is not the only parameter to look for, since an approach can have a worst revenue-to-cost ratio but a better long term acceptance ratio.

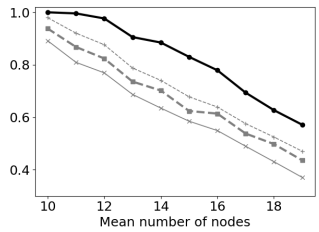
Overall, our results suggest that NEPA is the best suited method compared to state of the art algorithms when it comes to placing slices on real-world networks. We note however that although the GraphVine method struggled on random topologies, it is competitive when it comes to real networks, although not as good as NEPA overall. We think it would be a great area of future research to try to combine both methods, as NEPA might be able to leverage the addition of GraphVine's neural network for reusing information learned accross experiences. Results also suggests that the topology of the physical network has a great influence on the performances of each algorithms.

Starting from that observation, we investigate the key topological features that enable NEPA to peform so much better in those cases. Our data exploration reveals that real topologies tend to have a larger diameter and mean shortest path length (*e.g* overall longer paths) than generated ones. They also tend to have a lower link density (*e.g.* they have "less" edges). We depict statistics for these topologies compared to generated ones in 7, in Appendix C. The difference between real and synthetic topologies questions the appropriateness of the widely used in the litterature Waxman Generator for VNE studies.

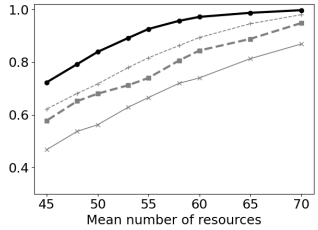
Furthermore, there is often a larger standard deviation in the length of shortest paths (*e.g.* distances) in real topologies than in synthetic ones. We notice that the differences between NEPA (which uses distance information a lot) and other algorithms is the most important for real topologies where the standard deviation in shortest path length is the largest. For example with



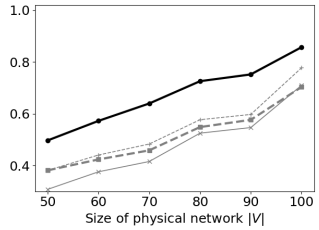
9.1 Acceptances for varying arrival rate (λ)



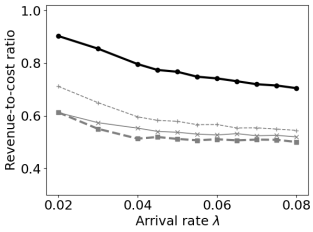
9.2 Acceptances for varying slice size



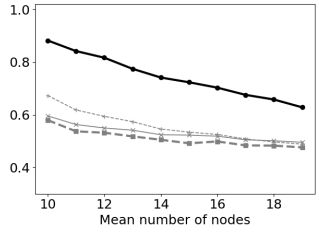
9.3 Acceptances (varying CPU & BW capacities)



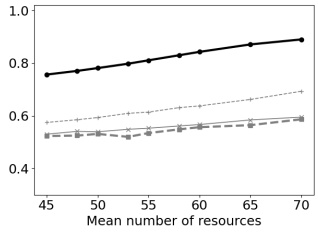
9.4 Mean acceptance ratios for varying physical network sizes



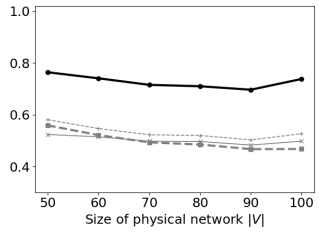
9.5 Revenue-to-cost ratio for varying λ



9.6 Revenue-to-cost ratio for varying slice size



9.7 Revenue-to-cost ratio (varying CPU & BW capacities)



9.8 Revenue-to-cost ratio (varying network sizes)

Figure 9: Results for sensitivity analysis experiments

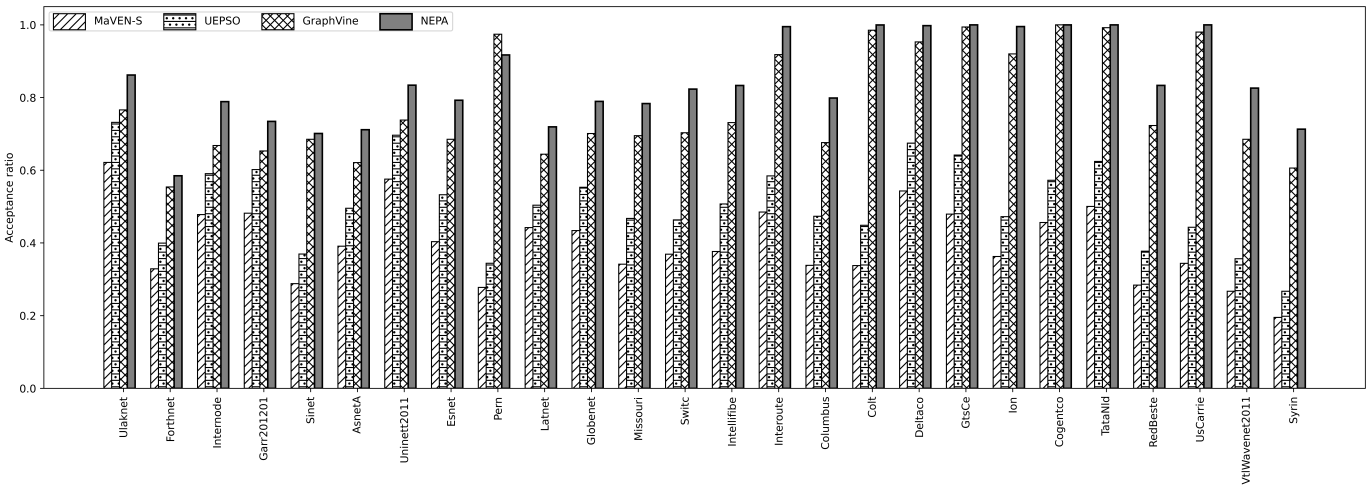


Figure 10: Acceptance on real physical networks. Results are ordered by increasing shortest-path length variance.

