A Novel Reinforcement Learning Algorithm for Virtual Network Embedding

Haipeng Yao[a,*], Xu Chen[a], Maozhen Li[b], Peiying Zhang[a], Luyao Wang[c]

[a]*State Key Lab. of Networking and Switching Tech., Beijing Univ. of Posts and Telecom., P.R. China*
[b]*Department of Electronic and Computer Engineering, Brunel University London, Uxbridge, UB8 3PH, UK*
[c]*Beijing Advanced Innovation Center for Future Internet Technology, Beijing University of Technology, Beijing, P.R. China*

Abstract

Network virtualization enables the share of a physical network among multiple virtual networks. Virtual network embedding determines the effectiveness of utilization of network resources. Traditional heuristic mapping algorithms follow static procedures, thus cannot be optimized automatically, leading to sub-optimal ranking and embedding decisions. To solve this problem, we introduce a reinforcement learning method to virtual network embedding. In this paper, we design and implement a policy network based on reinforcement learning to make node mapping decisions. We use policy gradient to achieve optimization automatically by training the policy network with the historical data based on virtual network requests. To the best of our knowledge, this work is the first to utilize historical requests data to optimize network embedding automatically. The performance of the proposed embedding algorithm is evaluated in comparison with two other algorithms which use artificial rules based on node ranking. Simulation results show that our reinforcement learning is able to learn from historical requests and outperforms the other two embedding algorithms.

Keywords: Virtual network embedding, reinforcement learning, policy network, policy gradient

*Corresponding author
Email address: yaohaipeng@bupt.edu.cn (Haipeng Yao)

1. Introduction

The combination of network virtualization and software defined networks is considered as the foundation towards the next generation of Internet architecture [1, 2]. Network virtualization enables the coexistence of multiple heterogeneous virtual networks on a shared network [3, 4, 5, 6]. For Internet Service Providers (ISPs), it enables new business models of hosting multiple concurrent network services on their infrastructures. Decisions for embedding are challenging problems for ISPs since it determines the effectiveness of utilization of network resources. A sub-optimal embedding algorithm will decrease the overall capacity of the infrastructure and lead to cost of revenue for ISPs. A virtual network consists of several virtual nodes (e.g. virtual routers), connected by a set of virtual links. The purpose of virtual network embedding is to map virtual networks to a shared physical network while providing the requests with adequate computing and bandwidth resources.

However, the virtual network embedding problem has been proved to be NP-hard [7]. As a result, a large number of heuristic algorithms have been proposed [8, 9, 10, 11], but most of them rely on artificial rules to rank nodes or make mapping decisions. The parameters in these algorithms are always fixed and cannot be optimized, making the embedding decisions sub-optimally. On the other hand, in prior works, the information about substrate network and the knowledge about virtual network embedding hidden in historical network request data have always been overlooked. Historical network requests are a good representation of temporal distribution and resource demands in the future.

In recent years, big data, machine learning and artificial intelligence have exciting breakthroughs achieving state of the art results such as natural language understanding and object detection. Machine learning algorithms process a large amount of data collected during a period and automatically learn the statistical information from the data to give classification or prediction. Reinforcement learning, as a widely-used technique in machine learning, has shown a great potential in dealing with complex tasks, e.g., game of go [12], or com-

plicated control tasks such as auto-driving and video games [13]. The goal of a reinforcement learning system (or an agent) is to learn better policies for sequential decision making problems with an optimal cumulative future reward signal [14].

In this paper, we introduce reinforcement learning into the problem of virtual network embedding to optimize the node mapping process. Similar to earlier works [8, 9], our work is based on the assumption that all network requests follow an invariable distribution. We divide our network request data into a training set and a testing set, to train our reinforcement learning agent (RLA) and evaluate its performance respectively. We devise an artificial neural network called policy network as the RLA, which observes the status of substrate network and outputs node mapping results. We train the policy network with historical network request data using policy gradient through back propagation. An exploration strategy is applied in the training stage to find better solutions, and a greedy strategy is applied in evaluation to fully evaluate the effectiveness of the RLA. Extensive simulations show that the RLA is able to extract knowledge from historical data and generalize it to incoming requests. To the best of our knowledge, this work is the first to utilize historical network requests data and policy network based reinforcement learning to optimize virtual network embedding automatically. The RLA outperforms two representative embedding algorithms based on node ranking in increasing long-term average revenue and acceptance ratio, while making a better utilization of network resources.

The rest of this paper is organized as follows. We present related works in Section 2. Section 3 gives a detailed introduction about the virtual network embedding problem and our network model. The design and implementation of the reinforcement learning agent together with its training and testing process are shown in Section 4. In Section 5, we evaluate the performance of the RLA, and we conclude this paper in Section 6.

