A Research on Algorithms for Vitual Network Embedding

Shubin Wu
Computer Science Department
Georgia State University
Atlanta, GA, United States
Swu17@student.gsu.edu

Abstract—**Network virtualization is recognized as an enabling technology for the future Internet. It aims to overcome the resistance of the current Internet to architectural change. Application of this technology relies on algorithms that can instantiate virtualized networks on a substrate infrastructure, optimizing the layout for service-relevant metrics. This class of algorithms is commonly known as "Virtual Network Embedding (VNE)" algorithms. In my project, my work is to do a survey of current research in the VNE algorithms and implement two of the algorithms into JAVA code and do some optimization. The Algorithms are called BFSN VNE Algorithms based on the Breadth-First Searching Tree technique and embed virtual networks on best-fit sub-substrate networks.**

Keywords—Virtual Network Embedding, Best-fit Sub-substrate Networks, Breadth-First-Searching Tree

I. Introduction

Network virtualization[4] is one of the most important features of the cloud computing, which allows creation of multiple virtual networks (VNs) on a single substrate network (SN) (Fajjari et al, 2013). In cloud computing, virtual network requests (VNRs) arrive dynamically over time with different topologies, resource requirements, and lifetime. Cloud providers accommodate VNRs by mapping each virtual node to a substrate node and each virtual link to a substrate path, such that a set of previously defined mapping constraints (e.g. topology constraints, link bandwidth, CPU capacity) is satisfied (Lischka & Karl, 2009). At the end of the VNR's lifetime, allocated SN's resources are released.

However, mapping virtual networks' resources to substrate network's resources is known to be nondeterministic polynomial-time hard (NP-hard) (Chowdhury et al, 2009; Chowdhury et al, 2012). This problem[5] is usually referred to as the Virtual Network Embedding (VNE) problem (Till Beck et al, 2013). In the last few years, many studies have proposed several algorithms for efficient and effective VNE (Lischka & Karl, 2009; Chowdhury et al, 2009; Chowdhury et al, 2012; Cheng et al, 2011; Zhang et al, 2013; Cheng et al, 2012). Effective VNE increases the utilization of the SN's resources and increases the revenues of the cloud computing datacenters. However, the process of allocating and releasing SN's resources fragments SN's resources (Fischer et al, 2013). SN's resources are considered fragmented if there are two or more sub-substrate networks connected by substrate paths with available bandwidths less than the minimum required VNR's bandwidth or substrate paths with lengths greater than the maximum allowed hops.

Fragmentation of the SN's resources reduces the acceptance ratio and reduces the long-term revenue. Substrate resources fragmentation can be improved by re-embedding one or more VNRs (Fischer et al, 2013). However, re-embedding running VNRs may reduce the quality of service provided to the customers and may violate the service level agreement (SLA).

Most of current researches ignore the SN's resources fragmentation problem (Chowdhury et al, 2009; Cheng et al, 2011; Zhang et al, 2013), while others propose reactive algorithms (Zhu & Ammar, 2006; Fajjari et al, 2011). Reactive algorithms just act when a VNR is rejected, and try to re-embed one or more VNRs to increase the acceptance ratio. Some of the current researches consider load balance of the SN's resources to avoid SN's resources fragmentation (Cheng et al, 2012; Sun et al, 2013). However, considering load balance of the SN's resources during embedding VNRs with small resources may cause rejecting VNRs with large resources in the future.

In our work, we looked into two VNE algorithms which aiming at improving the acceptability of fragmented substrate networks. The proposed algorithms increase the possibility of accepting future VNRs by embedding VNRs on best-fit sub-substrate networks[6]. The proposed algorithms are one stage (embed virtual nodes and virtual links at the same stage to allow coordination between them), online (deal with VNRs that arrive over time and do not require VNRs to be previously known), and backtracking algorithms. And then I implemented the VNE algorithms proposed by the authors but changed some functions to optimize the efficiency and the accuracy.

The algorithms exploit the virtualization feature of the cloud computing to embed more than one virtual node from the same VN on the same substrate node, which minimizes the cost of embedding VNRs by eliminating the cost of embedding virtual links between those virtual nodes.

The performance of the proposed algorithms have been evaluated using extensive simulations in JAVA, which show that the proposed algorithms outperforms some of the existing embedding algorithms.

The rest of this project report is organized as follows. In Section 2, we discuss the related work on general VNE problem. Section 3 presents the VN embedding model and problem formulation. Section 4 describes the proposed algorithms. In section 5 are the implemented JAVA code and results of the evaluation for the proposed VN embedding algorithms

.

II. BACKGROUND AND RELATED WORK

Many researches have been done for more efficient Virtual Network Embedding processThe authors in [1]–[2] try to minimize the resource allocation on a shared substrate network (SN) in an offline manner. A multicast mapping algorithm named MMPC is proposed in [1] to improve the efficiency of physical resource utilization. Multicast Virtual Network Embedding with strategies of waiting tolerant and load prediction (MVNEWL) is proposed for multicast services with delay constraints in [2]. In [3], end-delay and delay variation constraints are considered in the 3-step heuristic approach with Breadth- First Search (BFS) and Depth-First Search (DFS). In [13], multicast-aware graph embedding using the BFS-based static priority list is proposed.