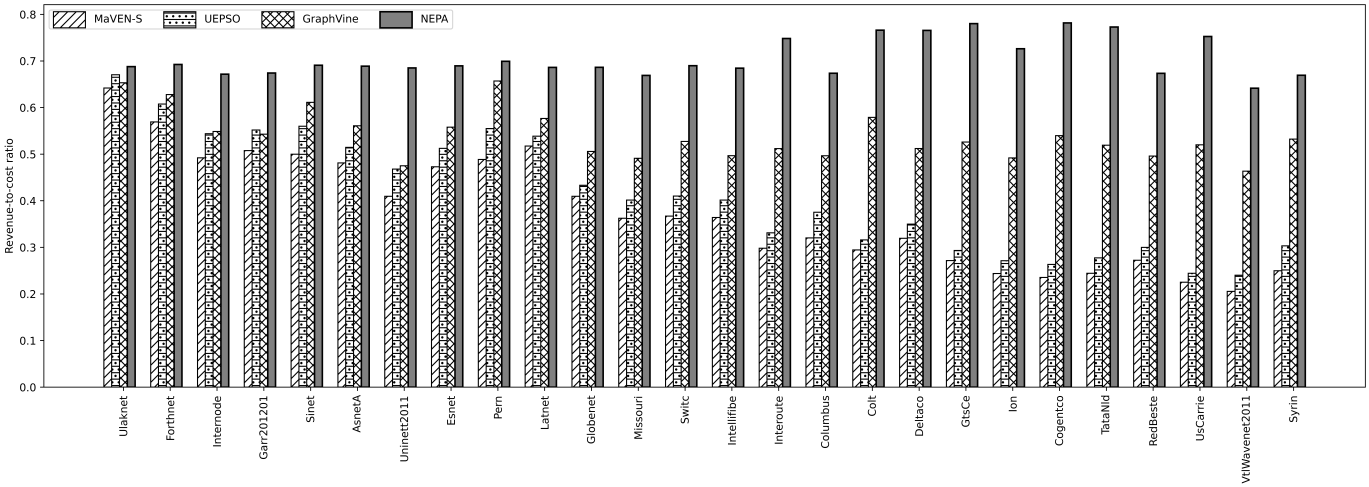


Figure 11: Revenue-to-cost ratios on real physical networks

	Mean distance	Diameter	Standard deviation of shortest path length	Clustering coefficient
Correlation	0.72	0.65	0.71	-0.17
p-value	3×10^{-5}	3×10^{-4}	5×10^{-5}	0.38

Table 4: Correlation between graph topological statistics and improvement ratio from NRPA to NEPA for real topologies.

Syrin, where acceptance rises from 0.18 - 0.23 for MaVEN-S and UEPSO to 0.77 with NEPA, and where the shortest path length standard deviation is 6.77. We observe the same pattern with VtlWavenet2011, UsCarrie, RedBeste, Cogentco or TataNld. On the other hand, when standard deviation is low (Ulaknet, Internode, Sinet, Forthnet, ...), we notice that the differences between algorithms are much lower, as the information to be leveraged from distances is less important, since choosing a "bad" placement would result in a smaller augmentation of the cost. Note however that NEPA still beats all other algorithms in those cases, although it is by a thinner margin. We quantify the advantage NEPA gets from exploiting distance information (*e.g.* using weight initialization and refinement) by calculating the augmentation ratio between the acceptance of NEPA and the acceptance of NRPA-W (which is depicted in Appendix B) for each real-topology scenario. We choose to compare against NRPA-W as it is the same algorithm, but with no help from distance-based information during node placement. We then calculate the correlation between the augmentation ratio and different topological measures for results on the real-world topologies.

Those correlation results (obtained using Pearson correlation coefficient) are depicted in Table 4. We find strong positive correlations of 0.65, 0.71 and 0.72 respectively for diameter, standard deviation of distances and mean distance, meaning distance information is particularly important to exploit when the physical network has a high standard deviation in the distribution of distances, such as in many of the real networks studied. This explains why our algorithm can perform so much better on these instances. This is relatively intuitive to understand: these cases correspond to instances where there are a lot of chances to make "high-cost mistakes", *e.g.* where a single virtual link could incur à lot of cost by being placed on two physical nodes that are far from one another. Our distance-based techniques explicitly mitigate this by ensuring virtual nodes are placed close to one another, which results in an even greater performance boost on those cases. Also notice that in this paragraph our analysis was focused on standard deviation but applies to the other distance related metrics, as mean distance, diameter and standard deviation all have a correlation between one another of 0.99, according to our measurements.

5.4. Specific case: Perfectly solvable scenarios

Third, we evaluate each algorithm on perfectly solvable scenarios (PSS). A PSS is a kind of scenario proposed by Fischer [34] that is generated such that there are only slice arrivals and no departure, and such that it is possible to place all slices. The scenario is generated so the only solutions where all slices are placed leave 0 remaining resources. Hence it is a very hard, but theoretically feasible scenario (*e.g.* 100% of acceptance is reachable).

