Topology-Awareness and Reoptimization Mechanism for Virtual Network Embedding

Nabeel Farooq Butt[1], Mosharaf Chowdhury[2], and Raouf Boutaba[1,3]

[1] David R. Cheriton School of Computer Science,
University of Waterloo,
Waterloo, ON N2L 3G1, Canada
nfbutt@uwaterloo.ca, rboutaba@uwaterloo.ca
[2] Computer Science Division,
University of California, Berkeley,
Berkeley, CA 94720, USA
mosharaf@cs.berkeley.edu
[3] Division of IT Convergence Engineering, POSTECH,
Pohang, KB 790-784, Korea

Abstract. Embedding of virtual network (VN) requests on top of a shared physical network poses an intriguing combination of theoretical and practical challenges. Two major problems with the state-of-the-art VN embedding algorithms are their indifference to the underlying substrate topology and their lack of reoptimization mechanisms for already embedded VN requests. We argue that topology-aware embedding together with reoptimization mechanisms can ameliorate the performance of the previous VN embedding algorithms in terms of acceptance ratio and load balancing. The major contributions of this paper are twofold: (1) we present a mechanism to differentiate among resources based on their importance in the substrate topology, and (2) we propose a set of algorithms for reoptimizing and re-embedding initially-rejected VN requests after fixing their bottleneck requirements. Through extensive simulations, we show that not only our techniques improve the acceptance ratio, but they also provide the added benefit of balancing load better than the previous proposals.

Keywords: Topology-awareness, Re-embedding, Virtual Network Embedding, Network Virtualization.

1 Introduction

Network virtualization is widely considered to be a potential candidate for providing the essential basis for future Internet architectures [1, 2, 3, 4]. One major challenge in network virtualization environment (NVE) is the allocation of substrate network resources, managed by infrastructure providers (InPs), to online VN requests by different service providers (SPs) [3, 4]. Unfortunately, the VN embedding problem reduces to the multi-way separator problem, which has been shown to be NP-hard [5]. From the point of view of an InP, a preferable allocation scheme should increase long-term revenue while reducing the cost of hosting

individual VNs. Consequently, mechanisms that can increase acceptance ratio are of great interest to InPs, because acceptance ratio is directly proportional to their revenue.

While embedding a VN request, existing proposals [6, 7, 8, 9, 10] do not distinguish between different substrate nodes and links. However, in practice, some substrate nodes and links are more *critical* than others. For example, resource depletion in bridges and articulation points in the substrate topology are expected to have deeper impact on future embeddings than a random resource near the edge of the network. Choosing amongst feasible VN embeddings, the one that uses fewer critical resources can be a key to improving acceptance ratio in the long run.

In this paper, we investigate how differentiating between substrate resources can increase the acceptance ratios of the existing VN embedding algorithms [6, 7, 8, 9, 10]. For differentiating resources, we design a mechanism for assigning weights to substrate nodes and links based on their residual capacities and their importance in the substrate topology. These weights help us in prioritizing certain VN embeddings over others. Other than links' or nodes' resource depletion being able to partition the network, attributes of the created partitions are also an important factor in weight assignment. In case of a substrate network partitioning, any virtual link from a VN request has to be turned down if its two ends can only be mapped in different partitions. We argue that distinguishing substrate resources is a key for better load balancing and for ameliorating the acceptance ratio.

Over time, as new VNs are embedded and old ones expire, resources in the substrate network become fragmented. The obvious consequence is the rejection of many VN requests lowering the acceptance ratio and revenue. This could be amended had the fragmented resources been consolidated using reoptimization techniques. We propose algorithms for identification and re-embedding of VNs that cause VN request rejection. Our reoptimization mechanism consists of two stages: detection of bottleneck links and nodes, followed by relocation of virtual links and nodes to free up resources on bottlenecks.

The remainder of this paper is organized as follows. Section 2 summarizes background and related work. Our proposals for distinguishing and reoptimizing critical resources are discussed in Section 3. Section 4 presents performance metrics, our experimental setup, and their results. We conclude in Section 5.

