Achieving High End-to-End Availability in VNF Networks

Enrique Rodicio, Deng Pan, Jason Liu
School of Computing and Information Sciences
Florida International University
Miami, FL 33199
Email: {erodi002, pand, liux}@fiu.edu

Bin Tang
Department of Computer Science
California State University Dominguez Hills
Carson, CA 90747
Email: btang@csudh.edu

Abstract—As software programs, Virtual Network Functions (VNFs) introduce new challenges to network availability due to their potential software failures. Existing models on the availability of VNF networks did not consider all the possible hardware and software failures, making them incapable of analyzing the end-to-end availability of a path. Furthermore, they did not capture the correlation between repeating nodes and links in the path, resulting in inaccurate analytical results. In this paper, we propose a new analytical model, which considers all hardware and software failures as well as the effect of repeating components, to effectively analyze the end-to-end availability of a flow path in VNF networks. On top of the analytical model, we formulate the Highest Availability Path (HAP) problem that finds the flow path with the highest end-to-end availability, and prove its NP-hardness by reduction from the Node-Weighted Steiner Tree problem. Next, we propose two algorithms for HAP: the first one based on a Steiner Tree approximation algorithm, having high time complexity and serving as a performance benchmark; the second one using a dynamic programming approach to search a multi-layer graph in polynomial time. Finally, we present extensive evaluation data to demonstrate the effectiveness of the Layered Search algorithm, which achieves comparable performance as that of the Steiner Tree based algorithm and runs faster by four orders of magnitude.

Index Terms—VNF, availability, failure

I. INTRODUCTION

Virtual Network Functions (VNFs) [42] implement network functions as virtual machines (VMs) [6] (or containers [19]) on commodity servers. By decoupling functions from underlying physical hardware [35], VNFs bring many advantages over traditional hardware-based network appliances, such as reduced cost, fast time-to-market, and elastic scalability [10]. VNFs have been adopted in various networks, including 5G [4], [7], Internet of things [17], [29], and smart communities [9], [33].

A Service Function Chain (SFC) [38] is a sequence of VNFs that will be visited by a flow. Because VMs have limited processing capabilities, a network is usually provisioned with multiple VMs for a given VNF to meet the capacity requirement [15], [41]. Hence, the flow path selects one among the multiple available VMs for each VNF in the SFC, and traverses them following the order specified by the SFC.

VNFs introduce new challenges to the availability of a flow path. A recent study on network reliability [31] found that 13% of the data center network incidents at Facebook from 2011 to 2018 were caused by hardware failures, such as faulty memory

TABLE I: Failures considered by existing studies.

	Hardware failures			Software failures	
	Switch	Link	Server	Hypervisor	VNF VM
[13], [35]		✓			
[34]			✓		
[14], [16], [23], [40]					✓
[32]	✓		✓		
[8], [25]		✓			✓
[2], [11], [21], [26], [37], [41], [43]			✓		✓

and ports, and 42% by software failures, such as software upgrading, misconfiguration, and bugs (besides 11%, 5%, and 29% caused by accidents, capacity planning, and undetermined reasons, respectively). As can be seen, software failures are more common than hardware ones, and the software nature of VNFs makes a flow path subject to additional potential failures, such as those to VNF software or virtual switches (running in hypervisors), which do not exist in traditional networks.

There has been an emerging research interest [8], [11], [14], [16], [21]–[23], [25], [26], [32], [37], [41], [43] to address the availability challenges in VNF networks. The basic approach is to utilize the flexibility of VNFs to be able to run at arbitrary locations, and select the proper hosting servers to achieve a high availability. Furthermore, backup VMs or replicas can be set up for a VNF to enhance the recovery probability in case of failures.

Existing studies on the VNF availability have several insufficiencies. To start with, none of the existing studies have considered all the possible hardware and software failures, as shown in Table I that summaries the failures considered by existing studies. However, a path fails if any node or link on the path fails, and therefore it is crucial to take all potential failures into consideration. As an example, if a (buggy) virtual switch randomly drops packets to certain hosted VNFs, existing solutions are incapable of modeling such a failure. We will show in Section III how the analytic model proposed in this paper models the failure.

Furthermore, existing studies [11], [25], [32], [40] calculate the availability probability of a path by multiplying the availability probability of each hop on the path, and do not consider
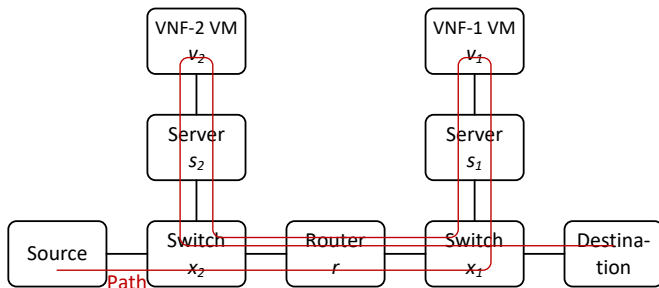
Fig. 1: Flow paths in VNF networks most likely contain repeating nodes and links.

the potential correlation between different hops. However, a path in NFV networks most likely contains repeating nodes and links, whose failures are correlated, for the following two reasons. First, since the hosting server is usually connected to its adjacent switch by a single link, and the VM is connected to the hosting server by a single (software) link, there is only one route between the switch and VM that will be used by the traffic to both accessing and leaving the VM. For example, Fig. 1 shows the path (in red) of a flow whose SFC consists of VNF-1 followed by VNF-2. The nodes x_1 and s_1, hardware duplex link (x_1, s_1), and software duplex link (s_1, v_1) are repeated in the path, because there is only one route between x_1 and v_1. Second, the network topology may require the path to traverse a path segment multiple times in order to visit the VNFs in the specified SFC order. In the example of Fig. 1, the nodes x_2, r, and x_1 and links (x_2, r) and (r, x_1) are traversed three times to visit VNF-1 before VNF-2.

In this paper, we propose a theoretical model to effectively analyze the end-to-end availability of a flow path in VNF networks. Different from existing studies, the proposed model considers all potential hardware and software failures, and is thus capable of analyzing the end-to-end availability of a path. Furthermore, the model captures the availability correlation between repeating nodes and links, and therefore makes the analytical results more accurate. To the best of our knowledge, the proposed model is the first to consider all hardware and software failures, and also the first to study the effect of repeating components.

In addition, based on the proposed model, we formulate the Highest Availability Path (HAP) problem as a graph optimization problem to find the path with the highest availability in a VNF network. We show that the problem is NP-hard by reduction from the Node-Weighted Steiner Tree (NWST) problem [20]. Next, we design two algorithms for the HAP problem. The first one extends an approximation algorithm for the NWST problem. It has exponential time complexity and is used as a performance benchmark. The second algorithm uses a dynamic programming approach to search a multi-layer graph, which achieves comparable performance and runs faster by four orders of magnitude.

Our main contributions are summarized as follows:

1) We propose a theoretical model to analyze the availability of a path in VNF networks, which considers all potential

hardware and software failures and captures the availability correlation between repeating components.