We argue evaluating algorithms on such scenarios is an important but often overlooked practice in the literature. Indeed, it is generally infeasible to evaluate the suboptimality gap as computing the exact placement would be computationally too expensive. We generate 10 PSS scenarios, using the additive algorithm from [34] . The generation is done by first generating slices, then "adding" them in order to form the physical network. The "addition" step is done by treating each virtual node iteratively, either reusing an already created physical node or creating a new one for the current virtual node (the choice is made probabilistically). Then, once all physical nodes have been created, they are linked so that if two nodes host neighboring virtual nodes, bandwidth is added to the link between them equal to the requirement of the corresponding virtual link.

Each scenario $PSSi$ is generated from a batch of 100 random slices of random size $7 + i$ to $10 + i$ and a probability of reusing existing nodes of 0.93. This parameter was chosen empirically as it enabled us to generate graphs of sizes similar to those we experimented with in the previous section, as shown in Table 5. Our experiments show the same kind of results (shown in figure 12) as for the previous part, *e.g.* that NEPA outperforms all other methods by an order of magnitude (consistently beating the second best method, UEPSO by accepting up to 35% more slices in PSS9), both in terms of acceptance and revenue-to-cost ratio. However, it is striking to note that we never achieve a result of 100% of acceptance (the best one is NEPA on PSS0 - the smallest case - with 69%), even though the revenue-to-cost ratio gets really close to 1 (up to 0.965 in the first scenario). This means that although we achieve an almost perfect online optimization objective, resources on the physical network are badly used, leaving a lot of "holes" which are unusable. We believe this shows the need for the VNE community to investigate better reward functions which could assess the quality of a solution with other metrics than pure resource usage (with the goal to define whether a virtual network "fits" its embedding or not). In that regard, a recent article [17] made a first step in that direction by proposing to enrich the reward function with degree information, which slightly helps improving acceptance depending on the algorithm used. However, the results shown in Appendix A demonstrate this reward function has no significant impact on the results of the NEPA placement, suggesting it is ineffective when the algorithm is already very good. Another possibility would be to place virtual networks in batches, which might enable us to combine placements better, at the cost of a higher computational complexity.

6. Discussion

Our results illustrate that the widely adopted idea of optimizing placement for reduced bandwidth [8] [9] [26] [20] consumption in order to let more resources for future virtual net-

Instance	PSS0	PSS1	PSS2	PSS3	PSS4	PSS5	PSS6	PSS7	PSS8	PSS9
$\|V\|$	67	86	73	94	111	104	124	121	135	150
$\|V^s\|$	7-10	8-11	9-12	10-13	11-14	12-15	13-16	14-17	15-18	16-19

Table 5: Number of nodes for physical and virtual networks of PSS scenarios

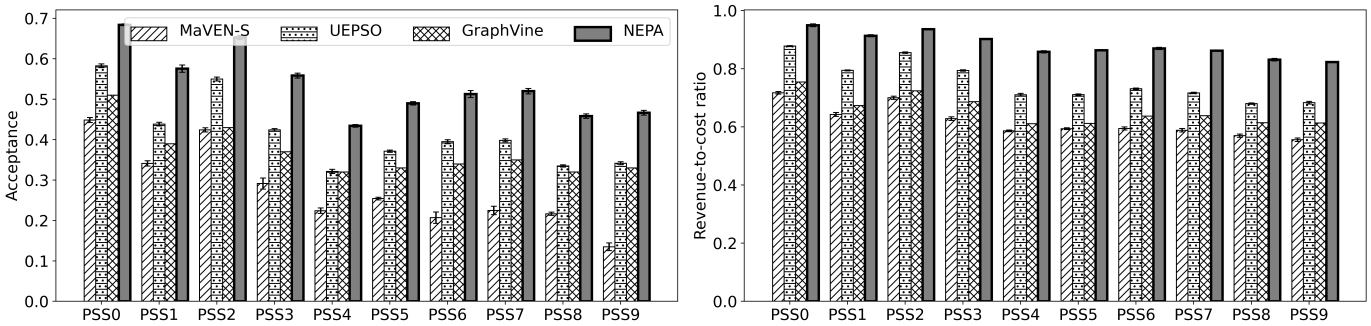


Figure 12: Acceptance and revenue-to-cost ratios for perfectly solvable scenarios

works good. We show that pushing this logic a step further by explicitly reducing the consumption of the found solution enables our algorithm to reach even better results. The main hurdle with the refinement step is the computational cost, which we overcome by selecting promising solutions to refine instead of trying to refine any solution. The NRPA algorithm is easily adaptable into NEPA due to its recursive nature. It is an open question whether other algorithms such as UEPSO could be modified in order to similarly select promising states to be refined, which would enable them to keep the computational cost low while finding better embeddings.

We shall now focus on the differences between MaVEN-S (which we call a mean-based approach) and NRPA and NEPA (which we call max-based approaches), in an effort to try to explain why max-based approaches perform so much better than the MCTS-based MaVEN-S algorithm (refer to Appendix B which shows the ablation study of NEPA, also demonstrating that NRPA without the improvements brought by NEPA outperforms MaVEN-S), while both types of algorithms are Monte Carlo Search algorithms that try to balance exploration and exploitation of the tree formed by the MDP underlying our embedding problem.