Zhu and Ammar (Zhu & Ammar, 2006) proposed two VN embedding algorithms. The first algorithm is static VNE algorithm, where allocated substrate resources are fixed throughout the VN lifetime. Heuristics and adaptive optimization strategies are used to improve the performance of the proposed algorithm. The second algorithm reconfigures the embedded VNs to increase the utilization of the underlying substrate resources. However, the proposed algorithms deal only with offline embedding problem (all VNRs are previously known). On the other hand, in cloud computing data centers, VNE problem is online problem, where new VNRs arrive over time.

Lischka and Karl (Lischka & Karl, 2009) presented online backtracking VNE algorithm. The proposed algorithm is one stage VNE algorithm (maps nodes and links during the same stage) and maps VN to a sub-physical network that is similar to the topology of the VN and achieves previously defined constraints (e.g. CPU capacity, link bandwidth). Nodes are mapped to substrate nodes sequentially after sorting it in descending order based on its required CPU. The proposed algorithm tries to map virtual links to substrate paths with minimal hops by incrementally increasing the maximum hop limit. However, the computational complexity is high due to multiple operations of the proposed algorithm. Di et al. (Di et al, 2010) improved performance and complexity of the proposed algorithm in (Lischka & Karl, 2009) by considering link mapping cost during the process of sorting nodes and choosing the maximal hop limit. Nogueira et al. (Nogueira et al, 2011) proposed heuristic-based VN embedding algorithm to deal with the heterogeneity of VNs and SN, in both links and nodes. The proposed algorithm is one stage VNE algorithm.

Cheng et al. (Cheng et al, 2011) proposed two online virtual network-embedding algorithms called RWMaxMatch and RW-BFS. Both of them rank nodes using topology-aware node ranking technique to reflect the topological structure of the VNs and the SN. RWMaxMatch algorithm is two stage embedding algorithm, which performs node mappings and link mappings at two different stages without coordination between them. However, mapping nodes without considering its relation to the link mapping might result in neighboring virtual nodes being widely separated in the SN, which leads to high consumption of the underlying SN's resources and reduces the acceptance ratio of the proposed algorithm. To avoid this problem, they proposed RW-BFS algorithm. RW-BFS algorithm is a backtracking one stage embedding algorithm, which embeds nodes and links at the same stage.

To improve the coordination between nodes mapping stage and links mapping stage, Chowdhury et al. (Chowdhury et al, 2009; Chowdhury et al, 2012) formulated the VNE problem as a mixed integer program (MIP), which is NP-hard. To obtain polynomial-time solvable algorithms, they relaxed the integer program to linear program, and proposed two VNE algorithms: D-VNE (deterministic VNE algorithm) and R-VNE (randomized VNE algorithm).

Zhang et al. (Zhang et al, 2013) proposed two VN embedding models: an integer linear programming model and a mixed integer programming model. To solve these models, the authors proposed VN embedding algorithm based on particle swarm optimization. The time complexity of the link mapping stage is reduced by using shortest path algorithm and greedy k-shortest paths algorithm.

Some of existing works proposed VN embedding algorithms to embed VNRs in distributed cloud computing environments (Samuel et al, 2013; Houidi et al, 2008a; Xin et al, 2011; Lv et al, 2010). Houidi et al. (Houidi et al, 2011) proposed exact and heuristics VN embedding algorithms, which split virtual network requests using max-flow min-cut algorithms and linear programming techniques. Leivadeas et al. (Leivadeas et al, 2013) proposed VN embedding algorithm based on linear programming. The proposed algorithm partitions VNRs using partitioning approach based on Iterated Local Search. Houidi et al. (Houidi et al, 2008b) proposed distributed VN embedding algorithm, which is performed by agent-based substrate nodes. The authors proposed VN embedding protocol to allow communication between the agent-based substrate nodes. However, the proposed algorithm assumes that all VNRs are previously known.

III. PROBLEM DEFINITION

Substrate network (SN): We model the substrate network as a weighted undirected graph = (N_S, L_S), where is the set of substrate nodes and is the set of substrate links. Each substrate node $n_S \in N_S$ is weighted by the CPU capacity, and each substrate link $l_S \in L_S$ is weighted by the bandwidth capacity. Fig.1 shows a simple SN example, where the available CPU resources are represented by numbers in rectangles and the available bandwidths are represented by numbers over the links.