2. Related Work

Virtual network embedding involves two stages - node mapping and link mapping. Some works, e.g., [10, 11], solve the problem using a one-stage approach and assign virtual nodes and links coordinately using linear programming or mixed integer programming(MIP). For example, a rounding-based approach is applied in R-ViNE and D-ViNE algorithms [10] to achieve a linear programming relaxation of the MIP. However, these methods demand certain additional constraints such as location requirements to extend the network topology to an augmented graph so that the computing space can be greatly reduced. In other works [8, 9, 15], node mapping and link mapping are solved independently. Firstly, the substrate nodes are ranked based on their availability measured with certain rules. Then, a greedy node mapping strategy is applied where the priority of mapping is decided by rank results. Substrate nodes with more available resources will be considered first in the node mapping stage. Finally, the virtual links are mapped to the shortest path that has enough bandwidth resources between fixed nodes. Yu et al. [8] focus on path splitting and migration in link mapping problem, which means a virtual link may be mapped to several substrate links and existing link mapping may change according to the condition of substrate network. However, those approaches require the support of the substrate network and might not be available. Inspired by PageRank [16] that ranks the relative importance of Web pages, the authors of [9] proposed an algorithm based on Markov random walk to solve the problem of node ranking and mapping. The availability of each substrate node as well as its neighbors is considered in node ranking. However, the node ranking methods mentioned above follow invariable procedures. Thus no automatic optimization can be performed, which leads to sub-optimal ranking and embedding results. Furthermore, the updating process of node ranking takes a rather long time to run and may not converge. The aforementioned virtual network embedding algorithms are carried out in a centralized manner, which means that a centralized controller is responsible for gathering information about substrate network

and making mapping decisions. In [17], a multi-agent based approach and a distributed protocol are proposed to ensure distributed negotiation and synchronization between substrate nodes. In addition, many works [18, 19] also consider the energy efficiency of VNE.

Haeri and Trajkovi [20] combined reinforcement learning and virtual network embedding. But different from our work, they employ the Markov decision process to solve the node mapping problem and use Monte-Carlo tree search (MCTS) as action policies. As a result, the MCTS has to be applied every time when a virtual request arrives, which requires a great amount of computing power and makes it less time-efficient. The works presented in [21, 22] also employ reinforcement learning. However, they focus on the dynamic resource management among virtual networks. Mijumbi et al. [21] applied a q-learning based reinforcement learning agent to build a decentralized resource management system, which takes the role to increase or decrease the resources allocated to a certain virtual network. Mijumbi et al. [23] employed an artificial network to make resources reallocation decisions and train the network with a q-table from reference [21]. In [22], a reinforcement learning based neuro-fuzzy algorithm is proposed. The aforementioned works apply machine learning and reinforcement learning approaches to achieving autonomous resource management in virtual networks. In our work, we utilize similar techniques, but mainly aim to improve the efficiency of virtual network embedding instead of dynamic resource management among virtual networks.

3. Network Modelling

In this Section, we present a network model and formulate the virtual network embedding problem with description of its components. The notations used in this section are shown in Table 1.

Figure 1 shows the mapping process of two different virtual network requests. A substrate network is represented as an undirected graph $G^S = (N^S, L^S, A_N^S, A_L^S)$, where N^S denotes the set of all the substrate nodes, L^S

G^S	Substrate network
N^S	Nodes of substrate network
L^S	Links of substrate network
A_N^S	Node attribute of substrate network
A_L^S	Link attribute of substrate network
G^V	Virtual network of a certain virtual request
N^V	Nodes of a virtual network
L^V	Links of a virtual network
A_N^V	Constraints of substrate nodes
A_L^V	Constraints of substrate nodes

Table 1: Frequently used Notations.

denotes the set of all the substrate links, A_N^S and A_L^S stand for the attributes of substrate nodes and links respectively. In consistency with earlier works[8, 9], in this paper we consider computing capability as node attribute and bandwidth capacity as link attribute. Let P^S denote the set of all the loop-free paths in substrate network. Figure 1(c) shows an example of a substrate network, where a circle denotes a substrate node, and a line connecting two circles denotes a substrate link. The number in a square box denotes the CPU(computing) capacity of that node, and the number next to a substrate link denotes the bandwidth of that link.

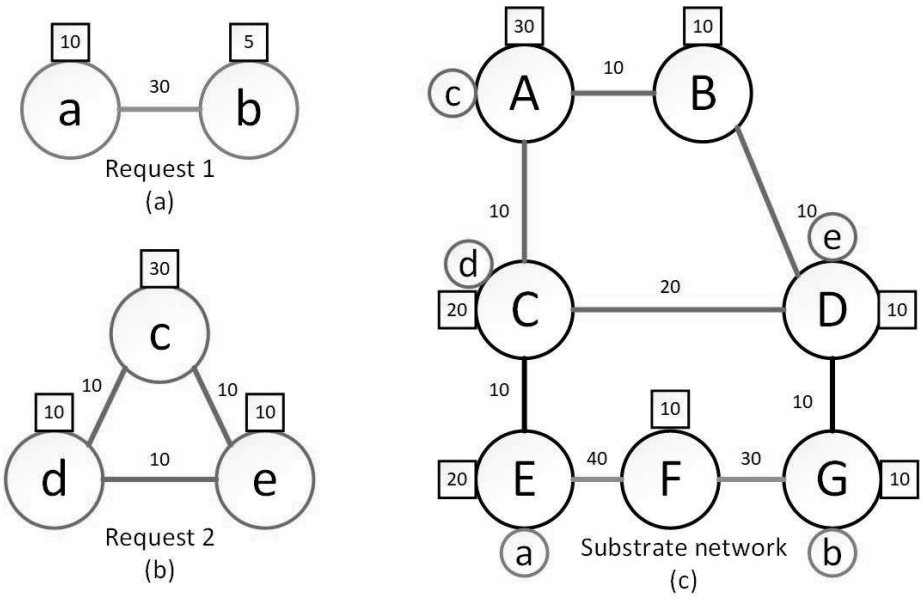


Figure.1 An example of virtual network embedding.