In figure 13, we illustrate with a toy example the potential results obtained after executing 6 random simulations (with a policy that could either be given by NRPA/NEPA's policy matrix or by MaVEN-S's tree). The tree represents the MDP, with the final values obtained through simulations at its leaves. This tree will serve us to illustrate the key difference between algorithms: max-based approaches assume that the best solutions lie near the single best solution found so far, hence they will explore regions of the search tree even with low expected value as long as they contain the best solution found so far. On the other hand, MCTS is designed to explore the states with the best expected (mean) value. Hence, on the tree from figure 13, MaVEN-S, would exploit more the states of the bottom subtree, since their mean value would be of 0.53, while max-based approaches would go for the top sub-tree since the maximum known value is of 0.9, even though the mean value would only be of 0.47.

We argue this is desirable for the VNE problem we solve, because optimizing for the mean expected reward is typically suited for problems where there is uncertainty, *e.g.* where taking one action from a given state can yield to several different states. This is not the case for the VNE, where a choice of action from a given state always yields to the same state: the model is a deterministic MDP. Hence it makes more sense to choose actions only according to the best sequence found so far and not according to the best mean value. This is what max-based approaches do since they optimize considering the best sequence found, as opposed to mean-based MaVEN-S, which partly explains why MaVEN-S is outperformed.

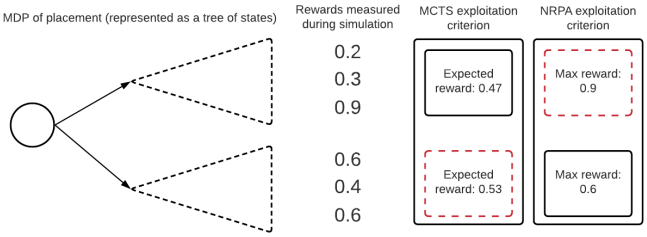


Figure 13: Toy example of MDP exploration choices

7. Conclusion

We have proposed a new approach for the virtual network embedding problem, which consists in placing virtualized networks ("slices") on a physical infrastructure (typically a 5G network). Our algorithm builds upon the NRPA (nested rollout policy adaptation) algorithm by combining its recursive reinforcement learning procedure with neighborhood search and by initializing the weights it uses for learning based on distances in the physical network. The resulting algorithm, called NEPA (Neighborhood Enhanced Policy Adaptation with Distance-weights) shows state-of-the-art results on commonly

used synthetic benchmarks as well as on real network topologies, while keeping the running time comparable to earlier algorithms. Experiments on real topologies show that since it exploits distances between placed nodes, NEPA brings acceptance and revenue-to-cost improvements of an order of magnitude compared other meta-heuristic and Monte Carlo search algorithms. It also beats a state-of-the-art graph neural network based approach (GraphVine). On random topologies, which are the most explored in the VNE litterature, we also showed that NEPA is the most robust of the tested approaches since it always reaches the best acceptance and revenue-to-cost ratio. These results will help in solving the resource allocations problems in future 5G networks, but also help the VNE community better evaluate its algorithms, since we characterized how topological features can induce enormous differences between results from different methods. In the future, we would like to demonstrate how to use NEPA on other combinatorial problems where good neighborhood search policies are also available, such as the TSP and the VRP. We also plan on incorporating offline learning by reusing the learned weights from past NEPA runs in order to learn better how to initialize future weights as currently these datas are not used once the placement is decided. In that regard, a combination with GraphVine would be particularly appealing. Implementing NEPA on a real 5G network is also planned in the near future. Finally, we contribute to the VNE community by making our set of instances and of implementations available online [36].

References

[1] Ibrahim Afolabi, Tarik Taleb, Konstantinos Samdanis, Adlen Ksentini, and Hannu Flinck. Network slicing and softwarization: A survey on principles, enabling technologies, and solutions. *IEEE Communications Surveys & Tutorials*, 20(3):2429–2453, 2018.

[2] Omar Houidi. *Algorithms for virtual network functions chaining*. PhD thesis, Institut polytechnique de Paris, 2020.

[3] Mosharaf Chowdhury, Muntasir Raihan Rahman, and Raouf Boutaba. Vineyard: Virtual network embedding algorithms with coordinated node and link mapping. *IEEE/ACM Transactions on networking*, 20(1):206–219, 2011.

[4] Xiang Cheng, Sen Su, Zhongbao Zhang, Hanchi Wang, Fangchun Yang, Yan Luo, and Jie Wang. Virtual network embedding through topology-aware node ranking. *ACM SIGCOMM Computer Communication Review*, 41(2):38–47, 2011.

[5] Andreas Fischer, Juan Felipe Botero, Michael Till Beck, Hermann De Meer, and Xavier Hesselbach. Virtual network embedding: A survey. *IEEE Communications Surveys & Tutorials*, 15(4):1888–1906, 2013.

[6] Matthias Rost and Stefan Schmid. Np-completeness and inapproximability of the virtual network embedding problem and its variants. *arXiv preprint arXiv:1801.03162*, 2018.

[7] Massinissa Ait Aba, Maxime Elkael, Badii Jouaber, Hind Castel-Taleb, Andrea Araldo, and David Olivier. A two-stage algorithm for the Virtual Network Embedding problem. In *LCN 2021: 46th conference on Local Computer Networks*, pages 395–398, Edmonton (online), Canada, October 2021. IEEE. doi: 10.1109/LCN52139.2021.9524968. URL `https://hal.archives-ouvertes.fr/hal-03524809`.

