JOINT VIRTUAL NETWORK FUNCTION PLACEMENT AND MIGRATION IN DYNAMIC CLOUD DATA CENTERS

A Project

Presented

to the Faculty of

California State University Dominguez Hills

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Jingsong Sun

Summer 2020

PROJECT: JOINT VIRTUAL NETWORK FUNCTION PLACEMENT AND MIGRATION IN DYNAMIC CLOUD DATA CENTERS

AUTHOR: JINGSONG SUN

APPROVED:

_____

Bin Tang, Ph.D

Project Committee Chair

_____

Liudong Zuo, Ph.D

Committee Member

_____

Brad Hollister, Ph.D

Committee Member

ACKNOWLEDGEMENTS

Project was conceived by Dr. Tang in 2019 fall semester. After 2 semesters' discussing and modifying, the implementation of the project finally shows beautiful algorithm and simulation results. I offer many thanks:

to Dr. Bin Tang of my project adviser, who offers excellent ideas about the further progress of my project and helpful feedback. Thanks for the encouragement and help when I feel discouraged by the stuck in implementation of my project. I learned a lot of research methods and project practical knowledge from him.

to Dr. Liudong Zuo and Dr. Brad Hollister, who attends my master defense and gives very helpful feedback about my project report.

to my wife who supports me to change my major, it is hard for me and her.

to my friends who studied with me during my master studies, who help me broaden my knowledge. Without their help, it is hard for me to finish my master studies.

Jingsong Sun

Summer 2020

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

We propose a new virtualization network function (VNF) optimization framework for policy-preserving data centers (PPDCs). In PPDCs, virtual machine (VM) traffic must traverse a sequence of VNFs for security and performance purposes, generating more network traffic and consuming more network resources compared to traditional cloud data centers. Our framework tackles this problem and achieves optimal or near optimal resource utilization for a PPDC's lifetime, while tackling diverse and dynamic VM traffic commonly existing in cloud data centers. It first places VNFs into PPDCs according to the diverse traffic rates of communicating VM pairs (i.e., VNF placement) and then adaptively migrate VNFs inside PPDCs in response of changing traffic rates of the VM pairs (i.e., VNF migration), with the goal of minimizing the VM communication cost and VNF migration cost. We formulate both problems and design optimal, approximation, and heuristic *policy-preserving* VNF placement and migration algorithms. Underlying VNF placement and migration problems are two new graph theoretical problems that have not been studied before. We show that VNF migration is an effective technique to tackle dynamic traffic in PPDCs by outperforming the existing VM migration traffic-mitigation techniques by three times, and our VNF placement algorithms outperform existing techniques by more than two times.

1. INTRODUCTION

Background and motivation. Network Function Virtualization (NFV) is an effective technique to reduce capital and operating expenses and to achieve flexible management in cloud computing environment [37]. With NFV, proprietary hardware middle-boxes(MBs) such as firewalls, intrusion detection and prevention systems (IDPSs), and load balancers can now be implemented as virtual network functions (VNFs) running as lightweight container on commodity hardware [17]. Being instantiated and deployed dynamically in cloud data centers, VNFs provide performance and security guarantees to cloud user applications in a flexible and cost-effective manner. In particular, *service function chains* (SFCs) (or *data center policies*) are established that require virtual machine (VM) application traffic to traverse a sequence of VNFs to achieve aforesaid guarantees [23], [24]. We refer to the cloud data centers that implement and enforce such data center policies as *policy-preserving data centers* (PPDCs). We use SFCs and data center policies interchangeably.

Fig. 1(a) shows a linear PPDC with two physical machines (PMs): pm_1 and pm_2 and five switches: sw_1, sw_2, ..., sw_5. There are two communicating VM pairs: (v_1, v_1') and (v_2, v_2'), with v_1 and v_1' stored at pm_1 and v_2 and v_2' at pm_2. It also shows an SFC consisting of an IDPS, denoted as vnf_1, and a cache proxy, denoted as vnf_2. As the VM traffic between v_1 and v_1' (and v_2 and v_2') must traverse vnf_1 and vnf_2 in that order, this policy first filters out malicious traffic and then caches the content to share

with other cloud users, improving both security and performance of the cloud user applications.

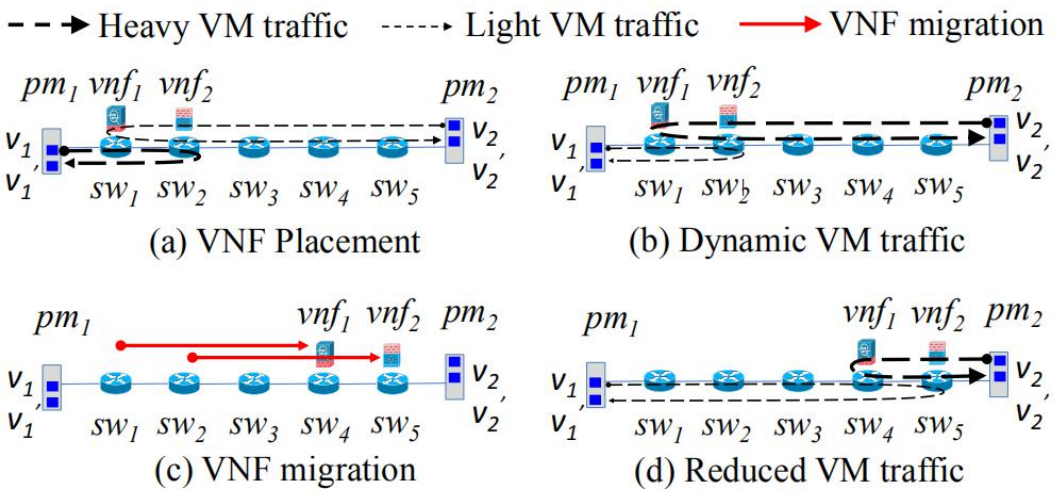


Fig. 1. An illustrating example for VNF placement and migration in PPDC.

VNF placement [29], [30], which allocates VNFs into PPDCs, and VNF migration [46], which moves VNFs around inside PPDCs, are two effective techniques achieving a variety of objectives in PPDCs including resource optimization, load balancing, and fault tolerance. Although they have attracted much attention, there are two limitations of the current research. First, VNF placement and migration are mostly studied in parallel without considering the effect of one upon the other [29], [30], [46]. This may attribute to different metrics and objectives that each targets. While VNF placement mainly aims to minimize the setup cost of VNFs [15], [41], [13], [7], or to minimize the communication cost of VMs such as energy, delays, or distances [48], [6], [10], [33], VNF migration is mainly to minimize its negative influences such as service down time [45], overall

migration time [26] and QoS degradation [18]. As achieving one goal often compromises the other, there is a need for joint optimization of VNF placement and migration to achieve overall optimal resource utilization in cloud data centers. Achieving optimal resource utilization is especially critical for PPDCs wherein VM traffic must traverse a sequence of VNFs, causing additional communication delays, generating more network traffic and consuming more network resource (e.g., energy and bandwidth) compared to traditional cloud data centers.

Second, recent findings and measurements from Facebook and other production data centers show that traffic loads (i.e., transmission rates) of VM applications are highly diverse and dynamic [9], [40]. One recent example is Zoom cloud video conferencing [3], where the Zoom Meeting Connector VMs [4] could support a few participants at one instance and hundreds of them (up to 350) at another, while time of each communication could vary greatly and traffic could vary from video, voice to data. Dynamic network traffic, if not dealt with well, could further exacerbate the network traffic in PPDCs.

In this report we propose a new framework that integrates VNF placement and migration in PPDCs to target diverse and dynamic traffic in cloud data centers. By modeling topology-aware costs, we are able to unify them into the same problem space for the first time and solve them in an integrated manner. Our approach is to first place VNFs into PPDCs according to the diverse traffic rates of communicating VM pairs and then adaptively migrate VNFs in response of changing traffic rates of the VM pairs. Our goal is to optimize the network resource usage of a PPDC for its lifetime.

An illustrating example. Take Fig. 1 for example, assume that the initial traffic rate among (vm_1 , vm_1') is much heavier than that of (vm_2 , vm_2'). To save the network traffic and their communication delay, we will place vnf_1 and vnf_2 on sw_1 and sw_2 respectively, as shown in Fig. 1(a). This way, the heavier traffic of (vm_1 , vm_1') along vnf_1 and vnf_2 (dark dashed line) is "shorter" than that of (vm_2 , vm_2') (light dashed line). Due to dynamic traffic in PPDCs, however, if next the traffic load of (vm_2 , vm_2') emerges as much heavier than that of (vm_1 , vm_1'), above VNF placement is no longer optimal. As vm_2 communicates with vm_2' via a route much longer than that of (vm_1 , vm_1'), shown in dark dashed line in Fig. 1(b), it generates heavy network traffic and consumes much of the network bandwidths. To tackle this problem, our key observation is that VNF migration can alleviate the network traffic. By migrating vnf_1 from sw_1 to sw_4 and vnf_2 from sw_2 to sw_5, as shown in Fig. 1(c), the heavy traffic is now confined in a small route while the light traffic taking a longer path, as shown in Fig. 1(d).

Our contributions. More formally, we identify and formulate two new VNF placement and migration problems in PPDCs. We refer to the VNF placement problem as PPP: *policy-preserving VNF placement* in PPDCs. Given a PPDC with communicating

VMs of diverse traffific rates, and a data center policy that each pair must satisfy, PPP studies how to place the VNFs inside PPDC to minimize the total communication cost of the VM pairs. With the dynamic VM traffic in PPDC, however, this initial VNF placement may become suboptimal after some time, thus there is a need for VNF migration. We refer to it as PPM: *policy-preserving VNF migration* in PPDCs. Given

existing placements of VNFs and VM pairs with dynamic traffic rates, and a data center policy that VM traffic must satisfy, the goal of PPM is to migrate VNFs to minimize the total cost of VNF migration and VM communication.

Consider that VNF migration itself incurs traffic overhead, and that a large scale PPDC typically has hundreds of thousands of communicating VMs with wide range of changing traffic rates, both PPP and PPM are challenging problems. We design optimal, approximation, and heuristic *policy-preserving* algorithms to solve both problems. One salient feature of our algorithms is that they potentially achieve ideal resource utilization for a PPDC's lifetime - after the PPP creates the VNF placement to optimize a PPDC's initial network resource utilization, the PPM then executes periodically to optimize a PPDC's network resource utilization in the events of dynamic VM traffic. Underlying PPP and PPM are two new graph-theoretical problems that have not been studied before.

Using traffic patterns and flow characteristics found in production data centers, we show that VNF migration is an effective technique to tackle dynamic traffic in PPDCs by outperforming the existing VM migration traffic-mitigation techniques by three times, and our VNF placement algorithms outperform existing techniques by more than two times.

2. RELATED WORK

There is a vast amount of literature of VNF placement in cloud data centers. Cohen *et al*. [15] was one of the first to tackle the NFV location problem considering VNF setup costs, connectivity costs of the clients, and capacity constraints of the network nodes.

They provided bi-criteria algorithms with constant approximation factors. Although it considered multiple types of VNFs, it did not consider the sequence of VNFs required in SFCs. Sang *et al*. [41] focused on minimizing the total number of VNF instances to provide a specific service to flows in a network. Chen *et al*. [13] improved it by considering limited capacity of each node and provided stronger algorithmic results in linear- and tree-structured networks. However, both work only considered multiple instances of the same VNF type and did not study the multiple VNF types addressed in SFCs.

Steering [48] was one of the first that considered a sequence of MBs and proposed a SFC placement problem. Built on SDN architecture and OpenFlow protocol, it proposed a heuristic SFC placement algorithm that minimizes the delay or distance traversed by all subscribers' traffic.Liu *et al*. [31] formally formulated the MB placement problem (that minimizes delay or bandwidth consumption) as a 0-1 programming problem. As it is NP-hard and no effective approximation, they proposed two heuristic algorithms called CalCostScore ORG and CalCostStore ED and show the latter performs better than the former. As their network models (including pairwise traffic and delay and bandwidth metrics) are the same as ours and it is one of the latest work, we compare our PPP algorithms with CalCostStore ED.

Bari *et al* [7], [6] studied VNF orchestration problem that determines the required number and placement of VNFs to optimize network operational costs. It provided an Integer Linear Programming (ILP) solution and an efficient greedy algorithm. Bhamare

et al. [10] studied the VNF placement problem that minimizes inter-cloud traffic and response time while satisfying deployment cost constraint and placement constraint. Ma *et al*. [33] considered the traffic changing effect of VNFs and studied the SFC deployment problems with the goal to load-balance the network. Gu *et al*. [21] designed a dynamic market auction mechanism for the transaction of VNF service chains that achieves near-optimal social welfare in the NFV ecosystem.

Different from all the existing work, the VNF placement identified in our report (i.e., PPP) is a new fundamental graph theoretic problem. PPP resembles but is significantly different from the classic *p*-median problem [38]. In *p*-median problem, it places *p* facilities in a network to minimize the sum of the demand-weighted distance between each demand node and its closest facility. In PPP, however, it not only identifies *p* locations to install the VNFs, but instead of accessing the closest facility, each VM pair needs to traverse all the *p* facilities (i.e., VNFs) in some order. We design optimal, approximation, and heuristic algorithms that outperform the existing work in different network scenarios.

VNF migration [45], [26], [32], [18], [25], [27], on the other hand, is not studied as extensively as VNF placement. As VNF migration disrupts service and incurs overhead traffic, one of its main goals was to minimize its influences including service down time [45], the overall migration time [26], and QoS degradation [18]. Eramo [18] proposed a comprehensive three-stage framework that instantiates, migrates VNFs and routes SFC requests to the appropriate VNFs. There is a different objective for each stage. In

particular, its VNF migration minimized the total energy consumption and the revenue loss of QoS degradation in VNF migration. As in our model the VM communication cost can be treated as total energy consumption of VM communication and the VNF migration cost can be treated as the revenue loss of QoS degradation, we compare our VNF migration (i.e., PPM) algorithms with theirs. Liu *et al*. [32] considered that existing cloud users can move around and new users can join in and maximized the service provider's profit. Huang *et al*. [25] proposed two VNF instance scaling techniques viz. horizontal scaling (that migrates existing VNF instances) and vertical scaling (that instantiates new VNF instances). The goal was to maximize the number of NFV-enabled requests while meeting their end-to-end delay requirements. Jia [27] studied policy-aware unicast request admissions with and without end-to-end delay constraints and minimized operational cost of admitting requests.

VNF migration studied in this report, however, differs from aforesaid work in both its goal and model. While existing VNF migration work achieved various objectives such as server consolidation and energy efficiency, QoS degradation minimization and throughput maximization, our work focuses on the dynamic communication traffic rates existing among VMs with the goal of minimizing total migration and communication cost. Theoretically, our VNF migration model (i.e., PPM) resembles but significantly differs from the dynamic facility location problem (DFLP) [35], [39]. The DFLP locates and possibly relocates facilities over time to minimize transportation costs in response to changing demands or distribution costs. The difference is that while in DFLP each

demand node only accesses its closest facility, in PPM each VM pair must traverse the entire facilities (i.e., VNFs) in some sequence.

In contrast to most of the existing work, one pronounced feature of our research is the integration of VNF placement and VNF migration, two fundamental VNF mechanisms, into one framework. By characterizing *topology-aware* costs for both VM communication and VNF migration, we are able to capture network traffic incurred in both cases accurately to seamlessly optimize the overall resource utilization in PPDCs.

Two lines of work specifically addressed dynamic network traffic in cloud data centers. The first line includes [44], [47], [42], which employed online learning methods to estimate upcoming traffic rates and to adjust VNF deployment. Our work instead adopted the dynamic traffic flow characteristics found in Facebook data centers [40] to emulate how VM traffic rates fluctuate. In the future if we consider the proactive prediction of dynamic traffic, this line of work could help. The other line of work is by Cui *et al*. [16] and Flores *et al*. [43], which proposed migrating communicating VMs instead of VNFs to ameliorate dynamic traffic. We observe that as migrating one VM pair does not affect communication cost of other pairs while migrating one VNF affects all VM pairs traversing it, VNF migration has a more decisive and effective impact on reducing dynamic network traffic than VM migration. We show in the experiments that VNF migration outperforms the existing VM migration traffic-mitigation techniques by 3 times.

3. PRELIMINARIES

System Model. We use fat trees [5] to illustrate the problems and solutions. However, as they are applicable to any data center topology, we model a PPDC as an undirected graph $G(V, E)$. $V = V_p \cup V_s$ is a set of PMs $V_p = \{pm_1, pm_2, ..., pm_{|Vp|}\}$ and a set of switches $Vs = \{sw_1, sw_2, ..., sw_{|Vs|}\}$. E is a set of edges, each connecting either one switch to another or a switch to a PM. Fig. 2 shows a $k = 4$ PPDC where k is the number of ports per switch.