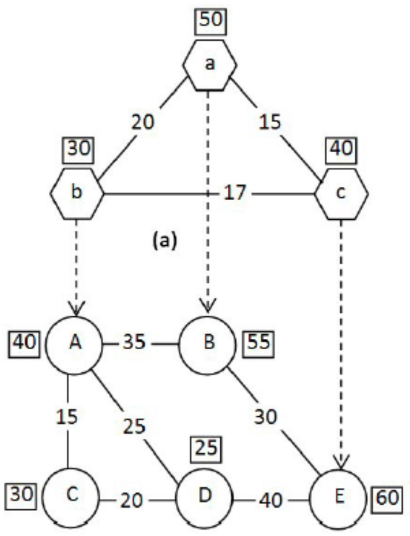

Fig 1: Example of VNE

Virtual network (VN): virtual network is modeled as a weighted undirected graphG_{vi}=(N_{vi},L_{vi}) , where is the set of virtual nodes and is the set of virtual links. Virtual nodes and virtual links are weighted by the required CPU and bandwidth, respectively. Fig.1 shows an example of VN with required CPU and bandwidth.

Virtual network requests (VNR): the i^{th} VN request in the set of all VN requests is modeled as (G_{vi},t_{ai}, t_{li}) where is the required VN to be embedded, $t_{ai}$is the arrival time, and t_{li} is the lifetime. When vnr_i arrives, substrate nodes' CPU and substrate links' bandwidth are allocated to achieve the vnr_i. If the substrate network does not have enough resources to achieve vnr_i, vnr_i is rejected. At the end of lifetime, all allocated resources to vnr_i are released.

Virtual Network Embedding (VNE): embedding VN_i on SN is defined as a map M : G_{vi}→(N_s', P'_s), where $N'_s \subseteq N_s$, and $P'_s \subseteq P_s$, where P_s is the set of all loop free substrate paths in G_s. Embedding can be decomposed into node and link mapping as follows:

Node mapping: M_N: $N_{vi} \rightarrow N'_s$

Link mapping: M_L: $L_{vi} \rightarrow P'_s$

For example, mapping of the virtual network in fig.1 on SN in fig.1 as well can be decomposed into:

Node mapping {(a, b)->{B, A}, (b, c)->{(A, D), (D, E)}, (c, a)->{(E, B)}}

Virtual Network Embedding Revenue: the revenue of embedding the revenue of embedding at time is defined as the sum of all required substrate CPU and substrate bandwidth by vnr_i at time t.

$$\text{R}(\,vnr_i,\text{t}) = \text{Life}(\,vnr_i,\text{t})(\sum n_{vi\in Nv_i} CPU(n_{vi}) + \sum l_{vi\in lv_i} BW(l_{vi}))$$

Where $CPU_{(n_{vi})}$ is the required CPU for the virtual node n_{vi}, BW(l_{vi}) is the required bandwidth for the virtual link, and Life (vnr_i, t) =1 if vnr_i is in its lifetime and substrate resources are allocated to it, otherwise Life(vnr_i, t) =0.

Virtual Network Embedding Cost: the cost of embedding vnr_i at time t is defined as the sum of all allocated substrate CPU and substrate bandwidth to vnr_i at time t.

$$\text{Cost}(\,vnr_i,\text{t}) = \text{Life}(\,vnr_i,\text{t})(\sum n_{vi\in Nv_i} CPU(n_{vi}) + \sum l_{vi\in lv_i} BW(l_{vi}) * Length(M_{lvi}(l_{vi})))$$

Where $Length(M_{lvi}(l_{vi})$ is the length of the substrate path that the virtual link l_{vi} is mapped to l_{vi}.

Objectives: the main objectives are to increase the revenue and decrease the cost of embedding virtual networks in the long run. To evaluate the achievement of these objectives, we use the following metrics:

-The long-term average revenue, which is defined by

$$lim_{T-\infty}(\frac{\sum_{t=0}^{T}\sum_{i=1}^{I} R(vnr_i,\text{t})}{T})$$

Where I=||VNR||, and T is the total time.

- The VNR acceptance ratio, which is defined by

$$\frac{||\text{VNRs}||}{||\text{VNR}||}$$

Where VNRs is the set of all accepted virtual network requests.

- The long term R/Cost ratio, which is defined by

$$lim_{T-\infty}(\frac{\sum_{t=0}^{T}\sum_{i=1}^{I} R(vnr_i,\text{t})}{\sum_{t=0}^{T}\sum_{i=1}^{I} Cost(vnr_i,\text{t})})$$

IV. Proposed Algorithms

We proposed two VNE algorithms to increase the acceptance ratio for fragmented SNs and minimize the remapping process. In the next sub-sections, we describe the motivation behind the proposed algorithms and describe the details of the proposed algorithms.