2) We formulate the HAP problem that finds the path with the highest end-to-end availability, prove its NP-hardness, and present two algorithms: one in exponential time complexity as a benchmark, the other achieving comparable performance in polynomial time complexity.

3) Finally, we evaluate the proposed algorithms with different network topologies and sizes, and present extensive results to show the effectiveness of our design.

The remaining of the paper is organized as follows. Section II briefly reviews related work and highlights our differences. Section III presents the analytical model, and formulates the HAP problem. Section IV proves the NP-hardness of the formulated problem. Section V presents a benchmark algorithm based on an approximation algorithm for the NWST problem, and Section VI proposes a dynamic programming based algorithm. Finally, Section VII conducts performance evaluation, and Section VIII concludes the paper.

II. RELATED WORK

In this section, we briefly review related work, and highlight our differences. There are two areas of related work: availability and routing in VNF networks.

In the area of VNF availability, existing studies focus on two categories: deploying VNFs to ensure high availability and setting up backup VMs for recovery in case of failures. Our work on the one hand differs from the existing work with a new analytical model that is more comprehensive and accurate, and on the one hand complements it by studying a new optimization problem. More specifically, the theoretical model proposed in this paper considers all possible hardware and software failures, and captures the correlation between repeating nodes and links. In addition, while existing work studies the deployment of VNFs and backup instances, the problem studied in this paper to achieve a high end-to-end availability with given VNF deployment has not been investigated before.

The first category of existing work on VNF availability studies the deployment of VNFs and backup VMs at selected locations to achieve a high availability for an SFC. Beck et al. [8] study how backup resources can be integrated into the deployment of VNFs to protect network services from failures, and propose to enhance link resilience with a link-disjoint backup path and VNF resilience with backup nodes. Fan et al. [16] define an optimal availability-aware SFC mapping problem and present a novel online algorithm that minimizes the physical resources consumption while guaranteeing the required high availability. Kong et al. [25] propose a coordinated protection mechanism with both link-disjoint backup paths and VNF replicas. nodes on the working and backup paths. Moualla et al. [32] propose a VNF placement algorithm for SFCs in a fat tree to enhance availability by placing VNF replicas in different pods. Wang et al. [39] study the deployment of parallelized SFCs in data center networks considering availability guarantee and resource optimization.

Mandal [30] compares VNF deployment with multiple host nodes, a single host node, and in the mixed mode by analyzing the SFC availability. In addition, multiple efforts are devoted to study VNF deployment for joint optimization of availability with latency [11], [34], [41], cost [13], [14], [35], [43], revenue [37], [40], resource utilization [26], [39], and scalability [2].

The second category of existing work on VNF availability assumes that primary VNF instances are already deployed, and the objective is to maximize the availability in case of failures by carefully selecting the backup instances. Taking advantage of the flexibility and resource-sharing abilities of VNFs, Kanizo et al. [22] propose to maintain only a few backup servers, each serving as the backup for multiple VNFs. Under this assumption, they define multiple optimization problems, and propose algorithms based on a graph theoretical model. Furthermore, they [23] develop VNF recovery schemes when a small number of VNFs fail simultaneously, based on a novel representation of solutions describing assignments of functions to VMs. He et al. [21] consider importance of different VNFs and failures for not only VNFs but also backup servers, and formulate a backup VNF assignment problem to minimize the worst weighted unavailability. They prove the problem to be NP-complete, and develop three heuristic algorithms with polynomial time.

In addition to VNF availability, the second area of related work optimizes routing in VNF networks to address the challenges of the SFC order and multiple available instances for a given VNF. A variety of solutions [3], [15], [18], [27], [36] are proposed in different contexts for different objectives. They adopt the convention model where the weight of a path is the sum of weights of all edges in the path. We consider a completely new model where the availability of a path is the production of the availabilities of unique vertices and edges in the path, excluding repeating occurrences.

III. PROBLEM STATEMENT

In this section, we describe the proposed model for availability analysis in VNF networks, and formulate the Highest Availability Path (HAP) problem. Table II summarizes the list of notations used in the paper for easy reference.

A. Availability Model for VNF Networks

Model the network as an undirected graph $G = (V, E)$, in which a vertex $v \in V$ may be a switch, server, VNF VM, or any hardware or software component whose availability needs to be considered, and edge $(u, v) \in E$ represents the link between two nodes $u, v \in V$. The availability of a vertex v or an edge (u, v) is denoted as a_v or $a_{(u,v)}$, which is the percentage of time, or in other words the probability, that the component is functional. The availability of a component can be calculated based on the Mean Time Between Failures (MTBF) and Mean Time To Repair (MTTR) [16], [34], [43], i.e., $availability = MTBF/(MTBF + MTTR)$. The availabilities of different components are assumed to be independent [26], [43], because the availability of a component

TABLE II: List of notations.

| Notation	Description
v, a_v	vertex v, availability of v
$(u, v), a_{(u,v)}$	edge (u, v), availability of (u, v)
$is(v, k)$	1 if v is VM of VNF-k, 0 otherwise
s, d, C	source, destination, SFC of flow
K	number of VNFs in SFC
VNF-k	k-th VNF in SFC
$v \in path, (u, v) \in path$	vertex v or edge (u, v) is on path $path$
a_{path}	end-to-end availability of path
n	number of hops on path
$path(i, v)$	1 if v is i-th hop on path, 0 otherwise
$loc(k, i)$	1 if VNF-k located at i-th hop, 0 otherwise
w_v	weight of vertex v in NWST
T	set of terminals in NWST
$v \in tree$	vertex v is in Steiner tree $tree$
w_{tree}	weight of Steiner tree in NWST
V_k	set of VMs for VNF-k
$\overline{uv}$	dummy vertex in NWST for edge (u, v)
$G^* = (V^*, E^*)$	multi-layer graph based on $G = (V, E)$
$v^k, (u^k, v^k)$	vertex, edge in layer k of G^*
$P(v^k), A(v^k)$	path and availability from s^1 to v^k
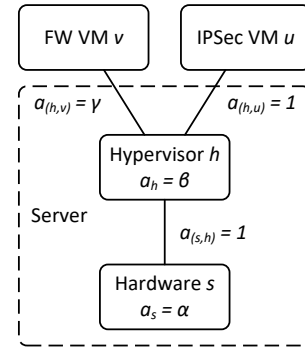
Fig. 2: The proposed model allows any vertex or edge to be associated with an availability value.

is calculated based on only its own failures but not external ones that may affect the component, such as power outages. Availability data are available from manufacturers [1], third party studies [31], or history statistics.

The above model is capable of representing all possible hardware and software failures, because a network is abstracted as a graph, and the model allows each vertex and edge in the graph to be associated with an availability value. The following example illustrates the difference between the proposed model and existing ones. Consider a hosting server, whose hardware box and hypervisor software have availabilities of α and β, respectively. The virtual switch in the hypervisor has an availability of γ when communicating with a hosted firewall VNF VM and an availability of 1 with other VNF VMs. The availability of such a server cannot be accurately described by existing models, but can be represented using the proposed model as shown in Fig. 2.

