Sādhanā (2021)46:33
https://doi.org/10.1007/s12046-020-01530-y

An online distributed approach to Network Function Placement in NFV-enabled networks

ANIX ANBIAH* and KRISHNA M SIVALINGAM

Department of Computer Science and Engineering, Indian Institute of Technology Madras, Chennai, India
e-mail: anix@cse.iitm.ac.in; skrishnam@cse.iitm.ac.in

MS received 22 January 2020; revised 24 October 2020; accepted 28 October 2020

Abstract. Network Function Placement (NFP) involves placing virtual network functions (VNFs) on the nodes of a network such that the data that flow through the network are processed by a chain of service functions along their path from source to destination. There are three aspects to this problem: (i) *routing* the flows efficiently through the network, (ii) *placement* of the VNFs on the nodes and (iii) *steering* each flow through a chain of VNFs, known as the service function chain (SFC). Routing must attempt to find "optimal" paths through the network (for e.g., shortest paths), possibly subject to constraints such as path latency and link bandwidth. The VNFs consume resources on the nodes where they are placed and are constrained by the capacity of the nodes. Steering must ensure that each flow has along its path a sequence of VNFs, likely in a certain order. One way to specify this problem is to define a multi-commodity flow problem with additional constraints based on the steering and placement requirements. Simultaneously solving all three aspects of this problem, trying to optimize various parameters and within the various constraints, is a hard problem, with even a simplified version shown to be NP-complete in this paper. Attempting to optimally solve this problem in real time while flows are getting provisioned and de-provisioned in parallel is an intractable problem, especially in large networks. Hence various types of heuristics have been used to solve this problem. In this paper we introduce a distributed, online solution that employs a message-passing protocol for nodes to negotiate the placement of the VNFs, with the minimization of the number of VNF instances being the primary objective. We compare the performance of the solution to that of the theoretically optimal solution and other proposed heuristics on both the Fat-tree topology and the BCube topology. The results show that this solution performs better than other heuristics. The average ratio of the result of the proposed solution to that of the optimal solution, taken as the approximation ratio, is found to be 1.5 for the tested scenarios.

Keywords. Software-Defined Networks; Network Function Virtualization; Network Function Placement; distributed.

1. Introduction

Software-Defined Network (SDN) and Network Function Virtualization (NFV) have gained much research and industry attention. The main objective of these research areas is to avoid proprietary- and hardware-based solutions in data and wireless networks. Software-Defined Networking aims to "open up" networking by creating abstractions of the control plane and data plane. This allows users to create their own implementation, enabling them to change the policy and behaviour of the networks according to their needs. NFV, on the other hand, allows for *middlebox* [1] hardware to be replaced by Virtual Network Functions (VNFs) [2]. The use of virtual functions has the advantage of upgrading the functions easily, making them elastic to be able to scale based on demand and allowing them to be placed or moved wherever resources are available within the network as needed to optimally meet the demand.

In this paper, we address Network Function Placement (NFP), a specific problem in NFV. NFV virtualizes network functions that operate on data flows into VNFs. They were traditionally implemented in dedicated hardware. Data flows through the network require a sequence of VNFs to be placed along their path. Such a sequence of VNFs is called a service function chain (SFC), sometimes referred to as a VNF Forwarding Graph (VNF-FG) [3]. The NFP problem involves (i) routing the flows through the network from source to destination, (ii) placing VNF instances in a network and (iii) steering the flows through SFCs such that the SFC requirements of all flows through a network are satisfied.

Part of this paper was presented as a Poster at the IEEE LCN 2017 Conference in Singapore.

*For correspondence

Published online: 05 February 2021

Each aspect of this problem has various possible objectives for optimization and a set of constraints. For example, routing the flows likely attempts to take the shortest path for each flow subject to constraints such as latency or bandwidth. Steering must follow an ordered sequence of VNFs for each flow with each VNF instance being able to handle a certain amount of flow, usually measured as bit rate. Placement must adhere to the resource constraints on the nodes (servers and switches) in the network and minimize the number of VNF instances in the network. Jointly optimizing all three aspects of this problem subject to all constraints is an intractable problem. In fact, later in this paper, we show that, even if the optimal paths for given set of flows through the network are fixed, placing the VNFs such that the number of VNF instances is minimized is an NP-complete problem. It is this definition of the problem (with fixed optimal paths for the flows and the objective to minimize the VNF instances) that we will solve in this paper. A distributed message-passing protocol that allows network nodes to negotiate the placement of VNF instances among themselves is presented as the proposed solution.

An illustration of placement of VNF instances given the flows, their paths through the network and their SFC requirements is shown in figure 1. F1–F3 are flows, each with a sequence of VNFs in its SFC. The VNFs are represented by the various shapes. Each instance of a VNF is shown within the node hosting the instance and placed on the flow(s) serviced by that instance. In this example, the total number of VNF instances is 4.

The main contributions of this paper are (i) proposing a distributed approach to NFP, where network nodes negotiate the placement of VNFs, (ii) presenting an algorithm to minimize number of VNF instances required to satisfy the SFC requirements of a given set of flows, (iii) showing that the solution works online by taking an "always on" approach where optimal VNF placement is continually computed and implemented even as flows get provisioned and de-provisioned. This paper is an extension of our earlier work titled *Funplace: A protocol for Network Function Placement* [4]. Our prior paper published our preliminary work on the Funplace protocol for NFP. This paper is a

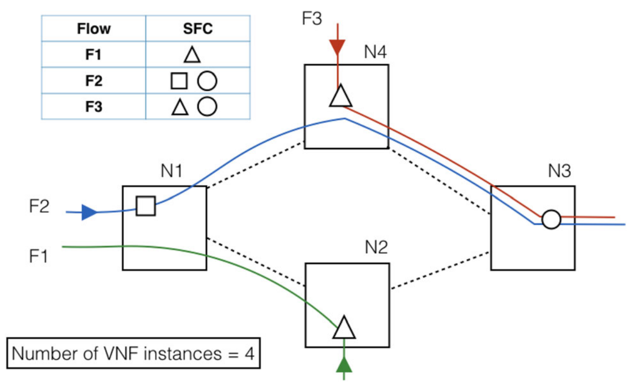


Figure 1. An illustration of Network Function Placement.

more detailed analysis of the NFP problem in general, including proof of its NP-completeness and comparison to other published heuristic solutions. Moreover, the current paper includes more detailed experimentation of the protocol on networks of various sizes and topologies and corresponding results.

The details of the content newly added in this paper are (i) the sequence diagram of exchange of protocol messages, shown in figure 3, (ii) a more detailed comparison of our work to other published work in this area, included in Section 2 and summarized in table 1 and (iii) more use cases of the protocol provided in Section 4.5, illustrating consolidation (Section 4.5a), branching of a request (Section 4.5c) and symmetry breaking (Section 4.5d), illustrated in figures 5, 7 and 8, respectively, (iv) an illustration of the use case involving a tie-breaker, as shown in figure 6, (v) testing of Funplace on much larger networks compared with the previous paper such as the Fat-tree topology of varying sizes, described in Section 5 and shown in figure 11, (vi) detailed results from testing Funplace on a 104-node network of the Fat-tree topology, shown in figures 13 and 14 and described in Section 5.2a, (vii) results from testing the protocol on a 656-node network of Fat-tree topology, shown in figures 15 and 16 and described in Section 5.2b, (viii) testing of Funplace on a 704-node network with a BCube topology, shown in figure 17 and described in Section 5.2c, (ix) the results of testing the distributed solution for an online (dynamic) case, described in Section 5.2d and shown in figure 18 and (x) NP-completeness proof of the NFP problem as given in Appendix I.

Based on the experimentation presented in this paper, it is observed that the convergence time of the distributed solution using Funplace does not increase with the number of flows in the network and depends only on the length of the SFCs. Test results for networks with Fat-tree [5] and BCube [6] topologies show an *approximation ratio* of 1.5, taken as the average ratio of the number of function instances computed by the proposed solution to that of the globally optimal solution. The optimal solution is computed using an Integer Linear Programming (ILP) formulation of this problem, which is shown in this paper to be NP-complete. Funplace is shown to be a dynamic (i.e. *online*) solution where optimization can proceed while flows are being provisioned in the network continually. This is a distinct advantage of this distributed solution over any ILP-based centralized solution. Moreover, a distributed solution where the network elements negotiate the placement of VNF instances among themselves using a protocol has not been addressed in any prior work by other researchers.

The rest of the paper is organized as follows. Section 2 presents the required background material and related work. Section 3 presents the NFP and its complexity. Section 4 describes the proposed distributed approach for solving the NFP problem using Funplace. It provides details of the protocol, the message format, the algorithm and its analysis. Section 5 provides the details of the simulation-

Table 1. Comparison with related work on NFP.

Authors	Online	Distributed	ILP/MIQCP	Heuristic	Flow based	VNF Migration
Beck and Botero [3]	×	×	✔	✔	✔	×
Charikar *et al* [7]	×	×	✔	×	✔	×
Baumgartner *et al* [10]	×	×	✔	×	×	×
Mehraghdam *et al* [11]	×	×	✔	×	✔	×
Kodirov *et al* [12]	×	×	✔	✔	✔	×
Moens and Turck [13]	×	×	✔	×	×	×
Mohammadkhan *et al* [14]	✔	×	✔	✔	✔	×
Lin *et al* [15]	×	×	✔	✔	✔	×
Addis *et al* [16]	×	×	✔	✔	✔	×
Ghaznavi *et al* [17]	✔	×	✔	✔	✔	✔
Our approach/solution	✔	✔	✔	✔	✔	✔

based performance study and the key findings. Section 6 summarizes the paper and presents the important conclusions.