There are n VNFs $M = \{vnf_1, vnf_2, ..., vnf_n\}$ serving as a SFC that need to be placed and then migrated inside the PPDC. We adopt the *bump-off-the-wire* design [28], which uses a policy-aware switching layer to explicitly redirect traffic along VNFs. Fig. 2 shows three VNFs vnf_1, vnf_2 and vnf_3 installed on different switches in the PPDC.[1] As a switch and its attached VNF are connected by high-speed optical fibers, the delay between them is negligible compared to that among switches and PMs [22].

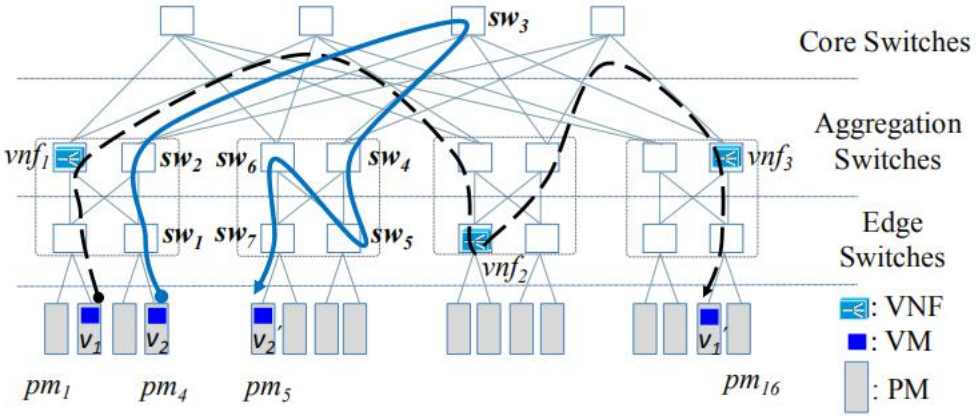


Fig. 2. A PPDC with 16 PMs: pm_1, pm_2, ..., and pm_{16}, 3 VNFs: vnf_1, vnf_2, and vnf_3, and two VM pairs: (v_1, v_1') and (v_2, v_2'). ● and ► indicate source and destination VM respectively.

As the east-west traffic is the predominant traffic in a data center and accounts more than 70 percent of its traffic [1], [2], and most east-west cloud traffic is pairwise [36], we focus on pairwise VM communication. We assume that there are l pairs of communicating VMs $P = \{(v_1, v_1'), (v_2, v_2'), ..., (v_l, v_l')\}$ already placed into the PMs, where $v \in V = \{v_1, v_1', v_2, v_2', ..., v_l, v_l'\}$ is placed at PM $s(v)$. For any VM pair (v_i, v_i'), v_i and v_i' are referred to as its *source* and *destination VM* and $s(v_i)$ and $s(v_i')$ as its *source* and *destination PM* respectively. Denote the traffic rate or transmission rate of (v_i, v_i') as λ_i and the *traffic rate vector* as $\vec{\lambda} = \langle \lambda_1, \lambda_2, ..., \lambda_l \rangle$. As the VM traffic rates change over time in a dynamic PPDC, $\vec{\lambda}$ is not a constant vector. In Fig. 2, there are two VM pairs: (v_1, v_1') and (v_2, v_2'), with $\vec{\lambda} = \langle 1, 100 \rangle$. Table 1 shows all the notations.

Table 1 Notation Summary

Notation	Description
V_p	$V_p = \{pm_1, pm_2, ..., pm_{\|V_p\|}\}$ is the set of $\|V_p\|$ PMs
V_s	$V_s = \{sw_1, sw_2, ..., sw_{\|V_s\|}\}$ is the set of $\|V_s\|$ switches
$\mathcal{M}$	$\mathcal{M} = \{vnf_1, vnf_2, ..., vnf_n\}$ is the set of n VNFs
$\mathcal{P}$	$\mathcal{P} = \{(v_i, v_i'), ..., (v_l, v_l')\}$ is the set of l VM pairs
$\mathcal{V}$	$\mathcal{V} = \{v_1, ..., v_l, v_1', ..., v_l'\}$
$s(v)$	$s(v)$ is the PM where VM $v \in \mathcal{V}$ is stored
λ_i	Traffic rate between v_i and v_i', $1 \leq i \leq l$
$\overrightarrow{\lambda}$	$\overrightarrow{\lambda} = \langle \lambda_1, \lambda_2, ..., \lambda_l \rangle$
$c(i,j)$	Communication cost between PMs (or switches) i and j
$p(j)$	Switch where vnf_j is placed at under VNF placement p
$\overrightarrow{\pi}$	$\overrightarrow{\pi} = \langle \pi^1, \pi^2, ..., \pi^l \rangle$
$C_c(p)$	Total VM communication cost with p
μ	VNF migration coefficient
$m(j)$	Switch where the vnf_j migrates to under VNF migration m
$C_m(m)$	Total VNF migration cost with migration m
$C_c(m)$	Total VM communication cost after migration m
$C_t(m)$	Total VNF migration and VM communication cost with m

Topology-Aware Cost Model. In our model, each edge $(u, v) \in E$ has a cost $c_{u,v}$ indicating either the delay or energy cost or bandwidth consumption on this edge caused by VM communication or VNF migration. Given any two PMs (and switches) u and v, let $c(u, v)$ denote the sum of the costs of all the edges traversed by VM communication (or VNF migration) from u to v. Thus the *communication cost* of any VM pair (v_i, v_i') is $\lambda_i \cdot c(s(v_i), s(v_i'))$ .The migration cost of migrating any VNF in M from switch u to switch v is $\mu \cdot c(i, j)$, where, μ is a VNF migration coefficient that depends on the relative size and bandwidth consumption of VMs and VNFs. Note that our topology-aware model is different from the well-known *pre-copy model* [14], [34]. It modeled the cost of migrating a VM or a VNF v as $M_s \cdot \frac{1-(P_r/B_a)^{n+1}}{1-(P_r/B_a)}$

, where M_s is the image size of v, P_r is its page dirty rate, B_a is the available bandwidth, and n is number of pre-copy phases. As this topology-aware cost model strives to accurately capture the incurred network traffic costs during VM communication and VNF migration, it is more suitable than the existing model for a large-scale dynamic cloud data center studied in this report.

Service Function Chainings (SFCs). As one SFC is generally sufficient to serve both security and performance purposes [2], we assume there is one SFC in a PPDC at a time. An SFC, denoted as (vnf_1, vnf_2, ..., vnf_n), requires that the VM traffic to go through the VNFs in that specific order. We refer to vnf_1 (and vnf_n) as ingress (and egress) VNF, and the switch where the ingress (and egress) VNF is installed as ingress (and egress) switch. In Fig. 2, (v_1, v_1') traverses VNFs under SFC (vnf_1, vnf_2, vnf_3), resulting in communication cost of 1 × 10 = 1000 (black dashed line). Note that we use such unweighted costs (i.e., number of edges) only for purpose of illustration. We assume that Flow Tags [19], a well-known SDN architecture enforcing network-wide middlebox policy, is available for consistent policy implementation during VNF migration. Next we quantitatively illustrate the benefit of using VNF migration to reduce network cost.

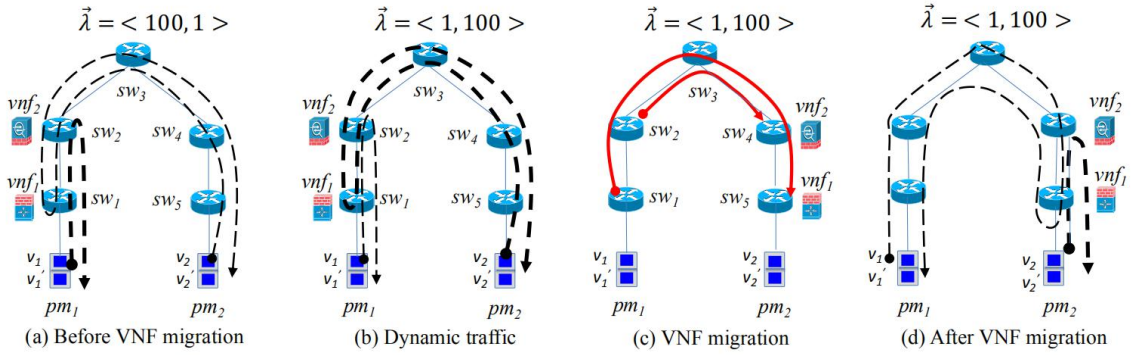


Fig. 3. VNF migration achieves 58.6% of total cost reduction in a $k = 2$ linear PPDC. The light and heavy black dashed lines refer to light and heavy VM traffic, respectively. The red solid lines refer to the VNF migration.

EXAMPLE 1: Fig. 3(a) shows a $k = 2$ fat tree PPDC with two PMs pm_1 and pm_2. There are two VM pairs (v_1, v_1') and (v_2, v_2'), with v_1 and v_1' at pm_1 while v_2 and v_2' at pm_2. $\vec{\lambda} = \langle 100, 1 \rangle$ and $\mu = 1$. There are two VNFs vnf_1 and vnf_2. An initial optimal VNF placement is installing vnf_1 on switch sw_1 and vnf_2 on switch sw_2, resulting in total communication cost of $100 \times 4 + 1 \times 10 = 410$ (shown in black dashed lines). However, due to dynamic traffic, $\vec{\lambda}$ next changes to $\langle 100, 1 \rangle$, as shown in Fig. 3(b). This results in a dramatic increase of total communication cost to $1 \times 4 + 100 \times 10 = 1004$. By migrating vnf_1 to sw_5 and vnf_2 to sw_4, shown in solid red line in Fig. 3(c), although it incurs migration cost of 6, the total VM communication cost (shown in Fig. 3(d)) reduces to $1 \times 10 + 100 \times 4 = 410$, a 58.6% of total cost reduction. This fat tree PPDC is indeed the same linear PPDC in Fig. 1.

4. PPP: POLICY-AWARE VNF PLACEMENT

1) Problem Formulation: We define a VNF placement function p: $M \rightarrow V_s$ that installs VNF $v \in M$ at switch $p(v) \in V_s$. For any VM pair communication, the ingress switch is

always $p(1)$ and the egress switch is always $p(n)$. Given any VNF placement p, denote the total communication cost of all the l VM pairs under p as $C_c(p)$. We have

$$C_c(p) = \sum_{i=1}^{l} \lambda_i \cdot \sum_{j=1}^{n-1} c\big(p(j), p(j+1)\big) + \sum_{i=1}^{l} \lambda_i \cdot \Big(c\big(s(v_i), p(1)\big) + c\big(p(n), s(v_i')\big)\Big). \quad (1)$$

The objective of PPP is to find a VNF placement p to minimize $C_c(p)$. Below we show that PPP is NP-hard by proving that its special case of $l = 1$ is NP-hard. We refer to this special case as PPP-1, and show below that it is equivalent to k-stroll problem [20], [11], which is NP-hard. k-stroll problem is defined as follows. Given a complete graph $G = (V, E)$ with non-negative length w_e on edge $e \in E$, two special nodes s and t, and an integer k, the goal of k-stroll is to find an s-t walk of minimum length that visits at least k distinct nodes excluding s and t. When $s = t$, it is called k-tour problem. Here it assumes that triangle inequality holds for the edges: for (x, y), (y, z), $(z, x) \in E$, $w(x, y) + w(y, z) \geqq w(z, x)$. Fig. 4 shows an optimal 2-stroll between s and t is a walk: s, D, t, C, and t, with a cost of 6.

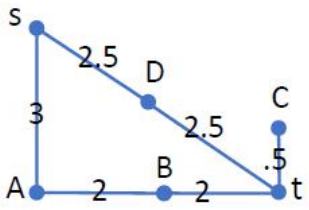


Fig. 4. k-stroll problem. Numbers on edges are their weights.

Theorem 1: PPP-1 is equivalent to *k*-stroll problem.

Proof: Although *k*-stroll problem is mostly studied on non-complete graph [20], [11], [12], we follow [8] and assume it is a complete graphs. We show that this assumption enables us to design a dynamic programming (DP) based heuristic algorithm that performs very close to the optimal.

Recall a data center is represented as a graph $G(V = \{V_p \cup V_s\}, E)$, where V_p is the set of PMs and V_s is the set of switches. Given an instance of PPP-1 where the only pair of VMs v_1 and v_1' are located at PMs $s(v_1)$ and $s(v_1')$ respectively, we construct a new graph $G'(V', E')$. Here $V' = V_s \cup \{s(v_1), s(v_1')\}$ and E' include all the edges in E that connect any two nodes in V'. That is, G' is a subgraph of G induced by V'. Next we construct the *metric completion* of $G'(V', E')$ and denote it as $G''(V'', E'')$. G'' is a complete graph with the same set of nodes V' (i.e., $V'' = V'$), while for any pair of nodes $u, v \in V''$, the cost of $(u, v) \in E''$ is the cost of the shortest path connecting u and v in G'. Fig. 5 shows the conversions for the linear data center G in Fig.3. considering only one VM pair (v_1, v_1')

We claim that an optimal *n*-stroll in G'' that starts at $s(v_1)$ and ends at $s(v_1')$ gives the optimal placement of n VNFs and minimum cost policy-preserving routing for (v_1, v_1') in G, and vice versa. First, as this walk visits at least n distinct other nodes when traversing from $s(v_1)$ to $s(v_1')$, let the first n nodes traversed be n_1, n_2, and ...n_n in that order. Then we place vnf_1 at n_1, vnf_2 at n_2, ..., and vnf_n at n_n, and let v_1 communicate with v_2 by following the same walk thus traversing vnf_1, vnf_2, ..., and vnf_n in that order. Second,

as this s(v_1)-s(v_1') walk is the minimum n-stroll in G'' , traversing the corresponding switches in G thus gives (v_1 , v_1') minimum communication cost in G. On the other hand, if a VNF placement gives minimum communication cost for (v_1 , v_1') in G, as each VNF is placed on a different node, the resulted s(v_1)-s(v_1') walk in G'' must be an optimal s(v_1)-s(v_1') stroll.

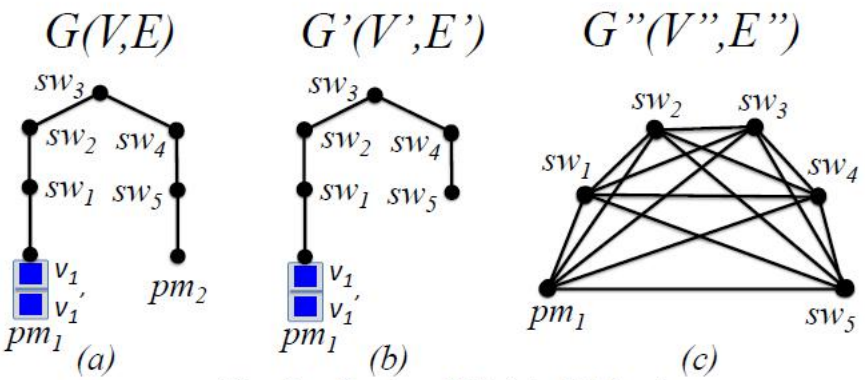


Fig. 5. Proving PPP-1 is NP-hard.

In Fig. 5, $s(v_1) = s(v_1') = pm_1$. As the optimal VNF placement for (v_1 , v_1') in G is placing vnf_1 on sw_1 and vnf_2 on sw_2, a minimum 2-tour for pm_1 in G'' is thus pm_1, sw_1, sw_2, and pm_1, and vice versa.

2) *VNF Placement Algorithms*: Below we present the VNF placement algorithms for PPP-1 and PPP respectively.

Algorithms for PPP-1. As PPP-1 is equivalent to k-stroll problem on a properly transformed graph of the PPDC, algorithms solving k-stroll thus can be used to solve PPP-1. There are extensive works solving k-stroll on both directed and undirected graphs [20], [11], [12], [8]. For undirected graphs, Chaudhuri *et al*. [11] designed a primal-dual based approximation algorithm with approximation ratio of $2 + \epsilon$. Chekuri [12] proposed

a bi-criteria approximation that finds an *s*-*t* walk of length at most $\max\{1.5D, 2LD\}$ that contains at least $(1 - \epsilon) \cdot k$ nodes. Here, L is the length of an optimal path and D the shortest path from s to t. For directed graph, k-stroll is APX-hard [12], meaning that a polynomial time approximation algorithm is unlikely. Chekuri *et al*. [12] and Bateni [8] *et al*. designed $O(\log^2 OPT)$ and polylogarithmic (i.e., $O(\log^k n)$) approximation algorithms respectively. As our data center network is modeled as an undirected graph, we adopt the primal-dual based approximation algorithm proposed by Chaudhuri *et al*. [11] to solve PPP-1 and places n VNFs for one VM pair (v_1, v_1').

Approximation algorithm for PPP-1. It consists of three steps.