2   Background and Related Work

The intra-domain VN embedding problem refers to the mapping of VN requests from different SPs on top of substrate network resources managed by an InP. A substrate network in this problem formulation is defined as an undirected graph with substrate nodes and links with CPU and bandwidth capacities, respectively. Similarly, a VN request is defined as an undirected graph with virtual nodes and links with CPU and bandwidth requirements and additional constraints.

The long-term objective of a VN embedding algorithm is to maximize the revenue of an InP while minimizing its cost of embedding online VN requests. Any virtual node can be embedded onto at most substrate node, whereas a virtual link can refer to a physical path in the substrate network. Further details on the basic VN embedding problem formulation can be found in [7, 8].

Allocating resources for Virtual Private Networks [11, 12] is the earliest, but simpler, incarnation of the VN embedding problem. It is different in its consideration of only the bandwidth constraints on virtual links. Since constraints on virtual nodes are ignored, this problem often simplifies to selecting paths that satisfy the given constraints.

VN embedding proposals in the existing literature address the problem by simplifying constraints in different dimensions. Szeto et al. [13] have presented algorithms for embedding only the virtual links, with an assumption that the virtual nodes have already been mapped. Off-line version of the VN embedding problem had been the focus of some of the existing research [6, 9]. Precluding admission control complexities by assuming infinite capacities of substrate nodes and links have been addressed in [6, 9, 14]. Lu et al. [9] have developed algorithms for specific topologies. Yu et al. [7] have provided a two stage algorithm for mapping the VNs. They start off by mapping the virtual nodes first and then proceed to map the virtual links using shortest paths and multi-commodity flow (MCF) algorithms. Chowdhury et al. [8] have recently presented a solution to the VN embedding problem by coordinating the node and the link mapping phases. by using mixed integer programming formulation. Backtracking algorithms in conjunction with graph isomorphism detection to map VN requests are used in a dynamic fashion in [15]. Distributed algorithms for simultaneously embedding virtual links and virtual nodes [10] have shown issues with scalability and poorer performance than their centralized counterparts.

Although all these approaches strive for improving VN embedding by various degrees, none of them foresee the possibility of narrowing down the chances for accepting future VN requests. They treat all the substrate nodes and links equally, which is not practical. Their indifference to the underlying topology, allows uninformed decisions while mapping resources on 'critical' substrate links and nodes that can affect the acceptance ratio in the future. Another potential candidate for improving the accepting ratio is the effective use of reoptimization mechanisms. By detecting and relocating bottleneck link and node embeddings to better alternates, we can achieve improvements. Ideally, we want to keep all the VNs optimized at all times (e.g., through periodic updating [6]). But this will result in relocation of a lot of virtual links and virtual nodes and incur significant overhead. A better way to deal with this problem is to take action only when it is inevitable [7]. However, unlike [7], we propose algorithms for relocation of virtual nodes as well as links.

Before proceeding to the next section, we want to mention that we use the same formulation for revenue as in the previous literature [6, 7, 8], but in upcoming sections we will provide a slightly better cost function.

3   Improving Acceptance Ratio and Balancing Load

Indifference to the attributes of the substrate network resources and lack of re-optimization mechanisms are the two major candidates for improvement in the existing VN embedding algorithms [6, 7, 8, 9, 15]. In the following sections, we present techniques to incorporate topology-awareness and reoptimization mechanisms for bottleneck embeddings in these algorithms to improve their overall performance.

3.1   Topology-Aware Embedding

We differentiate between substrate network resources by introducing scaling factors to their costs. The *scaling factor (SF)* of a resource refers to its likelihood of becoming a bottleneck. It is calculated as a combination of two weight factors, critical index and popularity index, explained later in this section.

We aim to incorporate topology-awareness in existing algorithms with minimal changes. In our solution, proposals using MCF only need to update their objective functions (constraints require no change) and proposals using shortest path algorithms should select paths with minimum total scaled costs to upgrade themselves.

Critical Index. The first weight function we define is the *critical index (CI)* of a substrate resource. The CI of a resource measures the likelihood of a residual substrate network partition[1] due to its unavailability. To understand the impact of such partitioning, consider the following scenario: suppose that a substrate network is partitioned into two - almost equal-sized - components. The probability of both the end nodes of any new virtual link to be embedded in different partitions is 0.5. In this case, we might have to reject almost 50% of the VN requests.