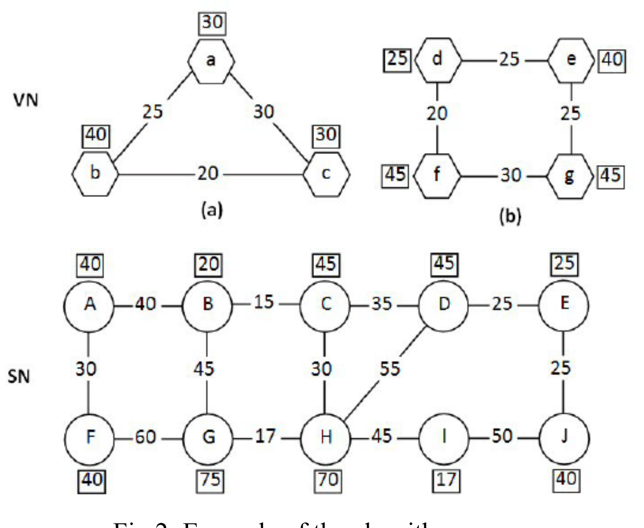

Fig 2: Example of the algorithms

4.1 An existing problem

For each VNR, we may have different VNE solutions. For example, table 1 shows some VNE solutions to embed VN in figure 2(a) on the SN in figure 2(c). Although, solution 2 consumes less SN's resources among others, it reduces the possibility to accept more future VN requests, such as VN in figure 2(b). Mapping VN in figure 2(b) on the SN in figure 2(c) requires remapping the VN in figure 2(a) to solution1. Then, the VN in figure 2(b) can be mapped as following:

Node mapping:{d->E, e-J, f->D, g->H}

Link mapping{(d,e)->{(E, J)}, (e, g)->{(J, I), (I, H)}, (g, f)->{(H, D)}, (f, d)->{(D, E)}}

From this example, we conclude that increasing the possibility to accept more future VN requests does not only depend on the mapping cost. This conclusion motivates us to propose VNE algorithms for selecting VNE solutions that increase the possibility to accept as many VNRs as possible. Even if the selected solutions maybe increase the embedding cost, the revenue is increased by accommodating more VNRs.

Table 1: Possible solutions to map fig2(a) to fig2(c)

	Node mapping	Link mapping
Solution 1	{a->A,b->F,c->G}	{(a, b)->{(A, F)}, (b, c)->{(F, G)}, (c, a)->{(G, B), (B, A)}}
Solution 2	{a->C,b->H,c->D}	{(a, b)->{(C, H)}, (b, c)->{(H, D)}, (c, a)->{(D, C)}}
Solution 3	{a->D,b->J,c->H}	{(a, b)->{(D, E), (E, J)}, (b, c)->{(J, I), (I, H)}, (c, a)->{(H, D)}}

4.2 BFSN Algorithm

The description for the algorithms are in the table 2 below.

Table 2: BFSN algorithms

Algorithm 1 The Details of the BFSN VNE algorithm
1: Build breadth-first searching tree of G_v from virtual node with largest resources. 2: Sort all nodes in each level in the created breadth-first tree in descending order according to their required resources. 3: Build candidate sub-substrate networks G_{sub} 4: for each $G_{subi} \in G_{sub}$ do 5: backtrack count=0 6: if Embed $(G_{vroot}, G_{subi}, M_{(G_v)})$ then 7: return true 8: end for 9: return false
Algorithm 1 The Details of the Embed function in the algorithm above
1: Build candidate substrate node list n_{vi} to C_i 2:for each n_s in C_i 3: Add((n_{vi}, n_s), $M_{(G_v)}$) 4: if Embed((n_{vi+1} , nG_{subi}), $M_{(G_v)}$) 5: else 6: Delete((n_{vi}, n_s), $M_{(G_v)}$) 7: if backtrack count > Max_backtrack then return false 8: end for 9: backtrack count ++ 10: return false

The table just shows the steps the algorithms take. The algorithm constructs breadth-first searching tree of G_v. The root node of the constructed tree is the virtual node with the largest resources (sum of CPU and BW). Nodes in each level in the created breadth-first searching tree are sorted in descending order based on their resources.

The set of candidate sub-substrate networks G_{sub} is built. Candidate sub-substrate networks are specified by visiting substrate nodes in N_s' sequentially and creating a breadth-first searching tree from a substrate node if it has not been included in any sub-substrate network yet. Substrate node is added to the sub-substrate network if it has not been included in any sub-substrate network, and has been connected with previously added substrate node through a substrate path in P_s' . G_{sub} is reduced by removing sub-substrate networks that do not have enough resources to embed G_{vi} . All remaining sub-substrate networks in G_{sub} are sorted in ascending order based on their total available resources.

The BFSN algorithm embeds VN on a sub-SN using Embed() function, which is described in figure 6. In the Embed() function, candidate substrate node list for each virtual node is built by collecting all substrate nodes that have available CPU capacity at least as large as the virtual node CPU and have a loop free substrate path to each substrate node contains one of the previously mapped neighbors. Each substrate path should satisfy the constraint of the maximum substrate path length, and have available bandwidth greater than or equal the bandwidth of the virtual link between the virtual node and its previously mapped neighbor.

Candidate substrate nodes for each virtual node are collected by creating a breadth-first search tree from each substrate node contains one of the previously mapped neighbors, and finding the common substrate nodes between the created trees. In the constructed trees, substrate nodes should satisfy the CPU constraints for virtual node, and substrate paths should satisfy the connectivity constraints to connect the virtual node with its neighbors. By this way, all candidate substrate nodes in the list satisfy all constraints (CPU and connectivity constraints).