Step 1. Given $G'(V', E')$ in Fig. 5, where $V' = \{s(v_1)\} \cup \{s(v_1')\} \cup V_s$, let x_v indicate if $v \in V_s$ is selected to place a VNF, and y_e denote whether an edge $e \in E'$ is on the path. And let $\delta(S)$ denote the set of edges with exactly one endpoint in set S. The primal ILP of PPP-1 is formulated as below.

$$(A) \quad \min \lambda_1 \cdot \sum_{e \in E'} c_e \times y_e \tag{2}$$

s.t.

$$x_v = \{0, 1\}, \quad \forall v \in V_s \tag{3}$$

$$y_e = \{0, 1\}, \quad \forall e \in E' \tag{4}$$

$$\sum_{e \in \delta(U)} x_e \geq 1, \quad \forall U \subseteq V : s(v_1') \in U, s(v_1) \notin U \tag{5}$$

$$\sum_{e \in \delta(S)} \geq 2 \cdot x_v, \quad \forall S \subseteq V_s, \forall v \in S \tag{6}$$

$$\sum_{v \in V_s} x_v \geq n, \quad \forall v \in V_s \tag{7}$$

Constraints 5 and 6 construct a path between PMs $s(v_1)$ and $s(v_1')$ by selecting an edge in every cut separating them, and Constraint 7 guarantees this path has least n switches.

Step 2. It considers the dual of the linear programming relaxation of above ILP (A), that is, $0 \leq x \leq 1$ and $0 \leq y_e \leq 1$, and relaxes the complementary slackness condition related to its dual variables.

Step 3. It iteratively adds edges, paying for them with increases to variables in the dual, and then deletes edges to obtain the final path that spans n switches. The running time of this algorithm is $O(|V'|^5 \cdot \log|V'|)$ [11]. Theorem 2 below shows it yields a solution to the primal integer problem that costs no more than $2 + \epsilon$ times the value of the feasible dual solution constructed, which implies that the primal solution is with a factor of $2 + \epsilon$ of optimal.

Theorem 2: Above primal-dual based algorithm for PPP-1 yields cost that is at most $2 + \epsilon$ of optimal.

Proof. Please refer to Section 4 of [11] for detailed proof.

Above approximation algorithm is inspired by a theoretical work that strives to achieve performance bound of algorithms, and involves complicated procedures (e.g., Step 3) that cannot be easily implemented. Besides, the objective function in ILP(A) counts the weight of each edge on the k-stroll only once, implying that the k-stroll must be a path that does not visit the same node twice. As a k-stroll can well be a walk wherein a node and an edge are visited multiple times, as shown in Fig. 4, this assumption is a strong one.

Dynamic programming (DP) for PPP-1. We thus propose a more practical VNF placement heuristic for PPP-1. Our key observation is that although finding a shortest s-t walk with k distinct nodes is NP-hard, finding a shortest s-t walk of k edges can be solved optimally and efficiently using DP. Algo. 1 below finds such a shortest s-t walk with $k + 1$ edges (lines 4-10) and checks if these $k + 1$ edges traverses k distinct nodes (lines 11-19). If not, it finds a shortest walk with $k + 2$ edges, so on and so forth, until k distinct nodes are found (lines 21).

To improve the search efficiency of finding more distinct nodes along the s-t walk, Algo. 1 adds two important improvement on top of its DP backbone. First, it applies on a complete graph $G''(V'', E'')$ shown in Fig. 5(c). As there does not always exist an s-t walk of exactly $k + 1$ edges in data center graph G while there always exists such a one in G'' (as long there are more than $k + 1$ edges in G''), using G'' overcomes an obstacle otherwise faced by using G. For example, if we directly feed the non-complete graph input shown in Fig. 4, it will find a 3-edge path of is s, A, B, t of cost 7, which is not optimal.

Second, even there exists $k + 1$ edges on the walk, it does not guarantee finding k distinct nodes on the walk as an edge can be traversed multiple times. To overcome this obstacle, Algo. 1 avoids the loop that traverses the same edge twice consecutively (line 6). The time complexity of Algo. 1 is $k \cdot |E''| \cdot |V''|^2$, which is $O(k \cdot |V'|^4)$, more efficient than that of above primal-dual. Note that Algo. 1 also works for k-tour problem where $s = t$

and the special case that at least *k* distinct nodes are already on the shortest path between *s* and *t*.

Algorithm 1: DP Algo. for k-stroll problem.
Input: A complete graph $G''(V'', E'')$, s, t, and k, $c_{(u,v)}$ for $(u, v) \in E''$.
Output: $stroll(s, t, k, p)$, the cost of an s-t walk in G'' visiting at least k distinct other nodes, stored in p.
Notations: e: index for edges; i, j, n: indices for nodes; $c(u, t, e)$: the cost of a u-t walk with exactly e edges, initially $+\infty$; $successor(u, t, e)$: successor of u in a u-t walk with e edges, initially -1; r: number of needed edges on s-t walk, initially $k + 1$; p: an array storing distinct nodes on s-t walk; num: the number of distinct nodes in p; $found$: true if it has found a s-t walk with at least k distinct nodes; initially false;

1. $V'' = \{u_1, ..., u_{|V''|}\}$, let $u_a = s$, $1 \leq a \leq |V''|$, $u_{|V''|} = t$;
2. $\forall u_i, u_j \in V''$ with $i \neq j$, $c(u_i, u_j, 1) = c_{u_i,u_j}$, $successor(u_i, u_j, 1) = u_j$, $successor(u_j, u_i, 1) = u_i$; $\forall u_i \in V''$, $c(u_i, u_i, 1) = +\infty$, $successor(u_i, u_i, 1) = -1$;
3. **while** $(\neg found)$
4. **for** $(e = 2; e <= r; e{+}{+})$ // # of edges in u_i-t walk
5. **for** $(i = 1; i \leq |V''| - 1; i{+}{+})$ // node u_i
6. **for** $\big($each u_n $s.t.$ $u_n \neq u_i \wedge u_n \neq t \wedge u_i \neq successor(u_n, t, e-1)\big)$
7. **if** $\big(c(u_i, t, e) > c_{u_i,u_n} + c(u_n, t, e-1)\big)$
8. $c(u_i, t, e) = c_{u_i,u_n} + c(u_n, t, e-1)$;
9. $successor(u_i, t, e) = u_n$;
10. **end if;**
11. $num = 0$; $p = \phi$ (empty set), $e--$;
12. $a = successor(s, t, e)$;
13. **while** $(e > 1)$
14. **if** $(a \neq s \wedge a \neq t \wedge a \notin p)$
15. $p[num] = a; num{+}{+}$;
16. **end if;**
17. $e--$;
18. $a = successor(a, t, e)$;

19. **end while;**
20. **if** $(num \geq k)$ found = true;
21. **else** $r++$; // less than k distinct nodes visited
22. **end while;**
23. $stroll(s,t,k,p) = c(u_a, u_b, r)$;
24. **RETURN** $stroll(s,t,k,p)$.

EXAMPLE 2: Fig. 2 shows a VM pair (v_2 , v_2') with v_2 and v_2' placed at pm_4 and pm_5 respectively. To find 7 VNFs between v_2 and v_2' is to find a 7-stroll between pm_4 and pm_5. Algo. 1 finds such an 8-edge path traversing 7 distinct switches (shown in blue solid line): pm_4, sw_1, sw_2, sw_3, sw_4, sw_5, sw_6, sw_7, and pm_5. Note there are other 8-edge walk between pm_4 and pm_5 that traverses only 5 distinct switches: pm_4, sw_1, sw_2, sw_1, sw_2, sw_3, sw_4, sw_7, and pm_5. This walk is not selected by Algo. 1 due to loop between sw_1 and sw_2.

By avoiding the loops between adjacent nodes and using a complete graph as input, Algo. 1 dramatically improves the efficiency of searching for distinct switches in *k*-stroll problem. We show in simulations that Algo. 1 constantly outperforms the performance guarantee of $2 + \epsilon$ provided the primal-dual algorithm and performs close to the optimal most of the time. However, despite its good heuristic performance, Algo. 1 is not optimal, as shown in Example 3. We attribute the sub-optimality of Algo. 1 to its finding a k-stroll between *s* and *t* by finding a walk of $k + 1$ edges, while an optimal *k*-stroll could have more than k +1 edges. Nonetheless, Theorem 3 below shows the sufficient condition for the optimality of Algo. 1.

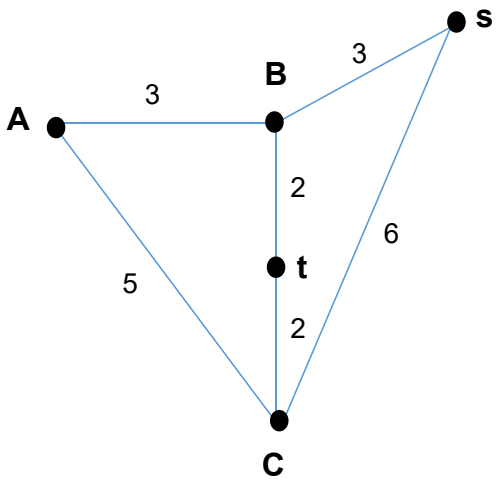


Fig. 6. Sub-optimality of Algo. 1. Numbers on edges are their weights

EXAMPLE 3: To find a 3-stroll between *s* and *t*, Algo. 1 gives s, B, A, B, t, C, and t, with total cost of 15. However, the optimal solution is s, B, A, C, and t, with total cost of 13. The reason for such optimality is because the 2-stroll from B to t is B, A, B, and t while the 1-stroll from A to t is A, B, and t. Thus edge (B, A) are visited multiple times in Algo. 1 while in optimal solution, it is only visited once.

Theorem 3: Let $successor_i$, $1 \leqq i \leqq k$, be the i^{th} switch along the $s(v_1) - s(v_1')$ stroll found by Algo. 1. If all the *i*-stroll between $successor_i$ and *t*, $1 \leqq i \leqq k$, do not visit the same edge twice, then Algo. 1 is optimal for *k*-stroll problem.

Proof. We give a proof sketch due to space constraint. If there exists a path of $k + 1$ edges between *s* and *t* that visits *k* distinct nodes, the two key characteristics of the DP viz. optimal substructure and overlapping subproblems guarantees that the DP is optimal; as it exists, it can find a shortest such path visiting *k* distinct nodes between *s* and *t*.

Algorithms for PPP. Next we solve PPP wherein $l > 1$. The key idea of Algo. 2 is to find all pairs of ingress switch $p(1)$ and egress switch $p(n)$ and then treat finding the rest *n*

- 2 switches to place VNFs as an (n - 2)-stroll problem with $s = p(1)$ and $t = p(n)$. Its time complexity is $O(k \cdot |V|^6 \cdot d)$.

Algorithm 2: VNF Placement Algorithm for PPP.

Input: A PPDC $G(V, E)$ with VM pair placement $s(v)$, $v \in \mathcal{V}$, and an SFC $(vnf_1, vnf_2, ..., vnf_n)$.

Output: A VNF placement p and the total comm. cost $C_c(p)$.

Notations: x: set of switches storing $vnf_2, vnf_3, ..., vnf_{n-1}$;

1. $C_c(p) = +\infty$;
2. **if** $(n == 1)$
3. **for** $(1 \leq i \leq |V_s|)$
4. $a = \sum_{k=1}^{l} \lambda_k \cdot \big(c(s(v_k), sw_i) + c(sw_i, s(v'_k))\big)$;
5. **if** $(a \leq C_c(p))$ $p(1) = sw_i$, $C_c(p) = a$;
6. **end for;**
7. **elseif** $(n == 2)$
8. **for** $(1 \leq i < |V_s|)$
9. **for** $(i + 1 \leq j \leq |V_s|)$
10. $a = \sum_{k=1}^{l} \lambda_k \cdot \big(c(s(v_k), sw_i) + c(sw_i, sw_j) +$
11. $c(sw_j, s(v'_k))\big)$;
12. **if** $(a \leq C_c(p))$
13. $p(1) = sw_i$, $p(2) = sw_j$, $C_c(p) = a$;
14. **end for;**
15. **end for;**
16. **else** // $n > 2$
17. **for** $(1 \leq i < |V_s|)$
18. **for** $(i + 1 \leq j \leq |V_s|)$
19. $a = \sum_{k=1}^{l} \lambda_k \cdot \big(c(s(v_k), sw_i) + c(sw_j, s(v'_k))\big)$;
20. $b = stroll(sw_i, sw_j, n - 2, x)$; // Call Algo. 1
21. **if** $\big((a + b) \leq C_c(p)\big)$
22. $C_c(p) = a + b$, $p(1) = sw_i$, $p(2) = sw_j$;
23. **for** $(2 \leq i \leq n - 1)$ $p(i) = x[i - 2]$;
24. **end for;**
25. **end for;**
26. **end else;**
27. **RETURN** p and $C_c(p)$.

Integer Linear Programming (ILP) for PPP. Below we present a ILP formulation (B) that optimally solves PPP. For any two nodes u and v with $(u, v) \in E$, as the flow could

go either from u to v or v to u, it necessitates the use of directed edges (u, v) and (v, u). Next we introduce a few decision variables. Let $x_{j,k}$ indicate if vnf_j is placed on switch sw_k or not. Let $y_{i,u,v}$ represent if directed edge (u, v) is traversed by VM pair (v_i, v_i'). Let $y_{i,j,e}$ indicate if an edge e is used to reach the vnf_j by VM pair (vm_i, vm_i').

$$(B) \qquad \min \sum_{i=1}^{l} \lambda_i \cdot \sum_{(u,v)\in E} (y_{i,u,v} \times c_{u,v}) \tag{8}$$

s.t.

$$y_{i,u,v} = \{0,1\}, \quad \forall 1 \le i \le l, (u,v) \in E \tag{9}$$

$$x_{j,k} = \{0,1\}, \quad \forall 1 \le j \le n, 1 \le k \le |V_s| \tag{10}$$

$$\sum_{k=1}^{|V_s|} x_{j,k} = 1, \quad \forall 1 \le j \le n \tag{11}$$

$$\sum_{j=1}^{n} x_{j,k} = 1, \quad \forall 1 \le k \le |V_s| \tag{12}$$

$$\sum_{(s(v_i),v)\in E} y_{i,s(v_i),v} = 1, \quad \forall 1 \le i \le l \tag{13}$$

$$\sum_{(u,s(v_i))\in E} y_{i,u,s(v_i')} = 1, \quad \forall 1 \le i \le l \tag{14}$$

$$\sum_{(v,u)\in E} y_{i,v,u} = \sum_{(u,v)\in E} y_{i,u,v}, \quad \forall u \in V_s, 1 \le i \le l \tag{15}$$

Eqns. 11 and 12 place each of the n VNFs onto different switches. Eqns. 13 and 14 ensure for each VM pair (v_i, v_i'), the traffic coming out of its source PM $s(v_i)$ arrives at its destination PM $s(v_i')$. Eqn. 15 enforces the flow conservation on all switches: if any VM pair(v_i, v_i') visits a switch, it must exit it. Obviously, the optimal solutions from ILP (B) can also be achieved by below exhaustive $O(|V|^k)$ algorithm.

Algorithm 3: Exhaustive VNF Placement Algo. for PPP.
Input: A PPDC $G(V, E)$ with VM pair placement $s(v)$,
$v \in V$, and an SFC $(vnf_1, vnf_2, ..., vnf_n)$.
Output: A VNF placement p and the total comm. cost $C_c(p)$.
1. $C_c(p) = +\infty$;
2. Among all $|V_s| \cdot (|V_s| - 1) \cdot ..., \cdot(|V_s| - n + 1)$ placements, find p that gives the minimum cost $C_c(p)$.
5. **RETURN** p and $C_c(p)$.

Although Algo. 3 is time-consuming, as it can be implemented easily, we compare it with other algorithms for benchmark.

3) *State-of-the-Art VNF Placement:* Steering [48] proposed a middlebox service (i.e. VNF) placement problem with the objective of minimizing the average time for the subscribers' traffic. It proposed a heuristic algorithm that considers service dependency between services. Two services are dependent if they appear consecutively in a service chain required by some subscribers, and the degree of this dependency is determined by the amount of traffic going through it. It then picks the service with the highest dependency degree, finds its best location (i.e., minimizing the average time), and finishes until all services are placed in the network. For a single SFC in our setup where all its VNFs have the same dependency, Steering just finds the best locations for each of the VNFs on by one.

Liu [31] et al. proposed a two-step greedy algorithm called CalCostStore ED. First, the MBs are sorted in descending order of their importance factor, which is the number of policies that uses this MB. Second, it calculates each MB's cost score for each switch and the switch with minimum cost score will be selected to place each MB. Here the *cost*

score is defined as the increment of the total end-to-end delay due to adding this MB plus the weighted average delay of all unplaced MBs to this MB.

We compare our PPP algorithms with above two works for the following two reasons. First, both works assume the ingress and egress locations for each subscriber's traffic, implying pairwise communication as in our report. Second, both works try to minimize the total communication cost in terms of end-to-end delay or bandwidth consumption, conforming to the goal of the VNF placement studied in this report.