We denote CI by ζ ($\zeta : x \rightarrow [0,1)$), where x is either a substrate link or a node. Higher value of $\zeta(x)$ means its highly likely that unavailability of x will partition the substrate graph and vice versa. The mathematical definition of CI is given in equations 1 and 2 for links and nodes respectively.

$$\zeta(e_s) = \begin{cases} \left(\dfrac{1 - |c_1 - c_2|}{2}\right) + \dfrac{1}{2} & e_s \in \text{cut-edges} \\[2ex] \dfrac{\phi + \psi}{4} & e_s \notin \text{cut-edges} \end{cases} \tag{1}$$

$$\zeta(n_s) = \begin{cases} \dfrac{1}{2}\left(\prod_{c \in C} P\{c\}\right)^{\frac{1}{|C|}} + \dfrac{1}{2} & n_s \in \text{cut-vertices} \\[2ex] \dfrac{\phi + \psi}{4} & n_s \notin \text{cut-vertices} \end{cases} \tag{2}$$

---

[1] We refer to residual substrate network while discussing partitioning in this paper. A residual substrate network is composed of residual capacities of substrate resources.

In equation 1, if e_s is a bridge (cut-edge) then removing it partitions the graph. c_1 and c_2 are the fractions of substrate nodes in each of these partitions. When e_s is not a bridge, then after removing it we get ϕ and ψ, where ϕ and ψ are the fractions of new cut-nodes or cut-edges to the total cut-nodes or cut-edges respectively. In equation 2, C is the set of components we get when we remove n_s. $P\{c\}$ here is the fraction of nodes present in c. In both equations 1 and 2, we make sure that a cut-resource has its CI value ≥ 0.5. Computing ζ is required only once in the beginning and it is updated whenever a substrate network is extended. Also, it is not computationally intensive.

Popularity Index. Resource saturation is the other major factor that can cause substrate network partition. *Popularity Index (PI)* measures "how many different VNs are affected when a link or a node is unavailable?" To be more specific, PI ($\rho : x \rightarrow [0, 1)$) is the time weighted average of occupied resources on a particular link or a node. It also takes into account the number of VNs that are mapped on that resource. In equation 3, $R_{E_{i-1}}$ is the percent of reserved bandwidth on link x, where the index $i - 1$ means the previous value of R_E; similarly, $R_{N_{i-1}}$ is the percent of reserved CPU capacity at the node x. The variables a and b ($a + b = 1$) are used to assign different weights to current and previous values. In equation 3, ν is the number of VNs mapped on top of x. The higher the value of ρ the higher the probability that mapping onto this link or node will saturate it and create a bottleneck.

$$\rho(x) = \begin{cases} \left(aR_{E_{i-1}} + bR_{E_i}\right)^{\frac{1}{\nu}} & x \in E_s \\[2ex] \left(aR_{N_{i-1}} + bR_{N_i}\right)^{\frac{1}{\nu}} & x \in N_s \end{cases} \tag{3}$$

Scaling Factor. Both CI and PI values are very crucial when mapping a particular link or node. Together they are referred to as the scaling factor and denoted by $\aleph$. In equation 4, we have multiplied both CI and PI by α and β respectively. We also use ω, which is a parameter and can be used to further tune the cost. The objective functions of the LP formulations of previous proposals [8] can be updated by scaling the cost of resources by $\aleph$.

$$\aleph(x) = 1 + \omega\big(\alpha\zeta(x) + \beta\rho(x)\big) \tag{4}$$

We use a slightly different cost function, $C(G_V)$, than that of the previous literature. In the modified formula, we are scaling up the cost by the scaling factor to avoid embedding onto critical resources whenever possible. In the formula below, n_s is the substrate node on which a virtual node n_v is mapped, $f_{e_s}^{e_V}$ is the flow on substrate link e_s for the virtual link e_v, and $c(n_v)$ is the cost of the virtual node n_v.

$$C(G_V) = \sum_{e_V \in E_V} \sum_{e_S \in E_S} \aleph(e_s) f_{e_s}^{e_V} + \sum_{n_V \in N_V} \aleph(n_s) c(n_V) \tag{5}$$

3.2   Reoptimizing Bottleneck Embeddings

Although efficient algorithms exist for embedding of VNs onto a substrate net-
work, arbitrary lifetimes of VNs can cause some embeddings to drift toward
inefficiency over time. Such embeddings can be reoptimized by relocating vir-
tual nodes and reassigning virtual links to better alternate embeddings. From
a practical point of view, it has been shown how to migrate virtual nodes (vir-
tual routers) from one physical node to another [16]. Link migration can also be
achieved by dynamically setting up and tearing down physical links; it is already
achieved using programmable transport networks [17].