There are multiple different types of VNFs in the network. As explained in Section I, a VNF may have multiple VMs in the network for enhanced processing capacity. For easy representation, we define a function $is()$ to denote the VNF type of a vertex: $is(v, k) = 1$ if the vertex $v \in V$ is a VM of the VNF type k, or VNF-k for short, and 0 otherwise.

A flow is a tuple (s, d, C) where $s \in V$ and $d \in V$ are the source and destination vertices, respectively, and C is the Service Function Chain (SFC) to be visited by the flow. Without loss of generality, assume that the SFC C consists of K VNFs in the order VNF-1 $\rightarrow$ VNF-2 $\rightarrow \ldots \rightarrow$ VNF-K. A valid flow path $path = < v_0, v_1, \ldots, v_n >$ starts and ends with the source s and destination d, respectively, i.e., $v_0 = s$ and $v_n = d$, and in between traverses a VM for each VNF in C in the specified order.

Since a path $path$ is functional only if all the vertices and edges on it are functional, the end-to-end availability a_{path} is the product of the availabilities of all vertices and edges on the path. With slight abuse of notations, use $v \in path$ and $(u, v) \in path$ to denote that the vertex v and edge (u, v) are on $path$, respectively. The end-to-end availability a_{path} can be calculated as

$$a_{path} = \prod_{v \in path} a_v \prod_{(u,v) \in path} a_{(u,v)} \tag{1}$$

Note that, in general, a_{path} is not the product of availabilities of consecutive vertices and edges on the path, i.e.

$$a_{path} \neq \prod_{i=0}^{n} a_{v_i} \prod_{i=0}^{n-1} a_{(v_i, v_{i+1})} \tag{2}$$

because there may be repeating vertices and edges on the path.

B. Problem Formulation

Given a graph $G = (V, E)$ and flow (s, d, C), the Highest Availability Path (HAP) problem finds a flow path $path$ that achieves the highest end-to-end availability a_{path}. There are three decisions variables. The first one, n, determines the number of hops on the path. The second one, $path(i, v)$, determines the vertex at each hop, and is defined as

$$path(i, v) = \begin{cases} 1, \text{if } v \in V \text{ is } i\text{-th hop on path.} \\ 0, \text{otherwise.} \end{cases} \tag{3}$$

The third one, $loc(k, i)$, determines the service location for the k-th VNF in the SFC, and is defined as

$$loc(k, i) = \begin{cases} 1, \text{if VNF-}k \text{ is located at } i\text{-th hop on path.} \\ 0, \text{otherwise.} \end{cases} \tag{4}$$

The problem to find the path with the highest end-to-end availability can thus be formulated as below.

$$\text{maximize } a_{path} \tag{5}$$

subject to:

$$path(s, 1) = 1 \tag{6}$$

$$path(d, n) = 1 \tag{7}$$

$$\forall 0 \leq i \leq n - 1, \sum_{(u,v) \in E} path(i, u)path(i + 1, v) = 1 \tag{8}$$

$$\forall 1 \leq k \leq K, \sum_{i=0}^{n} \sum_{v \in V} path(i, v)is(v, k)loc(k, i) = 1 \tag{9}$$

$$\forall 1 \leq k \leq K - 1,$$
$$\sum_{i=0}^{n} i \cdot loc(k + 1, i) \geq \sum_{i=0}^{n} i \cdot loc(k, i) \tag{10}$$

Brief explanation of the constraints is as follows. Equations (6) and (7) ensure that the path starts from the source s and ends at the destination d, respectively. Equation (8) states that two consecutive hops must be connected by an edge in the graph. Equation (9) guarantees that one VM for each VNF-k is located on the path. Finally, Equation (10) enforces the SFC order between VNFs.

As can be seen, our availability model considers hardware and software failures, such as faulty components, misconfiguration, and bugs, which are mostly not related with traffic volumes. Hence, the above problem formulation applies to multiple flows as well, as different flows do not affect the availability of each other.

IV. NP-HARDNESS

In this section, we show that the HAP problem is NP-hard by reduction from the Node-Weighted Steiner Tree (NWST) Problem [20].

Given a graph $G = (V, E)$, in which each vertex $v \in V$ has an associated weight $w_v \geq 0$, and a set of terminal vertices (abbreviated as terminals thereafter) $T \subseteq V$, the NWST problem computes a minimum weight tree that includes all the terminals in T, where the weight of the tree is the sum of the weights of all the vertices in the tree. The tree may include other vertices not in T as well.

Theorem 1. *The Highest Availability Path problem is NP-hard.*

Proof. For an instance of the NWST problem with a graph $G = (V, E)$, terminals T, and node weights w, an instance of the HAP problem can be constructed in polynomial time as follows. Without loss of generality, assume T has two or more terminals, i.e., $|T| = n + 1 \geq 2$, and denote the terminals as $T = \{t_0, \ldots, t_n\}$ where the order between the terminals is arbitrary.

1) The graph in the HAP problem instance is also $G = (V, E)$. The availability of each vertex is the exponentiation of the negative vertex weight in NWST, i.e., $\forall v \in V, a_v = e^{-w_v}$. Since $w_v \geq 0$, $0 < a_v \leq 1$. The availability of each edge is 1, i.e., $\forall (u, v) \in E, a_{(u,v)} = 1$.

2) For the flow in HAP, let its source and destination be t_0 and t_n, respectively, i.e., $s = t_0$ and $d = t_n$.

3) The SFC C of the flow consists of $K = n - 1$ VNFs in the order VNF-1 $\rightarrow \cdots \rightarrow$ VNF-K, and each VNF-k has a single VM located at t_k, i.e., $\forall 1 \leq k \leq n - 1, is(t_k, k) = 1$.

Next, it can be shown that if the NWST instance has a Steiner tree, denoted as $tree$, with a weight of w_{tree}, then HAP instance has a path with an availability of $e^{-w_{tree}}$, and reversely, if the HAP instance has a path $path$ with an end-to-end availability of a_{path}, the NWST instance has a Steiner tree with a weight of $-\ln a_{path}$. The detail is omitted due to space limitations.

$\square$

V. Steiner Tree based Algorithm

In this section, we present an algorithm for the HAP problem based on an approximation algorithm [20] for the NWST problem. The algorithm is expected to work only as a performance benchmark, as it has high time complexity.

The NWST problem is a classical NP-hard problem, and there have been a series of efforts [20], [24], [28] to develop approximation algorithms for it. The latest work [20] proposes three approximation algorithms with worst-case approximation factors of $1.5 \ln |T|$, $(1.35+\epsilon) \ln |T|$, and $1.6103 \ln |T|$, respectively. Among the three algorithms, we focus on the third one with an approximation factor of $1.6103 \ln |T|$, since it is the only one in polynomial time complexity. We do so because our approach solves a HAP problem instance by converting it into many instances of the NWST problem, and applying an algorithm in non-polynomial time complexity to each of the many instances requires a prohibitive amount of time to finish the simulations in Section VII. Nevertheless, it still works as a reasonable performance benchmark, as its approximation factor differs from the best approximation factor by only a constant.