Similarly, we also use an undirected graph $G^V = (N^V, L^V, C_N^V, C_L^V)$ to describe a virtual network request, where N^V denotes the set of all the virtual nodes in the request, L^V denotes the set of all the virtual links in the request, C_N^V and C_L^V stand for the constrains of virtual nodes and links respectively. To map a virtual node to a substrate node, the computing capacity of the substrate node must be higher than that is required by the virtual node. To map a virtual link to a set of substrate links, the bandwidth of each substrate link must be higher than that is required by the virtual link. Figure 1(a) and (b) show two different virtual requests. Additionally, we use t to denote the arrival time of a virtual request, and use t_d to denote the duration of the virtual request.

When a virtual request arrives, the objective is to find a solution to allocate different kinds of resources in the substrate network to the request while satisfying the requirements of the request. If such a solution exists, then the mapping process will be executed, and the request will be accepted. Otherwise the request will be rejected or delayed. The virtual network embedding process can be formulated as a mapping M from G^V to G^S : $G^V(N^V, L^V) \rightarrow G^S(N', P')$,

where $N' \subset N^S$, $P' \subset P^S$.

The main goal of virtual network embedding is to accept as many requests as possible to achieve maximum revenue for an ISP, when the arrival of virtual network requests follows an unknown distribution of time and unknown resource requirements. Consequently, the embedding algorithm must produce efficient mapping decisions within an acceptable period. As shown in Figure 1, virtual nodes a and b in request 1 are mapped to substrate nodes E and G respectively, and virtual nodes c, d and e in request 2 are mapped to substrate nodes A, C and D respectively. Note that the embedding result of request 1 is not optimal. For example, the cost of bandwidth in the substrate network can be significantly reduced by moving a to F.

To determine the performance of embedding algorithms, most works use certain metrics such as a long-term average revenue, a long-term acceptance ratio, and a long-term revenue to cost ratio. The revenue measures the profit of an ISP for accepting a certain virtual request, and it depends on the amount of requested resources and the duration of it. Similar to the earlier works presented in [8, 9], we define the revenue of accepting a virtual network request as follows:

$$R(G^v, t, t_d) = t_d \cdot [\sum_{n^V \in N^V} CPU(n^V) + \sum_{l^V \in N^V} BW(l^V)] \quad (1)$$

Where $CPU(n^V)$ and $BW(l^V)$ denote the computing resource that a virtual node n^V requires and the bandwidth resource that a virtual link l^V requires respectively. As shown in the formula, virtual requests having more resources requirements or lasting longer have more revenue.

The cost function measures the efficiency of utilizing substrate network resources. We define the cost of accepting a virtual request as follows:

$$C(G^v, t, t_d) = t_d \cdot [\sum_{l^V \in N^V} \sum_{l^V \in N^V} BW(l^V)] \quad (2)$$

Where $P_(l^V)'$ denotes the set of substrate links where virtual link l^V is embedded. $C(G^V, t, t_d)$ computes the actual consumption of bandwidth resource for embedding request G^V. When accepting a virtual request, the CPU con-

sumption is always fixed, but the bandwidth consumption may vary depending on the performance of embedding algorithm discussed above.

Following the works presented in [8, 9], we use a long-term average revenue to evaluate the overall performance of our embedding method defined as:

$$\lim_{T\to\infty} \frac{\sum_{t=0}^{T} R(G^v, t, t_d)}{T} \tag{3}$$

Where T is the time elapsed. A higher long-term average revenue leads to a higher profit for the ISP. Another important metric to evaluate the mapping algorithm is a long-term acceptance ratio, which means the ratio of accepted requests to the total number of requests arrived. A higher long-term acceptance ratio means the proposed algorithm manages to serve more virtual requests.

Finally, a better utilization of substrate network resources would lead to a high long-term average revenue with comparatively low cost of substrate network. The long-term revenue to cost ratio, defined as follows, measures the utilization of substrate network resources:

$$\lim_{T\to\infty} \frac{\sum_{t=0}^{T} R(G^v, t, t_d)}{\sum_{t=0}^{T} C(G^v, t, t_d)} \tag{4}$$

A higher long-term revenue to cost ratio shows that the proposed algorithm is able to generate more profit with a comparatively less cost to network resources.

We will use these metrics mentioned above to evaluate the performance of our embedding method in the following sections.

4. Embedding Algorithm

In this section, we present the details of the proposed policy network based reinforcement learning algorithm. Specifically, we apply the reinforcement learning agent in the node mapping stage to derive the probabilities of choosing nodes. The agent takes a feature matrix extracted from the substrate network as input, and makes decisions based on a policy network which is trained from historical data.

4.1. Feature Extraction

Every substrate node has several attributes, such as CPU capacity and the total amount of bandwidth of the adjacent links. A thorough knowledge of substrate network is crucial for the reinforcement learning agent to establish a basic understanding of its state and generate efficient mapping. To facilitate the agent to choose the substrate nodes, we need to extract features of each substrate node and use them as input to the policy network.

We extract four features for each substrate node listed as follows:

- Computing capacity (CPU): The CPU capacity of a substrate node n^S has a large impact on its availability. The substrate nodes with a higher computing capacity are likely to host more virtual nodes.