Our proposed solution can be divided into two stages. First, we detect the
virtual links and nodes that caused a VN request to be rejected. Second, we re-
locate/reassign these nodes/links to less critical regions of the substrate network.
We illustrate the basic idea behind our relocation and reassignment mechanism
using the example in Fig. 1 and Fig. 2. In Fig. 1 (a), VN request VN-6 is being
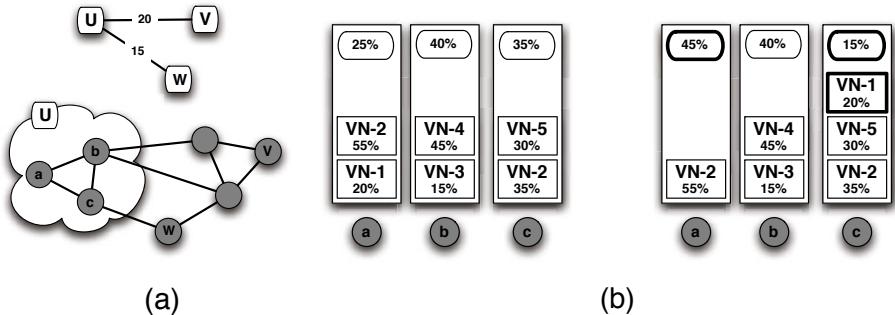rejected because one of its virtual nodes cannot be mapped, i.e., node U which

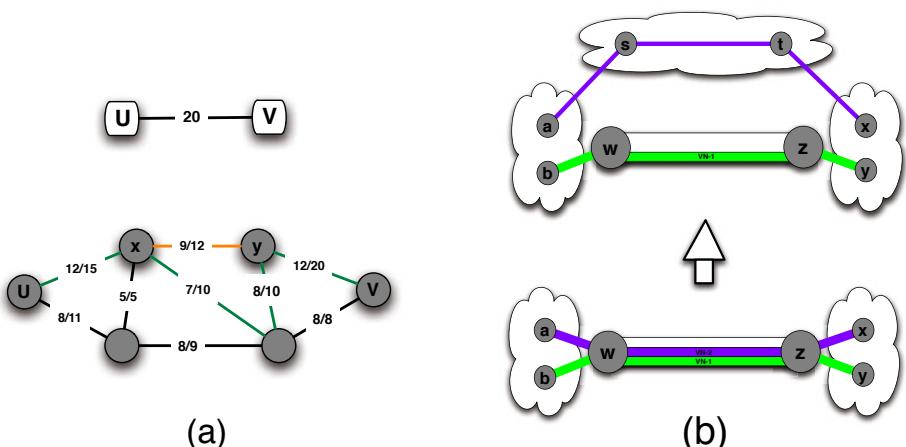**Fig. 1.** Relocating a virtual node of VN-1 to make room for virtual node U

Fig. 2. Reassigning links to make room for virtual link

Algorithm 1. Detecting V-Nodes that cannot be mapped

1: $N_{bad} \leftarrow$ virtual nodes that cannot be mapped
2: **for all** virtual nodes $v_n \in N_{bad}$ **do**
3:     $C_{N_S}(v_n) \leftarrow$ candidate nodes
4:     **for all** substrate node $n \in C_{N_S}(v_n)$ **do**
5:         $L_{low} \leftarrow$ All virtual nodes mapped on n that have lower priority than VN_{rej}
6:         **if** capacity(L_{low}) > required capacity **then**
7:             Save (n, L_{low}), for relocating later
8:         **end if**
9:     **end for**
10:    fails if v_n cannot be fixed
11: **end for**