2. Background and related work

This section presents the relevant background material and a summary of key related works.

2.1 *Background*

NFV allows network functions such as Firewall, Transcoder, Deep Packet Inspection (DPI), etc. to be implemented in software as VNFs instead of using proprietary hardware known as *middleboxes*. There has been a lot of work in this area addressing various aspects such as the virtualization, placement, scaling, performance, etc. In this paper, the focus is on the problem of placement of VNFs, commonly termed as Network Function Placement (NFP). It includes aspects of routing of flows, steering of flows through SFCs and placement of VNFs in the network. Prior instances of work on this problem have defined the problem and the objectives of solving it differently depending on the specific aspects of the problem being addressed. In this section, we will summarize the various approaches to this problem and the common assumptions made when solving this problem.

There is no single, well-known definition of this problem *per se*. Roughly speaking, the problem considers a physical network, a set of SFCs and a set of flows. The network comprises links with capacity to carry the flows and nodes (switches and servers) with capacity to host the VNFs. The objective is to place the VNFs and form the SFCs and simultaneously route the flows through the SFCs, with each flow being routed from its source to destination. This is similar to solving a multi-commodity flow problem, but with additional constraints [7]. Sometimes, a set of flows is not given; rather, the maximum possible aggregate flow through the SFCs is determined for a given placement of VNFs.

A key issue to be considered is the optimization criteria. The solution to the problem may try to optimize one of the following: (i) maximize the flow through the SFCs, (ii) minimize the maximum congestion on any link in the network for a given set of flows or (iii) minimize the number of VNF instances for a given fixed set of flows. In this paper, our optimization objective is to minimize the number of VNF instances for a given set of flows and their SFC requirements.

The assumptions made while solving the problem must also be noted. One of the advantages of virtualizing the network functions is that they can be implemented wherever processing and memory resources are available and scaled according to demand. Another possible advantage is that network function may be implemented as fine-granular functions and placed along the flows [8]. This also means that these fine-granular functions can be migrated dynamically from one network node to another to accommodate more flows or consolidate the instances in order to minimize the overall number of instances. Prior work in this area has shown that it is possible to dynamically migrate VNFs between nodes or servers [9]. In this paper, we assume that the VNF instances are fine-granular and can be migrated between nodes to facilitate optimization. We also assume that VNFs can be migrated with relatively low overhead and can be instantiated on generic nodes with minimal processing power and memory. This is possible since fine-grained VNFs can be implemented on *containers* or *unikernels* rather than separate servers or even virtual machines.

Another assumption is about the type of resources such as computation, network and storage available to process the flows. One approach is to assume, without loss of generality, that there is one type of resource and that one unit of processing is required to process a single unit of flow [7]. We make this assumption in this paper. Resources can also be available on network switches or servers connected to the switches. It must be noted that when it comes to routing, steering and placement this does not make a significant difference and this detail can be abstracted away as a method of simplification. In other words, the resources can be thought of as available on the network nodes that are either switches or switches with servers attached to them. This is another assumption followed in this paper, similar to the assumptions made in [10].

Some of the prior work considers or allows re-ordering of the SFC during optimization [11]. This is because it is possible that middleboxes can be placed in different orders along the path of flows and still achieve the same end-to-end behaviour. However, there may be a requirement for a partial order where certain functions must occur before certain other functions along the path. In this case, the functions can be sorted in a topological order before placement [12]. It must also be noted that network functions can increase, decrease or copy the flow as a result of the processing. As an example, a transcoder may result in a compressed flow as its output and load balancer may produce multiple streams of the same flow. This can be taken advantage of by placing the VNFs that compress flows closer to the source [11] and by placing the VNFs that increase or multiply the flow closer to the destination. Finding the optimal ordering of the VNFs is sometimes referred to as the *Chain Composition* problem [3]. In this paper, we assume that the VNFs are pre-ordered and provided as input.

Another factor to consider is whether to optimize the placement and the chaining for each individual flow or for a group of flows. In other words, a portion of the flow space can be considered as a single unit for which VNF placement is considered [8]. In this paper, we consider all flows between the same source–destination pair and the same SFC requirement as belonging to a *flow class*. Rather than optimizing VNF placement for individual flows, we optimize for *flow classes*. This greatly enhances the scalability of our solution.

Given that a particular definition of this problem has a maximization or minimization objective with several constraints to be satisfied, Integer or Mixed Integer Programming is frequently used as a method to solve the problem [10, 11, 13]. Due to the time required to compute optimal solutions using this method, it is not suitable for real-time use. Hence, several heuristic solutions have been developed [12, 14]. Our approach can be considered as a heuristic approach. However, it is completely distributed with none of the nodes working with global information about all flows or end-to-end SFC requirements and involves local negotiation between neighbouring nodes to optimize the placement. This is a unique approach compared with prior work in this area.

2.2 *Related work*

Most of the prior works on this problem include a form of Integer or Mixed Integer Programming with linear or polynomial constraints to solve the problem [3, 11, 15, 16]. They attempt to solve NFP as a unified problem handling routing, steering and placement. As mentioned earlier, the optimization criteria may differ based on the problem definition. The work by Mehraghdam *et al* [11] includes a Pareto analysis to compare various objectives and shows that the three metrics can have trade-offs but are not necessarily conflicting.

A detailed comparison of the work presented in this paper with similar work on the NFP problem by other researchers is shown in table 1. This table indicates whether or not the following characteristics can be attributed to each paper: (i) dynamic placement of VNFs is possible while flows are provisioned in parallel, indicating an *online* solution, (ii) a *distributed* solution was used where nodes negotiated the placement among themselves, (iii) an ILP or an MIQCP formulation and a solver were used to identify optimal solution, (iv) a heuristic solution was used, (v) the problem considered individual flows with each flow being routed between a source–destination pair and (vi) migration of VNF instances between nodes was considered or allowed. It can be observed from this table that this paper is unique in its *distributed* approach to the problem where network nodes negotiate the placement of VNF instances among themselves.

A survey of NFP-related research work is presented in [18]. Some of the more recent works on NFP such as [19, 20] have introduced new methods, but follow a familiar theme of ILP-based solution and heuristics. Prior works by the authors have considered other solutions for NFP in SDNs [21, 22].

3. The NFP problem

The objective of the NFP problem, as defined in this paper, is to minimize the number of instances required to satisfy the SFC requirements of the flows. Prior work primarily treated this as a constraint satisfaction problem, using linear programming to find optimal solutions. In contrast, a distributed approach that uses a message-passing network protocol to solve the problem will lend itself naturally to using the switches for computation and allow them to negotiate the placement of VNF instances among themselves. *Funplace* is used to demonstrate the distributed approach as a viable alternative. An *ILP* formulation is used to find optimal solutions for comparison and to show that the protocol yields fairly optimal solutions.

3.1 *Problem definition*

Before describing the details of our solution, the problem being solved is formally defined in this section. Let the network be represented as usual by a graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$. Each node $N_n \in \mathcal{N}$ in the network has the ability to host one or more VNF instances. C_n is the capacity of N_n to host VNFs given in flow units. Let $\mathcal{V}$ be the set of VNFs. Let $\mathcal{F}$

be a set of flows through the network, each being a single unit of flow. The path taken by flow F_f is given by $P_{f,n} = k$, where k is the position of node N_n along the path of F_f. SFC of flow F_f is denoted as $S_{f,v} = l$ where l is the position of the VNF V_v in the chain. The problem involves finding $L_{f,v,n}$ – the value being set to 1 if for flow F_f, VNF V_v is serviced at node N_n and 0 otherwise. Each flow must be serviced by the sequence of VNFs in its SFC, in the same order along the path taken by the flow through the network. Each node hosts at most one instance of a given VNF. The total number of VNF instances in the network is given by $\sum_n \sum_v (\max_f L_{f,v,n})$. The objective of solving the problem is to minimize the number of VNF instances in the network.

3.2 *NFP is NP-complete*

The NFP problem defined earlier is NP-complete. This can be proved by reduction from the *Set Cover* problem. This reduction proof is given in Appendix I.

3.3 *ILP formulation*

To find the optimality of the solutions computed by Funplace, we use an ILP formulation of the NFP problem to solve the exactly same problem instances as those of Funplace and compare the results. In this section, we describe this formulation. The input parameters, variables, constraints and the objective are listed in table 2.

The objective is to minimize the total number of VNF instances in the network to satisfy the SFC requirements of the flow. This is given by $\sum_n \sum_v (\max_f L_{f,v,n})$ in table 2c, where $L_{f,v,n}$ is the decision variable. This decision variable, as shown in table 2b, indicates whether a given function v is placed at a given node n for flow f.

The ILP parameters are given in table 2a and provide the input of flows, their paths and their SFC requirements. Parameter $P_{f,n}$ is used to identify the path taken by each flow through the network. For a given flow f, $P_{f,n}$ is the position of node n along the path taken by f. Parameter $S_{f,v}$ specifies the SFC of each flow in the network. For a given flow f, $S_{f,v}$ indicates the position of VNF v in the SFC of the flow.