- Degree (DEG): The degree of a substrate node n^S indicates the number of links connected to it. A substrate node with more adjacent links is more likely to find paths to other substrate nodes.

- Sum of bandwidth ($SUM^{(BW)}$): Every substrate node is connected to a set of links. A substrate node n^S has a sum of bandwidth resources of its neighboring links:

$$SUM^{(BW)}(n^S) = \sum_{l^S \in L(n^S)} BW(l^S) \tag{5}$$

  Where $L(n^S)$ is the neighboring links of n^S and $BW(l^S)$ is the bandwidth resource of a substrate link l^S. When a substrate node has access to more bandwidth, mapping a virtual node to it may lead to better link mapping options.

- Average distance to other host nodes $AVG^{(DST)}$: When mapping a virtual node, we also take into consideration the positions where other virtual nodes in the same request are mapped. By choosing a substrate node close to those already mapped, the cost of substrate link bandwidth can be reduced. We measure the distance between two substrate nodes in

terms of the number of links along the shortest path. The shortest path is computed following the FloydWarshall algorithm[24]. We take an average of the distance from a substrate node n^S to another host nodes $\widetilde{N}^S$ for the same request:

$$AVG^{(DST)}(n^S) = \frac{\sum_{\widetilde{n}^S \in \widetilde{N}^S} DST(n^S, \widetilde{n}^S)}{|\widetilde{N}^S| + 1} \tag{6}$$

where $DST(n^S, \widetilde{n}^S)$ is the distance from node n^s to node $\widetilde{n}^S$.

In fact, the features that we can extract from the substrate nodes would be far more than listed above. More features would bring more information about the substrate network which leads to a better performance of the learning agent. It should be noted that extracting more features from the substrate network adds complexity in computation.

After extracting the features of the k^{th} substrate node n_k^S, we take their normalized values and concatenate them into a feature vector v_k:

$$v_k = (CPU(n_k^S), DEG(n_k^S), SUM^{(BW)}(n_k^S), AVG^{(DST)}(n_k^S))^T \tag{7}$$

The purpose of normalization is to accelerate the training process and enable the agent to converge quickly. We concatenate all feature vectors of substrate nodes to produce a feature matrix M_f where each row is a feature vector of a certain substrate node:

$$M_f = (v_1, v_2 \cdots v_{|N^S|})^T \tag{8}$$

The feature matrix serves as an input to the learning agent. The feature matrix is updated along with the changing substrate network from time to time.

4.2. Policy Network

In this work, we implemented an artificial neural network called policy network as the learning agent. It takes the feature matrix as input and outputs the probabilities of mapping virtual nodes to substrate nodes.

For simplicity, we build a simple policy network with basic elements of an artificial neural network as shown in Figure 2. The policy network contains an input layer, a convolutional layer, a softmax layer and finally a node filter. For each virtual node that requires a mapping, we use the policy network to choose a substrate node for it.

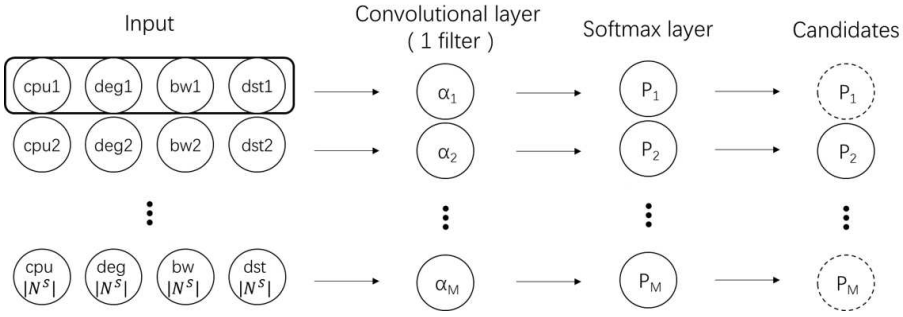


Figure.2 Policy network.

At the input layer, we compute the feature matrix and deliver it to the policy network. The policy network then passes the input feature matrix into a convolutional layer with one convolution kernel, where the policy network evaluates the resources of each substrate node. The convolutional layer performs a convolution operation on the input to produce a vector representing the available resources of each node:

$$h_k^c = \omega \cdot v_k + b \tag{9}$$

where h_k^c is the k^{th} output of the convolutional layer, is the convolution kernel weight vector, and b is bias.

Then the vector is transmitted to a softmax layer to produce a probability for each node which indicates the likelihood of yielding a better result if mapping a virtual node to it. For the k^{th} node, the probability p_k is computed as:

$$p_k = \frac{e^{h_k^c}}{\sum_i e^{h_i^c}} \tag{10}$$

The softmax function is a generalization of the logistic regression. It turns the n-dimensional vector into real values between 0 and 1 that add up to 1. The output of the softmax function indicates a probability distribution over n different possible mappings. Some of the nodes are not able to host the virtual

node in concern because they do not have enough computing resources. We add a filter to choose a set of candidate nodes with enough CPU capacities.