Substrate nodes in the candidate substrate node list are sorted in ascending order according to the total cost of embedding virtual links from the virtual node to all previously embedded neighbors. If the virtual node is a root node, the candidate substrate node list is a set of all substrate nodes that have enough resources to embed the virtual node. The candidate substrate nodes for the root are sorted in descending order according to the total available resources.

Virtual node is sequentially mapped to substrate nodes in its candidate substrate node list. If there is no appropriate substrate node in its candidate substrate node list, we backtrack to the previously mapped node, re-map it to the next candidate substrate node, and continue to the next node. Mappings of the virtual node and its virtual links are added to$M((G_v))$ by using the function Add() in line 3. In line 6, Delete() function is used to perform the backtracking process.

4.3 Example

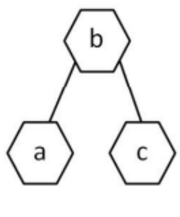

Fig 3: The sorted B-F-S tree for VN

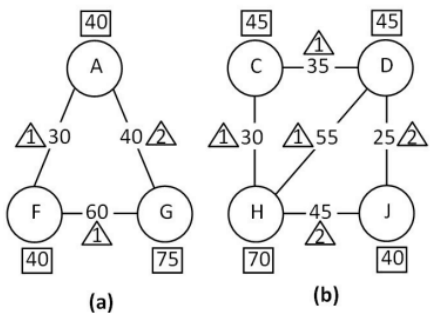

Fig 4: The sub-substrate candidates for VNE

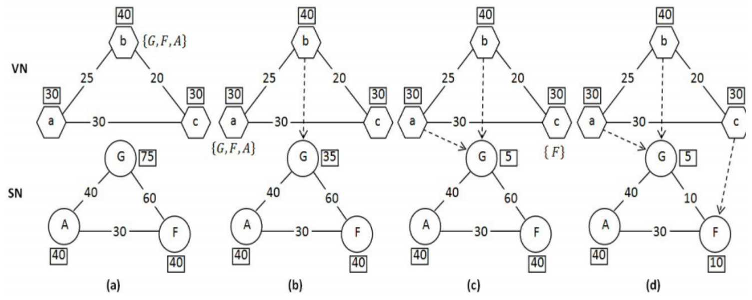

Fig 5: The process of Node mapping

For example, to map VN in figure 2 (a) on the SN in figure 2 (c), we construct a breadth-first searching tree from the virtual node b, which is the virtual node with largest resources. Virtual nodes in each level are sorted in descending order based on their resources as in figure 3. Figure 4 shows candidate sub-substrate networks that are specified, and sorted in ascending order based on its total available resources. Numbers in triangles represent numbers of hops for links. Figure 5 shows the embedding steps of the VN in figure 2 (a) on the first candidate sub-SN, which is the candidate sub-SN in figure 4 (a). The candidate substrate nodes list for the root node b is built as {G, F, A}. The root node b is mapped to the substrate node G as shown in figure 5(b). The next virtual node in the breadth first search tree in figure 3 is the virtual node a. The set of candidate substrate nodes for the virtual node a is specified as {G, F, A}, which is sorted in ascending order based on the cost of mapping the virtual node a to each of them. G substrate node is included because the proposed algorithm allows substrate nodes to include more than one virtual node from the same VN. Figure 5(c) shows the SN after mapping the virtual node a to the substrate node G. Finally, the virtual node c is mapped to the substrate node F, which is the only substrate node that can satisfy the connectivity constraints to the substrate node G (the total required bandwidth is 50).

V. Codes and Results

The Java code we implemented is shown as follows:

```
package computer network project;
```

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class NodeMapGreedy {
    private static String virtual;
    private static String substrate;
    private static double totalDistance = 0;
    private static List<String> multicastTree;

    private static double totalPathDistance = 0;

    public NodeMapGreedy(String sFile, String vFile){
        this.substrate = sFile;
        this.virtual = vFile;
    }

    public boolean mapping(List<Vertex> substrate, List<Vertex>
virtual) {

        Vertex[] mappedNodes = new Vertex[substrate.size()];
        showLists(substrate, virtual);

        mappedNodes = nodeMapping(substrate, virtual,
mappedNodes);

        if (mappedNodes==null) {
            return false;
        }

        List<Vertex> totalPath = linkMapping(substrate, virtual,
mappedNodes);

        if (totalPath.isEmpty()) {
            return false;
        }

        totalPath = updatePath(totalPath, substrate);



System.out.println("===========================================
===");
        System.out.println("Total path:" +
Arrays.toString(totalPath.toArray()));
        System.out.println("Total transmission distance:" +
totalPathDistance);
        totalPathDistance = updateCost(totalPath);
        System.out.println("Total transmission distance After
link update:" + totalPathDistance);

        return true;

    }

    private double updateCost(List<Vertex> totalPath) {

        double weight = 0;

        for (int i = 0; i < totalPath.size() - 1; i += 2) {
            List<Edge> links = new ArrayList<Edge>();
            links = totalPath.get(i).adjacencies;
            for (Edge edge : links) {
                if (edge.target.name == totalPath.get(i +
1).name) {
                    weight += edge.weight;
                }
//                                      System.out.println("Total
weight="+weight);
            }
//                            System.out.println("Total
weight="+weight);
        }



        return weight;
    }

    private List<Vertex> updatePath(List<Vertex> totalPath,
List<Vertex> substrate) {