In brief, the third approximation algorithm in [20] calculates the Steiner tree by iteratively merging the minimum ratio "spider" into a single terminal, where a spider is a tree with at most one vertex of degree more than two, and its ratio is the ratio between the total weight of all the vertices in the spider and the number of terminals in it. The time complexity of the algorithm is $O((|V| \log |V| + |E|)|T|^2)$, because there are up to $|T|$ iterations, and the complexity of each iteration is determined by the operation to find the minimum ratio spider, which involves computing the distances from each terminal to other terminals in $O((|V| \log |V| + |E|)|T|)$ [24] time. More efficient implementation is possible for a large $|T|$ by using Johnson's all-pairs shortest paths algorithm [12].

The basic idea of our Steiner Tree based algorithm is to create from the HAP problem instance multiple instances of the NWST problem, solve each individually using the NWST approximation algorithm, and select among the results the best one. From the proof of Theorem 1, it can be seen that when each VNF has only a single VM, a HAP instance can be converted to an NWST instance. Therefore, based on the given HAP instance, multiple sub-instances are generated, each having a single VM for each different VNF. Use V_k to denote the set of VMs for VNF-k, i.e., $V_k = \{v | is(v,k) = 1\}$. There will be $|V_1| \times |V_2| \times \cdots \times |V_K|$ possible ways to select one VM from each VNF, and correspondingly so many HAP sub-instances will be created. Each of the sub-instances will then be converted to an NWST instance, which will be solved by the approximation algorithm from [20]. Finally, the results of all the HAP sub-instances will be summarized to obtain the highest end-to-end availability. For the example in Fig. 3, since the HAP instance has $V_1 = \{v, w\}$ and $V_2 = \{u\}$, $|V_1| \times |V_2| = 2$ NWST instances are created.

Two challenges arise when converting a HAP instance to an NWST one. First, the vertex availability in HAP is additive
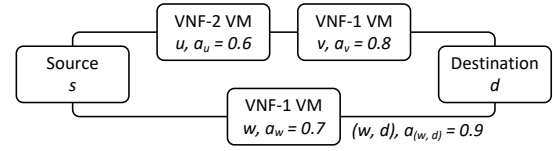
(a) Input HAP instance with $V_1 = \{v, w\}$ and $V_2 = \{u\}$

NWST instance 1: $T = \{s, d, v, u\}$

NWST instance 2: $T = \{s, d, w, u\}$
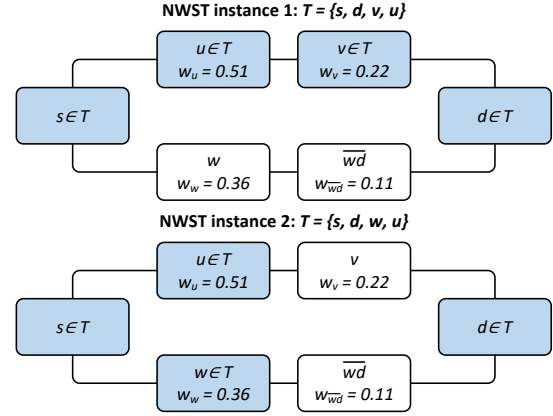
(b) Two (= $|V_1| \times |V_2|$) NWST instances created

Fig. 3: Based on the input HAP instance, $|V_1| \times |V_2| \times \cdots \times |V_K|$ NWST instances are created to apply the approximation algorithms.

and is a probability between 0 and 1, while the vertex weight in NWST is multiplicative and is a non-negative number. For conversion, the weight w_v of vertex $v \in V$ in NWST is set to be the negative logarithm of its availability a_v in HAP, i.e., $w_v = -\ln a_v$. Second, HAP considers availabilities for not only vertices but also edges, but NWST considers weights only for vertices. Thus, we need to convert an edge availability in HAP to a vertex weight in NWST. To do so, for each edge $(u,v) \in E$, a dummy vertex $\overline{uv}$ is created and is connected with u and v, so that a path traversing (u,v) will now traverse $\overline{uv}$. The weight of the dummy vertex $w_{\overline{uv}}$ is set to be the negative logarithm of the availability $a_{(u,v)}$ of the original edge (u,v), i.e., $w_{\overline{uv}} = -\ln a_{(u,v)}$. For the example in Fig. 3, we assume that all edges except (w,d) have availabilities of 1 for illustration purposes, and hence can be ignored. For the edge (w,d), a dummy vertex $\overline{wd}$ and corresponding edges $(w, \overline{wd})$ and $(\overline{wd}, d)$ are added to the NWST instances.

Algorithm 1 shows the pseudo code of the Steiner Tree based algorithm, which is explained as follows. The first step is to transform the graph in the HAP problem to one in the NWST problem. In detail, lines numbered 1 to 3 initialize the vertices, and lines 4 to 8 initialize the edges. Finally, line 9 initializes the currently known highest end-to-end path availability to be zero.

The next step is to generate the multiple instances of the NWST problem and apply the approximation algorithm to each. In detail, lines 10 to 13 specify a HAP sub-instance in which $v_1, \ldots, v_K$ are the only instances for VNF-1, $\ldots$, VNF-K, respectively, and based on it create an NWST instance with the terminals $T = \{s, d, v_1, \ldots, v_K\}$. Then, line 14 obtains the tree weight w_{tree} of the created NWST instance by applying

the approximation algorithm. Finally, lines 15 to 17 convert the NWST tree weight w_{tree} to the HAP path availability $e^{-w_{tree}}$, and replace the current highest availability a_{path} if $e^{-w_{tree}}$ is higher.

Algorithm 1 Steiner Tree based Algorithm

Input: $G = (V, E), (s, d, C), \cup_{k=1}^{K} V_k = \{v | is(v, k) = 1\}$
Output: path availability a_{path}

1: **for** $v \in V$ **do**
2:     $w_v = -\ln a_v$
3: **end for**
4: **for** $(u, v) \in E$ **do**
5:     $V = V \cup \{\overline{uv}\}$
6:     $E = (E \setminus \{(u, v)\}) \cup \{(u, \overline{uv}), (v, \overline{uv})\}$
7:     $w_{\overline{uv}} = -\ln a_{(u,v)}$
8: **end for**
9: $a_{path} = 0$
10: **for** $v_1 \in V_1$ **do**
11:     $\cdots$
12:     **for** $v_K \in V_K$ **do**
13:        create NWST instance with graph G and terminals $T = \{s, d, v_1, \ldots, v_K\}$
14:        w_{tree} = weight returned by NWST approximation algorithm
15:        **if** $a_{path} < e^{-w_{tree}}$ **then**
16:           $a_{path} = e^{-w_{tree}}$
17:        **end if**
18:     **end for**
19: **end for**

The time complexity of the Steiner tree based algorithm is $O(((|V|/K)^K ((|V| + |E|) \log(|V| + |E|))K^2)$, which is not polynomial due to the K in the exponent. In the worst case, each of the K VNFs may have $|V|/K$ VMs, so there are up to $(|V|/K)^K$ possible ways to select one VM from each VNF, or so many instances of the NWST problem will be created. The time complexity to solve each NWST instance by applying the NWST approximation algorithm is $O(((|V| + |E|) \log(|V| + |E|))K^2)$, because there are $|V| + |E|$ vertices including the dummy ones and $|T| = K + 2$.