4.3. Training and Testing

We first randomly initialize the parameters in the policy network, and train it for several epochs. For every virtual node in each iteration, a feature matrix is extracted from the substrate network which serve as input to the policy network. The policy network outputs a set of available substrate nodes as well as a probability for each node. The probability of each node represents the likelihood that mapping a virtual node to it will yield a better result. In the training stage, we cannot simply select the node with a maximal probability as the host because that the model is randomly initialized, which means the output could be biased and better solutions might exist. In other words, we need to reach a balance between the exploration of better solutions and the exploitation of current model. To this end, we generate a sample from the set of available substrate nodes according to their probability distribution that the policy network outputs, and select a node as the host. We repeat this process until all the virtual nodes in a virtual request are assigned and proceed to link mapping. If no substrate node is available, the mapping fails due to a lack of resources. For link mapping, we apply a breadth-first search to find the shortest paths between each pair of nodes.

In supervised learning, each piece of data in the training set corresponds to a label indicating the desired output of the model. With each output from model and the corresponding label, a loss value is computed which measures the deviation between them. The loss value for each piece of data in the training set sums up to an aggregated loss value, and the training stage aims to minimize the aggregated loss value. However, in reinforcement learning tasks such as virtual network embedding, data in the training set does not have corresponding labels. The learning agent relies on reward signals to know if it is working properly. A big reward signal informs the learning agent that its current action is effective and should be continued. A small reward signal or even a negative reward

signal shows that the current action is erroneous and should be adjusted. The choice of reward is critical in reinforcement learning as it directly influences the training process and determines the final policy. Here, we use the revenue to cost ratio of a single virtual request as the reward for every virtual node in this request because this metric represents the utilization efficiency of the substrate resources. Then we apply policy gradient method to train the policy network.

The actual implementation of the proposed algorithm is non-trivial since we cannot provide each output with a label. As a result, we temporarily consider every decision that the agent makes to be correct by introducing a *hand-crafted label* into our policy network. Assume that we choose the i^{th} node, then the *hand-crafted label* in policy network would be a vector y filled with zeros except the i^{th} position which is one. Then we calculate the cross-entropy loss:

$$L(\mathbf{y}, \mathbf{p}) = -\sum_{i} y_i log(p_i) \tag{11}$$

Where y_i and p_i are the i^{th} element of *hand-crafted label* and the output of policy network respectively. We use backpropagation to compute the gradients of parameters in the policy network. Since we use *hand-crafted label*, we stack the gradients g_f rather than applying them immediately. If our algorithm fails to embed a virtual request, the corresponding stacked gradients will be aborted since we cannot determine the reward signal. If a virtual request has been successfully mapped, we compute its revenue to cost ratio as a reward r. Then we multiply the stacked gradients by using the reward and an adjustable learning rate α to achieve the final gradients:

$$g = \alpha \cdot r \cdot g_f \tag{12}$$

The learning rate α is introduced to control the magnitude of gradients and the computation speed of training. If the gradients are too large, the model becomes unstable and may not improve through the training process. On the other hand, too small gradients make training extremely slow. Therefore the learning rate needs to be tuned carefully. It can be observed from Eq. (12) larger rewards make the corresponding gradients more significant than small

ones. As a result, the choices that lead to larger rewards have larger impact on the learning agent, making it more prone to make similar decisions. When we stack a batch of gradients, we apply them to parameters and update the policy network. There are two reasons for batch updating - one is that parameter updating normally takes a long time, but doing that in batches speeds up this process. Another reason is that batch updating averages over the gradients and is more stable. The training process is shown in Algorithm 1. Lines 7-10 show node mapping stage where we compute the gradients in line 10, lines 11-13 show the link mapping stage.

In the testing stage, we apply a greedy strategy where we directly choose the node with the highest probability as the host. The testing algorithm is shown in Algorithm 2.

5. Evaluation

We conducted a number of simulation tests to evaluate the performance of the proposed reinforcement learning algorithm and compared with other embedding algorithms.

We employed GT-ITM tool [25] to generate a substrate network with 100 nodes and approximately 550 links, which is about a middle-sized ISP. The computing capacity of every substrate node is a real number that follows a uniform distribution between 50 and 100, and the bandwidth of every link is a real number that follows a uniform distribution between 20 and 50, which is similar to parameters presented in related work [8, 9].

We also generated a number of virtual requests, each with 2 to 10 virtual nodes. The computing capacity requirement of every virtual node followed a uniform distribution between 0 and 50 units. The bandwidth requirement of every virtual link follows a uniform distribution between 0 and 50 units. Virtual nodes were connected with a probability of 0.5 forming an average of $\frac{n(n-1)}{4}$ virtual links, where n is the number of nodes. The virtual requests arrived following a Poisson distribution with an average of 4 requests over 100 time

Algorithm 1 Training process.

Input: Number of epochs, $numEpoch$; Learning rate, α; Training set;

Output: Trained parameters in policy network;

Initialize all the parameters in policy network;
while $iteration < numEpoch$ **do**
    for $req \in trainingSet$ **do**
        counter=0;
        for $node \in req$ **do**
            M_f=getFeatureMatrix();
            p=policyNet.getOutput(M_f) //Get the probability distribution from policy network;
            host=sample(p) //Sample from the probability distribution to choose a node as host;
            computeGradient(host);
        end for
        if isMapped(∀ node ∈ req) **then**
            bfsLinkMap(req);
        end if
        if isMapped(∀ node ∈ req, ∀ link ∈ req) **then**
            reward=revToCost(req) //Compute revenue to cost ratio;
            multiplyGradient(reward, α) //Compute the final gradients;
        else
            clear the stacked gradients;
        end if
        ++counter;
        if counter reach the batch size **then**
            apply gradients to parameters;
            counter=0;
        end if
    end for
    ++iteration;
end while

Algorithm 2 Testing process.