The constraints shown in table 2d are used to ensure that VNF instances are placed according to the SFC requirements of each flow in the given order along the part of the flow. The constraint C_1 given by $S_{f,v} = 0 \rightarrow \forall_n \; L_{f,v,n} = 0$ ensures that if a given VNF v is not applicable to a flow f, then it is not placed for that flow on any node in the network. The constraint C_2 given by $S_{f,v} > 0 \rightarrow \exists_n \; L_{f,v,n} = 1$ is the converse of C_1 and ensures that if VNF v is applicable to f, then a node n must exist where v is placed for the flow f. The final constraint C_3 is given by $\forall_{v_1,v_2,f} \; S_{f,v_1} < S_{f,v_2} \rightarrow \exists_{n_1,n_2} \; L_{f,v_1,n_1} = 1, L_{f,v_2,n_2} = 1, P_{f,n_1} \leq P_{f,n_2}$. For a given flow f and a given pair of VNFs v_1 and v_2, if v_1 occurs *before* v_2 in the SFC, the node n_1 where v_1 is placed for f must not occur *after* the node n_2 where v_2 is placed for f along the path of f. Constraints C_1 and C_2 ensure that a VNF instance is placed for a given flow if and only if required by its SFC. Constraint C_3 ensures that the VNF instances are placed in the same order as specified in the SFC, while C_4 is used to make sure that each node's capacity to host VNFs is not exceeded.

This ILP formulation was implemented using *AMPL* code, solved using a *Gurobi* solver and used to compute the globally optimal solution for instances of the NFP problem. Even though the ILP formulation is useful for this purpose, it is not a scalable and practical solution in a real network as the number of decision variables required is $O(|\mathcal{F}||\mathcal{V}||\mathcal{N}|)$ where $\mathcal{F}, \mathcal{V}$ and $\mathcal{N}$ are the sets of flows, VNFs and network nodes, respectively. Note that the formulation shown here is illustrative and includes non-linear constraints and objectives for readability. The actual formulation to solve the problems used standard conversion techniques to avoid non-linearity. The ILP approach yields optimal results for static instances of NFP. It cannot handle dynamic NFP when flows are being added and removed from the network. In contrast, *Funplace* is a dynamic protocol that constantly evaluates current network state and adjusts the placement of VNF instances.

4. Funplace: a distributed protocol for NFP

The NFP problem, as shown in Appendix A, is NP-complete even with a single network function. Therefore, for large networks having a large number of flows, the problem of finding the globally optimal solution with the least number of function instances is intractable. The problem therefore lends itself naturally to a distributed algorithm where switches negotiate locally optimal placements of the function instances. *Funplace* [4] is a message-passing protocol that allows switches to perform such negotiations in parallel throughout the network. The protocol allows the switches to compute fairly optimal placement of the network functions.

4.1 *Overview of the protocol*

Funplace assumes that each node (switch) in the network is aware only of its local state. The local state of a node comprises the set of flows passing through the node, the SFC of each such flow and the fragment of each SFC that is serviced at that node, which could be a null fragment (if none of the VNFs in the chain are serviced at that node). The node is thus aware of VNF instances that are serviced both at upstream and downstream nodes of each flow, but unaware of the exact location of such instances. This is illustrated in figure 2 for a single flow (F1) passing through three nodes (N1–N3). At each node, a start (S) and an end (E) pointer delineate the

Table 2. ILP formulation.

$\forall_{n,N_n \in \mathcal{N}}, \forall_{v,V_v \in \mathcal{V}}, \forall_{f,F_f \in \mathcal{F}}$		
(a) Parameters		
P_1	$P_{f,n}$	The path taken by the flow. $P_{f,n}$ is the position of N_n in the path taken by F_f.
P_2	$S_{f,v}$	The SFC for flow F_f. $S_{f,v}$ is the position of VNF V_v in the SFC of flow F_f.
P_3	C_n	The capacity of N_n to host VNFs measured in flow units.
(b) Decision variable		
D_1	$L_{f,v,n}$	The placement of VNFs along the path of each flow. $L_{f,v,n} = 1$ indicates that for flow F_f, VNF V_v has been placed on node N_n.
(c) Objective		
O_1		Minimize $\sum_n \sum_v (\max_f L_{f,v,n})$
(d) Constraints		
C_1		$S_{f,v} = 0 \rightarrow \forall_n\ L_{f,v,n} = 0$ (If a VNF is not required for a flow, it must not be placed.)
C_2		$S_{f,v} > 0 \rightarrow \exists_n\ L_{f,v,n} = 1$ (If a VNF is applicable to a flow, it must be placed on a node.)
C_3		$\forall_{v_1,v_2,f}\ S_{f,v_1} < S_{f,v_2} \rightarrow \exists_{n_1,n_2}\ L_{f,v_1,n_1} = 1, L_{f,v_2,n_2} = 1, P_{f,n_1} \leq P_{f,n_2}$ (VNFs for a flow must be placed in the same order as in the SFC.)
C_4		$\forall_{n,v,f} \sum_n L_{f,v,n} \leq C_n$

fragment of the SFC hosted locally. Each node is aware of the VNFs hosted in the upstream (U/s) and downstream (D/s) directions.

In each round of the algorithm, each node considers its current local state and the potential reduction in the number of VNF instances by possibly negotiating the migration of VNF instances from upstream and downstream nodes for the various flows; thus, it coalesces more flows for processing locally. VNF transfers are negotiated one at a time with the upstream or downstream neighbour. Funplace therefore ensures that the SFC sequence for every flow is maintained at all times. At the core of the algorithm is the concept of *utility* of a VNF at a node, defined as the number of flows serviced by that VNF at that node. Each node computes both its current utility of each VNF as well as the potential maximum utility by considering flows that are currently not serviced by the VNF locally, but can potentially be serviced locally instead of an upstream or downstream node. The difference between the current utility and the maximum utility is taken as the *differential*, which

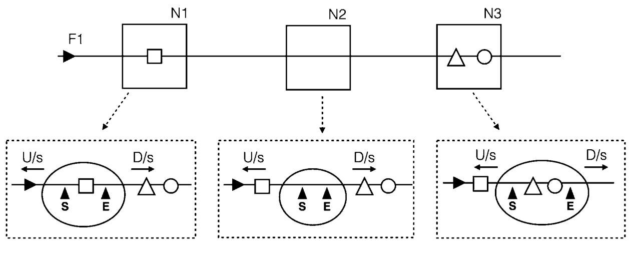


Figure 2. An illustration of a flow and the SFC fragments at each node along the flow.

becomes useful for tie-breaking in certain use cases that are explained later.

If a node identifies a better maximum *utility* of a local VNF instance (the number of flows processed by that instance) or finds it meaningful to spawn a local instance of a VNF, it sends a request to “transfer” the VNF from neighbouring nodes to itself. Conceptually, each node “pulls” VNF instances from either direction to optimize their utility. To increase the scalability of the protocol, Funplace operates on flow classes rather than flows. Each flow class is defined as the set of flows between the same pair of source and destination using the same path through the network, and having identical SFC. This avoids the protocol having to address individual flows and greatly reduces the size and the number of the protocol messages. For the rest of the paper, while discussing the protocol and its use cases, the term “flow” is used to mean a flow class for simplicity.

Each node generates transfer requests based on its local state to its neighbours. A neighbouring node that receives such a request makes a decision based on its own local state and takes one of the following actions: (i) rejects the transfer request since it has a higher utility of the VNF instance being requested, i.e. it can service more flows with the VNF instance than the requesting node, (ii) yields to the transfer request and offers to transfer the VNF instance to the requesting node or (iii) forwards the transfer request along the same direction in which the request was generated (either upstream or downstream) for the next node along the path of the flow(s) to consider the request. A three-way handshake is required to complete the transfer of a VNF instance at a node to another requesting node to reliably complete the transfer. A successful *transfer request* is responded to with a *transfer offer* in the opposite direction and finally the offer is accepted using a *transfer confirmation*, which proceeds in the same direction as the request. These three messages and their contents essentially constitute the protocol.

4.2 *An illustrative example*

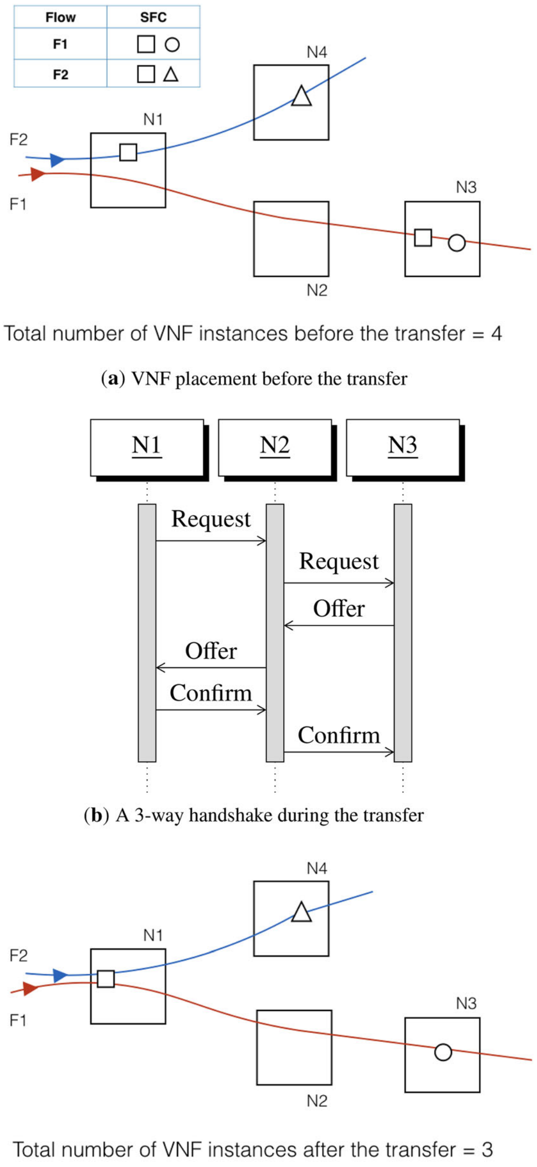
An illustration of how the nodes use the messages to negotiate VNF placement is presented in figure 3. At each node N_n there is at most one VNF instance of each type, each instance servicing one or more flows. There are two flows F1 and F2 in the network and their SFC requirements are as shown. The VNFs are represented by the various shapes: □, △ and ○. The initial placement of the VNFs results in a total of four VNF instances in the network.