[8] Soroush Haeri and Ljiljana Trajković. Virtual network embedding via monte carlo tree search. *IEEE transactions on cybernetics*, 48(2):510–521, 2017.

[9] Zhongxia Yan, Jingguo Ge, Yulei Wu, Liangxiong Li, and Tong Li. Automatic virtual network embedding: A deep reinforcement learning approach with graph convolutional networks. *IEEE Journal on Selected Areas in Communications*, 38(6):1040–1057, 2020.

[10] Mahdi Dolati, Seyedeh Bahereh Hassanpour, Majid Ghaderi, and Ahmad Khonsari. Deepvine: Virtual network embedding with deep reinforcement learning. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 879–885. IEEE, 2019.

[11] Maxime Elkael, Hind Castel-Taleb, Badii Jouaber, Andrea Araldo, and Massinissa Ait Aba. Improved monte carlo tree search for virtual network embedding. In *2021 IEEE 46th Conference on Local Computer Networks (LCN)*, pages 605–612. IEEE, 2021.

[12] Christopher D Rosin. Nested rollout policy adaptation for monte carlo tree search. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[13] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[14] Zhongbao Zhang, Xiang Cheng, Sen Su, Yiwen Wang, Kai Shuang, and Yan Luo. A unified enhanced particle swarm optimization-based virtual network embedding algorithm. *International Journal of Communication Systems*, 26(8):1054–1073, 2013.

[15] Tristan Cazenave and Fabien Teytaud. Application of the nested rollout policy adaptation algorithm to the traveling salesman problem with time windows. In *International Conference on Learning and Intelligent Optimization*, pages 42–54. Springer, 2012.

[16] Tristan Cazenave, Jean-Yves Lucas, Hyoseok Kim, and Thomas Triboulet. Monte carlo vehicle routing. In *ATT at ECAI 2020*, 2020.

[17] Christian Aguilar-Fuster and Javier Rubio-Loyola. A novel evaluation function for higher acceptance rates and more profitable metaheuristic-based online virtual network embedding. *Computer Networks*, 195: 108191, 2021.

[18] Marcio Melo, Susana Sargento, Ulrich Killat, Andreas Timm-Giel, and Jorge Carapinha. Optimal virtual network embedding: Node-link formulation. *IEEE Transactions on Network and Service Management*, 10(4): 356–368, 2013.

[19] Matthias Rost and Stefan Schmid. On the hardness and inapproximability of virtual network embeddings. *IEEE/ACM Transactions on Networking*, 28(2):791–803, 2020.

[20] Farzad Habibi, Mahdi Dolati, Ahmad Khonsari, and Majid Ghaderi. Accelerating virtual network embedding with graph neural networks. In *2020 16th International Conference on Network and Service Management (CNSM)*, pages 1–9. IEEE, 2020.

[21] Andreas Blenk, Patrick Kalmbach, Johannes Zerwas, Michael Jarschel, Stefan Schmid, and Wolfgang Kellerer. Neurovine: A neural preprocessor for your virtual network embedding algorithm. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 405–413. IEEE, 2018.

[22] Xiuming Mi, Xiaolin Chang, Jiqiang Liu, Longmei Sun, and Bin Xing. Embedding virtual infrastructure based on genetic algorithm. In *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 239–244. IEEE, 2012.

[23] Ilhem Fajjari, Nadjib Aitsaadi, Guy Pujolle, and Hubert Zimmermann. Vne-ac: Virtual network embedding algorithm based on ant colony metaheuristic. In *2011 IEEE international conference on communications (ICC)*, pages 1–6. IEEE, 2011.

[24] Xiang Cheng, Sen Su, Zhongbao Zhang, Kai Shuang, Fangchun Yang, Yan Luo, and Jie Wang. Virtual network embedding through topology awareness and optimization. *Computer Networks*, 56(6):1797–1813, 2012.

[25] Ashraf A Shahin. Memetic multi-objective particle swarm optimization-based energy-aware virtual network embedding. *arXiv preprint arXiv:1504.06855*, 2015.

[26] Long Gong, Yonggang Wen, Zuqing Zhu, and Tony Lee. Toward profit-seeking virtual network embedding algorithm via global resource capacity. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2014.

[27] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[28] Matthew Andrews and Lisa Zhang. Hardness of the undirected edge-disjoint paths problem. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 276–283, 2005.

[29] Minlan Yu, Yung Yi, Jennifer Rexford, and Mung Chiang. Rethinking virtual network embedding: Substrate support for path splitting and migration. *ACM SIGCOMM Computer Communication Review*, 38(2):17–29, 2008.

[30] Bernard Fortz, Luís Gouveia, and Martim Joyce-Moniz. Models for the piecewise linear unsplittable multicommodity flow problems. *European Journal of Operational Research*, 261(1):30–42, 2017.

[31] Fatemeh Hosseini, Alexander James, and Majid Ghaderi. Probabilistic virtual link embedding under demand uncertainty. *IEEE Transactions on Network and Service Management*, 16(4):1552–1566, 2019.

[32] Walid Ben-Ameur and Adam Ouorou. Mathematical models of the delay constrained routing problem. *Algorithmic Operations Research*, 1(2):94–103, 2006.

[33] Muntasir Raihan Rahman, Issam Aib, and Raouf Boutaba. Survivable virtual network embedding. In *International Conference on Research in Networking*, pages 40–52. Springer, 2010.

[34] Andreas Fischer. *An evaluation methodology for virtual network embeddING*. PhD thesis, Universität Passau, 2017.