Input: testing set;

Output: long-term average revenue, acceptance ratio, long-term revenue to cost ratio;

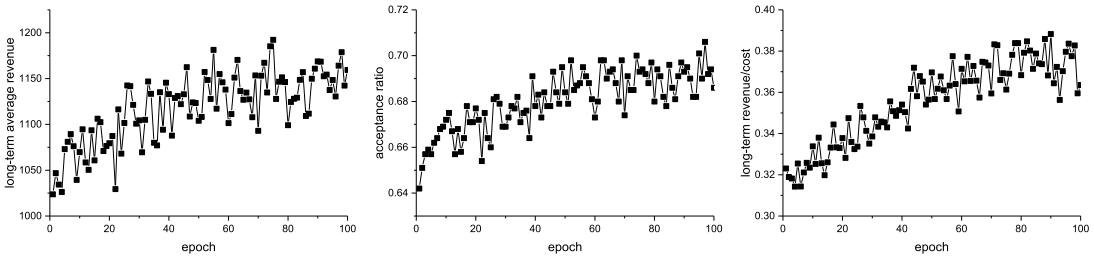
```
Initialize all the parameters in policy network;
for req ∈ testSet do
    for node ∈ req do
        M_f=getFeatureMatrix();
        host=maxProbablity(p) //Greedy strategy;
    end for
    bfsLinkMap(req);
    if isMapped(∀ node ∈ req, ∀ link ∈ req) then
        signal(SUCCESS);
    end if
end for
```

units. The duration of every requests followed an exponential distribution with an average of 1000 time units. We generate a timeline that lasted around 50000 time units containing about 2000 requests. We divided the requests equally into two sets - training set and testing set.

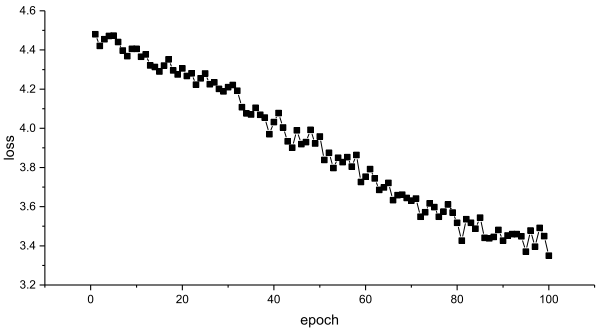
We employed TensorFlow [26] to build the policy network. We first assembled the layers mentioned in Section 4 and followed a normal distribution to initialize their parameters. Then we defined a tensor for gradient of every training step using the *compute_gradients* method of stochastic gradient descent [27] optimizer. When doing batch updates, we added all the gradients for each parameter and multiply them by a reward, and called the *apply_gradients* method of SGD optimizer to update all parameters in policy network. We train our agent for 100 epochs using gradient descent with a learning rate of 0.005. The batch size was set to 100 which means the parameters are updated every 100 requests.

5.1. Effectiveness of Reinforcement Learning

Compared to supervised learning, reinforcement learning has proven to be hard to train. It may take a very long time for the learning agent to stabilize, especially for a complicated problem such as virtual network embedding. We run the learning agent on the training data set for 100 epochs and observe its performance. Figure 3(a) shows the change of a long-term average revenue, the acceptance ratio and a long-term revenue to cost ratio during the training process. In the very beginning, the learning agent performs poorly because all the parameters in the policy network are initialized randomly. As the training goes, the random sampling allows the learning agent to explore different possibilities. The learning agent may find a good solution occasionally and receive a great reward which helps the policy network to learn to make better decisions. Consequently, the performance starts to get better, proving the effectiveness of reinforcement learning on the task. The exploration strategy sometimes leads our agent into bad choices causing a fluctuation in its performance as the training proceeds. But such cases will lead to small rewards and have small impact on the learning agent. In the later stage of training process, the performance stopped improving due to the limited capacity of functional complexity that the policy network can handle. Eventually the learning agent reaches a certain point, and the performance stabilizes in a range. Figure 3(b) shows the cross-entropy loss during the training process. Clearly, the loss decreases through the training stage and eventually starts to stabilize in the last 10 epochs.



(a) Performance on training set.



(b) Loss on training set.

Figure.3 Training process.

The result shows that the proposed reinforcement learning based algorithm is getting better performance as the training goes, which means the learning agent can adapt itself to training data.

We have proved that the reinforcement learning method can improve the performance of the learning agent on training set. But it is still unclear if the learning agent actually learns how to optimize node mapping, or simply adjusts to existing data. In order to test the generalization ability of the learning agent, we separated a testing data set that consisted of different requests from the training set and run the learning agent on it.

Different from the training process, we run the learning agent without a random sampling and applied a greedy strategy to choose a node with the maximal probability. The performance over time on the testing data set is shown in Figure 4. We compare the learning agent with another two rule-based node ranking algorithms. The first one is a baseline algorithm proposed in [8] using equation:

$$H(n^S) = CPU(n^S) \sum_{l^S \in L(n^S)} BW(L^S) \tag{13}$$

to rank substrate nodes, where $H(n^s)$ measures the availability of substrate node n^s. The other is proposed in [9] using NodeRank algorithm to measure the importance of nodes. All the three methods followed the same breadth-first search link mapping algorithm. We measured the performance of these methods

with three metrics mentioned in Section 3 - a long-term average revenue, an acceptance ratio and a long-term revenue to cost ratio.