Algorithm 2. Detecting V-Links that cannot be mapped

1: $L_{bad} \leftarrow$ All virtual links v such that MaxFlow(v) < ReqCap(v)
2: **for all** virtual links $v \in L_{bad}$ **do**
3:     **repeat**
4:         **for all** substrate link $e \in MinCut(v)$ **do**
5:             $L_{low} \leftarrow$ All virtual links mapped on e that have lower priority than VN_{rej}
6:             Save (e, L_{low})
7:         **end for**
8:     **until** $Capacity(MinCut(v)) < ReqCapacity(v)$
9:     fails if v cannot be fixed
10: **end for**

---

requires 43% of CPU capacity. There are three possible embeddings for this node (i.e., nodes a, b, and c), but none with the required capacity for U. In Fig. 1 (b), relocating a VN-1 node from node a to node c makes enough room to accommodate U. All the associated virtual links of VN-1 are also reassigned to the new location. Now suppose that we want to embed a virtual link U-V. We can see that it cannot be mapped because of the bottleneck substrate link w-z (Fig. 2 (a)). As shown in Fig. 2 (b), VN-2 is reassigned to free up some bandwidth on the substrate link w-z so that the virtual link U-V can be accommodated.

Detecting Bottleneck Nodes and Links. For bottleneck detection, we deal with VN requests that cannot be embedded due to lack of residual capacities in substrate resources. Contrary to the periodic reconfiguration scheme presented in [6], we have taken a reactive approach in detecting bottlenecks – we reoptimize whenever a VN request gets rejected.

Algorithm 1, finds a list of virtual nodes that cannot be mapped, and for each one of them it stores corresponding bottleneck substrate nodes. After detecting the candidate substrate nodes for an unmapped virtual node, it figures out whether enough capacity can be salvaged on any of the candidate nodes or actually on L_{low}. L_{low} is the set of virtual nodes of other VN requests with less priority than the rejected VN request. If a single virtual node cannot be mapped

---

Algorithm 3. Relocating a Virtual Node

1: Q_{v_n} = v-nodes that are adjacent to v_n
2: C_{v_n} = candidate s-nodes for v_n
3: status = **false**
4: **for all** substrate node $n \in C_{v_n}$ **do**
5:     status = **true**
6:     **for all** virtual link $v \in Q_{v_n}$ **do**
7:         Map-Link($Substrate(v), v_n$)
8:         **if** map fails **then**
9:             status = **false**
10:             break
11:         **end if**
12:     **end for**
13:     **if** status **then**
14:         Relocate v_n to n
15:     **end if**
16: **end for**

---

we do not proceed any further. Finally, L_{low} and associated substrate node n are stored for later use in the relocation phase. The worst case running time of algorithm 1 is $O(n^2)$, but on the average it would be much faster than that.

Next, we use algorithm 2 to determine the unmapped virtual links, L_{bad}, and corresponding bottleneck links. A virtual link is rejected if there is not enough capacity between its two end nodes. The maximum flow from the source to the sink[2] and the minimum s-t cut can be computed in $O(V^3)$ using the Edmond-Karp maximum flow algorithm [18]. Algorithm 2 iterates through L_{bad} and for every virtual link it finds the minimum cut[3]. Re-assigning some virtual links on the minimum cut can increase the maximum flow between source and sink. Note that even if we made enough room in the first minimum cut it might still not guarantee the required flow between the source and the sink. Since the minimum cut can change over time, we have to iteratively compute the minimum cut. To keep the running time of algorithm 2 small, we repeat this only a constant number of times. Next we re-assign links in L_{low} to free up some capacity on these links. It is useless to proceed if we fail to map only a single virtual link; in this case, we can just reject this request immediately.

Nodes and Links Selection and Placement. Algorithm 3, relocates a virtual node v_n which is currently mapped on a substrate node s_n to some other substrate node, provided all the virtual links incident on v_n can also be reassigned. Q_{v_n} is a set of virtual nodes adjacent to v_n and C_{v_n} is a list of substrate nodes which are potential hosts for v_n. Note that C_{v_n} is sorted according to the overhead cost. The algorithm iterates on C_{v_n} and for every substrate node n in C_{v_n}, it checks whether it can map all the virtual links after v_n is relocated to n.

---

[2] Source and sink are the end nodes of the rejected virtual link.
[3] For information on Max-Flow Min-Cut theorem, readers are referred to [18].