5. PPM: POLICY-PRESERVING VNF MIGRATION

1) *Problem Formulation*: In PPM, the initial VNF placement is given by a *placement function* p : [*1, 2, ..., n*] $\rightarrow V_s$, indicating that VNF vnf_j is at switch $p(j) \in V_s$. The total communication cost of all the l VM pairs with placement p is thus $C_c(p)$ (Eq. 1). Next, define a VNF *migration function* as m : [1, 2., ...n] $\rightarrow V_s$, meaning that VNF vnf_j will be migrated from switch $p(j)$ to switch $m(j)$ ($m(j) = p(j)$ if vnf_j does not migrate). Let $C_m(m) = \mu \cdot \sum_{j=1}^{n} c(p(j), m(j))$ be the *total migration cost* of all the n VNFs with migration m. Let $C_c(m)$ be the *total communication cost* of all VM pairs *after* m. Let $C_t(m)$ be the total cost of *VNF migration* and *VM communication cost* after m. Then

$$C_t(m) = C_m(m) + C_c(m) = \mu \cdot \sum_{j=1}^{n} c(p(j), m(j)) + \sum_{i=1}^{l} \lambda_i \cdot \sum_{j=1}^{n-1} c(m(j), m(j+1))$$
$$+ \sum_{i=1}^{l} \lambda_i \cdot (c(s(v_i), m(1)) + c(m(n), s(v_i'))). \quad (16)$$

The objective of PPM is to find a VM migration m that minimizes $C_t(m)$. Below we show that PPP is a special case of PPM, thus PPM is also NP-hard.

Theorem 4: PPP is a special case of PPM with $\mu = 0$.

Proof: Plug $\mu = 0$ into Eq. 16, we get $C_t(m) = \sum_{i=1}^{l} \lambda_i \cdot \sum_{j=1}^{n-1} c(m(j), m(j+1)) + \sum_{i=1}^{l} \lambda_i \cdot (c(s(v_i), m(1)) + c(m(n), s(v_i')))$. As PPP is to find a VNF placement, we replace *m* with *p* in r.h.s. of above equation and get $C_t(m) = \sum_{i=1}^{l} \lambda_i \cdot \sum_{j=1}^{n-1} c(p(j), p(j+1)) + \sum_{i=1}^{l} \lambda_i (c(s(v_i), p(1)) + c(p(n), s(v_i'))) \overset{Eq.1}{=} C_c(p)$.

2) *VNF VM Migration Algorithm for PPM*: Below we present our VNF migration algorithm for PPM. Our key observation is that minimizing the total cost of VM communication and VNF migration is to strike a balance between these two costs, as VNF migration increases migration costs while decreasing VM communication cost. In this regards, the PPM is similar to a multi-objective optimization problem (MOP) as it tries to minimize migration and communication cost simultaneously. Pareto front is one of the solutions of MOPs, which is a set of non-dominated solutions, being chosen as optimal, if no objective can be improved without sacrificing at least one other objective. We thus design below step-wise algorithm to approximate Pareto front, and show it performs close to the optimal.

Algorithm 4: VNF Migration Algorithm for PPM.
Input: A PPDC $G(V, E)$ with VM pair placement $s(v)$, $v \in \mathcal{V}$, and VNF placement $p(j)$, $1 \leq j \leq n$, migration coefficient μ, traffic rate vector $\overrightarrow{\lambda}$;
Output: A VNF migration m and the total cost $C_t(m)$.
Notations: $pareto$: an array of n elements;
$pareto = [p(1), p(2), ..., p(n)]$; // initial VNF placement
$pareto|\{j, a\} = [p(1), ..., p(j-1), a, p(j+1), ..., p(n)]$ is $pareto$ with its j^{th} element being replaced by a;

0. $C_m(m) = C_c(m) = 0; C_t(m) = \infty$;
1. Let m be the new VNF placement found by Algo. 2, that is, $m(j)$ is the switch that vnf_j is migrated to;
2. Let $hops_j$ denote the number of hops between $p(j)$ and
3. $m(j)$; let $hops_{max}$ be the maximum of $hops_j$;
4. **for** $(0 \leq i \leq hops_{max})$
5. **for** $(1 \leq j \leq n)$
6. **if** $(hops_j \geq 1)$
7. Let the switch that is one hop closer from $p(j)$ towards $m(j)$ be a;
8. $hops_j --$; $pareto = pareto|\{j, a\}$;
9. $C_m(pareto) += \mu \cdot c\big(pareto[j], a\big)$;
10. **end if;**
11. **end for;**
12. $C_t(pareto) = C_c(pareto) + C_m(pareto)$;
13. **if** $\big(C_t(pareto) \leq C_t(m)\big)$
14. $C_t(m) = C_t(pareto)$; $m = pareto$;
15. **end if;**
16. **end for;**
17. **RETURN** m and $C_t(m)$.

EXAMPLE 4: Fig. 7 illustrates how StepWise Algorithm (i.e., Algo. 4 works. The SFC consists of 5 VNFs: vnf_1,...,vnf_5. Due to dynamic VM traffic, the new optimal VNF placement is computed by VNF placement algorithm (i.e., Algo. 2) to minimize the total communication cost among VM pairs. Note that some VNFs could have the same current and new location such as vnf_2. However, to migrate each VNF towards its new placed location to decrease the total communication cost, the total migration cost increases.

Therefore, Algo. 4 attempts to find all the Parent fronts to strike a balance between these two costs.

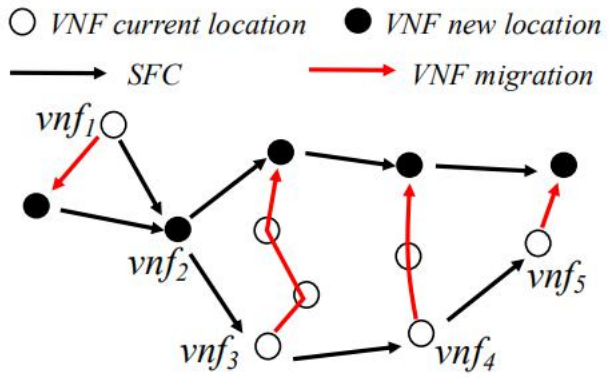


Fig. 7. Illustrating StepWise Algorithm.

We also design Benefit algorithm: It is similar to greedy placement algorithm from Liu [31] *et al*. In this algorithm, we try to migrate one VNF to all possible positions each time to find the minimum total cost of migrate and communicate. The migrating order follows the original SFCs.

Algorithm 5: Benefit Migration Algorithm for PPM.
Input: A PPDC $G(V, E)$ with VM pair placement s(v), $v \in V$, and VNF placement $p(j)$, $1 \leqq j \leqq n$, migrate coefficient μ, traffic rate vector $\vec{\lambda}$;
Notations: *pareto*: an array of n elements;
pareto = [$p(1)$, $p(2)$, ..., $p(n)$]; // initial VNF placement
pareto|{j, a}} = [$p(1)$, ..., $p(j \mid 1)$, a, $p(j + 1)$, ..., $p(n)$] is
pareto with its j *th* element being replaced by a;
Output: The best VNF migration m and the total cost $C_t(m)$.

0. Cm(m) = Cc(m) = ; Ct(m) = ∞;
1. **for** ($1 \leqq j \leqq n$)
2. **for** ($1 \leqq i \leqq |V|$)
3. **if** ($v_i \notin p(j)$)

4. $pareto|\{j, v_i\} = [p(1), ..., p(j \mid 1), v_i, p(j+1), ..., p(n)]$
5. $C_t(pareto) = C_c(pareto) + C_m(pareto)$;
6. **if** ($C_t(pareto) < C_t(m)$)
7. $C_t(m) = C_t$(pareto); $m = pareto$;
8. **end if;**
9. **end if;**
10. **end for;**
11. **end for;**
12. **RETRUN** m and $C_t(m)$

3) *State-of-the-Art Tackling Dynamic Traffic*: As our work is the first that migrates VNFs to specifically attack dynamic traffic in cloud data centers, we couldn't find any existing VNF migration work for meaningful comparison. Two recent research viz. PLAN [16] and PAM [43] instead proposed migrating communicating VMs to reduce dynamic traffic. That is, given the dynamic traffic rates among VM pairs and the placement of VNFs inside the data center, they propose to migrate VMs in order to minimize the sum of total communication cost among VMs and the total migration of of VMs. Their main idea is to migrate VM pairs with high traffic rates closer to SFCs by migrating source VM of each pair closer to the ingress switch of the SFC and destination VM closer to the egress switch. Below we briefly review these two VM migration techniques.

PLAN [16] is a heuristic algorithm that works in rounds. In each round, it computes that which VM is migrated to which PM with available resources, such that the *utility* of this migration is the maximum among all the VMs that have not been migrated.The utility of migrating a VM is defined as the reduction of the VM's communication cost minus its migration cost. This continues until all the VMs are migrated, or no more VM migration gives any positive utility. The goal of PLAN is to find a migration scheme that

maximizes the total utility of migrating all the VMs. PAM [43] showed that minimizing the total communication and migration cost of VMs is equivalent to minimum cost flow problem , which can be solved optimally and efficiently. As such, it is shown that PAM performs better than that PLAN.

In contrast, we observe that migrating one VM pair does not affect communication cost of other pairs while migrating one VNF affects all VM pairs traversing it. As such, VNF migration is more influential on reducing dynamic network traffic than VM migration. We show in the experiments that VNF migration outperforms the existing VM migration traffic-mitigation techniques by 3~4 times.

6. PERFORMANCE EVALUATION

Simulation Setup. We compare our algorithms with existing work. For *k*-stroll problem (i.e., PPP-1), we refer to the our DP algorithm (i.e., Algo. 1) as **DP-Stroll**. We compare them with the $2 + \epsilon$ performance guarantee (i.e., two times of the Optimal) by the primal-dual algorithm, referred to as **PrimalDual**. For the general VNF placement (i.e., PPP), we refer to our DP-based algorithm (Algo. 2) as **DP** and compare it with **Steering** [48], a seminal MB placement algorithm, and **Greedy**, a greedy algorithm proposed in [31]. For VNF migration, we also compare **StepWise** with two latest VM migration techniques viz. PLAN [16] and PAM [43]. For all the cases, we refer to the ILP-based optimal algorithm (e.g., ILP (A)) as **Optimal**. They are summarized in Table II.

TABLE II
SUMMARY OF COMPARED ALGORITHMS.

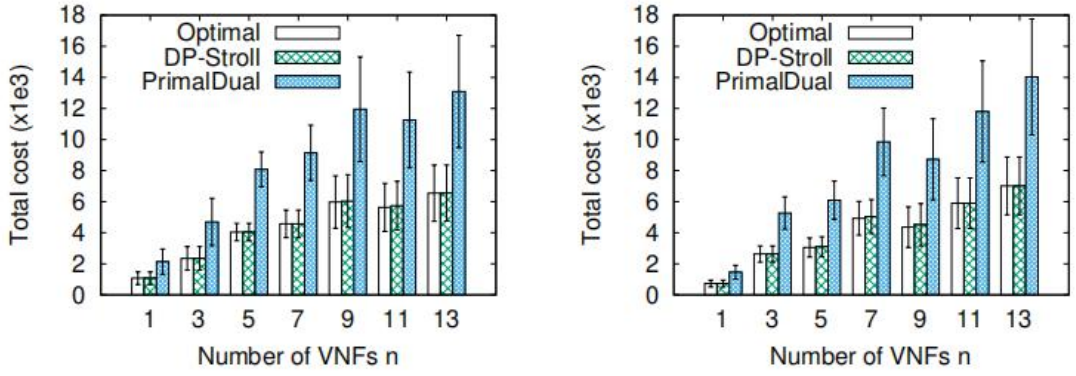
	Our solutions	Existing work
k-stroll	DP-Stroll, Optimal	PrimalDual [11]
VNF placement	DP, Optimal	Steering [48], Greedy [31]
VNF migration	StepWise, Optimal	PLAN [16], PAM [43]

We consider fat-tree PPDCs of size $k = 4$ with 16 PMs, $k = 8$ with 128 PMs, and $k = 16$ with 1024 PMs. As 80% of cloud data center traffic originated by servers stays within the rack [9]. we place 80% of the VM pairs into PMs under the same edge switches. Recent flow characteristics found in Facebook data centers [40] show that 25% of flows sends less than 1 KB and lasts less than a second, 70% sends less than 10 KB and lasts less than 10 seconds, and less than 5% of the flows are larger than 1 MB or last longer than100 seconds. We therefore assume the traffic rates of VM pairs are in the range of [0, 1000], and 25% of VM pairs have light traffic rates in [0, 300), 70% medium traffic rates in [300, 700], and 5% heavy rates in (700,1000]. Each data point in the plots is an average of 20 runs with 95% confidence interval.

SFC Use Cases [2]. SFCs in a real-world data center are broadly categorized into two types [2]. Access SFCs include segment firewalls and deep packet inspectors that perform stateful inspection of traffic and policy identification, and web optimization control that optimizes the link bandwidth usage. Application SFCs include application-specific firewalls and application delivery controllers that distribute traffic across a pool of application servers. As real-world SFCs could have 5 to 6 access

functions and 4 to 5 application function in a typical SFC [2], we consider up to 13 VNFs in a SFC.

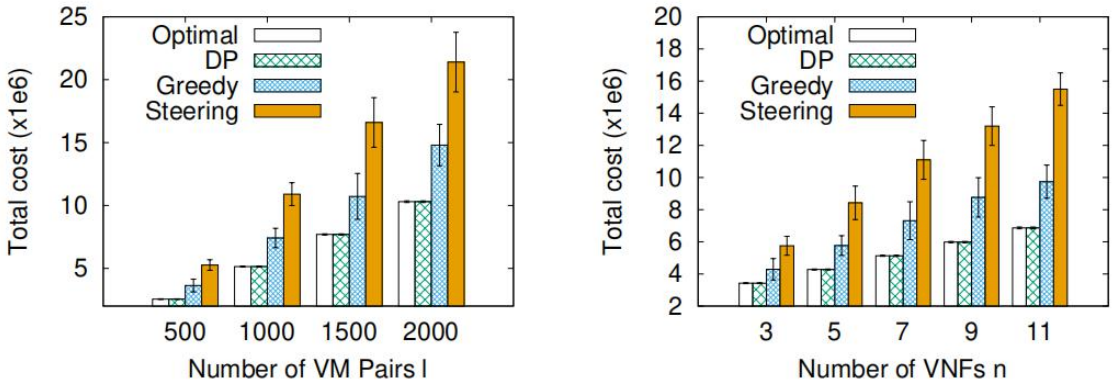
***k*-stroll (i.e., PPP-1) Algorithms.** Fig. 8 compares *k*-stroll (i.e., PPP-1 for one VM pair) algorithms viz. Optimal, PrimalDual, and DP-Stroll in $k = 4$ and $k = 8$ data centers by varying number of VNFs l required for this VM pair to traverse. It shows that DP-Stroll performs very close to Optimal, yielding only 4% of more cost, and outperforms the $2 + \epsilon$ performance guarantee of PrimalDual. With the increase of l, the communication cost of this VM pair increases as its traffic needs to traverses more VNFs. However, its cost decreases when l increases from 9 to 11 in Fig. 8(a) and from 7 to 9 in Fig. 8(b). The reason is that the communication frequencies of this VM pair are in general get smaller (due to randomness) for those two transitions.



(a) Varying n, $k = 4$. (b) Varying n, $k = 8$.

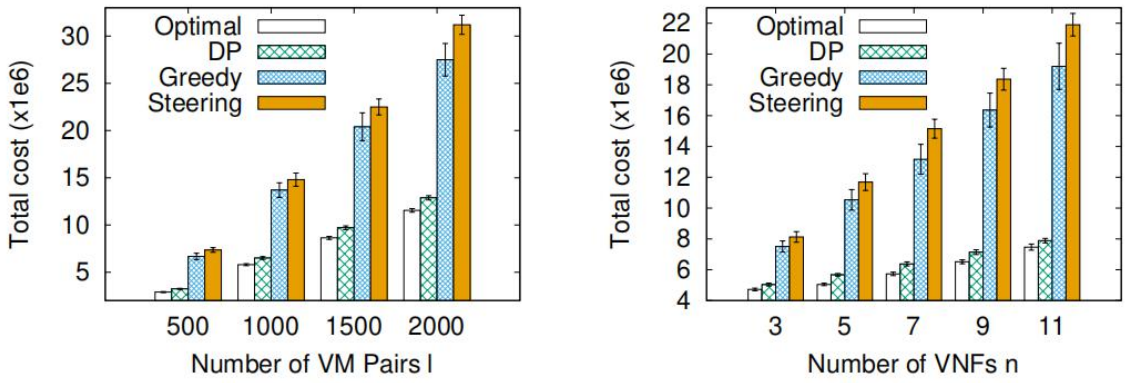
Fig.8. Comparing *k*-stroll algorithms, $l = 1$.