VI. LAYERED SEARCH ALGORITHM

In this section, we propose a practical heuristic algorithm in polynomial time complexity, whose performance is comparable to that of the Steiner Tree based algorithm as will be demonstrated in Section VII.

The basic idea is to generate a graph with multiple layers, each of which is a copy of the original graph and contains the VMs of one VNF in the SFC. Dynamic programming is then applied to calculate a high-availability path across the layers. Finally, the path in the layered graph will be mapped back to the original graph, resulting in a path traversing each required VNF.

Our algorithm is inspired by the shortest path algorithm in [15], but has the following two major differences. First, the shortest path algorithm considers only edge weights, while

our algorithm considers both vertex and edge availabilities. Second, the shortest path algorithm adds weights of all edges, including those of repeated edges, on the path, but our algorithm multiplies the availabilities of unique components only, i.e., excluding those of repeated vertices and edges.

A. Generating Layered Graph

Before applying the algorithm, the first step is to generate a graph $G^* = (V^*, E^*)$ with $K + 1$ layers. Each of the first K layers contains the instances of the corresponding VNF in the SFC, and the $(K + 1)$-th layer contains the destination.

In detail, $K + 1$ copies of the original graph G are created, each being one of the $K + 1$ layers. For easy representation, we add a superscript k to denote the vertices in the k-th layer. For example, v^k and (u^k, v^k) denote the vertex and edge in layer k corresponding to v and (u, v) in the original graph G, respectively. The availability of a vertex v^k or edge (u^k, v^k) is equal to that of its counterpart in the original graph, i.e., $a_{v^k} = a_v$ and $a_{(u^k, v^k)} = a_{(u,v)}$. Hence, $V_k^k = \{v^k \in V^* | is(v, k) = 1\}$ is the set of VMs of VNF-k that are located in the k-th layer, and the path must traverse one of them to access the VNF-k service.

Next, the layers are connected by vertical edges with an availability of one. In detail, the k-th layer is connected with the $(k + 1)$-th layer via the edges (v^k, v^{k+1}) where $v^k \in V_k^k$, and their availabilities are one, i.e., $\forall v^k \in V_k^k, a_{(v^k, v^{k+1})} = 1$. Thus, only after a path visits one of the vertices in V_k^k, it can proceed to the next layer.

For the example in Fig. 4, since $K = 2$, a 3-layer graph is generated. Given $V_1 = \{v, w\}$, the vertical edges (v^1, v^2) and (w^1, w^2) connect the layers 1 and 2. Similarly, as $V_2 = \{u\}$, the vertical edge (u^2, u^3) connects the layers 2 and 3.

B. Dynamic Programming based Search Algorithm

Once the multiple-layer graph $G^* = (V^*, E^*)$ has been created, a dynamic programming based search algorithm will be applied to calculate the high-availability path.

Algorithm 2 shows the pseudo code of the Layered Search algorithm, in which $P(v^k)$ denotes the current path from the source s^1 to the vertex v^k and $A(v^k)$ denotes the availability of the path. Explanation of the pseudo code is as follows. The first step, lines 1 to 16, creates the $(K + 1)$-layer graph $G^* = (V^*, E^*)$. In detail, lines 3 to 5 generate the vertices, lines 6 to 8 generate the edges, and lines 9 to 15 generate the vertical edges.

The next step, lines 17 to 28, conducts initialization. In detail, line 17 initializes the source vertex s^1 in the first layer, by setting the path $P(s^1)$ to reach it as $[s]$ and the path availability $A(s^1)$ to be a_{s^1}. As the highest availability to s^1 is known, line 18 removes it from the vertex set V^*. Lines 19 to 28 initialize other vertices $v^k \in V^*$. There are three possible scenarios. First, lines 20 to 21 deal with the scenario where $s = v$ or in other words (s^1, v^k) is a vertical edge, and the path and availability of v^k are set to those of s^1. Second, lines 22 to 24 deal with the scenario where there is a non-vertical edge between s^1 and v^k, in which case the path to v^k is set

Algorithm 2 Layered Search Algorithm

Input: $G = (V, E), (s, d, K)$
Output: high-availability path $P(d^{K+1})$

1: $V^* = \emptyset$, $E^* = \emptyset$
2: **for** $k = 1$ **to** $K + 1$ **do**
3:    **for** $v \in V$ **do**
4:       $V^* = V^* \cup \{v^k\}$, $a_{v^k} = a_v$
5:    **end for**
6:    **for** $(u, v) \in E$ **do**
7:       $E^* = E^* \cup \{(u^k, v^k)\}$, $a_{(u^k, v^k)} = a_{(u,v)}$
8:    **end for**
9:    **if** $k > 1$ **then**
10:      **for** $v^{k-1} \in V^*$ **do**
11:         **if** $is(v, k-1) = 1$ **then**
12:            $E^* = E^* \cup \{(v^{k-1}, v^k)\}$, $a_{(v^{k-1}, v^k)} = 1$
13:         **end if**
14:      **end for**
15:    **end if**
16: **end for**
17: $P(s^1) = [s]$, $A(s^1) = a_{s^1}$
18: $V^* = V^* \setminus \{s^1\}$
19: **for** $v^k \in V^*$ **do**
20:    **if** $(s^1, v^k) \in E^*$ **and** $s = v$ **then**
21:       $P(v^k) = P(s^1)$, $A(v^k) = A(s^1)$
22:    **else if** $(s^1, v^k) \in E^*$ **then**
23:       $P(v^k) = P(s^1).append(v)$
24:       $A(v^k) = A(s^1)a_{(s^1, v^k)}a_{v^k}$
25:    **else**
26:       $A(v^k) = 0$
27:    **end if**
28: **end for**
29: **while** $d^{K+1} \in V^*$ **do**
30:    select u^l from V^* with highest availability, i.e., $\forall v^k \in V^*, A(u^l) >= A(v^k)$
31:    $V^* = V^* \setminus \{u^l\}$
32:    **for** $(u^l, v^k) \in E^*$ **do**
33:      **if** $u = v$ **then**
34:         **if** $A(v^k) < A(u^l)$ **then**
35:            $P(v^k) = P(u^l)$, $A(v^k) = A(u^l)$
36:         **end if**
37:      **else if** $(u, v) \in P(u^l)$ **then**
38:         **if** $A(v^k) < A(u^l)$ **then**
39:            $P(v^k) = P(u^l).append(v)$, $A(v^k) = A(u^l)$
40:         **end if**
41:      **else if** $v \notin P(u^l)$ **then**
42:         **if** $A(v^k) < A(u^l)a_{(u^l, v^k)}a_{v^k}$ **then**
43:            $P(v^k) = P(u^l).append(v)$
44:            $A(v^k) = A(u^l)a_{(u^l, v^k)}a_{v^k}$
45:         **end if**
46:      **end if**
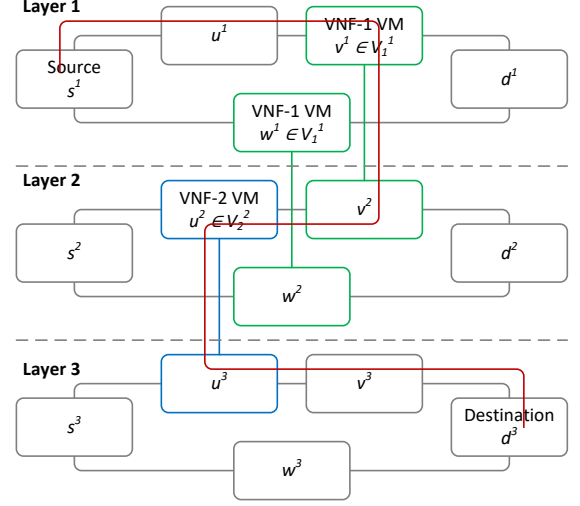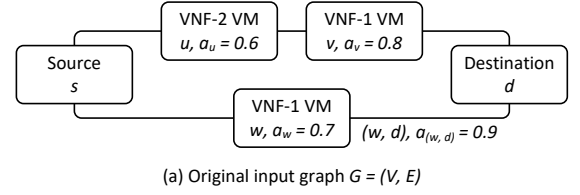47:   **end for**
48: **end while**