[35] Ieee standard for ethernet - amendment 3: Media access control parameters for 50 gb/s and physical layers and management parameters for 50 gb/s, 100 gb/s, and 200 gb/s operation. *IEEE Std 802.3cd-2018 (Amendment to IEEE Std 802.3-2018 as amended by IEEE Std 802.3cb-2018 and IEEE Std 802.3bt-2018)*, pages 1–401, 2019. doi: 10.1109/IEEESTD.2019.8649797.

[36] Maxime Elkael. vne.jl. `https://github.com/melkael/vne`, 2021.

[37] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.

[38] Javier Rubio-Loyola, Christian Aguilar-Fuster, Gregorio Toscano-Pulido, Rashid Mijumbi, and Joan Serrat-Fernández. Enhancing metaheuristic-based online embedding in network virtualization environments. *IEEE Transactions on Network and Service Management*, 15(1):200–216, 2017.

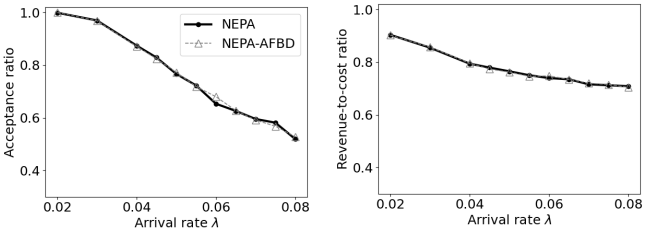
Appendix A: Comparison of results for NEPA with and without the Alternative Reward Function Based on Degrees

In this Appendix, we expose our simulation results on all tested instances when we use the AFBD (alternative function based on degrees) proposed in [17] in the reward function. This function uses a combination of the sum of Bandwidth used and the degrees of the used nodes as a cost function. The formula for the AFBD cost function for virtual network H^x placed on G is the following :

$$AFBD(G, H^x) = \sum_{\forall (v_i, v_j) \in E} B\bar{W}^x_{v_i, v_j} + \sum_{v_i^x \in V^x} deg(host(v_i^x)) - deg(v_i^x)$$

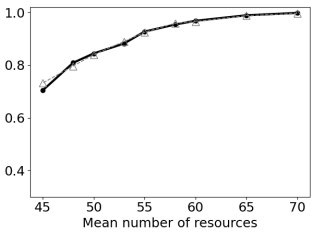
Where $deg(v_i)$ is the degree of node v_i and $host(v_i^x)$ is the physical node hosting virtual node v_i^x.
The idea behind that choice is to keep minimizing the length of the used paths, but while preserving resources on high degree nodes when possible. This revolves around the intuition that higher degree nodes tend to offer more link embedding possibilities, hence, if a virtual network can be placed by using more constraining physical nodes, it should be done, since some future virtual networks might require less constrained ones in order to be placed. In [17], the authors claim to achieve improvements (in the order of a several percents of acceptance) that could be transferred to other meta-heuristic algorithms.
We try to find out if this is the case with NEPA, since after our investigations it is the best performing algorithm at our disposal. Note that the reference meta-heuristic used in [17] is Harmony Search, which has comparable performances to UEPSO, as shown by the same authors in [38] (in that article UEPSO is referred to as PSOI and shows very close acceptance ratio with Harmony Search based methods).

Since NEPA tries to maximize a reward function, and the AFBD function is a cost function, which should be minimized, we take $\frac{1}{AFBD(G,H^x)}$ as the reward function in this appendix. We run NEPA on all the cases of section 5.2 (synthetic networks) and section 5.3 (real topologies), but only depict a subset of those instances due to space constraints (the rest can be found online in [36]). Our results (figs. 141516) show there is no significative difference when using AFBD with NEPA.

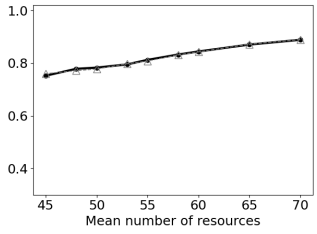


15.1 Acceptances for varying arrival rate (λ)

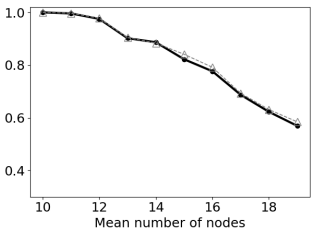
15.5 Revenue-to-cost ratio for varying λ



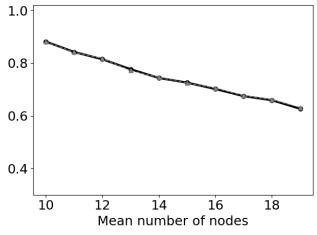
15.2 Acceptances for varying CPU/BW capacities



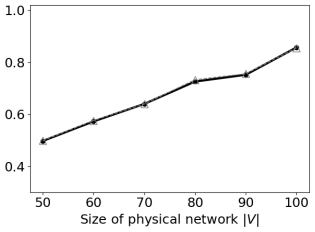
15.6 Revenue-to-cost ratio for varying CPU/BW capacities