        List<String> link = new ArrayList<String>();
        for (int i = 0; i < totalPath.size() - 1; i++) {
            if (totalPath.get(i) != null && totalPath.get(i +
1) != null) {
                link.add(totalPath.get(i) + "-" +
totalPath.get(i + 1));
            }
        }

        for (int i = 0; i < link.size() - 1; i++) {
            if (link.get(i).equalsIgnoreCase(reverse(link.get(i
+ 1)))) {
                link.remove(i);
                i--;
            }
        }

        //
System.out.println("Links:"+Arrays.toString(link.toArray()));
        multicastTree = link;
        List<Vertex> tempNodeList = new ArrayList<Vertex>();
        for (String string : link) {


tempNodeList.add(substrate.get(Integer.valueOf(string.substring(
0, 1))));

tempNodeList.add(substrate.get(Integer.valueOf(string.substring(
string.length() - 1, string.length()))));
        }

        // System.out.println("Last node
        // list:"+Arrays.toString(tempNodeList.toArray()));

        return tempNodeList;
    }

    private String reverse(String string) {
        String temp = "";
        for (int i = 0; i < string.length(); i++) {
            temp += string.charAt(string.length() - 1 - i);
        }
        // System.out.println("Coming String:"+string);
        // System.out.println("Reverse of it:"+temp);
        return temp;
    }

    private Vertex[] nodeMapping(List<Vertex> substrate,
List<Vertex> virtual, Vertex[] mappedNodes) {

        while (virtual.size() != 0) {

            Vertex node = virtual.get(0);
            CandidateSet cSet = new CandidateSet();
```

```
//            List<Vertex> candidateNodes =
CandidateSet.getPossibleNodes(CandidateSet.getCandidateList(subs
trate, node),node);
            List<Vertex> candidateNodes =
cSet.getPossibleNodes(cSet.getCandidateList(substrate,
node),node);
            node = findBestNode(candidateNodes, mappedNodes);
            System.out.println("Virtual node " + virtual.get(0)
+ " is mapped onto Substrate Node:" + node);
            if (node != null) {
                mappedNodes[node.name] = virtual.get(0);
            } else {
                System.out.println("There is no available node
to map virtual node " + virtual.get(0));
                return null;
            }
            System.out.println("Mapped Nodes:" +
Arrays.toString(mappedNodes));
            virtual.remove(0);
        }

        return mappedNodes;
    }

    private List<Vertex> linkMapping(List<Vertex> substrate,
List<Vertex> virtual, Vertex[] mapped) {

        List<Vertex> totalPath = new ArrayList<Vertex>();
        List<Vertex> tempPath = new ArrayList<Vertex>();
        List<Vertex> mappedNodes = new ArrayList<Vertex>();
        List<Vertex> IntermediateNodes = new
ArrayList<Vertex>();

        for (int i = 0; i < mapped.length; i++) {
            if (mapped[i] != null) {
                mappedNodes.add(substrate.get(i));
            }
        }


        System.out.println("Mapped substrate nodes:" +
Arrays.toString(mappedNodes.toArray()));

        IntermediateNodes.addAll(mappedNodes);

        System.out.println("Intermediate Nodes:" +
Arrays.toString(IntermediateNodes.toArray()));

        for (int i = 0; i < mappedNodes.size(); i++) {

            int maxBW =
findMaxBW(mapped[mappedNodes.get(i).name]);
            Vertex node = mappedNodes.get(i);
            tempPath = findShortestPath(node, IntermediateNodes,
maxBW);
//                          tempPath = findShortestPath(node,
mappedNodes, maxBW);

            if (tempPath.isEmpty()) {
                return null;
            }
            totalPath.addAll(tempPath);
            for (Vertex V : tempPath) {
                int found = 0;
                for (Vertex I : IntermediateNodes) {
                    if (I.name == V.name) {
                        found = 1;
                        break;
                    }
                }

                if (found == 0) {
                    System.out.println("Adding New Intermediate
Vertex" + V.name);
                    IntermediateNodes.add(V);
                }
            }
            System.out.println("Updated Intermediate Nodes " +
Arrays.toString(IntermediateNodes.toArray()));
            System.out.println("Total Path:" +
Arrays.toString(totalPath.toArray()));
            totalPath.add(null);
        }

        return totalPath;
    }

    private Vertex findBestNode(List<Vertex> candidateNodes,
Vertex[] mapped) {

        Vertex[] allNodes = new Vertex[candidateNodes.size()];

        for (int i = 0; i < candidateNodes.size(); i++) {
            allNodes[i] = candidateNodes.get(i);
        }

        Sort.bubbleSort(allNodes);