Fig. 4: Based on the input graph $G = (V, E)$, the Layered Search algorithm generates a $(K + 1)$-layer graph $G^* = (V^*, E^*)$.

by appending v to $P(s^1)$, and the path availability $A(v^k)$ is set by multiplying $A(s^1)$ with the availability $a_{(s^1, v^k)}$ of the edge and the availability a_{v^k} of v^k. Finally, lines 25 to 26 deal with the scenario where there is no edge between s^1 and v^k, and the availability of v^k is set to zero, i.e., not yet reachable.

The final step, lines 29 to 48, enters a loop to find the highest-availability path to one vertex in V^* at a time. In detail, line 30 selects from V^* the vertex u^l that has the highest availability, i.e., $\forall v^k \in V*, A(u^l) \geq A(v^k)$, and line 31 removes u^l from V^*. Next, lines 32 to 47 conduct the relaxation process. In other words, for each adjacent vertex v^k of the previously selected vertex u^l, if the new path of $P(u^l)$ appended with v has a higher availability, the path $P(v^k)$ and its availability are updated accordingly. There are also three possible scenarios. First, lines 33 to 36 deal with the scenario where $u = v$ or in other words (u^l, v^k) is a vertical edge, and the path and availability of v^k are set to those of u^l directly. Second, lines 37 to 40 deal with the scenario where the edge (v, u) is already in the path $P(u^l)$, in which case appending v^k to the path will not change the availability, since the edge (v, u) and vertex u are repeating components. Finally, lines 41 to 46 deal with the scenario where u is not in the path $P(u^l)$.

For the example in Fig. 4, the path returned by the Layered Search algorithm is s-u-v-u-v-d with an end-to-end availability of 0.48, as shown by the red line. In detail, when the path is projected in the layered graph G^*, it starts from s^1 in layer 1, traverses u^1, and visits v^1 to access VNF-1 service. After that, it advances to layer 2 via the vertical edge (v^1, v^2), and

continues from v^2 to u^2 to access VNF-2 service. Finally, the path advances to layer 3 via the vertical edge (u^2, u^3), traverses v^3, and reaches the destination d^3.

C. Time and Optimality Analysis

The time complexity of the Layered Search algorithm is $O(|V|K\log(|V|K) + |E|K)$, as it has the same time complexity as the Dijkstra's algorithm, and the layered graph $G^* = (V^*, E^*)$ has $O(|V|K)$ vertices and $O(|E|K)$ edges.

Because Theorem 1 shows that the HAP problem is NP-hard, the Layered Search algorithm in polynomial time complexity will not be able to always deliver the optimal result. The reason is that the sub-problems in the dynamic programming process are not independent but correlated. Specifically, whether the path from the source s^1 to an intermediate vertex v^k is part of an optimal solution also depends on the remaining path from v^k to the destination d^{K+1}, because an optimal path from s^1 to v^k may provide fewer repeating components for the remaining path, and hence results in a sub-optimal end-to-end availability.

For the example in Fig. 4, if the VNF-1 VM w has an availability of $a_w = 0.85$, the Layered Search algorithm will return a sub-optimal solution s-w-s-u-s-w-d, because the sub-path from s to u via w, i.e., s^1-w^1-w^2-s^2-u^2 as projected in G^*, has a higher availability than the alternative via v, i.e., s^1-u^1-v^1-v^2-u^2. However, after multiplying the availability of the remaining path from u to d, the entire path via w has a lower end-to-end availability than that of the alternative via v, because the latter can take advantage of repeating components in the remaining path.

VII. Performance Evaluation

In this section, we present extensive evaluation data to demonstrate the effectiveness of the Layered Search algorithm, which achieves comparable performance to that of the Steiner Tree based algorithm with much reduced running time, and outperforms other compared algorithms.

A. Benchmark Algorithms

The following five algorithms are evaluated and compared against each other.

1) The *Steiner Tree based algorithm* in Section V breaks the input into many NWST instances, and applies the NWST approximation algorithm [20] to each instance.
2) The *Layered Search algorithm* in Section VI uses dynamic programming to search a multi-layer graph.
3) *Greedy* uses a greedy approach to determine the VM for the next VNF in the SFC, i.e., applying Dijkstra's algorithm from the current vertex, and selecting the first visited VM of the next VNF, which is also the VM with the highest availability from the current vertex.
4) *Random Steiner Tree* (RST) selects a random VM for each VNF in the SFC, and applies the NWST approximation algorithm [20] to determine the path.
5) *Random* calculates the path by randomly selecting the next hop until the path has visited one VNF for each VNF in the SFC.

B. Topologies

To comprehensively evaluate the algorithms, we consider three different topologies as described below.

1) The *Fat Tree* topology [5] is widely used in data center networks due to its large bi-section bandwidth and multi-path connectivity. We consider an 8-pod fat tree with 128 servers (all at the leaves), 80 switches, and 272 links.
2) The *75-node US mesh*, abbreviated as *US-75*, topology [25] represents a real-world network across the US continent. Each of the 75 nodes is an interconnecting switch, and has one or two servers connected to it.
3) The *Binary Tree* topology is a classical topology that abstracts the tree structure used by traditional networks. We consider an 8-layer binary tree with 128 servers (all at the leaves), 127 switches, and 254 links.