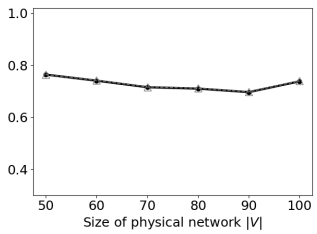
15.2 Acceptances for varying slice size



15.7 Revenue-to-cost ratio for varying slice size



15.4 Mean acceptance ratios for varying physical network sizes



15.8 Revenue-to-cost ratio (varying network sizes)

Figure 15: Results for sensitivity analysis experiments

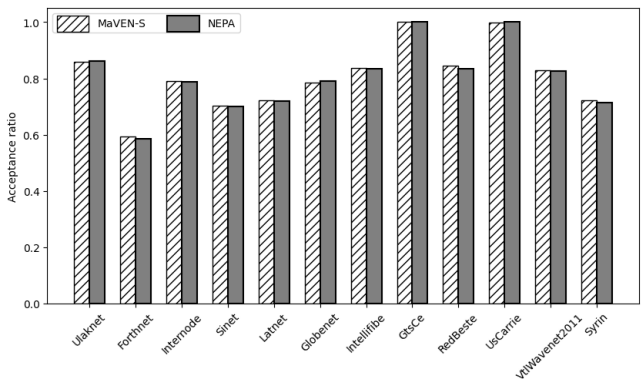


Figure 14: Acceptance for several real topologies

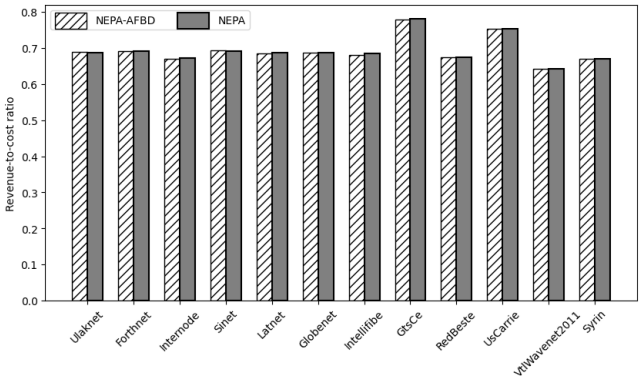


Figure 16: Revenue-to-cost ratio for several real topologies

Appendix B: Ablation study of NEPA components

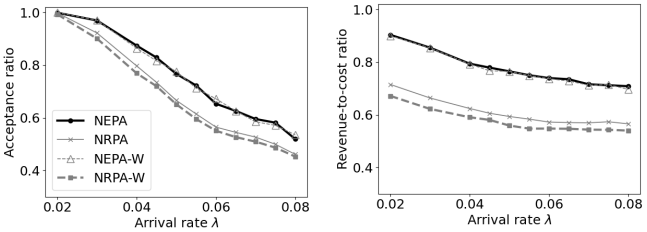
In this appendix, we compare NEPA and NRPA with their counterpart that do not use weight initialization. The presented results were ran using the parameters and instance generation described in the main article. However, due to space constraints, we only present a representative subset (10 topologies) of the ablation study experiments for real networks. The rest can be found on [36]. We call NRPA-W and NEPA-W the versions of our algorithms that do not use our weight initialization function (but initialize all weights to 0)

Note that we chose parameters $l = 3$, $N = 7$ for NRPA and NRPA-W, as we observed that this resulted in the same runtime for NEPA/NEPA-W (with parameters $l = 3, N = 5$) and these approches.

Our results from figure 18 show that NRPA outperforms standard NRPA-W by a thin margin on random topologies, hinting that the weight heuristic helps slightly for improving the results. For NEPA-W, there is no significative difference in results for randomly generated topologies compared to NEPA, meaning that the refining operation already exploits well the shortest path information. However, on real topologies (figure 17), the difference is more significative. NEPA outperforms NEPA-W by a large acceptance margin on several topologies, such as Pern and UsCarrie. In terms of revenue-to-cost ratios (figure 19, NEPA also outperforms NEPA-W on real topologies. It means on some real topologies, the extra exploitation of shortest path information is of importance. This makes sense intuitively as, as we have seen before in the paper, real topologies typically have larger diameters, mean distances and distance standard deviation, meaning an error of placement related to distances can be much more costly that on randomly generated topologies.

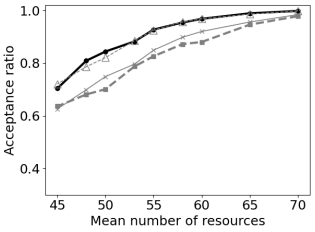
When we compare NRPA and NRPA-W on real topology, we also observe that on most of the cases, NRPA beats NRPA-W both in terms of acceptance and revenue-to-cost ratio by a larger margin than what was observed with random topologies. Similarly here, we believe this is largely due to less errors caused by choosing distant nodes when other less costly solutions existed.

Finally, we observe that on all cases, NEPA significantly outperforms NRPA, meaning that neigborhood-based enhancement is effective at finding better solutions.

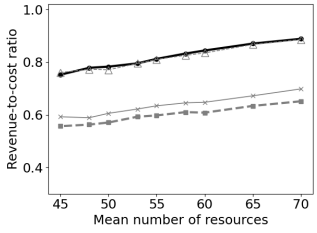


18.1 Acceptances for varying arrival rate (λ)

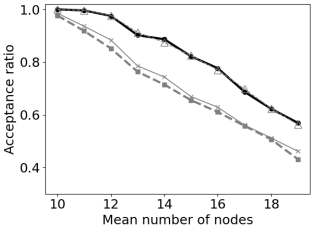
18.5 Revenue-to-cost ratio for varying λ



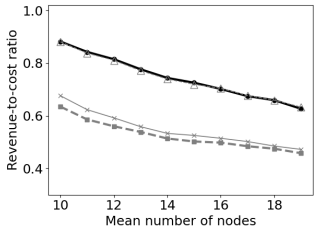
18.2 Acceptances for varying CPU/BW capacities