The sequence of messages that are used to negotiate a more efficient VNF placement is shown. Node N1 identifies □ as a VNF that can be "pulled" from the downstream nodes (Of flow F1) and initiates a request message. This message indicates a *utility* of 2 flows. Since N2 does not host this VNF it forwards the message further downstream to N3, which responds to the request with an offer message since its own *utility* for □ is lower than that of the requesting node. The offer message is forwarded hop-by-hop back to N1. On seeing the offer message, N1 modifies its VNF-node for □ to include F1 and responds with a confirm message back to N3. N3 completes the transfer by removing F1 from its corresponding VNF-node and proceeds to destroy it since there are no other flows using it. All three nodes update their SFC fragments for the flows in the confirm stage of the transfer. After the transfer, the total number of VNF instances in the network is reduced to 3.

4.3 *The messages*

In this section we will describe the various messages that are used in the protocol, namely the *transfer request*, the *transfer offer* and the *transfer confirmation* messages. These messages are used by nodes along the paths of the flows to negotiate the placement of the VNF instances such that their utility is maximized. The messages are identical in format, which is shown in figure 4. However the semantics of the fields vary based on the type of message, which are discussed in this section.

The message has a fixed part and a variable part. The fixed part comprises the type of the message (transfer request, offer or confirmation), its length, the VNF being requested, offered or confirmed, source and destination node IDs, maximum utility and the differential. The fixed part of the message requires 24 bytes as shown. The variable part contains the flow IDs of the applicable flows for which the VNF transfer is being requested, offered or confirmed by the message. This part requires 4 bytes for each flow included in the message.


Flow
SFC
F1 □ ○
F2 □ △
N4
N1
F2
F1
N3
N2
Total number of VNF instances before the transfer = 4
N1
N2
N3
Request
Request
Offer
Offer
Confirm
Confirm
N4
N1
F2
F1
N3
N2
Total number of VNF instances after the transfer = 3

(**a**) VNF placement before the transfer

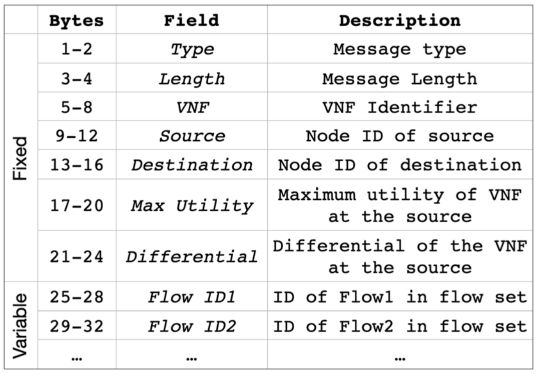
(**b**) A 3-way handshake during the transfer

(**c**) VNF placement after the transfer

Figure 3. An illustration of a VNF transfer.

The *Src node ID* field contains the ID of the node that originates the message. Thus, for the request and the confirmation messages, it is the requesting node. For the offer message, it is the node responding to a transfer request.

The *Dst node ID* field contains the ID of the node to which the message is addressed. When a node generates a transfer request, it addresses it to *all nodes* by setting a wildcard value to this field. This is because it does not know which downstream or upstream node will respond to the request. However, when an offer message is generated by a responding node, it sets the ID of the requesting node



	Bytes	Field	Description
Fixed	1-2	*Type*	Message type
	3-4	*Length*	Message Length
	5-8	*VNF*	VNF Identifier
	9-12	*Source*	Node ID of source
	13-16	*Destination*	Node ID of destination
	17-20	*Max Utility*	Maximum utility of VNF at the source
	21-24	*Differential*	Differential of the VNF at the source
Variable	25-28	*Flow ID1*	ID of Flow1 in flow set
	29-32	*Flow ID2*	ID of Flow2 in flow set
	…	…	…

Figure 4. Funplace message format.

in this field. Similarly, in the confirm message, the ID of responding node (which made the offer) is used.

The *max utility* and *differential* fields are applicable only for the request message and not used in the other two messages. They are used by the requesting node to advertise its maximum utility for a VNF to its neighbours and its difference from the current utility. The maximum utility serves as the primary decision making parameter by the receiving node. It essentially yields to a request if the requesting node has a higher maximum utility for a VNF. However, if the two nodes have the same maximum utility, the node that has the higher differential (and is therefore currently servicing flows to a lesser extent) yields to the other node.

4.4 *Funplace algorithm*

In this section, we will use pseudo-code to describe the distributed algorithm executed by the nodes in the network. The algorithm proceeds in rounds with each node receiving messages, taking actions on them and generating messages for neighbouring nodes in each round. Algorithm 1 shows the steps taken by a given node in a given round. The handling of incoming offers, confirmations and requests are addressed by algorithms 2, 3 and 4, respectively. These algorithms handle some special cases such as two offers "crossing in the wire", which are illustrated in a subsequent section. Based on the incoming requests and local maximum utility, generating either local requests or forwarded requests is handled by 5 and 6.

Algorithm 1 Process a round at a node

```
procedure Node.processRound()
    // handle incoming offers/confirms
    handleOffers()
    handleConfirmations()
    // calculate max utility for all VNFs
    calculateMaxUtilities()
    // now, handle requests
    handleRequests()
    // finally, generate requests
    generateRequests()
    Done
```

Algorithm 2 Handle an offer

```
procedure Node.handleOffer(msg)
    if checkCrossedOffers() then
        // offers crossed on wire
        if local node ID < msg.srcNodeId then
            // node with lower ID rejects the offer
            return
    // accept the offer by "pulling" the VNF
    pullVnf(msg)
    // adjust the SFC fragment markers
    adjustSfcFragment()
    // send the confirmation
    sendConfirmation()
    Done
```

Algorithm 3 Handle a confirmation

```
procedure Node.handleConfirmation(msg)
    // accept the confirmation by "pushing" the VNF
    pushVnf(msg)
    // adjust the SFC fragment markers
    adjustSfcFragment()
    Done
```

Algorithm 4 Handle a request

```
    procedure Node.handleRequest(msg)
2:      if !localInstance(msg.vnf) then
            // there is no local instance of the VNF–
4:          // forward the request along the flows
            forwardRequest(msg)
6:      if localMaxUtility > msg.maxUtility then
            // reject the request
8:          return
        else if localMaxUtility < msg.maxUtility then
10:         // accept the request
            sendOffer()
12:     else if local node ID > msg.srcNodeId then
            // node with larger node ID yields
14:         // when utilities are equal
            sendOffer()
16:     Done
```

Algorithm 5 Forward a request

```
    procedure Node.forwardRequest(msg)
2:      // get the set of flows from incoming message
        flows ← msg.flows
4:      for each flow in flows do
            // direction is upstream or downstream based on
6:          // whether rx port was ingress or egress of flow
            dir ← direction of flow at the rx port
8:          // save the request in the flow to be used
            // later to generate forwarded requests
10:         flow.saveForwardRequest(dir, msg)
        Done
```

Algorithm 6 Generate transfer request

```
    procedure Port.generateRequest(vnf)
2:      // this method compares local max utility with values
        // in received requests (already saved in the flows) and
4:      // generates either local or forwarded requests
        if localMaxUtility(vnf) > received max utility then
6:          generateLocalRequest(vnf)
        else
8:          flowGroups ← groupFlowsByNodeId(vnf)
            // flows grouped by srcNodeId of received requests
10:         for each flowGroup in flowGroups do
                generateForwardedRequest(flowGroup, vnf)
12:     Done
```

Note that Funplace starts with the flows already routed and SFC for each flow provisioned. It is assumed that the path computation and provisioning are already done. The placement of the VNFs can be done using another trivial algorithm that places the VNF instances along the path of each flow. For example VNFs can be placed sequentially starting from the source, one instance at each node. Alternatively, the algorithm can place as many VNFs as possible at each node before proceeding to the next node. Similarly, when a new flow enters the network, it is routed and VNFs are placed using this initial method. Obviously, such an algorithm will result in a highly inefficient placement as an initial state. Funplace is an online solution that continuously iterates and improves the efficiency of this placement, starting from this initial state.

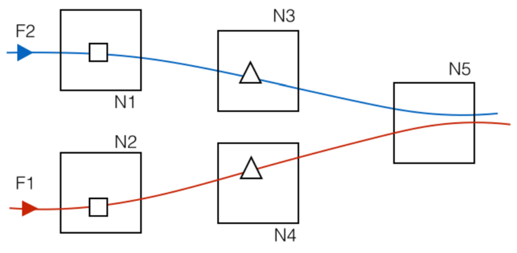
It must also be noted that Funplace was tested using discrete event simulation for the purpose of presenting the results in this paper, where the simulation proceeds in rounds. The messages generated by a node in a round reach the neighbour in the subsequent round. The time duration of a round in a real implementation may vary based on the network size, number of flows, etc. For example this duration can be set to 500 ms, which determines the frequency at which the transfer requests are generated.