        System.out.println("All nodes were ordered by
CPU+BW+Degree");
        System.out.println(Arrays.toString(allNodes));

        for (int i = 0; i < allNodes.length; i++) {
            System.out.println("Node " + allNodes[i]);
            System.out.println("Mapped[" + allNodes[i].name +
"]=" + mapped[allNodes[i].name]);

            if (mapped[allNodes[i].name] == null) {
                return allNodes[i];
            }

        }

        return null;
    }

    private void showTotalPath(List<String> totalPath) {

        for (String path : totalPath) {
            System.out.println(path);
        }

    }

    private List<Vertex> findShortestPath(Vertex closestVertex,
List<Vertex> mappedList, int maxBW) {

        List<Vertex> path = new ArrayList<Vertex>();
        double minDistance = Double.POSITIVE_INFINITY;
        List<Vertex> savedPath = new ArrayList<Vertex>();

        for (Vertex vertex : mappedList) {

            if (closestVertex.name != vertex.name) {

                ShortestPath pathObj = new
ShortestPath(substrate);
                path =
pathObj.defineShortestPath(closestVertex.name, vertex.name,
maxBW);
                System.out.println("Path:" +
Arrays.toString(path.toArray()));
                System.out.println("Path from " + closestVertex
+ " to " + vertex + " with BW=" + maxBW);

                double distance = pathObj.totalDistance(path);
                System.out.println("Distance:" + distance);
                if (distance < minDistance) {
                    minDistance = distance;
                    savedPath = path;
                    System.out.println("Saved Path:==>" +
Arrays.toString(path.toArray()));
                    System.out.println("Minimum Distance:" +
minDistance);
                }
            } // end-if
        } // end-for mappedList

        totalPathDistance += minDistance;

        return savedPath;
    }

    private void showLists(List<Vertex> substrate, List<Vertex>
virtual) {
        System.out.println("Substrate Vertex List:" +
Arrays.toString(substrate.toArray()));
        System.out.println("Virtual Vertex List:" +
Arrays.toString(virtual.toArray()));
    }

    private List<Vertex> checkCandidateSet(List<Vertex> list,
List<Vertex> mappedList, Boolean[] mapped) {

        List<Vertex> updateList = new ArrayList<Vertex>();

        for (Vertex vertex : list) {
            Boolean status = false;

            for (int i = 0; i < mapped.length; i++) {
                if (mapped[i] != null && vertex.name == i) {
                    status = true;
                }
            }

            if (!status) {
                updateList.add(vertex);
            }

        }

        return updateList;
    }

    private int findMaxBW(Vertex vertex) {

        int max = Integer.MIN_VALUE;
        for (Edge link : vertex.adjacencies) {

            System.out
                    .println("Edge from " + vertex.toString() +
" to " + link.target.toString() + "=" + link.bandWidth);
            if (link.bandWidth >= max) {
                max = link.bandWidth;
            }

        }

        return max;
    }

    public boolean map(int source, int[] dest) {

        multicastTree = new ArrayList<String>();
        GenerateVertex gVertex = new GenerateVertex();
        List<Vertex> substrate =
gVertex.setVertices(this.substrate);
        List<Vertex> virtual =
gVertex.setVertices(this.virtual);

        List<Vertex> mappedList = new ArrayList<Vertex>();
        List<Vertex> totalPath = new ArrayList<Vertex>();
        Vertex[] mappedNodes = new Vertex[substrate.size()];
        System.out.println("Virtual List:" +
Arrays.toString(virtual.toArray()));

        List<Vertex> updatedList = new ArrayList<Vertex>();
        Vertex[] virtualNodes = new Vertex[virtual.size()];

        for (int i = 0; i < virtual.size(); i++) {
            virtualNodes[i] = virtual.get(i);
        }

        System.out.println("Virtual Node Array:" +
Arrays.toString(virtualNodes));

        Sort.bubbleSort(virtualNodes);

        System.out.println("Sorted Virtual Node Array:" +
Arrays.toString(virtualNodes));

        for (Vertex vertex : virtualNodes) {
            updatedList.add(vertex);
        }

        System.out.println("Virtual Priority List:" +
Arrays.toString(updatedList.toArray()));

        return mapping(substrate, updatedList);

    }

    public double getTotalDistance() {
        double distance = totalPathDistance;
        totalPathDistance = 0;
        return distance;
    }