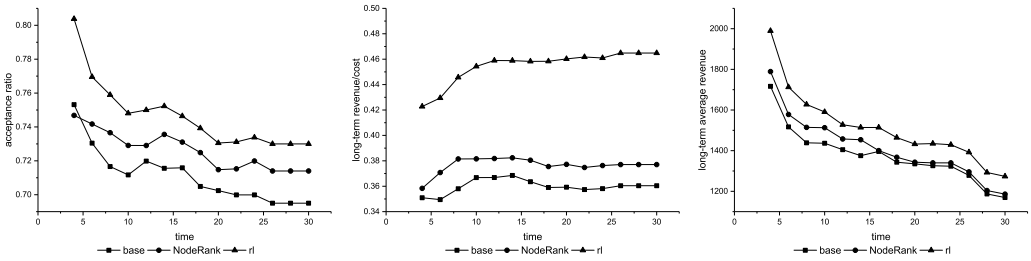


Figure.4 Performance on testing set.

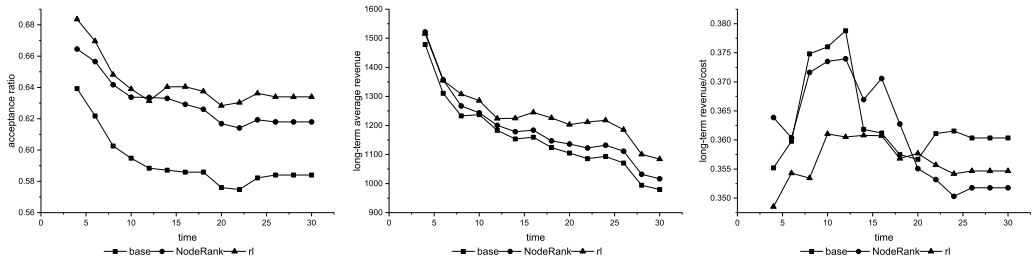
At the beginning, the performance of all the three algorithms on a long-term average revenue and an acceptance ratio decrease because the amount of resources of the substrate network decreases as more requests arrive. The long-term revenue to cost ratio is stable because it is irrelevant to the amount of available resources. Then the performance of all algorithms on all metrics starts to stabilize because the resources of the substrate network is depleted. The results in Figure 4 show that the learning agent outperforms the other two algorithms in all the three metrics. The training data set and testing data set consist of different requests, but the learning agent is able to perform well on both data sets. The conclusion is that the learning agent does not simply adjust itself to the training set, rather it is actually capable of generalizing from the training process to acquire knowledge about the substrate network and node mapping. Note that the performance improved evidently compared to the result on the training data set, because the exploration in training process may lead to bad embedding results.

5.2. Stress tests.

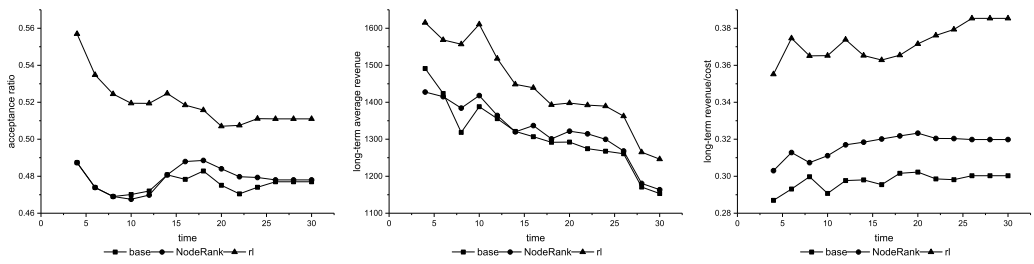
We further expanded our experiments by increasing resource demands to examine the performance of the proposed algorithm under more stress. By doing this we also expected to find the circumstances where our algorithm is robust.

Figure 5(a) shows the performances of the three algorithms when we dou-

bled the CPU requirements of the virtual nodes. Figure 5(b) shows the results when the bandwidth of virtual links were uniformly distributed between 25 and 50 units. As shown in Figure 5, the proposed reinforcement learning based algorithm performs better in the later circumstance, and outperforms the other two methods against the three metrics. In the computing-intensive environment however, the proposed algorithm achieves similar performance to the other two methods in terms of a long-term revenue to cost ratio while getting better results in the long-term average revenue and acceptance ratio. The proposed reinforcement learning based algorithm works in node mapping phase, but larger CPU requirements means less nodes with enough computing resource to choose from, which leads to relatively worse performance in computing-intensive environment. The conclusion is that the proposed algorithm can achieve comparatively better performance for bandwidth-intensive requests rather than computing-intensive ones.



(a) Performance In a computing-intensive environment.



(b) Performance In a bandwidth-intensive environment.

Figure.5 Stress tests.

6. Conclusions

In this paper, we have presented a novel virtual network embedding based on reinforcement learning. The embedding is conducted through a training process using historical data rather than relies on any hand-crafted rules. Simulation results showed that the reinforcement learning agent outperformed two existing algorithms.