4.5 *Use cases*

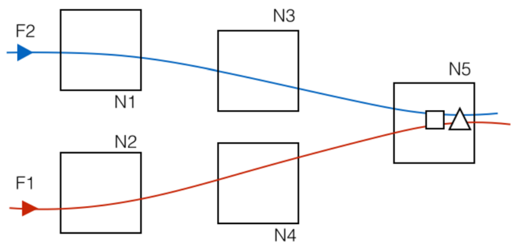
In this section we present a few use cases as further illustration of the algorithm and some of the complex scenarios that must be handled by the algorithm.

4.5a *Consolidation*: When two or more flows converge at a common node, but the VNF instances are placed downstream or upstream from the common node, the VNFs can be consolidated at the common node. This is illustrated in figure 5.

4.5b *Tie-breaking*: In some cases, two nodes compete with each other for a VNF instance with both computing the



(**a**) VNF placement before the consolidation



(**b**) VNF placement after the consolidation

Figure 5. Consolidation of VNFs at a common node.

same value as the *utility* for the VNF. In this case, the difference between the current utility and the target utility of a VNF instance at a node is used to break the tie. This difference is termed as the *differential*. This is illustrated in figure 6. Nodes N1 and N3 both have a *utility* of 2 flows for the VNF represented by □. However, node N1 has a differential of zero. This indicates that it is already servicing 2 flows. Node N3 generates a transfer request to N1, which is rejected as N3 has a higher differential, avoiding an unnecessary transfer.

4.5c *Branching of a request*: Consider the VNF placement shown in figure 7. Here, N4 sends a transfer request to N3 since a consolidation of □ is desirable for the two flows. When the request is received at N3, it must be forwarded along both flows in their respective directions away from N4. The "pull" from N4 is essentially transmitted after branching the request into two separate requests, one towards N1 and the other towards N2. The *utility* and *differential* are preserved in the forwarded requests to indicate the utility of the VNF instances in their intended target node (N4). In this case a successful completion of the individual transfers is allowed, consolidating □ at *N*4.

4.5d *Symmetry breaking*: Figure 8 illustrates a use case where using the differential as tie-breaker is not effective. Here, both N1 and N2 have the same *utility* (2 flows) and *differential* (1 flow). In this case, the node with the higher node ID yields to the one with the lower node ID. In this case, N1 ends up hosting the VNF for both flows in the illustration.

4.5e *Simultaneous requests*: *(a race condition)* Sometimes two nodes, one downstream from the other for one or more flows, may generate simultaneous transfer requests for two different VNF instances, one in each direction. This is illustrated in figure 9. Here, N1 has a higher utility for △ and requests a transfer for F1 from its upstream neighbour. Also, N2 has a higher utility for □ and may simultaneously generate a transfer request for it in the downstream direction. Clearly, only one of these requests must be allowed to be completed to avoid swapping the VNF instances and violating the SFC requirement. This race condition cannot be handled by either node when the requests are received since neither one knows if the other node is going to respond with an offer. Thus, both nodes are allowed to generate their respective offers. However, when the offers are received at the two ends, one of the nodes (the one with the lower ID) rejects the offer, thus ending the handshake process in one of the directions, resolving the race condition.

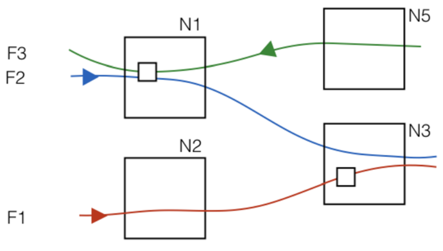


Figure 6. Differential used as tie-breaker.

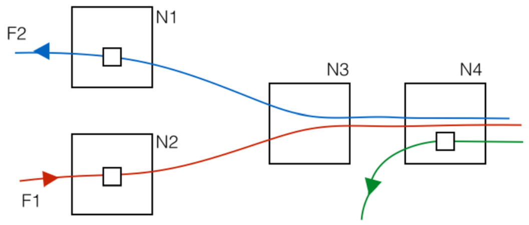


Figure 7. Branching of an incoming request.

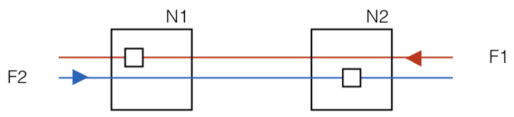


Figure 8. Symmetry breaking using node ID.

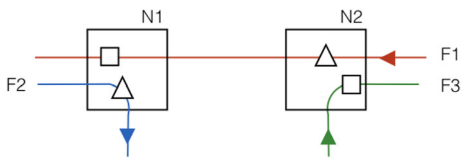


Figure 9. Simultaneous requests leading to race conditions.

Thus Funplace attempts to incrementally and continually increase the utility of each VNF instance in the network, resulting in overall optimization of the number of VNF instances in the entire network. It does so by consolidating VNF instances on different nodes wherever possible considering both upstream and downstream directions of flows as seen from each node. The protocol includes techniques to break ties, handle symmetry and prevent race conditions.

4.6 *Analysis of the algorithm*

In this section, we prove the correctness of the Funplace algorithm by showing the following lemmas given a set of flows and an SFC for each flow.

Lemma 1 *A VNF transfer completed by Funplace will never result in a higher number of VNF instances in the network.*

Lemma 2 *Whenever it is possible to lower the overall number of VNF instances by transferring a VNF instance from one node to another, Funplace will complete the transfer.*

Lemma 3 *Funplace always reaches a steady state where no transfer offers are being made.*

These lemmas can be proved by considering a single arbitrary flow F and an arbitrary VNF denoted by ○ in its SFC. Let N be the node where the VNF is hosted at the start of round R_0 of the algorithm. Since Funplace works symmetrically in both upstream and downstream directions, we will consider potential transfers of this VNF in the downstream direction only without loss of generality. Let us assume that ○ is the next VNF available for transfer in the downstream direction at the end of some round R_t and therefore, at the start of round R_{t+1}. This VNF is transferred to some downstream node N' as shown in figure 10, if and only if (i) N$'$ has a higher utility for ○ than N OR (ii) N and N$'$ have the same utility but N$'$ has a higher differential OR (iii) N and N$'$ have the same utility and differential, but N$'$ has a lower node ID. The higher utility for this VNF at N$'$ implies an overall reduction in the number of VNF instances since, by definition of utility, more flows can be serviced by the VNF instance at N$'$. Clearly, Lemma 1 and Lemma 2 are followed by this transfer based on the conditions. Also note that once a downstream transfer is completed for ○, it cannot be transferred back in the upstream direction based on the conditions. Moreover the transfer will be completed by round R_{t+3h}, where h is the number of hops from N to N$'$ since $3h$ is the number of rounds taken to complete the 3-way handshake. Assuming that there are no loops in the flow and since the path of the flow has a finite length the VNF instance will reach a node where no further transfer offers are made. Therefore, Lemma 3 is also followed for this flow and VNF. Since we showed the lemmas to be true for an arbitrary flow and VNF, it can be shown to be true for all flows and their SFCs by extension.

Since Funplace is a distributed solution, it is meaningful to analyse the average time complexity of Funplace on any node in the network. If $\mathcal{F}_{\mathcal{N}}$ is the set of flows through node N, L is the average length of the SFCs and P is the average path length of the flows, then Funplace reaches steady state in $O(|\mathcal{F}_{\mathcal{N}}|LP)$. This is because the node may participate in the transfer of each VNF instance in the SFC along the entire path of the flow.

5. Performance results

In this section we present the details of the discrete event simulation of the protocol, various test cases and the results from the simulation, along with comparison of the performance of the protocol to that of the optimal solution generated by the ILP formulation.

5.1 *Test network*

Fat-tree topology [5] and *BCube* topology [6] are among the most commonly used topologies in modern networks such as Data Center Networks (DCNs). They are highly meshed and present the most challenging conditions for testing the efficiency of a protocol like Funplace. Figure 11 illustrates the Fat-tree topology that was used for testing the protocol. Networks of varying sizes of this topology are used for testing by varying the height of the tree (the illustration shows a height of 3) and the number of pods in the topology. A network with k pods and a height of h has $O(k^h)$ nodes and $O(k^{h+1})$ links. Later in this section, the results from a topology generated with $k = 8$ and $h = 3$ with 104 nodes and 512 links and also from a topology generated with $k = 16$ and $h = 3$ with 656 nodes and 6144 links are presented.

Funplace is also tested on the BCube topology shown in figure 12, which is also used in DCNs. A BCube topology is recursively defined using two parameters k and n, where a BCube$_k$ is constructed using n^k switches and n^{k-1} instances of BCube$_{k-1}$. At the lowest level, a BCube$_0$ has n host nodes that are capable of switching and a single n-port switch. Overall, a BCube network has $(n + k + 1)n^k$ nodes and $(k + 1)n^{k+1}$ links. For testing the protocol, a topology with $k = 2$ and $n = 8$ is used, resulting in 704 nodes and 1536 links. The results for this topology are also presented later in this section.

Flows (flow classes) were provisioned between various *{src, dst}* pairs of switches. Equal Cost Multi-Path (ECMP) routing was used to ensure that the various paths through the network were used to evenly distribute the flows. The simulation was conducted with a short SFC (a sequence of 3 VNFs), a medium SFC (5 VNFs) and a long SFC (8 VNFs). The default algorithm places the VNFs in the SFC of each flow, one VNF at each node starting with the source node of the flow in sequence, essentially, in a highly inefficient configuration. Funplace was then used to optimize the placement. The output thus generated is labelled as

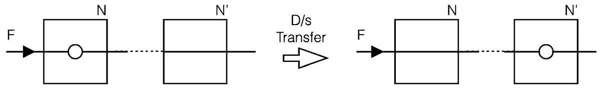


Figure 10. Transfer of an arbitrary VNF instance of a flow from node N to N'.