Algorithm 4. Reassigning a Virtual Link

1: $R_E(e_S) = 0$
2: (u, v) be the virtual nodes of e_V
3: $c = MaxFlow(u, v)$
4: **if** $c = 0$ **then**
5:      **return false**
6: **end if**
7: **if** $ReqCap(e_V) <= c$ **then**
8:      Re-embed virtual link e_V
9: **else**
10:     Free c amount of flow from e_S
11:     Augment c amount of flow elsewhere
12: **end if**
13: Restore and Adjust $R_E(e_S)$
14: **return true**

If it can map all these links then v_n can be relocated to the substrate node used to map these links; finally, it relocates v_n.

For reassigning a virtual link, algorithm 4 takes the virtual link e_V which is mapped to substrate link e_S. First it sets the available bandwidth on e_S to zero, so that the freed capacity on e_S is not allocated to e_V, when e_V is reassigned. It then finds the maximum flow between the end nodes of e_V and if its zero than nothing can be done. If the maximum flow is greater than the required bandwidth capacity of the link e_V then it just free the old mapping of e_V and remaps it again. If the maximum flow is less than the required capacity then we can free c amounts of capacity from e_S and map c units of capacity elsewhere by using the computed flow. Finally, we restore and adjust the freed bandwidth capacity on e_S (which was set to zero at the beginning). These link mappings should be done using topology-aware LP formulation, to keep the overhead cost of reassigning e_V to minimum.

4   Evaluation

Our evaluation focuses primarily on quantifying the effectiveness of our techniques when applied to the VN embedding algorithms in the existing literature. We consider the following four algorithms in our evaluation: D-ViNE and R-ViNE [8] map nodes deterministically and randomly, respectively; in both the algorithms, links are embedded using a modified MCF. G-SP [6] and G-MCF [7] both use greedy node mapping; for link embedding, the former uses shortest path algorithm, whereas the latter uses MCF.

4.1   Experimental Setup and Performance Metrics

We have extended the simulator of [8] to include our proposed techniques and algorithms. We have used GT-ITM [19] for generating topologies for the underlying substrate networks. Substrate networks in our experiments have 100 nodes
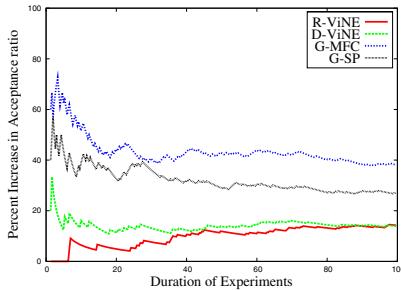
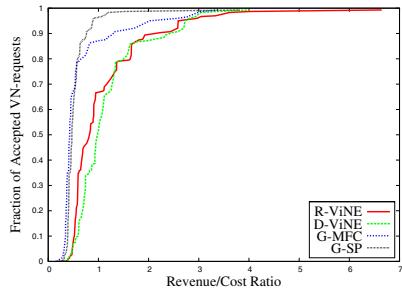**Fig. 3.** Improvement in Acceptance Ratios for Different Algorithms

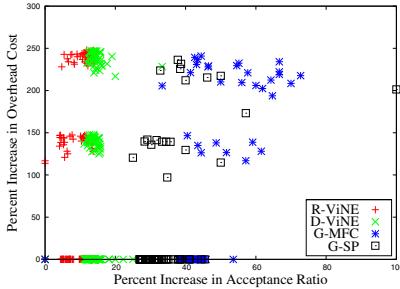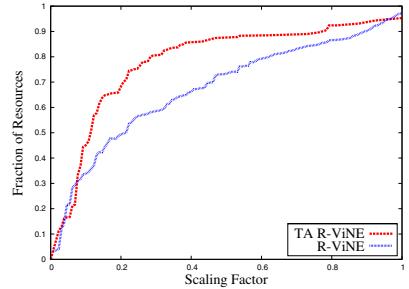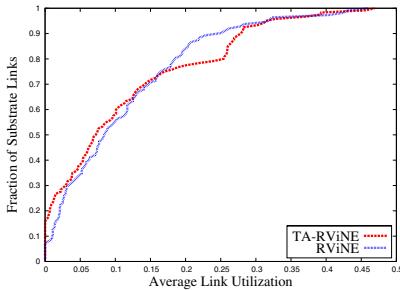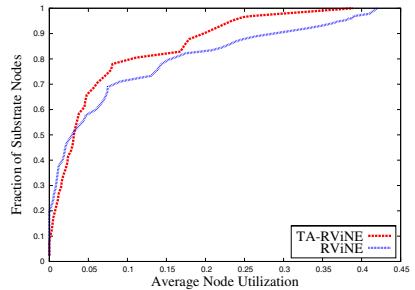**Fig. 4.** Revenue-Cost Ratio for Different Algorithms