C. Simulation Settings

The simulations parameters are set up as follows. There are a total of ten different VNFs, and every VNF has three to five VMs in the network. Each VNF VM is hosted by a random server. Each flow has a random source and destination, and an SFC consisting of four to six VNFs randomly selected from the available ones. For comprehensive evaluation at different availability levels, the simulations are conducted with four component availability ranges [20]: $[0.9, 0.99]$, $[0.99, 0.999]$, $[0.999, 0.9999]$, and $[0.9999, 0.99999]$, and the availability of any node or link is a random number from the selected range. The simulations are conducted on computers each configured with four Intel Xeon E5-2650L CPUs with a max frequency at 2.5 GHz and 503 GB memory. The presented data are collected from 1000 simulation runs.
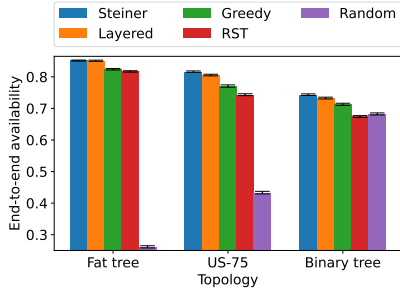
D. End-to-End Availability Data

First, we compare the end-to-end availability results of different algorithms. Fig. 5(a) shows the mean and 99% confidence interval for the availability of different algorithms with the component availability range of $[0.9, 0.99]$. As can be seen, the Steiner Tree based algorithm achieves the highest end-to-end availability at 20.1%, 12.6%, and 5.2% in the fat tree, US-75, and binary tree topologies, respectively. The Layered Search algorithm achieves a comparable availability at 19.4%, 11.7%, and 4.6%, respectively, with a difference equal to or less than 1%, but at much reduced time complexity as will be shown by the running time data in Fig. 6. On the other hand, the Layered Search algorithm outperforms all the remaining algorithms: Greedy, RST, and Random. As an extreme case, the availability of Random is close to zero in the fat tree and US-75 topologies due to its inefficient path search strategy.

Fig. 5(b), (c), and (d) show the end-to-end availability results with the component availability ranges of $[0.99, 0.999]$, $[0.999, 0.9999]$, and $[0.9999, 0.99999]$, respectively. As the availability of each component increases, the end-to-end availability increases for all the algorithms, and the confidence interval becomes tighter. The performance rank among different algorithms is mostly consistent that the Steiner Tree algorithm achieves the highest availability, and the Layered
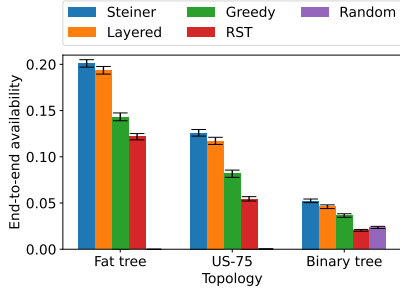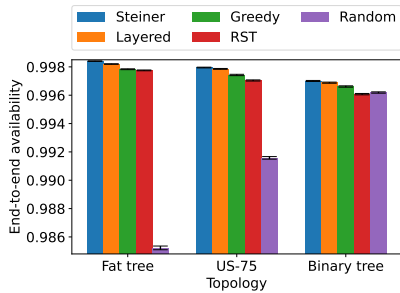
(a) Component availability range [0.9, 0.99]

(b) Component availability range [0.99, 0.999]
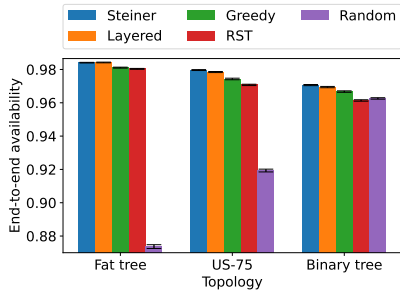
(c) Component availability range [0.999, 0.9999]

(d) Component availability range [0.9999, 0.99999]

Fig. 5: Mean and 99% confidence interval for end-to-end availability with different component availability ranges.

Search algorithm achieves a comparable result, followed by Greedy, RST, and Random.

Next, we compare the end-to-end availability results under different topologies. As shown by all the figures in Fig. 5, in general, the availability of an algorithm in the fat tree is higher than that in the US-75 mesh, which is higher than that in the binary tree, all other things being equal. This can be explained by the different levels of connectivity between servers enabled by the different topologies. The fat tree has the best connectivity because multiple paths exist between any
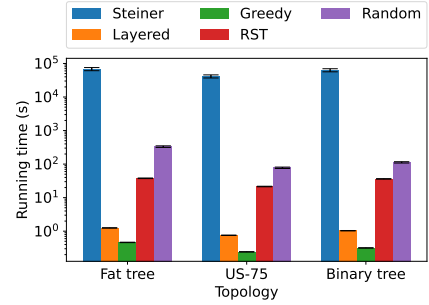

Fig. 6: Mean and 99% confidence interval for running time with component availability range of [0.9, 0.99].

pair of servers. Similarly, the US-75 mesh is well connected with multiple available paths. The binary tree has the worst connectivity with only a single path between any pair of servers. Better connectivity enables more potential paths, and hence an efficient algorithm is more likely to find a shorter path, which translates to a higher end-to-end connectivity. However, the Random algorithm is an exception, whose end-to-end availability in the fat tree is lower than that in the US-75 mesh, which is lower than that in the binary tree. Ironically, with the in efficient random search strategy, better connectivity leads to a longer random walk, and hence a lower end-to-end availability.

E. Running Time Data

Fig. 6 compares the simulation running time of different algorithms with the component availability range of [0.9, 0.99]. Although the Steiner Tree based algorithm achieves the best end-to-end availability, it has the longest running time as well due to its high time complexity, which are 68251, 41344, and 63102 seconds on average in the fat tree, US-75 mesh, and binary tree, respectively. The Layered Search algorithm runs much faster by four orders of magnitude, at 1.24, 0.75, and 1.03 seconds, respectively. The Greedy algorithm has the shortest running time without the need to search multiple layers. Except Greedy, the Layer Search algorithm is faster than RST and Random by one and two orders of magnitude, respectively. Different component availability ranges do not affect the algorithm running time, and hence the data for other ranges are omitted.

VIII. Conclusions

The software nature of VNFs introduces new challenges to the availability of flow paths. Existing research on VNF availability is not capable of analyzing all the possible hardware and software failures, and has not considered the correlation between repeating components in a path. In this paper, we propose a theoretical model to effectively analyze the end-to-end availability of a path in VNF networks. To the best of our knowledge, the proposed model is the first to comprehensively consider all possible hardware and software failures, and the first to model repeating components. On top of the model, we formulate the Highest Availability Path problem to find

the path with the highest availability in a VNF network, and prove its NP-hardness by reduction from the Node-Weighted Steiner Tree problem. Next, we propose two algorithms, one based on an NWST approximation algorithm, the other using a dynamic programming approach to search a multi-layer graph. Finally, we present extensive evaluation data to demonstrate the effectiveness of the Layered Search algorithm, which achieves a high end-to-end availability in polynomial time. In our future work, we plan to enhance the availability analytical model by considering failures caused by traffic congestion to make it more realistic, in which case the capacity of a node and link will also be constraints for optimization.

REFERENCES

[1] "Cisco MGX 8250 Reliability, Availability, and Serviceability (RAS)," https://www.cisco.com/en/US/docs/switches/wan/mgx/mgx_8850/software/mgx_r1.1.3/mgx_8250/overview/mgx8ps9.pdf.

[2] M. A. Abdelaal, G. A. Ebrahim, and W. R. Anis, "High availability deployment of virtual network function forwarding graph in cloud computing environments," *IEEE Access*, vol. 9, pp. 53 861–53 884, 2021.

[3] B. Addis *et al.*, "Virtual network functions placement and routing optimization," in *IEEE CloudNet*, 2015, pp. 171–177.