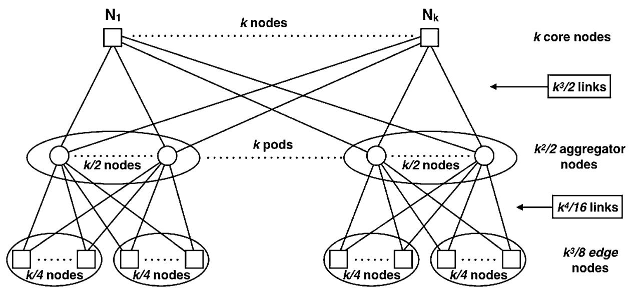


Figure 11. Fat-tree topology used for testing.

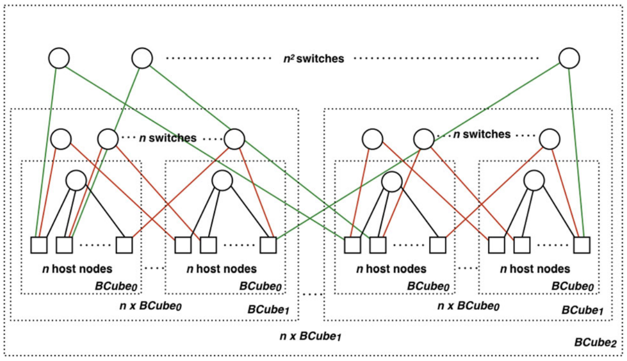


Figure 12. BCube topology used for testing.

Funplace in the graphs and compared with (i) a heuristic placement specified in [12] as NetPack, (ii) another heuristic algorithm specified in [14] as Heuristic-A and (iii) the optimal solutions generated by the ILP formulation (labelled as optimum). NetPack and Heuristic-A algorithms are described in Sections 5.1a and 5.1b, respectively. The average ratio of the output of Funplace to the optimal solution for various instances of the problems tested is taken as the *approximation ratio* of this distributed solution. This ratio was also plotted for the various test scenarios. The simulation was also used to collect information about the convergence speed (the number of iterations or time steps taken by the protocol to converge).

5.1a *NetPack heuristic NetPack* algorithm [12] attempts to optimize bandwidth usage in the network by following a sequence of optimization steps. First, VNFs are sorted in topological order. Then, *network-locality* is used in trying to place all VNFs of the chain on the same server. If this is impossible, it explores servers on the same rack, then servers within the same cluster, etc. Finally, *server-locality* is used to attempt to re-use previously allocated servers when placing consecutive VNFs.

5.1b *Heuristic-A Heuristic-A* [14] is a multi-step greedy algorithm that attempts to place VNFs one flow at a time. VNFs are placed along the shortest path of each flow, wherever capacity is available. While doing so, the algorithm tries to share the capacity already allocated on the shortest path. If this is not possible, the algorithm looks at neighbouring switches to use available capacity on those switches and tries to reuse the capacity allocated on them. If after these steps a flow does not have all the necessary services, Heuristic-A adds a node from the neighbouring switches to the shortest path and repeats all the aforementioned steps.

5.2 *Observations*

In this section, the observations made on the performance of Funplace on networks of various sizes and topology are given.

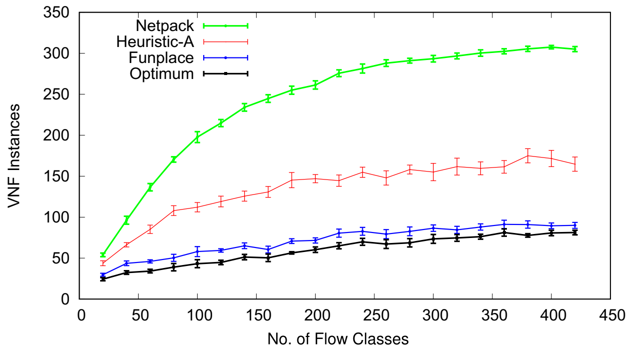
5.2a *Fat-tree topology with 104 nodes*: Figure 13 compares the number of VNF instances as computed by Funplace in comparison with other heuristics and also the globally optimal solution computed by the ILP formulation on a Fat-tree topology with 8 pods and 3 levels. It can be observed that Funplace greatly optimizes the number of VNF instances compared with other heuristic solutions. As expected, performance of Funplace (the number of VNF instances in each case) is worse than that of the ILP formulation but the Funplace-to-optimum ratio ranges from 1.22 to 1.5 for this network as shown in figure 13d. Given that the underlying problem is an NP-complete problem, this is reasonable performance. It can be noted that this performance degrades with the length of the SFC, based on the increasing value of the ratio.

As seen in figure 14, for a given network, the convergence time does not scale with the number of flows, but only with the length of SFC for the flows. This is because of the protocol operating in parallel and optimizing the placement of VNF instances at various parts of the network simultaneously. It is clear that a distributed protocol like Funplace will scale much better compared with a centralized solution as the size of the network and the number of flows become larger. The latency involved in ferrying state from the nodes to the centralized controller and the computation time involved in finding a globally optimal solution will eventually degrade the performance of a centralized solution.

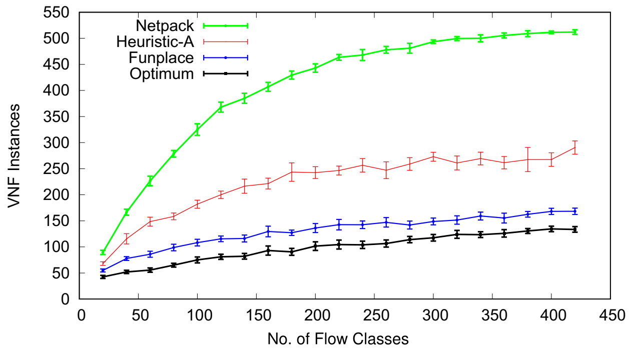
5.2b *Fat-tree topology with 656 nodes*: Figures 15 and 16 show the results from similar testing on a Fat-tree topology with 16 pods and 3 levels. It can be observed the results are similar to that of a smaller network. In the case of this larger network, the Funplace-to-optimum ratio ranges from 1.44 to 2.0. The overall average of this ratio for all test cases for Fat-tree networks presented here was computed to be 1.5 and taken as the approximation ratio.

5.2c *BCube topology with 704 nodes*: Funplace was tested on a BCube topology with $k = 2$ and $n = 8$, which has 704 nodes and 1536 links. The performance observed for SFC length of 3 on this topology is similar to that of the Fat-tree topology with Funplace outperforming both the NetPack and the Heuristic-A algorithms, as shown in figure 17. Note that comparison with the optimum was possible only for SFC of length 3. For longer service chains, with 200 or more flow classes, the ILP solver takes in excess of 24 h to find the optimum solution for each instance of the problem on a 32-core server with 2-GHz Intel Xeon processors and 128 GB of RAM running 64-bit Ubuntu 18.04.3 LTS due to the size of the optimization problem. This further underscores the value of a distributed solution such as Funplace.

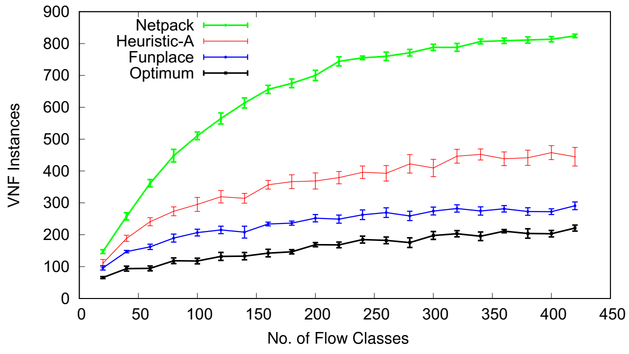
5.2d *Illustration of dynamic solution*: Figure 18 illustrates Funplace solving the NFP problem when flows are added to the network dynamically. In this example, 20 new flows are provisioned in every 50 rounds of the protocol simulation.



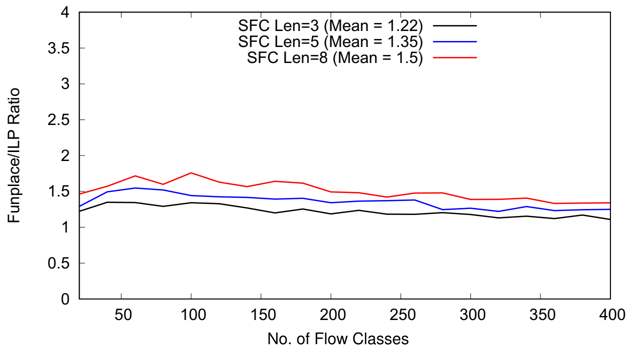
(**a**) Results for SFC Length = 3



(**b**) Results for SFC Length = 5



(**c**) Results for SFC Length = 8



(**d**) Ratio between Funplace and optimum solution

Figure 13. Performance of Funplace vs. other heuristics and the optimum on a Fat-tree topology ($k = 8, h = 3$) with 104 nodes, 512 links (CI = 90%).