and around 400 links, on the average. Node CPU capacities and link bandwidth capacities are randomly chosen between 50 to 100. For VN requests, we used a similar setup as previously used in [6, 7, 8]. VN requests arrive in a Poisson process with an inter-arrival time of 20 time units. Lifetimes of the VN requests follow an exponential distribution with mean 1000 time units. Virtual nodes for VN requests are chosen uniformly between 5 to 10. CPU and bandwidth requirements are distributed uniformly between 0 to 20 and 0 to 50 respectively. The metrics we use to measure performance are increase in acceptance ratio, revenue-cost ratio, incurred cost, and distribution of utilization of resources.

4.2 Evaluation Results

Improvement in Acceptance Ratio. In our first set of experiments, we compare the increase in acceptance ratios of all the algorithms. As shown in Fig. 3, in steady state, our mechanisms improve acceptance ratios by almost 40% for G-MCF and just below 35% for G-SP. For D-ViNE and R-ViNE, the improvements are smaller but still a sizeable 17%. We believe that the lower increase for D-ViNE and R-ViNE is due to the fact that they already have much higher acceptance ratio than G-SP and G-MCF without the proposed improvements [8].

Revenue/Cost Ratio. We present the per request revenue/cost ratio for the compared algorithms in Fig. 4. We see that quite a few readings are close to zero which indicates that those VN requests are being mapped to one or more very critical links or nodes. On the average, the revenue/cost ratio of D-ViNE and R-ViNE is slightly above 2 for 90% of the accepted VN requests, which is significantly higher than that of G-SP and G-MCF. This shows that our techniques not only improve the acceptance ratios (Fig. 3), but they also keep the revenue/cost ratio within acceptable range (specially for D-ViNE and R-ViNE).

Acceptance Ratio vs Incurred Cost. An important factor in evaluating the effectiveness of our techniques is the cost incurred by increasing the acceptance

Fig. 5. Acceptance Ratio vs Incurred Cost

Fig. 6. CDF of SFs of all links and nodes

Fig. 7. Average link utilization

Fig. 8. Average node utilization

ratio. In Fig. 5, we have plotted the percentage increase in acceptance ratio against the percentage increase in cost for making room for these VN requests. Here the incurred cost is the sum of the cost of moving already embedded VN requests and accepting VN requests rejected earlier. Fig. 5 shows that the maximum increase in cost is close to 250%, but the range of acceptance ratio improvement varies from $7 - 77\%$ for various algorithms. For all the algorithms there are several readings with very low or even zero incurred cost. This is because the VNs required to move to make room for rejected VN requests have a lot of alternative embeddings of equal cost and incurred no additional cost.

Differentiating Resources. In Fig. 6, we have plotted the CDF of the scaling factors of all links and nodes for R-ViNE [8] and R-ViNE incorporated with topology-aware embedding (hereinafter referred to as "TA-R-ViNE"). The other algorithms, compared with their extended versions, also show similar characteristics. SF is a good measure because it gives an indication of potential bottlenecks; an SF value close to 1 means that a resource is more likely to become a bottleneck. We can see here that TA-R-ViNE has significantly fewer resources being indicated as bottlenecks than R-ViNE. With TA-R-ViNE, only 15% resources have SF values above 0.4; whereas with R-ViNE, it is almost 35%. This graph shows that topology-awareness plays a significant role in identifying bottlenecks.