VNF Placement (i.e. PPP) Algorithms. Fig. 9 compares Optimal, DP, Greedy, and Steering in k = 8 fat tree data centers. Fig. 9(a) varies number of VM pairs l from 500, 1000, 1500, to 2000 while fixing number of VNFs n = 7 whereas Fig. 9(b) varies n while fixing l. Our data shows that DP performs the same as Optimal does, and both outperform Greedy, which outperforms Steering. For fair comparison Fig. 10 further adopts the same parameter settings used in Greedy [31] that the link delays are a uniform distribution with a mean value of 1.5 ms and variance of 0.5 ms. Under this setting, DP yields between 5.8% and 12.4% more costs than that of the Optimal in Fig. 10(a) and between 11.5% and 15.5% more costs than that of Optimal in Fig. 10(b). Both however, outperform the other two by three to four times.



(a) Varying l, $n = 7$. (b) Varying n, $l = 1000$.

Fig. 9. Comparing VNF placement, k = 8.



(a) Varying l, $n = 7$. (b) Varying n, $l = 1000$.

Fig. 10. Comparing VNF placement with time delays, $k = 8$.

Effects of VNF Migrations.

Fig. 11 shows the performance comparison of Benefit, MCF, and PLAN by varying migration coefficient μ. It shows that Benefit performs three or four times better than that of the other two algorithms. MCF performs better than PLAN, as it is an optimal VM migration algorithm.

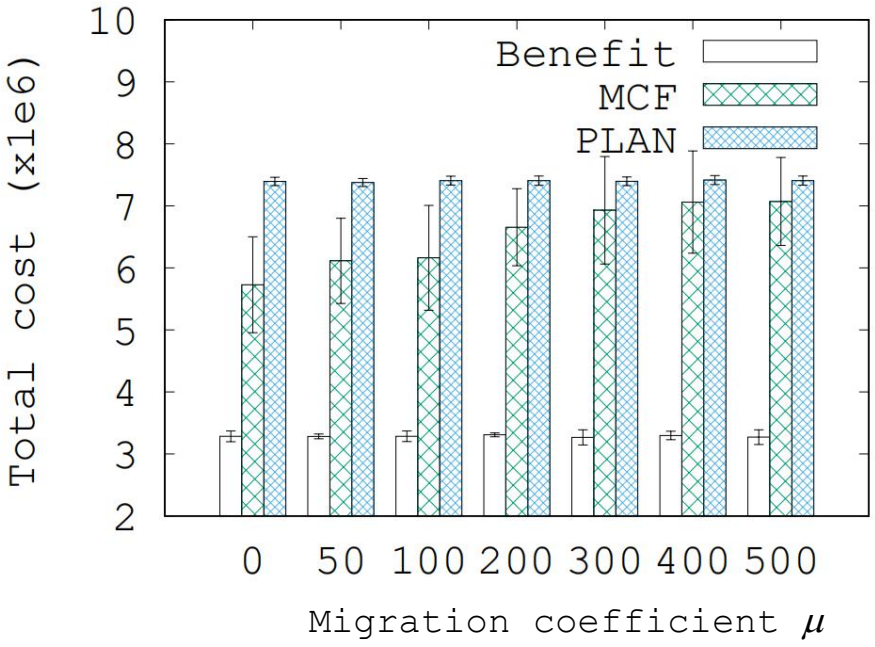


Fig. 11. Comparison between Benefit, MCF and PLAN algorithms

The dynamic traffic in data centers is generally cycle-stationary that exhibits strong diurnal patterns [18]. We thus adopt cycle-stationary traffic pattern with N stationary intervals (N = 12 is the typical value for daily traffic) where the bandwidth of the SFC is modulated by the scale factors in the h^{th} interval (h = 0, ..., N - 1) chosen according to the classical sinusoidal trend and given by the following expression:

$$\tau_h = \begin{cases} 1 & h = 0, \\ 1 - 2\frac{h}{N}(1 - \tau_{min}) & h = 1, ..., \frac{N}{2}, \\ 1 - 2\frac{N-h}{N}(1 - \tau_{min}) & h = \frac{N}{2} + 1, ..., N - 1. \end{cases} \quad (17)$$

Fig. 12 shows the performance comparison under dynamic scenarios. Besides above cycle-stationary traffic pattern, we also notice that during one day's period, the east coast is three hours ahead than west coast, thus the VMs submitted by users in the east coast always started earlier in terms of increasing or decreasing their traffic. We thus assume half of the VM pairs in the data center belong to the users of east coast and the other half west coast. It shows that StepWise performs the same as the Optimal under some scenarios, and both outperform PLAN and MCF by at least two to three times.

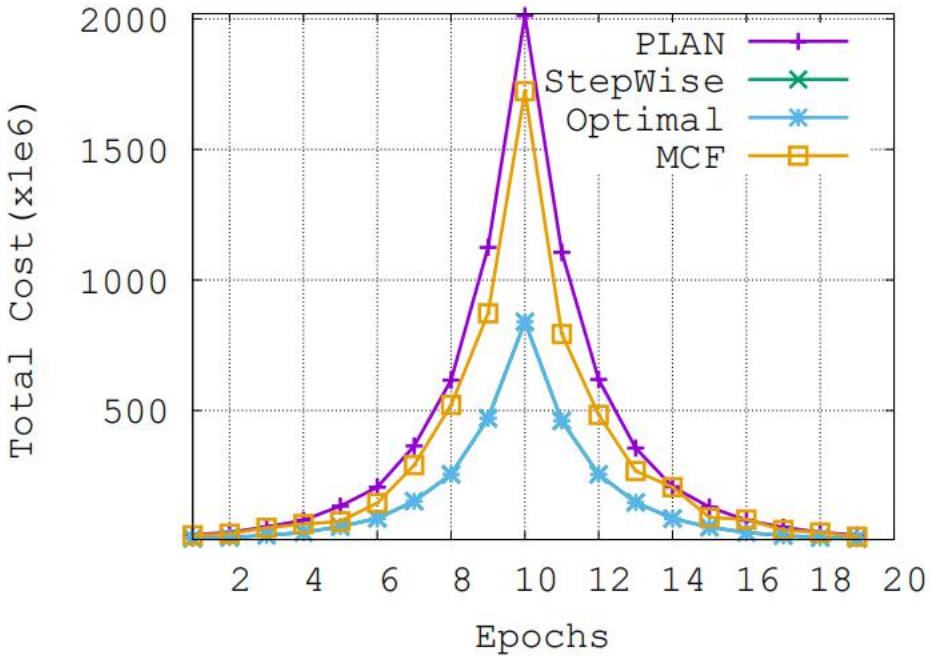


Fig. 12. Comparing with MCF and PLAN.

7. CONCLUSIONS AND FUTURE WORK

In this report we proposed a new integrated framework of VNF placement and migration in PPDCs. It targets diverse and dynamic VM traffic that commonly exists in cloud data centers. It consists of two problems viz. PPP, which places VNFs into PPDCs

according to the diverse traffic rates of communicating VM pairs, and PPM, which then adaptively migrates VNFs inside PPDCs in response of changing traffic rates of the VM pairs. We designed optimal, approximation, and heuristic algorithms to solve them. Working together, PPP and PPM are able to achieve resource optimization for a PPDC's lifetime. We believe PPP and PPM are theoretically fundamental problems as they generalize or vary the well-know *p*-median problem and dynamic facility location problem. Because of these theoretical roots, the algorithms proposed in this report could be applicable to any applications that address SFC-based communication in dynamic traffic environments such as edge computing or mobile crowd sourcing. In addition, it provides an algorithmic framework that can be further improved and augmented by considering more parameters in SFCs such as capacities of network nodes and edges. For example, to address the capacity constraint of both network nodes and VNFs, we assume that each VNF is located on a different node. In the future we will consider that each node can store multiple VNFs thus how to consolidate VNF instances while respecting resource constraints of each node becomes an interesting problem.

8. REFERENCES

[1] Cisco global cloud index: Forecast and methodology, 2016 to 2021 white report. https://www.cisco.com/c/en/us/solutions/service-provider/global-cloud index-gci/white-report-listing.html.

[2] Service function chaining use cases in data centers (ietf). https://tools.ietf.org/html/draft-ietf-sfc-dc-use-cases-06section-3.3.1.

[3] Zoom cloud meetings. https://zoom.us/.

[4] Zoom meeting connector core concepts. https://support.zoom.us/hc/en-us/articles/201363113-Meeting-Connector-Core-Concepts.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, 2008.

[6] F. Bari, S. R. Chowdhury, R. Ahmed, R. Boutaba, and O. C. M. B. Duarte. Orchestrating virtualized network functions. *IEEE Transactions on Network and Service Management*, 13(4):725–739, 2016.

[7] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba. On orchestrating virtual network functions. In *Proc. of the 2015 11th International Conference on Network and Service Management (CNSM)*.

[8] M. Bateni and J. Chuzhoy. Approximation algorithms for the directed k-tour and k-stroll problems. In *Proc. of APPROX/RANDOM 2010*, 2010.

[9] T. Benson, A. Akella, and D. A. Maltz. Network traffific characteristics of data centers in the wild. In *Proc. of ACM IMC 2010*.

[10] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, and H. A. Chan. Optimal virtual network function placement in multi-cloud service function chaining architecture. *Computer Communications*, 102:1 – 16, 2017.

[11] K. Chaudhuri, B. Godfrey, S. Rao, and K. Talwar. Paths, trees, and minimum latency tours. In *Proc. of IEEE FOCS 2003*.

[12] C. Chekuri, N. Korula, and M. Pal. Improved algorithms for orienteering and related problems.

[13] Y. Chen, J. Wu, and B. Ji. Virtual network function deployment in tree-structured networks. In *Proc. of ICNP 2018*.

[14] C. Clark, K. Fraser, and S. Hand. Live migration of virtual machines. In *Proc. of NSDI 2005*.

[15] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz. Near optimal placement of virtual network functions. In *Proc. of IEEE INFOCOM 2015*.

[16] L. Cui, F. P. Tso, D. P. Pezaros, W. Jia, and W. Zhao. Plan: Joint policy- and network-aware vm management for cloud data centers. *IEEE Transactions on Parallel and Distributed Systems*, 28(4):1163–1175, 2017.

[17] R. Cziva and D. P. Pezaros. Container network functions: bringing nfv to the network edge. *IEEE Communications Magazine*, 55(6):24–31, 2017.

[18] V. Eramo, E. Miucci, M. Ammar, and F. G. Lavacca. An approach for service function chain routing and virtual function network instance migration

in network function virtualization architectures. *IEEE/ACM Transactions on Networking*, 25(4):2008–2025, 2017.

[19] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *Proc. of USENIX NSDI 2014*.13

[20] N. Garg. Saving an : A 2-approximation for the k-mst problem in graphs. In *Proc. of ACM STOC 2005*.

[21] S. Gu, Z. Li, C. Wu, and C. Huang. An effificient auction mechanism for service chains in the nfv market. In *Proc. of IEEE INFOCOM 2016*.

[22] A. Gushchin, A. Walid, and A. Tang. Scalable routing in sdn-enabled networks with consolidated middleboxes. In *Proc. of ACM Hotmiddlebox*, 2015.

[23] H. Hantouti, N. Benamar, T. Taleb, and A. Laghrissi. Traffific steering for service function chaining. *IEEE Communications Surveys Tutorials*, 21(1):487–

507, 2019.

[24] H. Huang, S. Guo, J. Wu, and J. Li. Service chaining for hybrid network function. *IEEE Transactions on Cloud Computing*, 7:1082–1094, 2019.

[25] M. Huang, W. Liang, Y. Ma, and S. Guo. Maximizing throughput of delay-sensitive nfv-enabled request admissions via virtualized network function

placement. *IEEE Transactions on Cloud Computing*, 2019.

[26] B. Jaumard and H. Pouya. Migration plan with minimum overall migration time or cost. *J. Opt. Commun. Netw.*, 10:1 – 13, 2018.

[27] M. Jia, W. Liang, M. Huang, Z. Xu, and Y. Ma. Routing cost minimization and throughput maximization of nfv-enabled unicasting in software-defifined

networks. *IEEE Transactions on Network and Service Management*, 15(2):732–745, 2018.

[28] D. A. Joseph, A. Tavakoli, and I. Stoica. A policy-aware switching layer for data centers. In *Proc. of ACM SIGCOMM 2008*.

[29] A. Laghrissi and T. Taleb. A survey on the placement of virtual resources and virtual network functions. *IEEE Communications Surveys Tutorials*,

21(2):1409–1434, 2019.

[30] X. Li and C. Qian. A survey of network function placement. In *IEEE CCNC 2016*.

[31] J. Liu, Y. Li, Y. Zhang, L. Su, and D. Jin. Improve service chaining performance with optimized middlebox placement. *IEEE Transactions on Services*

Computing, 10(4):560–573, 2017.

[32] J. Liu, W. Lu, F. Zhou, P. Lu, and Z. Zhu. On dynamic service function chain deployment and readjustment. *IEEE Transactions on Network and Service*

Management, 14(3):543–553, 2017.

[33] W. Ma, J. Beltran, D. Pan, and N. Pissinou. Traffific aware placement of interdependent nfv middleboxes. *IEEE Transactions on Network and Service Management*, 16(4):1303–1317, Dec. 2019.

[34] V. Mann, A. Gupta, P. Dutta, A. Vishnoi, P. Bhattacharya, R. Poddar, and A. Iyer. Remedy: Network-aware steady state vm management for data centers. In *Proc. of the NETWORKING 2012*.

[35] E. Melachrinoudisa and H. Min. The dynamic relocation and phase-out of a hybrid, two-echelon plant/warehousing facility: A multiple objective approach. *European Journal of Operational Research*, 123(1):1 – 15, 2000.

[36] X. Meng, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffific-aware virtual machine placement. In *Proc. of IEEE INFOCOM 2010*.

[37] R. Mijumbi, J. Serrat, J. L. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys and Tutorials*, 18(1), 2015.

[38] J. Reese. Solution methods for the p-median problem: An annotated bibliography. *Networks*, 48(3):125–142, 2006.

[39] Amber Rae Richte. *Dynamic Facility Relocation and Inventory Management for Disaster Relief*. Ph.D. Thesis, UC Berkeley, 2016.

[40] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proc. of ACM SIGCOMM 2015*.

[41] Y. Sang, B. Ji, G. R. Gupta, X. Du, and L. Ye. Provably effificient algorithms for joint placement and allocation of virtual network functions. In *Proc. of INFOCOM 2017*.

[42] L. Tang, X. He, P. Zhao, G. Zhao, Y. Zhou, and Q. Chen. Virtual network function migration based on dynamic resource requirements prediction. *IEEE Access*, 7:112348–112362, 2019.

[43] H. Flores V. Tran and B. Tang. Pam & pal: Policy-aware virtual machine migration and placement in dynamic cloud data centers. In *Proc. of IEEE INFOCOM 2020*.

[44] X.Fei, F. Liu, H. Xu, and H. Jin. Adaptive vnf scaling and flflow routing with proactive demand prediction. In *Proc. of IEEE INFOCOM 2018*.

[45] B. Yi, X. Wang, M. Huang, and A. Dong. A multi-criteria decision approach for minimizing the inflfluence of vnf migration. *Computer Networks*, 159:51– 62, 2019.

[46] F. Zhang, G. Liu, X. Fu, and Ramin Yahyapour. A survey on virtual machine migration: Challenges, techniques, and open issues. *IEEE Communications Surveys & Tutorials*, 20:1206–1243, 2018.

[47] X. Zhang, C. Wu, Z. Li, and F. C.M. Lau. Proactive vnf provisioning with multi-timescale cloud resources: Fusing online learning and online optimization. In *Proc. of IEEE INFOCOM 2017*.

[48] Y. Zhang, N. Beheshti, L. Beliveau, G. Lefebvre, R. Manghirmalani, R. Mishra, R. Patney, M. Shirazipour, R. Subrahmaniam, C. Truchan, and M. Tatipamula. Steering: A software-defifined networking for inline service chaining. In *Proc. of IEEE ICNP 2013*.