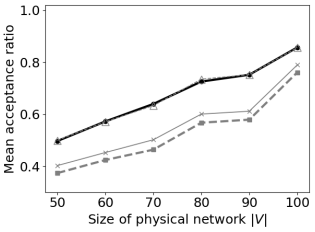
18.6 Revenue-to-cost ratio for varying CPU/BW capacities



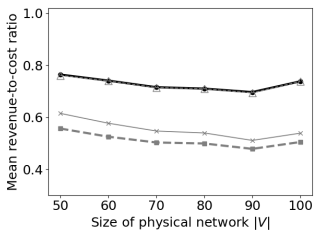
18.2 Acceptances for varying slice size



18.7 Revenue-to-cost ratio for varying slice size



18.4 Mean acceptance ratios for varying physical network sizes



18.8 Revenue-to-cost ratio (varying network sizes)

Figure 18: Results for sensitivity analysis experiments

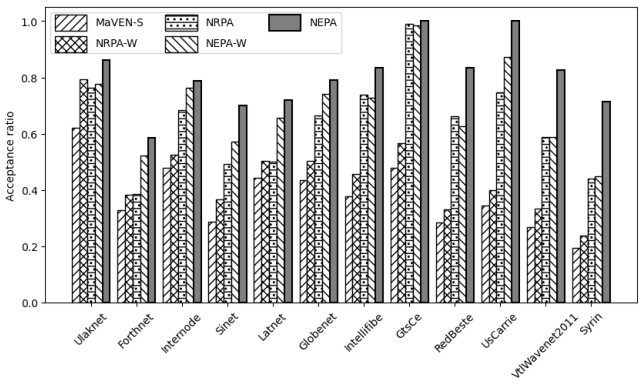


Figure 17: Acceptance for several real topologies

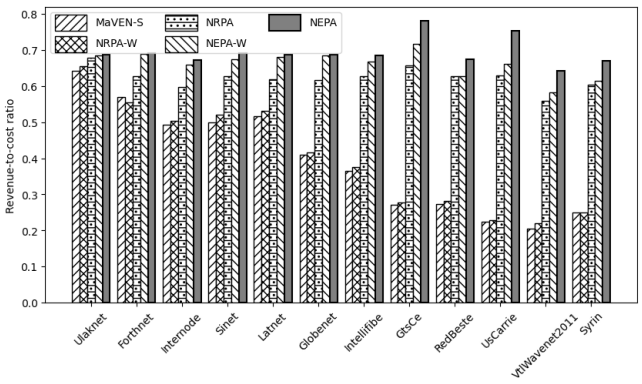


Figure 19: Revenue-to-cost ratio for several real topologies

Appendix C: statistics on network topologies

Instance	Mean distance	Diameter	Distance Standard Deviation	Clustering Coefficient
Intellifibe	6.33	15	2.99	0.088
Latnet	3.96	12	2.17	0.058
VtlWavenet2011	13.07	31	6.50	0.000
Syrin	11.95	31	6.77	0.000
Globenet	5.23	15	2.46	0.081
Forthnet	3.28	7	1.06	0.016
GtsCe	9.83	21	4.18	0.082
Ulaknet	2.44	4	0.59	0.000
Sinet	3.93	7	1.41	0.000
Internode	3.60	6	1.14	0.013
UsCarrie	12.09	35	6.46	0.058
RedBeste	10.59	28	5.66	0.009
Missouri	6.23	14	2.73	0.025
Interoute	7.62	17	3.39	0.105
Columbus	7.24	18	3.62	0.045
Garr201201	3.62	8	1.28	0.054
Cogentco	10.51	28	5.10	0.012
Deltaco	7.16	23	3.80	0.107
AsnetA	3.78	8	1.43	0.127
Switc	6.09	13	2.80	0.110
Uninett2011	4.25	9	1.62	0.018
Ion	10.14	25	4.79	0.011
Pern	4.55	8	1.89	0.004
Esnet	4.32	9	1.68	0.034
Colt	9.35	20	3.70	0.040
TataNld	9.85	28	5.17	0.065

Table 6: Statistics on each real topology

Instance	Mean Distance	Diameter	Distance standard deviation	Clustering coefficient
Waxman 50	2.58	6	0.88	0.169
Waxman 60	2.45	4	0.76	0.108
Waxman 70	2.50	5	0.77	0.152
Waxman 80	2.41	5	0.73	0.142
Waxman 90	2.44	4	0.70	0.127
Waxman 100	2.27	4	0.66	0.160
Erdos-Renyi 0.03	4.33	9	1.60	0.032
Erdos-Renyi 0.04	3.66	8	1.27	0.042
Erdos-Renyi 0.06	3.27	7	1.09	0.019
Erdos-Renyi 0.08	2.74	6	0.87	0.054
Erdos-Renyi 0.11	2.34	4	0.68	0.109
Erdos-Renyi 0.16	2.01	3	0.55	0.142
Erdos-Renyi 0.2	1.89	3	0.49	0.182
PSS 1	1.93	4	0.62	0.385
PSS 2	1.82	4	0.61	0.476
PSS 3	1.91	4	0.59	0.376
PSS 4	1.94	4	0.59	0.358
PSS 5	1.83	4	0.56	0.425
PSS 6	1.98	4	0.62	0.413
PSS 7	1.92	4	0.60	0.398
PSS 8	1.91	4	0.58	0.360

Table 7: Statistics on example simulated topologies