Link and Node Utilization. A comparison between link utilization of TA-R-ViNE and R-ViNE is shown in Fig. 7. It presents an average of multiple runs of carefully designed experiments that made sure that both techniques could map the same VN requests. We can see that for R-ViNE, almost 90% of the links are utilizing less than 25% of bandwidth, whereas for TA-R-ViNE it is almost 80%. An explanation would be that these links are alternative links to bottlenecks; hence they are used more often. We can also see that TA-R-ViNE used almost 8% fewer substrate links than R-ViNE. In Fig. 8, almost 50% of the nodes have less utilization than 50% of the nodes mapped by TA-R-ViNE, but the other 50% have much higher utilization. This figure shows that to avoid bottlenecks TA-R-ViNE has distributed load among nodes.

5   Conclusion

In this paper, we have presented mechanisms for differentiating various resources (links and nodes) according to their impact on the connectivity of the substrate network, and their utilization coupled with the number of VNs utilizing them. We have also proposed algorithms for reoptimizing the already embedded VN requests in order to improve the overall acceptance ratio. Through simulation we have shown that our techniques significantly improve the performance of the existing algorithms in terms of increase in acceptance ratio, revenue-cost ratio, cost incurred by these techniques and load balancing based on criticalness of resources. Experimental results endorse the fact that our techniques have enough potential to be seriously considered as an integral part of any future VN embedding algorithm design.

Acknowledgment

References

1. Anderson, T., Peterson, L., Shenker, S., Turner, J.: Overcoming the internet impasse through virtualization. Computer 38(4), 34–41 (2005)
2. Turner, J., Taylor, D.: Diversifying the internet. In: Global Telecommunications Conference, 2005. GLOBECOM 2005, November-2 December, vol. 2, p. 6. IEEE, Los Alamitos (2005)
3. Feamster, N., Gao, L., Rexford, J.: How to lease the Internet in your spare time. ACM SIGCOMM Computer Communication Review 37(1), 61–64 (2007)
4. Chowdhury, N.M.M.K., Boutaba, R.: A survey of network virtualization. Computer Networks (2010) (in press), `http://dx.doi.org/10.1016/j.comnet.2009.10.017`

5. Andersen, D.: Theoretical approaches to node assignment.,
   `http://www.cs.cmu.edu/~dga/papers/andersen-assign.ps`
6. Zhu, Y., Ammar, M.: Algorithms for assigning substrate network resources to virtual network components. In: IEEE INFOCOM, April 2006, pp. 1–12 (2006)
7. Yu, M., et al.: Rethinking virtual network embedding: substrate support for path splitting and migration. SIGCOMM CCR 38(2), 17–29 (2008)
8. Chowdhury, N.M.M.K., Rahman, M.R., Boutaba, R.: Virtual network embedding with coordinated node and link mapping. In: IEEE INFOCOM (2009)
9. Lu, J., Turner, J.: Efficient mapping of virtual networks onto a shared substrate. Washington University, Tech. Rep. WUCSE-2006-35 (2006)
10. Houidi, I., Louati, W., Zeghlache, D.: A distributed virtual network mapping algorithm. In: IEEE ICC, pp. 5634–5640 (2008)
11. Gupta, A., et al.: Provisioning a virtual private network: A network design problem for multicommodity flow. In: ACM STOC, pp. 389–398 (2001)
12. Ricci, R., Alfeld, C., Lepreau, J.: A solver for the network testbed mapping problem. ACM SIGCOMM CCR 33(2), 65–81 (2003)
13. Szeto, W., Iraqi, Y., Boutaba, R.: A multi-commodity flow based approach to virtual network resource allocation. In: IEEE GLOBECOM, pp. 3004–3008 (2003)
14. Fan, J., Ammar, M.: Dynamic topology configuration in service overlay networks: A study of reconfiguration policies. In: IEEE INFOCOM (2006)
15. Lischka, J., Karl, H.: A virtual network mapping algorithm based on subgraph isomorphism detection. In: ACM SIGCOMM VISA Workshop, pp. 81–88 (2009)
16. Wang, Y., et al.: Virtual routers on the move: Live router migration as a network-management primitive. In: ACM SIGCOMM, pp. 231–242 (2008)
17. Agrawal, M., et al.: Routerfarm: towards a dynamic, manageable network edge. In: ACM SIGCOMM INM Workshop, pp. 5–10 (2006)
18. Diestel, R.: Graph theory. Springer, New York (1997)
19. Zegura, E.: How to model an Internet. In: IEEE INFOCOM, pp. 594–602 (1996)