9. APPENDIX

9.1. Data Center for FatTree Topology

```
/**
 * <h1>FatTree Topology </h1>
 * This is the data center for FatTree Topology to store the structure of the FatTree
 * and other data
 * @author sunjingsong
 * @version 1.0
 * @since 8-3-2020
 */
public class FatTree {
    // Data fields
    // The structure of the tree
    public ArrayList<String> coreSwitches; // core switches
    public ArrayList<String> aggrSwitches; // aggregation switches
    public ArrayList<String> edgeSwitches; // edge switches
    public ArrayList<String> allSwitches; // allSwitches = core + aggregation +
edge
    public ArrayList<String> allPositions; // core + aggregation + edge + hosts
    public ArrayList<String> pods; // pods
    public ArrayList<String> hosts; // physical machines
    public HashMap<String, HashSet<String>> links; // the link between two nodes
    public int numberOfPorts;   // number of ports
    public int numberOfPods;    // Pod number in Fat Tree
    public int coreSwitchNumber; // The number of core switches
    public int aggrSwitchNumber; // The number of aggregation switches
    public int edgeSwitchNumber; // The number of edge switches
    public int hostNum;         // The number of hosts
    public int numberOfPairs;
    public int numberOfVNFs;
    public double minEnd = Double.MAX_VALUE;

    // The services of the tree
    public ArrayList<int[]> pairs;  // the communication pairs
    public int[] VNFs;   // middle boxes
```

```
    public double migrateCoefficient;   // migrate coefficient
    public int totalFrequency;      // the total communication frequency along the
VNFs chain

    public HashMap<Integer, Double> start; // {Possible first VNF position: total
cost of hosts->1st VNF}
    public HashMap<Integer, Double> end;   // {Possible last VNF position: total
cost of last VNF->hosts}
    public HashMap<String, Double> distance; // {node1 + node2: the distance of
two nodes}
    public double[][] costTable;
    public int[] optimalVNFs;
    public int[] switchesTable;
    public HashMap<Integer, String> indexNodeMap;
    public HashMap<String, Integer> nodeIndexMap;
    public HashMap<String, String> hostEdgeMap;
    public HashSet<String> left;
    public HashSet<String> right;
    public int operation; // placement and migration
    public int maxFrequency;

    /** Initialize the topology */
    public FatTree(int operation, int numberOfPorts, int numberOfPairs, int
numberOfVNFs, int freq) {
        this.operation = operation;
        this.numberOfPorts = numberOfPorts;
        this.numberOfPairs = numberOfPairs;
        this.numberOfVNFs = numberOfVNFs;
        this.maxFrequency = freq;
        // Make operations
        setOperation();
        // Initial variables
        initialVariables();
        createTopo(); // Create a topology tree
        generate_VM_pairs(); // Generate VM pairs
        Distance ds = new Distance(this); // Calculate the distance
        generateVNFs(); // Generate VNFs
        // setMigrationCoefficient(); // set migration coefficient
    }

    /**
```

```
     * Initializing the FatTree Data
     */
    public void initialVariables() {
        this.coreSwitches  = new ArrayList(); // core switches
        this.aggrSwitches  = new ArrayList();// aggregation switches
        this.edgeSwitches  = new ArrayList(); // edge switches
        this.allSwitches   = new ArrayList(); // core + aggregation + edge
        this.allPositions  = new ArrayList(); // core + aggregation + edge + hosts
        this.pods          = new ArrayList(); // pods
        this.hosts         = new ArrayList(); // physical machines
        this.links         = new HashMap<>(); // the link between two nodes
        this.indexNodeMap  = new HashMap();
        this.nodeIndexMap  = new HashMap();
        this.hostEdgeMap   = new HashMap();
        this.left = new HashSet();
        this.right = new HashSet();
        // The services of the tree
        this.migrateCoefficient = 0.0;   // migrate coefficient
        this.totalFrequency     = 0;      // the total communication frequency
along the VNFs chain
        this.maxFrequency = 1000;
        start             = new HashMap(); // {Possible first VNF position: total
cost of hosts->1st VNF}
        end               = new HashMap();   // {Possible last VNF position: total
cost of last VNF->hosts}
        distance          = new HashMap(); // {node1 + node2: the distance of two
nodes}
    }

    /** Set the operation: placement and migration */
    public void setOperation() {
    }

    /** Create a fat tree based on the number of ports from user input */
    public void createTopo() {


        int k = this.numberOfPorts;

        this.numberOfPods = k;
        this.coreSwitchNumber = (int)Math.pow(k/2, 2);
```

```
        this.aggrSwitchNumber = (int)(k * k / 2);
        this.edgeSwitchNumber = (int)(k * k / 2);
        this.hostNum          = (int)(k * Math.pow(k / 2, 2));

        // Generate core switches
        for (int i = 0; i < this.coreSwitchNumber; i++) {
            this.coreSwitches.add("cs" + i);
            this.allSwitches.add("cs" + i);
            this.allPositions.add("cs" + i);
        }

        // Traversal each pod
        for (int j = 0; j < this.numberOfPods; j++) {
            this.pods.add("po" + j);
            // Aggregation switches
            for (int l = 0; l < (int)(this.aggrSwitchNumber / this.numberOfPods);
l++) {
                String aggrSwitch = "as" + j + "_" +  l;
                this.aggrSwitches.add(aggrSwitch);
                this.allSwitches.add(aggrSwitch);
                this.allPositions.add(aggrSwitch);
                // Each pod contains k / 2 aggregation switches
                for (int m =  (int)(k * l / 2); m < (int)(k * (l + 1) / 2); m++)
{
                    links.putIfAbsent(aggrSwitch, new HashSet<>());
                    links.putIfAbsent(this.coreSwitches.get(m), new
HashSet<>());
                    links.get(aggrSwitch).add(this.coreSwitches.get(m));
                    links.get(this.coreSwitches.get(m)).add(aggrSwitch);
                }
            }

            // Edge switches
            for (int l = 0; l < (int)(this.edgeSwitchNumber / this.numberOfPods);
l++) {
                String edgeSwitch = "es" + j + "_" +  l;
                this.edgeSwitches.add(edgeSwitch);
                this.allSwitches.add(edgeSwitch);
                this.allPositions.add(edgeSwitch);
                // Each pod contains k / 2 edge switches
                for (int m =  (int)(this.edgeSwitchNumber * j / this.numberOfPods);
```

```
                    m < (int)(this.edgeSwitchNumber * (j + 1) /
this.numberOfPods); m++) {
                links.putIfAbsent(edgeSwitch, new HashSet<>());
                links.putIfAbsent(this.aggrSwitches.get(m), new
HashSet<>());
                links.get(edgeSwitch).add(this.aggrSwitches.get(m));
                links.get(this.aggrSwitches.get(m)).add(edgeSwitch);
            }

            // Physical Machines
            for (int m = 0; m < (int)(this.hostNum / this.numberOfPods /
(this.edgeSwitchNumber / this.numberOfPods));
                    m++){
                String host = "pm" + j + "_" + l + "_" + m;
                this.hosts.add(host);
                this.allPositions.add(host);
                links.putIfAbsent(edgeSwitch, new HashSet<>());
                links.putIfAbsent(host, new HashSet<>());
                links.get(edgeSwitch).add(host);
                links.get(host).add(edgeSwitch);
                this.hostEdgeMap.put(host, edgeSwitch);


            }
        }
    }

    int index = 0;
    for (String pos: this.allPositions) {
        this.indexNodeMap.put(index, pos);
        this.nodeIndexMap.put(pos, index);
        index++;
    }

    index = 0;
    this.switchesTable = new int[this.allSwitches.size()];
    for (String swit: this.allSwitches) {
        this.switchesTable[index++] = nodeIndexMap.get(swit);
    }
```

```
        this.costTable = new
double[this.allPositions.size()][this.allPositions.size()];


    }

    /**
     * Generate VM pairs and their frequency. The VM pairs following the pattern
80% under
     * same edge switch. Their frequency follows Facebook's pattern
     */
    public void generate_VM_pairs() {
        this.pairs = new ArrayList();
        int size = this.hosts.size();
        int num = numberOfPairs;
        int number25 = (int)(num * 0.25);
        int number80 = (int)(num * 0.80);
        int number95 = (int)(num * 0.95);
        int totalNumber = 0;
        Random rand = new Random();

        while (num > 0) {
            totalNumber += 1;
            int i = rand.nextInt(size - 1);
            String host1 = this.hosts.get(i);
            int j = rand.nextInt(size - 1);
            String host2 = this.hosts.get(j);

            if (numberOfPairs >= 5) {
                if (totalNumber <= number80) {
                    i = rand.nextInt(size - 1);
                    host1 = this.hosts.get(i);
                    int index = rand.nextInt(this.numberOfPorts / 2);
                    host2 = host1.substring(0, host1.lastIndexOf('_'))
                            + '_' + index;
                } else {
                    i = rand.nextInt(size - 1);
                    host1 = this.hosts.get(i);
                    ArrayList<String> str = new ArrayList();

                    for (String sw: this.edgeSwitches) {
```

```
            if (sw.equals(host1.substring(0, host1.lastIndexOf('_')))) {
                continue;
            }
            str.add(sw);
            }

            j = rand.nextInt(str.size());
            int index = rand.nextInt(this.numberOfPorts / 2);
            host2 = "pm" + str.get(j).substring(2) + "_" + index;
        }
    } else {
        int prob = rand.nextInt(100) + 1;
        if (prob <= 80) {
            i = rand.nextInt(size - 1);
            host1 = this.hosts.get(i);
            int index = rand.nextInt(this.numberOfPorts / 2);
            host2 = host1.substring(0, host1.lastIndexOf('_'))
                    + '_' + index;
        } else {
            i = rand.nextInt(size - 1);
            host1 = this.hosts.get(i);
            ArrayList<String> str = new ArrayList();

            for (String sw: this.edgeSwitches) {
            if (sw.equals(host1.substring(0, host1.lastIndexOf('_')))) {
                continue;
            }
            str.add(sw);
            }

            j = rand.nextInt(str.size());
            int index = rand.nextInt(this.numberOfPorts / 2);
            host2 = "pm" + str.get(j).substring(2) + "_" + index;
            System.out.println(host2);
        }
    }


    int k = this.maxFrequency / 10;
    int frequency = 0;
    if (numberOfPairs >= 5) {
```

```
                if (totalNumber <= number25) {
                    frequency = rand.nextInt(3 * k);
                    this.totalFrequency += frequency;
                } else if (totalNumber <= number95) {
                    frequency = rand.nextInt(7 * k  - 3 * k) + 3* k;
                    this.totalFrequency += frequency;
                } else {
                    frequency = rand.nextInt(10 * k + 1 - 7 * k) + 7 * k;
                    this.totalFrequency += frequency;
                }
            } else {
                int prob = rand.nextInt(100) + 1;
                if (prob <= 25) {
                    frequency = rand.nextInt(3 * k);
                    this.totalFrequency += frequency;
                } else if (prob <= 95) {
                    frequency = rand.nextInt(7 * k - 3 * k) + 3 * k;
                    this.totalFrequency += frequency;
                } else {
                    frequency = rand.nextInt(10 * k + 1 - 7 * k) + 7 * k;
                    this.totalFrequency += frequency;
                }
            }
            this.pairs.add(new int[]{this.nodeIndexMap.get(host1),
this.nodeIndexMap.get(host2), frequency});
            num -= 1;
        }

        int i;
    for (i = 0; i < this.hosts.size() / 2; i++) {
        this.left.add(this.hosts.get(i));
    }

    for (i = this.hosts.size() / 2; i < this.hosts.size(); i++) {
        this.right.add(this.hosts.get(i));
    }
    }

    /**
     * Update VM pairs randomly
     */
```

```
public void update_VM_pairs_Frequency() {
    this.totalFrequency = 0;
    int num = numberOfPairs;
    int number25 = (int)(num * 0.25);
    int number80 = (int)(num * 0.80);
    int number95 = (int)(num * 0.95);
    int totalNumber = 0;
    Random rand = new Random();
    HashSet<Integer> set = new HashSet();
    for (int i = 0; i < numberOfPairs; i++) {
        set.add(i);
    }
    int size = numberOfPairs;
    int idx = Integer.MAX_VALUE;
    while (num > 0) {
        totalNumber += 1;
        int r = rand.nextInt(size);
        int n = 0;
        for (int it: set) {
            if (r == n) {
                idx = it;
                set.remove(it);
                size--;
                break;
            }
            n++;
        }


        int frequency = 0;
        int k = this.maxFrequency / 10;
        if (numberOfPairs >= 5) {
            if (totalNumber <= number25) {
                frequency = rand.nextInt(3 * k);
                this.totalFrequency += frequency;
            } else if (totalNumber <= number95) {
                frequency = rand.nextInt(7 * k - 3 * k) + 3 * k;
                this.totalFrequency += frequency;
            } else {
                frequency = rand.nextInt(10 * k + 1 - 7 * k) + 7 * k;
                this.totalFrequency += frequency;
```

```
                }
            } else {
                int prob = rand.nextInt(100) + 1;
                if (prob <= 25) {
                    frequency = rand.nextInt(3 * k);
                    this.totalFrequency += frequency;
                } else if (prob <= 95) {
                    frequency = rand.nextInt(7 * k - 3 * k) + 3 * k;
                    this.totalFrequency += frequency;
                } else {
                    frequency = rand.nextInt(10 * k + 1 - 7 * k) + 7 * k;
                    this.totalFrequency += frequency;
                }
            }
            int h1 = this.pairs.get(idx)[0];
            int h2 = this.pairs.get(idx)[1];
            this.pairs.set(idx, new int[] {h1, h2, frequency});
            num -= 1;
        }

        System.out.println("After Changing the frequency");
        String str = "";
        str += "=================================================\n";
        str += "The VM pairs shows as below(Frequency 0~300: 25%; 300~700: 70%;
700~1000: 5%):\n";
        for (int[] obj: this.pairs) {
            str += "[" + this.indexNodeMap.get(obj[0]) +  ", " +
this.indexNodeMap.get(obj[1]) +
                    ", Frequency: " + obj[2] + "]\n";
        }

        str += "=================================================\n";
        System.out.println(str);

    }

    /** Update VM pairs in the report with 12 epoch
     * @param epoch
     * @param cycle count the epoch
     */
    public void update_VM_pairs_Frequency(int epoch, int cycle) {
```

```
System.out.println("epoch " + epoch);
if (epoch == 0) {
    for (int i = 0; i < this.pairs.size(); i++) {
        int h1 = this.pairs.get(i)[0];
        int h2 = this.pairs.get(i)[1];
        String host1 = this.indexNodeMap.get(h1);
        String host2 = this.indexNodeMap.get(h2);
        int oldFreq = this.pairs.get(i)[2];
        int freq = oldFreq;
        if (cycle <= 6) {
            freq *= 2;
        } else {
            freq /= 2;
        }
        if (left.contains(host1) || left.contains(host2)) {
            this.pairs.set(i, new int[] {h1, h2, freq});
            this.totalFrequency += (freq - oldFreq);
        }
    }
} else {
    for (int i = 0; i < this.pairs.size(); i++) {
        int h1 = this.pairs.get(i)[0];
        int h2 = this.pairs.get(i)[1];
        String host1 = this.indexNodeMap.get(h1);
        String host2 = this.indexNodeMap.get(h2);
        int oldFreq = this.pairs.get(i)[2];
        int freq = oldFreq;
        if (cycle <= 6) {
            freq *= 2;
        } else {
            freq /= 2;
        }
        if (right.contains(host1) || right.contains(host2)) {
            this.pairs.set(i, new int[] {h1, h2, freq});
            this.totalFrequency += (freq - oldFreq);
        }
    }
}
```

```
        System.out.println("After Changing the frequency");
        String str = "";
        str += "=================================================\n";
        str += "The VM pairs shows as below(Frequency 0~300: 25%; 300~700: 70%;
700~1000: 5%):\n";
        for (int[] obj: this.pairs) {
            str += "[" + this.indexNodeMap.get(obj[0]) +  ", " +
this.indexNodeMap.get(obj[1]) +
                    ", Frequency: " + obj[2] + "]\n";
        }

        str += "=================================================\n";
        System.out.println(str);

    }

    /**
     * Generate VNFs chain randomly
     */
    public void generateVNFs() {
        // randomly generate VNFs based on the user input
        int num = numberOfVNFs;
        this.VNFs = new int[num];
        this.optimalVNFs = new int[num];
        if (num > this.allSwitches.size()) {
            System.out.println("Too many VNFs without enough positions to put");
        }
        int size = this.allSwitches.size();
        HashSet<Integer> used = new HashSet();

        int k = 0; // The index of VNFs
        while (num > 0) {
            Random rand = new Random();
            String tmp = this.allSwitches.get(rand.nextInt(size));
            int index = nodeIndexMap.get(tmp);
            if (!used.contains(index)) {
                this.VNFs[k++] = index;
                used.add(index);
                num -= 1;
            }
        }
```

```
}

public void setMigrationCoefficient(int num) {
    this.migrateCoefficient = num;
}

/**
 * Print the FatTree structure and the data stored in FatTree
 */
public String toString() {
    String str = "=====================================================\n";
    str += "The Structure of the fat tree is: \n";
    int num = 0;
    str += "Core Switches: {\n";
    str += "\t";
    for (String coreSwitch: this.coreSwitches) {
        num++;
        str += coreSwitch + "   ";
        if (num == 10) {
            num = 0;
            str += "\n";
            str += "\t";
        }
    }
    str += "\n}\n\n";

    num = 0;
    str += "PODS: {\n";
    str += "\t";
    for (String pod: this.pods) {
        num++;
        str += pod + "   ";
        if (num == 10) {
            num = 0;
            str += "\n";
            str += "\t";
        }
    }
    str += "\n}\n\n";

    num = 0;
```

```
str += "Aggregation Switches: {\n";
str += "\t";
for (String aggrSwitch: this.aggrSwitches) {
    num++;
    str += aggrSwitch + "   ";
    if (num == 10) {
        num = 0;
        str += "\n";
        str += "\t";
    }
}
str += "\n}\n\n";

num = 0;
str += "Edge Switches: {\n";
str += "\t";
for (String edgeSwitch: this.edgeSwitches) {
    num++;
    str += edgeSwitch + "   ";
    if (num == 10) {
        num = 0;
        str += "\n";
        str += "\t";
    }
}
str += "\n}\n\n";

num = 0;
str += "Physical Machines: {\n";
str += "\t";
for (String host: this.hosts) {
    num++;
    str += host + "   ";
    if (num == 10) {
        num = 0;
        str += "\n";
        str += "\t";
    }
}
str += "\n}\n\n";
```

```
        str += "=================================================\n";
        str += "The VM pairs shows as below(Frequency 0~300: 25%; 300~700: 70%;
700~1000: 5%):\n";
        for (int[] obj: this.pairs) {
            str += "[" + indexNodeMap.get(obj[0]) +  ", " + indexNodeMap.get(obj[1])
+
                    ", Frequency: " + obj[2] + "]\n";
        }

        str += "=================================================\n";

        return str;
    }

}
```