    public List<String> getMulticastTree() {
        return multicastTree;
    }
}
```

5.1 data structure we use

In the initialization part, what we did is to define the data structure of each node. For the substrate node, it contains CPU capacity. And, the virtual node contains the CPU demand. The connectivity. bandwidth and distance are shown in the adjacent matrix. We use adjacent matrix (AM[][]) to represent a connected graph. The size of the matrix depends on the number of nodes in the graph. A graph within N nodes, the size of the matrix is N * N. If node m and node n is connected, the value of AM[m][n] and AM[n][m] becomes 1. As for tree, we only take the upper side which does not contain itself. For each column, there exists one 1 to make sure there is no hoop in the tree and the connectivity of the tree. The bandwidth demand/capacity and distance are also matrixes which transform from the connectivity adjacent matrix.

As for building the substrate candidates part, what we did is to remove the substrate link which does not satisfy the bandwidth demand of the virtual network. In the coding part, we just need to check the bandwidth matrix and delete the links which are less than the requirement. And then, we need to check connectivity of the connectivity matrix when deleting those links.

The implementation of candidate set part is to compare the CPU demand of the virtual node with existing substrate nodes. If the substrate node's CPU capacity is lager or equal to the demand. It is one candidate of that virtual node. Then adding that substrate node into the candidate list.

In the Choosing Physical node part, what we did is to call the distance matrix to calculate each CCR of all candidates of that most important node for the distance matrix also shows the connectivity.

After one loop, decreasing one from the size of virtual node continues a new loop till the size of virtual node becomes 0. When finishing the mapping part, the next step is to construct connections based on virtual topology using the shortest path in the physical network. When all connections are built, the left work is to calculate the total distance spread of the virtual network and the cost of finishing data transmitting. Then, the whole work has done.

5.2 data and comparison

In the simulation and experimentation, our algorithm is compared to Greedy Node Mapping as well as First Fit Node Mapping.

In random generated graph, 30 nodes in substrate network graph with 90 links that are randomly connected. Each node in the graph are property-specific in line with our algorithm and performance can be obtained in guided simulation in current test benches. Properties introduce by our algorithm are random generated in a range in each virtual node. Specifically, computing Resource are generated in the range of [5-40] cost units. Bandwidth capacity of each substrate link are generated in the range of [5-35] cost units. Large number of virtual multicast tree requests with [3 - 10]

VN nodes are generated. Computing resource in each request of every virtual node is randomly assigned in the range of [10 - 30] cost units. And bandwidth demand of the virtual multicast request is in the range of [5 - 25] cost units

For the testing platform, we simulate our algorithm using Java. And for each series of experiments, we vary the CPU and Bandwidth based on different VN request to obtain the average results. In the first set of simulations, by giving requested bandwidth demands from 5 to 25 with the maximum CPU as 15, we represent the total bandwidth usage, number of hops and number of link duplication. In the second set of simulations, we show the total bandwidth usage, number of hops while varying the number of virtual nodes from 3 to 10 with bandwidth demand as 20. The results are shown in the Fig. 6.

In the final result, our proposed algorithm outperforms GNM and FF algorithms at least 50% when increasing Bandwidth through 15. The virtual multicast request onto SN with less bandwidth usage is further verified by the smaller number of hops. When Bandwidth is bigger than 15, the performance of our algorithm is as much as three times better than GNM and FF, as shown on the right graph.

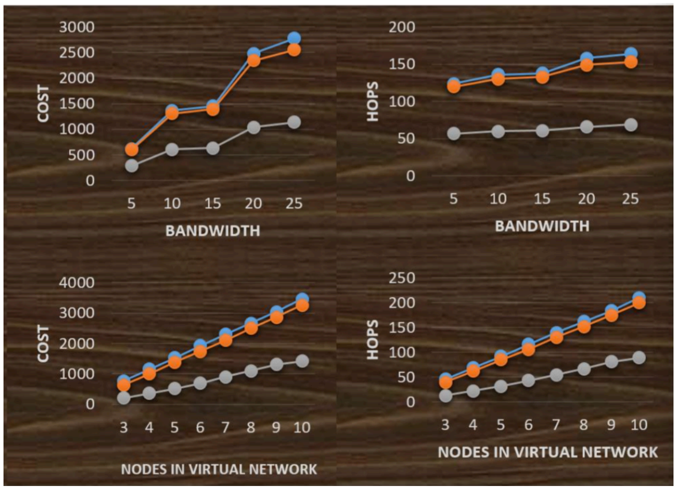

Fig 6: the result & comparisons for the algorithm

VI. References

[1] Miao, Yuting, et al. "Multicast virtual network mapping for supporting multiple description coding-based video applications." *Computer Networks* 57.4 (2013): 990-1002..

[2] Liao, Dan, et al. "Opportunistic provisioning for multicast virtual network requests." *2014 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 2014.

[3] Ayoubi, Sara, Khaled Shaban, and Chadi Assi. "Multicast Virtual Network Embedding in Cloud Data Centers with Delay Constraints." *2014 IEEE 7th International Conference on Cloud Computing*. IEEE, 2014..

[4] Chowdhury, NM Mosharaf Kabir, Muntasir Raihan Rahman, and Raouf Boutaba. "Virtual network embedding with coordinated node and link mapping." *INFOCOM 2009, IEEE*. IEEE, 2009.R. Nicole, “Title of paper with only first word capitalized,” J. Name Stand. Abbrev., in press.

[5] Cheng, Xiang, et al. "Virtual network embedding through topology-aware node ranking." *ACM SIGCOMM Computer Communication Review* 41.2 (2011): 38-47.

[6] Abascal, Federico, Rafael Zardoya, and David Posada. "ProtTest: selection of best-fit models of protein evolution." *Bioinformatics* 21.9 (2005): 2104-2105.