In future works, we will extract more features for each substrate node to form a more complex feature matrix, and verify the effectiveness of each feature. Meanwhile, we will make efforts to build more complex structure for the policy network to expand its capacity of functional complexity by increasing the number of layers or neurons of each layer. We will also investigate the use of the connection matrix of the substrate network instead of using extracted feature matrix as input for the policy network to provide more information for the learning agent.

References

[1] D. Drutskoy, E. Keller, J. Rexford, Scalable network virtualization in software-defined networks, IEEE Internet Computing 17 (2) (2013) 20–27.

[2] R. Jain, S. Paul, Network virtualization and software defined networking for cloud computing: a survey, Communications Magazine IEEE 51 (11) (2013) 24–31.

[3] A. Fischer, J. F. Botero, M. T. Beck, H. D. Meer, X. Hesselbach, Virtual network embedding: A survey, IEEE Communications Surveys & Tutorials 15 (4) (2013) 1888–1906.

[4] C. Liang, F. R. Yu, Wireless network virtualization: A survey, some research issues and challenges, IEEE Communications Surveys & Tutorials 17 (1) (2015) 358–380.

[5] N. M. K. Chowdhury, R. Boutaba, Network virtualization: state of the art and research challenges, IEEE Communications magazine 47 (7).

[6] N. M. M. K. Chowdhury, R. Boutaba, A survey of network virtualization , Computer Networks 54 (5) (2010) 862–876.

[7] Y. Zhu, M. Ammar, Algorithms for assigning substrate network resources to virtual network components, in: INFOCOM 2006. IEEE International Conference on Computer Communications. Proceedings, 2007, pp. 1–12.

[8] M. Yu, Y. Yi, J. Rexford, M. Chiang, Rethinking virtual network embedding: substrate support for path splitting and migration, Acm Sigcomm Computer Communication Review 38 (2) (2008) 17–29.

[9] X. Cheng, S. Su, Z. Zhang, H. Wang, F. Yang, Y. Luo, J. Wang, Virtual network embedding through topology-aware node ranking, Acm Sigcomm Computer Communication Review 41 (2) (2011) 38–47.

[10] N. M. M. K. Chowdhury, M. R. Rahman, R. Boutaba, Virtual network embedding with coordinated node and link mapping, Proceedings - IEEE INFOCOM 20 (1) (2009) 783–791.

[11] S. Shanbhag, A. R. Kandoor, C. Wang, R. Mettu, T. Wolf, Vhub: Single-stage virtual network mapping through hub location, Computer Networks 77 (2015) 169–180.

[12] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al., Mastering the game of go with deep neural networks and tree search, Nature 529 (7587) (2016) 484–489.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, Human-level control through deep reinforcement learning, Nature 518 (7540) (2015) 529.

[14] S. Mozer, M. C, M. Hasselmo, Reinforcement learning: An introduction, Machine Learning 8 (3-4) (1992) 225–227.

[15] A. Razzaq, M. S. Rathore, An approach towards resource efficient virtual network embedding, in: International Conference on Evolving Internet, 2010, pp. 68–73.

[16] L. Page, The pagerank citation ranking : Bringing order to the web, Stanford Digital Libraries Working Paper 9 (1) (1998) 1–14.

[17] I. Houidi, W. Louati, D. Zeghlache, A distributed virtual network mapping algorithm, in: IEEE International Conference on Communications, 2008, pp. 5634–5640.

[18] J. M. H. Elmirghani, L. Nonde, T. E. H. Elgorashi, Energy efficient virtual network embedding for cloud networks, Journal of Lightwave Technology 33 (9) (2015) 1828–1849.

[19] M. Melo, S. Sargento, U. Killat, A. Timm-Giel, J. Carapinha, Optimal virtual network embedding: Energy aware formulation, Computer Networks 91 (C) (2015) 184–195.

[20] S. Haeri, L. Trajkovic, Virtual network embedding via monte carlo tree search., IEEE Transactions on Cybernetics (99) (2017) 1–12.

[21] R. Mijumbi, J. L. Gorricho, J. Serrat, M. Claeys, F. D. Turck, S. Latre, Design and evaluation of learning algorithms for dynamic resource management in virtual networks, in: Network Operations and Management Symposium, 2014, pp. 1–9.

[22] R. Mijumbi, J. L. Gorricho, J. Serrat, M. Shen, K. Xu, K. Yang, A neuro-fuzzy approach to self-management of virtual network resources, Expert Systems with Applications An International Journal 42 (3) (2015) 1376–1390.

[23] R. Mijumbi, J. L. Gorricho, J. Serrat, M. Claeys, J. Famaey, F. D. Turck, Neural network-based autonomous allocation of resources in virtual networks, in: European Conference on Networks and Communications, 2014, pp. 1–6.

[24] S. Hougardy, The floydcwarshall algorithm on graphs with negative cycles, Information Processing Letters 110 (8) (2010) 279–281.

[25] M. Thomas, E. W. Zegura, Generation and analysis of random graphs to model internetworks, College of Computing volume 63 (4) (1994) 413–442(30).

[26] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al., Tensorflow: Large-scale machine learning on heterogeneous distributed systems, arXiv preprint arXiv:1603.04467.

[27] L. Bottou, Online algorithms and stochastic approximations, in: D. Saad (Ed.), Online Learning and Neural Networks, Cambridge University Press, Cambridge, UK, 1998, revised, oct 2012.
URL `http://leon.bottou.org/papers/bottou-98x`