9.2. DP algorithm for placement

```
/**
 * <h1>DP k-stroll Algorithm </h1>
 * The ShortestPathDPA program implements the DP k-stroll algorithm to find the
best
 * Service Function chains with lowest cost. The major part of the algorithm uses
 * a 4-dimension DP to represent different ingress and egress combinations, edges,
 * and cost.
 * This method also uses another 3-DP to store the current VNF locations to prevent
the
 * duplicated edges
 * @author sunjingsong
 * @version 1.0
 * @since 8-3-2020
 */
public class ShortestPathDPA {
    // Data field
    public int operation;
    public Cost cost;
    public int[] originalVNFs;
    public int[] optimalVNFs;
    public double originalCost;
    public double optimalCost;
    public double[][] graph;
```

```
    public double minCostStart = Double.MAX_VALUE;
    public double minCostEnd = Double.MAX_VALUE;


    /**
     * This is a construct method for instantiating object
     * @param cost A model for calculating various cost
     */
    public ShortestPathDPA(Cost cost) {
        this.cost = cost;
    }

    /**
     * This is the entry point method to input FatTree, original VNFs chains,
     * current optimal VNFs chains and their correspond cost
     * Two steps: (1) create adjacency matrix of switches, (2) Start the calculating
     * @param ft FatTree Topology with the data(i.e. VM pairs, VNFs, Frequency,and
switches)
     * @param originalVNFs Initial VNFs chains
     * @param optimalVNFs  Initial Optimal VNFs chains
     * @param originalCost Initial total cost
     * @param optimalCost  Optimal cost
     * @return optimal cost based on DP Algorithm
     */
    public double execute(FatTree ft,  int[] originalVNFs,
        int[] optimalVNFs, double originalCost, double optimalCost) {
        this.originalVNFs = originalVNFs;
        this.optimalVNFs  = optimalVNFs;
        this.originalCost = originalCost;
        this.optimalCost  = optimalCost;
        createGraph(ft);
        shortestPath(ft);
        return this.optimalCost;
    }

    /**
     * This is the method to traversal all possible ingress and egress combinations
     * which is the Algorithm 2 in the report
     * @param ft FatTree with the data related VM pairs and frequency
     */
    public void shortestPath(FatTree ft) {
```

```
        double lowestCost = this.optimalCost;
        int[] bestVNFs = new int[this.optimalVNFs.length];
        for (int i = 0; i < bestVNFs.length; i++) {
            bestVNFs[i] = this.optimalVNFs[i];
        }

        int k = ft.VNFs.length - 1;
        double[][][][] dp = new double[graph.length][graph.length][k + 1][3];
        int[][][][] visited = new int[graph.length][graph.length][k +
1][graph.length];
        for (Map.Entry<Integer, Double> ingress: ft.start.entrySet()) {
            for (Map.Entry<Integer, Double> egress: ft.end.entrySet()) {
                if (ingress.getKey() != egress.getKey()) {
                    ft.VNFs[0] = ingress.getKey();
                    this.optimalVNFs[0] = ingress.getKey();
                    ft.VNFs[ft.VNFs.length - 1] = egress.getKey();
                    this.optimalVNFs[ft.VNFs.length - 1] = egress.getKey();
                    this.minCostStart  = ingress.getValue();
                   this.minCostEnd  = egress.getValue();
                   HashMap<Integer, Integer> map = new HashMap<>();
                for (int i = 0; i < ft.switchesTable.length; i++) {
                    map.put(ft.switchesTable[i], i);
                }

                   int u = map.get(ft.VNFs[0]);
                int v = map.get(ft.VNFs[ft.VNFs.length - 1]);
                if (dp[u][v][k][2] != 0 && dp[u][v][k][2] != Double.MAX_VALUE) {
                    this.optimalCost = dp[u][v][k][2] + this.minCostStart +
                            this.minCostEnd;
                    int i = u;
                    int j = v;
                    int e = k;
                    int idx = 0;
                    ft.VNFs[idx] = ft.switchesTable[i];
                    this.optimalVNFs[idx] = ft.switchesTable[i];
                    idx++;
                    while (e != 1) {
                        i = (int)dp[i][j][e][0];
                        j = (int)dp[i][j][e][1];
                        ft.VNFs[idx] = ft.switchesTable[i];
                        this.optimalVNFs[idx] = ft.switchesTable[i];
```

```
                    idx++;
                    e--;

                }
            } else {
                    DP(ft, dp, visited);
            }

                if (lowestCost > this.optimalCost) {
                    lowestCost = this.optimalCost;
                    for (int i = 0; i < bestVNFs.length; i++) {
                        bestVNFs[i] = this.optimalVNFs[i];
                    }
                }

            }
        }
    }

    this.optimalCost = lowestCost;
    for (int i = 0; i < bestVNFs.length; i++) {
        this.optimalVNFs[i] = bestVNFs[i];
        ft.VNFs[i] = bestVNFs[i];
    }
}

/**
 * The major part for DP Algorithm (Algorithm 1 in the report)
 * @param ft FatTree object
 * @param dp 4-DP array
 * @param visited stored occupied switches
 */
public void DP(FatTree ft, double[][][][] dp, int[][][][] visited) {
    boolean isBiggerThanPrev = false;

    HashMap<Integer, Integer> map = new HashMap<>();
    for (int i = 0; i < ft.switchesTable.length; i++) {
        map.put(ft.switchesTable[i], i);
    }

    if (ft.VNFs.length == 1) {
```

```
            for (int swit: ft.switchesTable) {
                ft.VNFs[0] = swit;
                double minCost = cost.calculateTotalCost(ft);
                if (this.optimalCost > minCost) {
                    this.optimalCost = minCost;
                    this.optimalVNFs[0] = swit;
                }
            }
            ft.VNFs[0] = this.optimalVNFs[0];
            return;
        }

        int u = map.get(ft.VNFs[0]);
        int v = map.get(ft.VNFs[ft.VNFs.length - 1]);
        int k = ft.VNFs.length - 1;
        int len = graph.length;
        for (int e = 1; e <= k; e++) {// edges
            double globalMinVal = Double.MAX_VALUE;
            for (int i = 0; i < len; i++) {// sources
                for (int j = 0; j < len; j++) {// destinations
                    dp[i][j][e] = new double[] {i, j, Double.MAX_VALUE};

                    if (e == 1) {
                        visited[i][j][e][i] = 1;
                        visited[i][j][e][j] = 1;
                        dp[i][j][e][2] = graph[i][j];
                    }

                    if (e > 1) {

                        int minIndex = -1;
                        double minVal = Double.MAX_VALUE;
                        if (dp[i][j][e][2] != 0 && dp[i][j][e][2] !=
Double.MAX_VALUE) {
                            continue;
                        }
                        for (int a = 0; a < len; a++) {
                            if (a != u && a != v && i != a && j != a && dp[i][j][e
- 1][2]
                                    != Double.MAX_VALUE) {
                                if ((minVal > graph[i][a] + dp[a][j][e - 1][2]) &&
```

```
                                    (visited[a][j][e - 1][i] == 0)) {
                                minIndex = a;
                                minVal = graph[i][a] + dp[a][j][e - 1][2];
                            }

                        }
                    }

                    globalMinVal = Math.min(globalMinVal, minVal);
                    if (minIndex != -1) {
                        for (int m = 0; m < visited[i][j][e].length; m++) {
                            visited[i][j][e][m] = visited[minIndex][j][e -
1][m];
                        }
                        visited[i][j][e][i] = 1;
                        dp[i][j][e] =  new double[]{minIndex, j, minVal};
                    }
                }
            }
        }
        if (globalMinVal > this.optimalCost) {
            isBiggerThanPrev = true;
            break;
        }
    }
    if (isBiggerThanPrev) {
        this.optimalCost = Double.MAX_VALUE;
        return;
    }
    this.optimalCost = dp[u][v][k][2] + this.minCostStart + this.minCostEnd;
    int i = u;
    int j = v;
    int e = k;
    int idx = 0;
    ft.VNFs[idx] = ft.switchesTable[i];
    this.optimalVNFs[idx] = ft.switchesTable[i];
    idx++;
    while (e != 1) {
        i = (int)dp[i][j][e][0];
        j = (int)dp[i][j][e][1];
        ft.VNFs[idx] = ft.switchesTable[i];
```

```
            this.optimalVNFs[idx] = ft.switchesTable[i];
            idx++;
            e--;

        }
    }

    /**
     * Create an adjacency matrix between different switches
     * @param ft FatTree ojbect
     */
    public void createGraph(FatTree ft) {
        graph = new double[ft.switchesTable.length][ft.switchesTable.length];
        for (int i = 0; i < ft.switchesTable.length; i++) {
            for (int j = 0; j < ft.switchesTable.length; j++) {
                graph[i][j] =
                  ft.costTable[ft.switchesTable[i]][ft.switchesTable[j]] *
ft.totalFrequency;
            }
        }

    }


}
```

9.3. Exhaustive algorithm for placement and migration

```
/**
 * <h1>Exhaustive Algorithm </h1>
 * The Exhaustive program implements exhaustive algorithm to placing VNFs or
migrating VNFs.
 * This method try each possible VNFs chain and calculate the cost. It is NP-hard.
 * However, our Algorithm makes some optimzing by cutting branches based on current
 * cost and future minimum steps
 * @author sunjingsong
 * @version 1.0
 * @since 8-3-2020
 */
public class Exhaustive {
    // Data Field
    public int operation;
```

```
    public Cost cost;
    public int[] originalVNFs;
    public int[] optimalVNFs;
    public double originalCost;
    public double optimalCost;

    /**
     * This is the construct method for instantiating this model
     * @param operation 0-Placement, 1-Migration
     * @param cost The model for calculating cost
     */
    public Exhaustive(int operation, Cost cost) {
        this.operation = operation;
        this.cost      = cost;
        if (operation == 0) {
            System.out.println("Exhaustive Method To Placement");
        } else {
            System.out.println("Exhaustive Method To Migration");
        }

    }

    /**
     * This is the entry point method to input FatTree, original VNFs chains,
     * current optimal VNFs chains and their correspond cost
     * This method start the exhaustive algorithm
     * @param ft FatTree Topology with the data(i.e. VM pairs, VNFs, Frequency,and
switches)
     * @param originalVNFs Initial VNFs chains
     * @param optimalVNFs  Initial Optimal VNFs chains
     * @param originalCost Initial total cost
     * @param optimalCost  Optimal cost
     * @return optimal cost based on DP Algorithm
     */
    public double execute(FatTree ft,  int[] originalVNFs,
        int[] optimalVNFs, double originalCost, double optimalCost) {
        this.originalVNFs = originalVNFs;
        this.optimalVNFs  = optimalVNFs;
        this.originalCost = originalCost;
        this.optimalCost  = optimalCost;
```

```
        int index = 0;
        HashSet<Integer> used = new HashSet();
        help(ft, index, used);
        for (int i = 0; i < ft.optimalVNFs.length; i++) {
            ft.optimalVNFs[i] = this.optimalVNFs[i];
        }
        return this.optimalCost;
    }

    /**
     * This is the major part for exhaustive algorithm(Optimal solution in report)
     * @param ft FatTree Object
     * @param index indicates the length of VNFs chain
     * @param used stored occupied switches to prevent duplicated placement
     */
    public void help(FatTree ft, int index, HashSet<Integer> used) {
        if (index == this.optimalVNFs.length) {
            optimize(ft);
            return;
        }

        for (int i = 0; i < ft.switchesTable.length; i++) {

            if (!used.contains(i)) {
                int temp = ft.switchesTable[i];
                ft.VNFs[index] = ft.switchesTable[i];
                if (index < this.optimalVNFs.length && !optimize(ft, index + 1))
{
                    ft.VNFs[index] = temp;
                    continue;
                }
                used.add(i);
                help(ft, index + 1, used);
                used.remove(i);
            }
        }
    }

    /**
     * Find if the cost of partial VNFs chain is greater than optimal cost
     * If so then return true, if not return false
```

```
     * @param ft FatTree Object
     * @param n current number of VNFs
     * @return
     */
    public boolean optimize(FatTree ft, int n) {
        double operationCost = this.operation == 0? 0: cost.migrateCost(ft,
this.originalVNFs);
        double positionCost  = cost.calculateTotalCost(ft, n);
        double cost          = positionCost + operationCost;

        if (cost < this.optimalCost) {
            return true;
        } else {
            return false;
        }
    }

    /**
     * Find the optimal cost
     * @param ft FatTree Object
     */
    public void optimize(FatTree ft) {
        double operationCost = this.operation == 0? 0: cost.migrateCost(ft,
this.originalVNFs);
        double positionCost  = cost.calculateTotalCost(ft);
        double cost          = positionCost + operationCost;

        if (cost < this.optimalCost) {
            this.optimalCost = cost;
            for (int i = 0; i < this.optimalVNFs.length; i++) {
                this.optimalVNFs[i] = ft.VNFs[i];
            }

        }
    }


}
```

9.4. Greedy algorithm for placement

```
/**
```

```
 * <h1>Greedy Algorithm For Placement </h1>
 * A two-step greedy algorithm for placement in the report
 * @author sunjingsong
 * @version 1.0
 * @since 8-3-2020
 */
public class Greedy {
    public int operation;
    public Cost cost;
    public int[] originalVNFs;
    public int[] optimalVNFs;
    public double originalCost;
    public double optimalCost;
    public double[][] graph;
    public double minCostStart = Double.MAX_VALUE;
    public double minCostEnd = Double.MAX_VALUE;


    public Greedy(int operation, Cost cost) {
        this.operation = operation;
        this.cost      = cost;
        if (operation == 0) {
            System.out.println("Greedy Algorithm To Placement");
        } else {
            System.out.println("Greedy Algorithm To Migration");
        }
    }

    public double execute1(FatTree ft,  int[] originalVNFs,
        int[] optimalVNFs, double originalCost, double optimalCost) {
        System.out.println("Algo 1");
        this.originalVNFs = originalVNFs;
        this.optimalVNFs  = optimalVNFs;
        this.originalCost = originalCost;
        this.optimalCost  = optimalCost;
        algorithm1(ft);
        return this.optimalCost;
    }

    public void algorithm1(FatTree ft) {
        int n = ft.VNFs.length;
```

```
    HashSet<Integer> set = new HashSet<>();
    // Without VNFs
    // double cost = this.cost.totalCostWithoutVNF(ft);
    // Greedy placement VNF one by one
    for (int i = 1; i <= n; i++) {
        int minIndex = -1;
        // double baseCost = this.cost.calculateTotalCost(ft, i - 1);
        double minCost = Double.MAX_VALUE;
        for (int j = 0; j < ft.switchesTable.length; j++) {
            ft.VNFs[i - 1] = ft.switchesTable[j];
            if (!set.contains(ft.switchesTable[j])) {
                double newCost = this.cost.calculateTotalCostGreedy(ft, i);

                if (newCost < minCost) {
                    minCost = newCost;
                    minIndex = j;
                }
            }
        }
        set.add(ft.switchesTable[minIndex]);
        ft.VNFs[i - 1] = ft.switchesTable[minIndex];
        this.optimalVNFs[i - 1] = ft.switchesTable[minIndex];
    }
    this.optimalCost = this.cost.calculateTotalCost(ft);
}

public double execute2(FatTree ft,  int[] originalVNFs,
    int[] optimalVNFs, double originalCost, double optimalCost) {
    System.out.println("Algo 2");
    this.originalVNFs = originalVNFs;
    this.optimalVNFs  = optimalVNFs;
    this.originalCost = originalCost;
    this.optimalCost  = optimalCost;
    algorithm1(ft);
    return this.optimalCost;
}

public void algorithm2(FatTree ft) {
    int n = ft.VNFs.length;
    HashSet<Integer> set = new HashSet<>();
    // Without VNFs
```

```
        double oldCost = this.cost.totalCostWithoutVNF(ft);
        // Greedy placement VNF one by one
        for (int i = 1; i <= n; i++) {
            int minIndex = -1;
            double minScore = Double.MAX_VALUE;
            for (int j = 0; j < ft.switchesTable.length; j++) {
                ft.VNFs[i - 1] = ft.switchesTable[j];
                if (!set.contains(ft.switchesTable[j])) {
                    double newCost = this.cost.calculateTotalCostGreedy(ft, i);
                    double scoreOrg = newCost - oldCost;
                    double scoreEd = this.cost.calculateED(ft, i,
ft.switchesTable[j]);
                    double score = scoreOrg + scoreEd;
                    if (score < minScore) {
                        minScore = score;
                        minIndex = j;
                    }
                }
            }
            set.add(ft.switchesTable[minIndex]);
            ft.VNFs[i - 1] = ft.switchesTable[minIndex];
            this.optimalVNFs[i - 1] = ft.switchesTable[minIndex];
        }
        this.optimalCost = this.cost.calculateTotalCost(ft);
    }
}
```