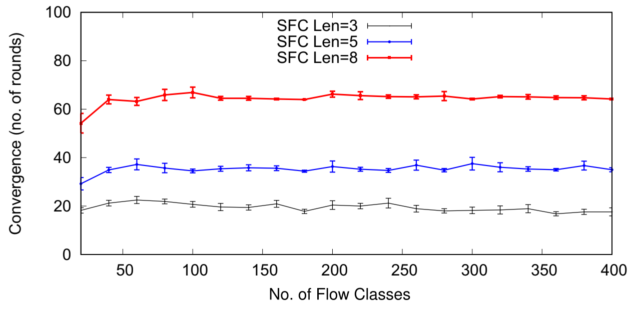


Figure 14. Convergence speed of Funplace on a Fat-tree topology with 104 nodes, 512 links ($k = 8, h = 3$) (CI = 90%).

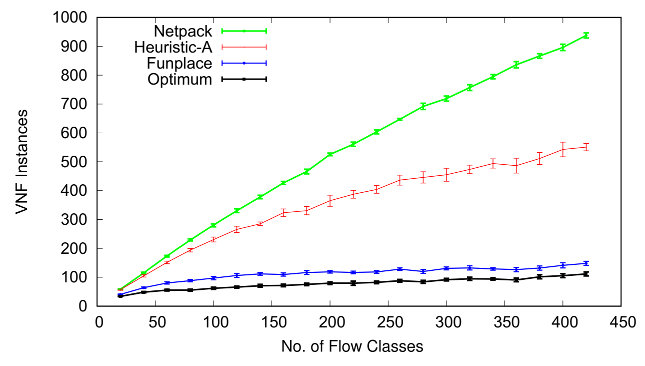
It can be observed that the number of VNF instances spikes after each round of provisioning, followed by Funplace quickly optimizing the number of VNF instances. Each flow had an SFC of length 5 in this experiment.

5.3 *Impact on network throughput*

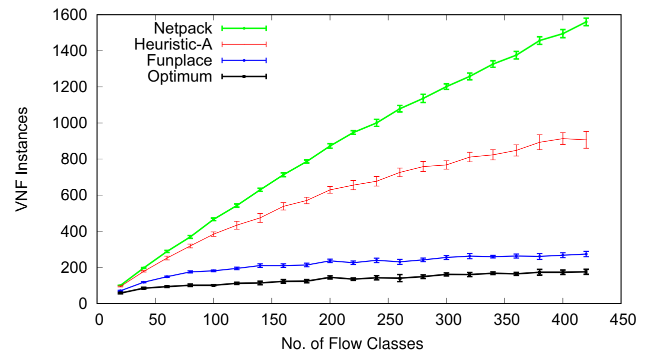
In this section, the impact of Funplace on the throughput of the network is discussed. As explained in Section 4.3, each Funplace message instance has a fixed component of length 24 bytes and a variable component of flow IDs with each ID taking 4 bytes. The total number of bytes transferred by Funplace for various numbers of flows in a 656-node Fat-tree topology and the corresponding average number of bytes per network link are shown in figure 19. As the number of flows increases, even though the total bandwidth usage increases, the average usage per link decreases due to the flows being distributed over a larger part of the network and more network links participating in the protocol. Note that the average amount of data transferred by the protocol for 400 flows and SFC length of 5 is about 8 kbytes over approximately 20 iterations of the protocol (as observed from figure 16). Assuming a time duration of 500 ms for each round, this translates to less than 1 kbps data rate. This is a small fraction of the bandwidth of a network link, which is typically 100 Mbps, 1 Gbps or higher. It can be concluded that the impact of Funplace on the network throughput is negligible.

5.4 *Effect of migration cost*

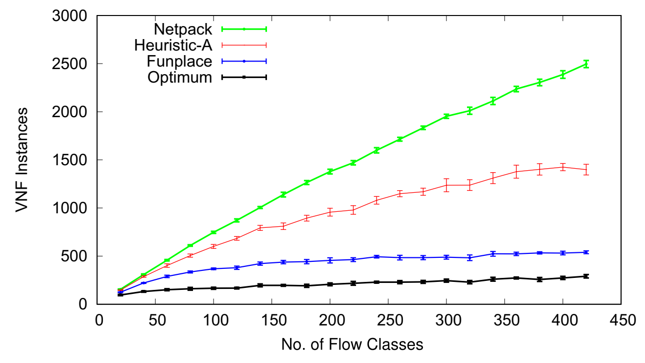
In this section, we present the effect of migration cost on the result computed by Funplace. When studying VNF placement, it may be necessary to take the cost of migrating VNFs into account especially when the VNFs are stateful and migration can result in data loss [23]. Even though the default implementation of Funplace does not take migration cost into account it is relatively easy to account for migration cost by offsetting the utility value received in the



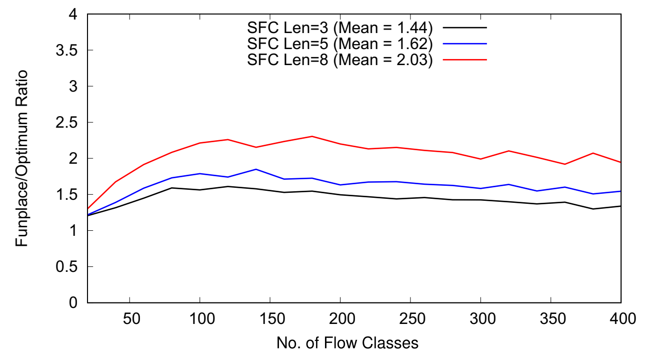
(**a**) Results for SFC Length = 3



(**b**) Results for SFC Length = 5



(**c**) Results for SFC Length = 8



(**d**) Ratio between Funplace and optimum solution

Figure 15. Performance of Funplace vs. other heuristics and the optimum on a Fat-tree topology ($k = 16, h = 3$) with 656 nodes, 6144 links (CI = 90%).

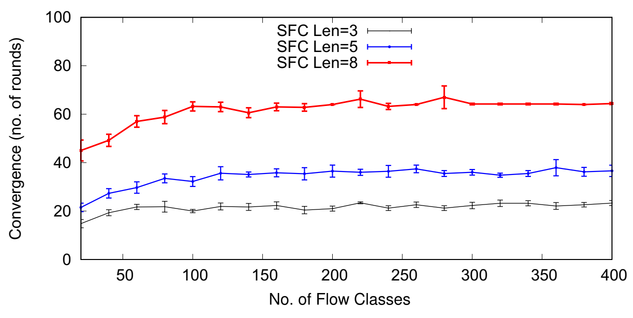
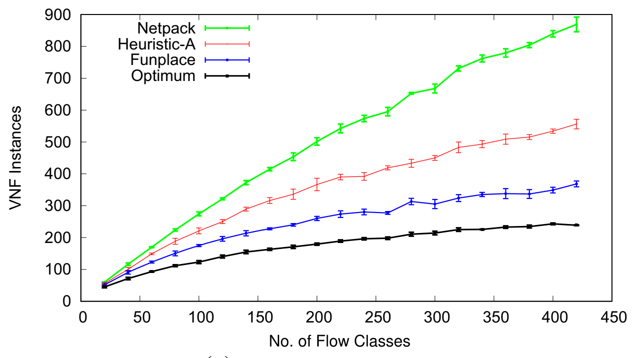
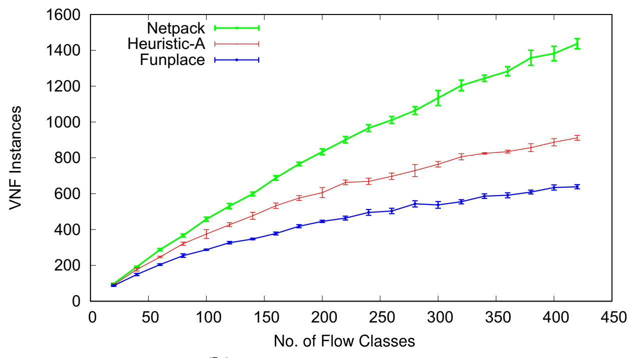


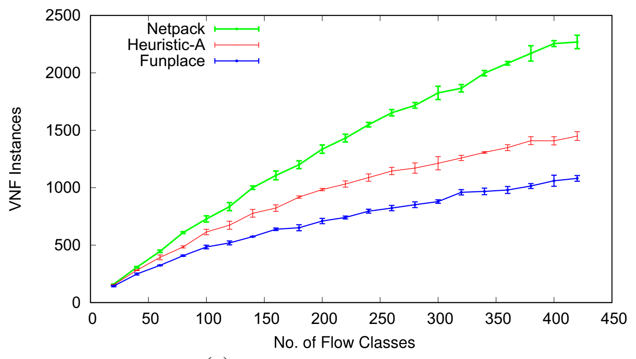
Figure 16. Convergence speed of Funplace on a Fat-tree topology with 656 nodes, 6144 links ($k = 16, h = 3$) (CI = 90%)



(**a**) Results for SFC Length = 3



(**b**) Results for SFC Length = 5



(**c**) Results for SFC Length = 8

Figure 17. Performance of Funplace vs. other heuristics and the optimum on a BCube topology ($k = 2, n = 8$) with 704 nodes, 1536 links (CI = 90%)