[4] S. Agarwal *et al.*, "Joint VNF placement and CPU allocation in 5G," in *INFOCOM*. IEEE, 2018, pp. 1943–1951.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM computer communication review*, vol. 38, no. 4, pp. 63–74, 2008.

[6] A. K. Alnaim, A. M. Alwakeel, and E. B. Fernandez, "A pattern for an nfv virtual machine environment," in *2019 IEEE International Systems Conference (SysCon)*. IEEE, 2019, pp. 1–6.

[7] A. A. Barakabitze *et al.*, "5g network slicing using sdn and nfv: A survey of taxonomy, architectures and future challenges," *Computer Networks*, vol. 167, p. 106984, 2020.

[8] M. T. Beck, J. F. Botero, and K. Samelin, "Resilient allocation of service function chains," in *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2016, pp. 128–133.

[9] L. Ben Azzouz and I. Jamai, "Sdn, slicing, and nfv paradigms for a smart home: A comprehensive survey," *Transactions on Emerging Telecommunications Technologies*, vol. 30, no. 10, p. e3744, 2019.

[10] M. S. Bonfim, K. L. Dias, and S. F. Fernandes, "Integrated nfv/sdn architectures: A systematic literature review," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–39, 2019.

[11] Y. Chen and J. Wu, "Latency-efficient vnf deployment and path routing for reliable service chain," *IEEE Transactions on Network Science and Engineering*, 2020.

[12] T. H. Cormen *et al.*, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[13] W. Ding, H. Yu, and S. Luo, "Enhancing the reliability of services in nfv with the cost-efficient redundancy scheme," in *IEEE international conference on communications (ICC)*, 2017, pp. 1–6.

[14] N.-T. Dinh and Y. Kim, "An efficient availability guaranteed deployment scheme for iot service chains over fog-core cloud networks," *Sensors*, vol. 18, no. 11, p. 3970, 2018.

[15] A. Dwaraki and T. Wolf, "Adaptive service-chain routing for virtual network functions in software-defined networks," in *ACM SIGCOMM Workshop on Hot topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, 2016, pp. 32–37.

[16] J. Fan *et al.*, "Availability-aware mapping of service function chains," in *IEEE Conference on Computer Communications (INFOCOM)*, 2017, pp. 1–9.

[17] I. Farris *et al.*, "A survey on emerging sdn and nfv security mechanisms for iot systems," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 812–837, 2018.

[18] X. Fei *et al.*, "Adaptive vnf scaling and flow routing with proactive demand prediction," in *IEEE INFOCOM*, 2018, pp. 486–494.

[19] D. Gedia and L. Perigo, "Performance evaluation of sdn-vnf in virtual machine and container," in *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2018, pp. 1–7.

[20] S. Guha and S. Khuller, "Improved methods for approximating node weighted steiner trees and connected dominating sets," *Information and computation*, vol. 150, no. 1, pp. 57–74, 1999.

[21] F. He, T. Sato, and E. Oki, "Optimization model for backup resource allocation in middleboxes with importance," *IEEE/ACM Transactions on Networking*, vol. 27, no. 4, pp. 1742–1755, 2019.

[22] Y. Kanizo *et al.*, "Optimizing virtual backup allocation for middleboxes," *IEEE/ACM Transactions on Networking*, vol. 25, no. 5, pp. 2759–2772, 2017.

[23] ——, "Designing optimal middlebox recovery schemes with performance guarantees," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 10, pp. 2373–2383, 2018.

[24] P. Klein and R. Ravi, "A nearly best-possible approximation algorithm for node-weighted steiner trees," *Journal of Algorithms*, vol. 19, no. 1, pp. 104–115, 1995.

[25] J. Kong *et al.*, "Guaranteed-availability network function virtualization with network protection and vnf replication," in *IEEE Global Communications Conference*, 2017, pp. 1–6.

[26] D. Li *et al.*, "Availability aware vnf deployment in datacenter through shared redundancy and multi-tenancy," *IEEE Transactions on Network and Service Management*, vol. 16, no. 4, pp. 1651–1664, 2019.

[27] L. Liu *et al.*, "Joint dynamical vnf placement and sfc routing in nfv-enabled sdns," *IEEE Transactions on Network and Service Management*, vol. 18, no. 4, pp. 4263–4276, 2021.

[28] C. Lund and M. Yannakakis, "On the hardness of approximating minimization problems," *Journal of the ACM (JACM)*, vol. 41, no. 5, pp. 960–981, 1994.

[29] Z. Lv and W. Xiu, "Interaction of edge-cloud computing based on sdn and nfv for next generation iot," *IEEE Internet of Things Journal*, vol. 7, no. 7, pp. 5706–5712, 2019.

[30] P. Mandal, "Comparison of placement variants of virtual network functions from availability and reliability perspective," *IEEE Transactions on Network and Service Management*, 2022.

[31] J. Meza *et al.*, "A large scale study of data center network reliability," in *ACM Internet Measurement Conference (IMC)*, 2018, pp. 393–407.

[32] G. Moualla, T. Turletti, and D. Saucez, "An availability-aware sfc placement algorithm for fat-tree data centers," in *IEEE International Conference on Cloud Networking (CloudNet)*, 2018, pp. 1–4.

[33] B. K. Mukherjee *et al.*, "An SDN based distributed iot network with nfv implementation for smart cities," in *International Conference on Cyber Security and Computer Science*. Springer, 2020, pp. 539–552.

[34] L. Qu *et al.*, "Reliability-aware service function chaining with function decomposition and multipath routing," *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 835–848, 2019.

[35] L. Qu, M. Khabbaz, and C. Assi, "Reliability-aware service chaining in carrier-grade softwarized networks," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 3, pp. 558–573, 2018.

[36] G. Sallam *et al.*, "Shortest path and maximum flow problems under service function chaining constraints," in *IEEE INFOCOM*, 2018, pp. 2132–2140.

[37] S. Sharma *et al.*, "Vnf availability and sfc sizing model for service provider networks," *IEEE Access*, vol. 8, pp. 119 768–119 784, 2020.

[38] M. M. Tajiki *et al.*, "Joint energy efficient and qos-aware path allocation and vnf placement for service function chaining," *IEEE Transactions on Network and Service Management*, vol. 16, no. 1, pp. 374–388, 2018.

[39] M. Wang *et al.*, "Availability-and traffic-aware placement of parallelized sfc in data center networks," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 182–194, 2021.

[40] Y. Wu *et al.*, "Reliability-aware vnf placement using a probability-based approach," *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 2478–2491, 2021.

[41] L. Yala, P. A. Frangoudis, and A. Ksentini, "Latency and availability driven vnf placement in a mec-nfv environment," in *IEEE Global Communications Conference (GLOBECOM)*, 2018, pp. 1–7.

[42] B. Yi *et al.*, "A comprehensive survey of network function virtualization," *Computer Networks*, vol. 133, pp. 212–262, 2018.

[43] J. Zhang *et al.*, "Raba: Resource-aware backup allocation for a chain of virtual network functions," in *IEEE Conference on Computer Communications (INFOCOM)*, 2019, pp. 1918–1926.