9.5. Calculate FatTree cost

```
public class Cost {
    public void endCosts(FatTree ft) {
        for (int pos: ft.switchesTable) {
            double start = 0, end = 0;
            for (Integer[] item : ft.pairs) {

                start += ft.costTable[pos][item[0]] * item[2];
                end   += ft.costTable[pos][item[1]] * item[2];
            }
            ft.start.put(pos, start);
            ft.end.put(pos, end);
            ft.minEnd = Math.min(ft.minEnd, end);
```

```
    }
}



public double totalCostWithoutVNF(FatTree ft) {
    double result = 0;
    for (Integer[] pair: ft.pairs) {
        result += ft.costTable[pair[0]][pair[1]] * pair[2];
    }
    return result;
}

public double calculateTotalCost(FatTree ft) {
    double totalCost = 0.0;
    double totalDistance = 0.0;
    totalCost += ft.start.get(ft.VNFs[0]);
    totalCost += ft.end.get(ft.VNFs[ft.VNFs.length - 1]);

    if (ft.VNFs.length > 0) {
        for (int i = 0; i < ft.VNFs.length - 1; i++) {
            totalDistance += ft.costTable[ft.VNFs[i]][ft.VNFs[i + 1]];
        }
        totalCost += totalDistance * ft.totalFrequency;
    }
    return totalCost;
}

public double calculateTotalCostGreedy(FatTree ft, int k) {
    double totalCost = 0.0;
    double totalDistance = 0.0;
    totalCost += ft.start.get(ft.VNFs[0]);
   totalCost += ft.end.get(ft.VNFs[ft.VNFs.length - 1]);

    if (ft.VNFs.length > 0) {
        for (int i = 0; i < ft.VNFs.length - 1; i++) {
            totalDistance += ft.costTable[ft.VNFs[i]][ft.VNFs[i + 1]];
        }
        totalCost += totalDistance * ft.totalFrequency;
    }
    return totalCost;
```

```
}

public double calculateTotalCost(FatTree ft, int k) {
    double totalCost = 0.0;
    double totalDistance = 0.0;
    totalCost += ft.start.get(ft.VNFs[0]);
    totalCost += ft.minEnd;

    if (k > 1) {
        for (int i = 0; i < k - 1; i++) {
            totalDistance += ft.costTable[ft.VNFs[i]][ft.VNFs[i + 1]];
        }
        totalDistance += ft.VNFs.length - k;
        totalCost += totalDistance * ft.totalFrequency;
    }
    return totalCost;
}

public double calculateED(FatTree ft, int k, int l) {
    int n = ft.pairs.size();
    HashSet<Integer> set = new HashSet<>();
    set.add(l);
    for (int i = 0; i < k; i++) {
        set.add(ft.VNFs[i]);
    }
    double totalDistance = 0;
    if (ft.VNFs.length > 0) {
        for (int i = 0; i < ft.switchesTable.length; i++) {
            if (!set.contains(ft.switchesTable[i])) {
                totalDistance += ft.costTable[ft.switchesTable[i]][l];
            }
        }
    }
    return totalDistance / (ft.switchesTable.length - set.size()) * n;
}

public double migrateCost(FatTree ft, int[] oldVNFs) {
    double distance = 0.0;
    int len = oldVNFs.length;
    for (int i = 0; i < len; i++) {
        distance += ft.costTable[ft.VNFs[i]][oldVNFs[i]];
```

```
        }
        double cost = distance * ft.migrateCoefficient;
        return cost;
    }

    public double twoNodesMigrateCost(FatTree ft, int node1, int node2) {
        double distance = ft.costTable[node1][node2];
        double cost = distance * ft.migrateCoefficient;
        return cost;
    }

    public double printMigrateCost(FatTree ft, int[] oldVNFs) {
        double totalCost = 0.0;
        int len = oldVNFs.length;
        for (int i = 0; i < len; i++) {
            double cost =  ft.costTable[ft.VNFs[i]][oldVNFs[i]]*
ft.migrateCoefficient;
            System.out.println("The migration cost of " +
ft.indexNodeMap.get(ft.VNFs[i]) + " from " + ft.indexNodeMap.get(oldVNFs[i])
                    + " to " + ft.indexNodeMap.get(ft.VNFs[i]) + " is " + cost);
            totalCost += cost;
        }
        System.out.println("The total migration cost is: " + totalCost);
        return totalCost;
    }
}
```

9.6. Calculate FatTree hops between switches

```
public class Distance {
    public Distance(FatTree ft) {
        int i = 0;
        int j = 0;
        for (String node1: ft.allPositions) {
            for (String node2: ft.allPositions) {
                ft.distance.put(node1 + node2, calculateDistance(ft, node1,
node2));
                ft.costTable[i][j++] = calculateDistance(ft, node1, node2);
```

```
            }
            i++;
            j = 0;

        }
    }


    public double calculateDistance(FatTree ft, String node1, String node2) {
        // same nodes
        if (node1 == node2) {
        return 0;
        } else if (!isNode(node1) || !isNode(node2)) {
        return -1;
        }

        // cs--cs
        if (node1.substring(0, 2).equals("cs") && node2.substring(0,
2).equals("cs")) {
        if (sameAggr(ft, node1, node2)) {
            return 2;
        } else {
            return 4;
        }
        }

        // cs--as or as--cs
        if (node1.substring(0, 2).equals("cs") && node2.substring(0,
2).equals("as")||
          node1.substring(0, 2).equals("as") && node2.substring(0,
2).equals("cs")) {
        if (ft.links.get(node1).contains(node2)) return 1;
        else return 3;
        }

        // cs--es or es--cs
        if (node1.substring(0, 2).equals("cs") && node2.substring(0,
2).equals("es")||
          node1.substring(0, 2).equals("es") && node2.substring(0,
2).equals("cs")) {
        return 2;
```

```
        }

        // cs--pm or pm--cs
        if (node1.substring(0, 2).equals("cs") && node2.substring(0,
2).equals("pm")||
          node1.substring(0, 2).equals("pm") && node2.substring(0,
2).equals("cs")) {
        return 3;
        }

        // as--as
        if (node1.substring(0, 2).equals("as") && node2.substring(0,
2).equals("as")) {
        if (node1.substring(0, 3).equals(node2.substring(0, 3))) { // same pod
            return 2;
        } else { // different pod
            // connected to same core switch
            for (String item: ft.links.get(node1)) {
                if (item.substring(0, 2).equals("cs")) {
                    if (ft.links.get(node2).contains(item)) return 2;
                }
            }
            // different core switch
            return 4;
        }
        }

        // as--es or es--as
        if (node1.substring(0, 2).equals("as") && node2.substring(0,
2).equals("es")||
                node1.substring(0, 2).equals("es") && node2.substring(0,
2).equals("as")) {
                if (node1.substring(2, 3).equals(node2.substring(2, 3))) { // same
pod
                    return 1;
                } else { // different pod
                    return 3;
                }
        }

        // as--pm or pm--as
```

```
        if (node1.substring(0, 2).equals("as") && node2.substring(0,
2).equals("pm")||
          node1.substring(0, 2).equals("pm") && node2.substring(0,
2).equals("as")) {
        if (node1.substring(2, 3).equals(node2.substring(2, 3))) { // same pod
            return 2;
        } else { // different pod
            return 4;
        }
        }


        // es--es
        if (node1.substring(0, 2).equals("es") && node2.substring(0,
2).equals("es")) {
        if (node1.substring(0, 3).equals(node2.substring(0, 3))) { // same pod
            return 2;
        } else { // different pod
            return 4;
        }
        }

        // es--pm or pm--es
        if (node1.substring(0, 2).equals("es") && node2.substring(0,
2).equals("pm")||
          node1.substring(0, 2).equals("pm") && node2.substring(0,
2).equals("es")) {
        if (node1.substring(2, 3).equals(node2.substring(2, 3))) { // same pod
            if (node1.substring(3, 5).equals(node2.substring(3, 5))) {// same edge
                return 1;
            } else { // different edge
                return 3;
            }
        } else { // different pod
            return 5;
        }
        }

        // pm--pm
        if (node1.substring(0, 2).equals("pm") && node2.substring(0,
2).equals("pm")) {
```

```
        if (node1.substring(0, 3).equals(node2.substring(0, 3))) { // same pod
            if (node1.substring(3, 5).equals(node2.substring(3, 5))) {// same edge
                return 2;
            } else {
                return 4;
            }
        } else { // different pod
            return 6;
        }
        }

        return -1;

    }


    public boolean sameAggr(FatTree ft, String node1, String node2) {
        for (String item: ft.links.get(node1)) {
            if (item.substring(0, 2).equals("as")) {
                if (ft.links.get(node2).contains(item)) return true;
            }
        }
        return false;
    }
    public boolean isNode(String node) {
        return node.substring(0,2).equals("pm") ||
node.substring(0,2).equals("cs") ||
            node.substring(0,2).equals("as") ||
node.substring(0,2).equals("es");
    }

}
```

9.7. Benefit Algorithm For Migration

```
public class Benefit {
    public int[] originalVNFs;
    public int[] optimalVNFs;
    public double originalCost;
    public double optimalCost;
```

```
    public FatTree ft;
    public Cost cost;
    public Benefit(FatTree ft, int[] originalVNFs, int[] optimalVNFs,
              double originalCost, double optimalCost) {
        this.ft = ft;
        this.originalVNFs = originalVNFs;
        this.optimalVNFs = optimalVNFs;
        this.originalCost = originalCost;
        this.optimalCost = optimalCost;
        cost = new Cost();
    }

    public void execute() {
        migrateVNFs();
        for (int i = 0; i < this.optimalVNFs.length; i++) {
            ft.optimalVNFs[i] = this.optimalVNFs[i];
        }
    }

    public void migrateVNFs() {
        HashSet<Integer> set = new HashSet();
        for (int VNF: this.originalVNFs) {
            set.add(VNF);
        }
        int[] tempVNFs = new int[this.originalVNFs.length];
        for (int i = 0; i < tempVNFs.length; i++) {
            tempVNFs[i] = this.originalVNFs[i];
        }

        double minCost = this.optimalCost;
        for (int i = 0; i < tempVNFs.length; i++) {
            for (int pos: ft.switchesTable) {
                double tempCost = 0;
                if (!set.contains(pos)) {
                    int temp = tempVNFs[i];
                    tempVNFs[i] = pos;
                    tempCost += cost.calculateLocationCost(ft, tempVNFs);
                    tempCost += cost.migrateCost(ft, this.originalVNFs,
tempVNFs);
                    if (tempCost < minCost) {
                        minCost = tempCost;
```

```
                    set.remove(temp);
                    set.add(pos);
                } else {
                    tempVNFs[i] = temp;
                }
            }
        }
    }

    for (int i = 0; i < tempVNFs.length; i++) {
        this.optimalVNFs[i] = tempVNFs[i];
    }
  }

}
```

9.8. StepWise Algorithm For Migration

```
public class Stepwise {
    public int[] originalVNFs;
    public int[] optimalVNFs;
    public double originalCost;
    public double optimalCost;
    public int maxHops;
    public FatTree ft;
    public Cost cost;
    HashMap<Integer, HashSet<Integer>> graph;
    ArrayList<ArrayList<int[]>> paths;
    public Stepwise(FatTree ft, int[] originalVNFs, int[] optimalVNFs,
                    double originalCost, double optimalCost) {
        this.ft = ft;
        this.originalVNFs = originalVNFs;
        this.optimalVNFs = optimalVNFs;
        this.originalCost = originalCost;
        this.optimalCost = optimalCost;
        cost = new Cost();
    }

    public void execute() {
        createGraph();
        findPaths();
        migrateVNFs();
```

```
    for (int i = 0; i < this.optimalVNFs.length; i++) {
        ft.optimalVNFs[i] = this.optimalVNFs[i];
    }
    //shortestPath();
}

public void migrateVNFs() {
    HashSet<Integer> visited = new HashSet();
    for (int VNF: this.originalVNFs) {
        visited.add(VNF);
    }

    int[] tempVNFs = new int[ft.VNFs.length];
    for (int i = 0; i < tempVNFs.length; i++) {
        tempVNFs[i] = this.originalVNFs[i];
    }

    double minCost = this.originalCost;
    for (int i = 0; i < paths.size(); i++) {
        ArrayList<int[]> path = paths.get(i);
        for (int j = 0; j < path.size(); j++) {
            if (!visited.contains(path.get(j)[0]) ) {
                visited.add(path.get(j)[0]);
                tempVNFs[i] = path.get(j)[0];
                double totalCost = totalCost(tempVNFs);
                if (totalCost < minCost) {
                    minCost = totalCost;
                    updateOptimalVNFs(tempVNFs);
                }
            }
        }
    }
    this.optimalCost = minCost;
}

public void updateOptimalVNFs(int[] tempVNFs) {
    for (int i = 0; i < tempVNFs.length; i++) {
        this.optimalVNFs[i] = tempVNFs[i];
    }
}
public double totalCost(int[] modifiedVNFs) {
```

```
        double hop = 0.0;
        double totalCost = 0.0;
        for (int i = 0; i < modifiedVNFs.length; i++) {
            hop += ft.costTable[modifiedVNFs[i]][this.originalVNFs[i]];
        }
        totalCost += hop * ft.migrateCoefficient;
        totalCost += cost.calculateLocationCost(ft, modifiedVNFs);
        return totalCost;
    }

    public void findPaths() {
        paths = new ArrayList();
        for (int i = 0; i < this.originalVNFs.length; i++) {
            ArrayList<int[]> path = shortestPath(this.originalVNFs[i],
this.optimalVNFs[i]);
            paths.add(path);
        }

    }

    public ArrayList<int[]> shortestPath(int s, int t) {
        int[] nodes = new int[ft.costTable.length];
        Arrays.fill(nodes, Integer.MAX_VALUE);
        nodes[s] = 0;

        HashSet<Integer> visited = new HashSet();
        ArrayList<int[]> path = new ArrayList<>();

        PriorityQueue<ArrayList<int[]>> pq = new PriorityQueue<ArrayList<int[]>>(
            (ArrayList<int[]> a, ArrayList<int[]> b) -> {
                return a.get(a.size() - 1)[1] - b.get(b.size() - 1)[1];
            });

        int chosenIndex = s;
        path.add(new int[] {chosenIndex, nodes[chosenIndex]});
        visited.add(chosenIndex);

        while (chosenIndex != t) {
            for (int adj: graph.get(chosenIndex)) {
            if (!visited.contains(adj)) {
                int hop = nodes[chosenIndex] + 1;
```

```
                if (hop < nodes[adj]) {
                    nodes[adj] = hop;
                    ArrayList<int[]> temp = new ArrayList();
                    temp.addAll(path);
                    temp.add(new int[] {adj, nodes[adj]});
                    pq.offer(temp);
                }
            }
            }
            if (!pq.isEmpty()) {
            path.clear();
            ArrayList<int[]> temp = pq.poll();
            path.addAll(temp);
            chosenIndex = temp.get(temp.size() - 1)[0];
            visited.add(chosenIndex);
            } else {
            return null;
            }
        }
        return path;

    }

    public void createGraph() {
        this.graph = new HashMap();
        for (int i = 0; i < ft.costTable.length; i++) {
            for (int j = 0; j < ft.costTable[i].length; j++) {
                if (ft.costTable[i][j] == 1.0) {
                    graph.putIfAbsent(i, new HashSet<>());
                    graph.putIfAbsent(j, new HashSet<>());
                    graph.get(i).add(j);
                    graph.get(j).add(i);
                }
            }
        }
    }
}
```