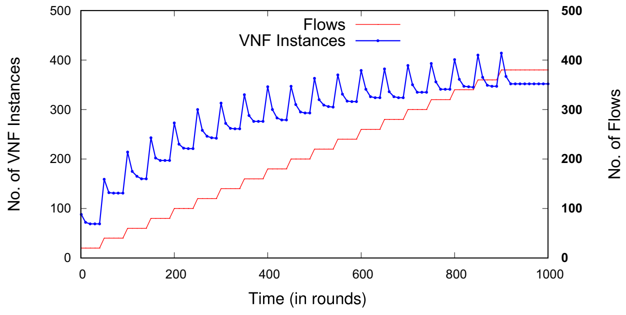
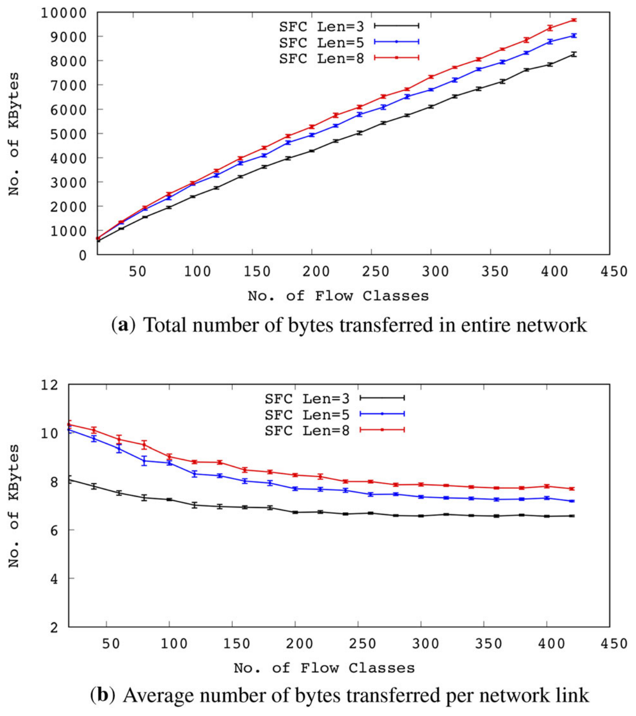


Figure 18. Illustration of Funplace solving NFP problem when flows are dynamically provisioned.



(**a**) Total number of bytes transferred in entire network

(**b**) Average number of bytes transferred per network link

Figure 19. Bandwidth usage in bytes by Funplace protocol messages on a Fat-tree topology with 656 nodes, 6144 links (CI = 90%)

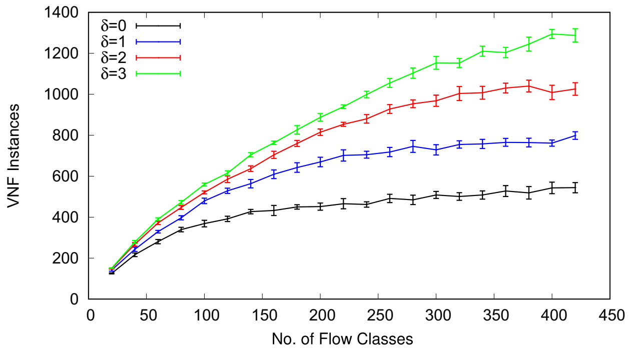


Figure 20. VNF instances computed by Funplace on a Fat-tree topology with 656 nodes, 6144 links for various values of migration cost (CI = 90%).

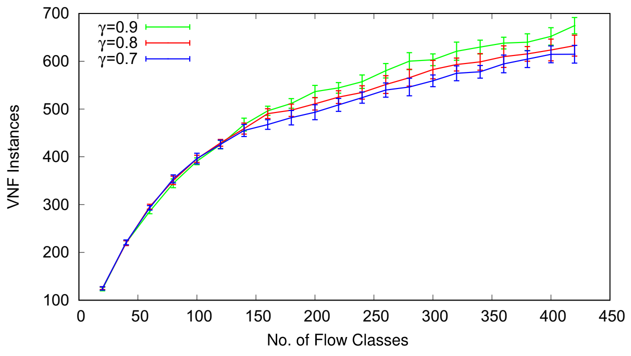


Figure 21. VNF instances computed by Funplace on a Fat-tree topology with 656 nodes, 6144 links for various distributions of flow size (CI = 90%).

transfer request by a certain value δ to represent this cost, before comparison with local utility at a node while deciding to accept or reject the request. The results are shown in figure 20 for various values of δ in the 656-node Fat-tree topology to indicate the impact of migration cost on the number of VNF instances computed by Funplace.

It must be noted that a fixed migration cost has been used to arrive at these results. However, the migration of a VNF may cost a variable amount of time and resources depending on its complexity and the size of the flows. For example, migrating VNFs servicing smaller flows (mice flows) may take shorter time than those servicing larger flows. Moreover, large number of mice flows that are short-lived may result in frequent migrations leading to possible loss of data throughput. For the purpose of this paper, only a simulation study of the protocol for migration is studied. Actual migration of VNFs is not studied. However, a Funplace implementation using implementation of real VNFs is a suitable topic for future work.

5.5 *Flow size distribution*

While defining the placement problem in Section 3.1, we assumed that each flow is a single unit of flow. However, in a realistic network, the set of flows comprise small (mice) flows and very large (elephant) flows. In this section we study the effect of having a certain fraction of elephant flows, each assumed to be 1000 times the size of a mice flow. This was accomplished by redefining the *utility* of a VNF at a node to be the aggregate data rate of the flows serviced by the VNF at that node. Figure 21 shows the results of testing Funplace with various distributions of flows. Each value of γ indicates the fraction of the total number of flows that are mice flows. It can be observed that with a small percentage of elephant flows (10%, with $\gamma = 0.9$), the number of VNF instances is marginally higher compared with a larger percentage of elephant flows. This is due to the small number of elephant flows dominating the placement decisions of the VNFs and preventing a more optimal solution overall.

5.6 *Summary of results*

The results from the testing of Funplace show that for a given SFC length, the distributed algorithm does not degrade with the number of flows in the network. The ratio of the VNF instances as computed by Funplace to that computed by ILP does not increase with the number of flows in the network. Moreover, this ratio is on an average 1.5 for the various SFC lengths tested on the Fat-tree networks. Taking into consideration the fact that the NFP problem is NP-complete, this shows that Funplace can generate reasonably optimal solutions. Similarly, the number of iterations taken by the protocol to converge on a solution also does not increase with the number of flows. This result indicates that a distributed algorithm will perform better beyond a certain number of flows, since a centralized solution will degrade due to the increased messaging between the nodes and a centralized controller with increasing number of flows and the resulting latency in processing the messages centrally.

6. Conclusions

NFP is a complex problem for a large network with a large number of flows in the network. Attempting to solve the problem dynamically and in parallel when flows are being provisioned is an intractable problem. A distributed solution lends itself well since the nodes can negotiate the placement locally and make the decision to place or migrate the functions themselves. This paper presented such a solution, using *Funplace* as the protocol for nodes to negotiate the placement among themselves. The results of testing the protocol among various networks of different topologies and sizes show that it performs better when compared with other heuristics and reasonably well in comparison with the optimal solution computed using ILP formulation of the problem and a corresponding solver. Future work on this problem involves actual dynamic migration of VNF instances between nodes. Possible loss of data due to unnecessary and frequent migration can be studied, especially in the presence of short-lived mice flows.

Acknowledgements

This work was supported by a DST grant (EMR/2016/003016) from Government of India (2017–2020) and an IRDA grant from IIT Madras (2017–2020).

Appendix I. NP-completeness proof of NFP problem

In this section, a simplified form of the NFP problem that involves a single network function is taken into consideration. This simplified NFP problem is termed as the Single-NFP (S-NFP) problem. It is shown that the S-NFP problem is NP-complete.

Appendix I.1 The Single Network Function Placement (S-NFP) problem

Given a network, a set of paths (representing flows through the network), a network function and a bound on the number of instances of the network function, the S-NFP problem involves finding whether the instances can be placed in the nodes of the network such that each path has at least one node (vertex) with the function placed on it. This is more formally defined here.

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph representing a network and $\mathcal{P} = \{p_1, p_2, \ldots, p_m\}$ be a set of m paths through the network. Each path p_j connects two vertices s_j and t_j, where $s_j, t_j \in \mathcal{V}$. The *S-NFP problem* is as follows: given a constant c, is there an $\mathcal{R} \subseteq \mathcal{V}$ s.t. $|\mathcal{R}| \leq c$ and $\forall p_j \in \mathcal{P}, \exists r \in \mathcal{R}$ s.t. r is a vertex on path p_j?

Appendix I.2 Reduction from Set Cover problem to S-NFP

The Set Cover problem can be reduced to the S-NFP problem, proving the NP-Completeness of the S-NFP problem. The Set Cover problem is as follows. Given $\mathcal{U} = \{1, 2, \ldots, m\}$ and a family of l subsets of $\mathcal{U}$, $\mathcal{F} = \{A_1, A_2, \ldots, A_l\}$ and a constant c, is there a set $\mathcal{C} \subseteq \mathcal{F}$ s.t. $|\mathcal{C}| \leq c$ and $\forall j \in \mathcal{U}, \exists A \in \mathcal{C}$ s.t. $j \in A$.

Given an instance of the Set Cover problem, a corresponding instance of the S-NFP problem can be constructed as follows.

Let $\mathcal{U}, \mathcal{F}$ and c of a Set Cover problem instance be given. A corresponding graph $\mathcal{G}$ and a set of m paths $\mathcal{P}$ through the graph can be defined based on this input. Let s_j and t_j be a pair of nodes in the graph corresponding to each element $j \in \mathcal{U}$. s_j and t_j $(1 \leq j \leq m)$ are the two end nodes of path p_j through the graph. Therefore, there are m such pairs and m corresponding paths through the graph. Additional nodes in the graph are now defined based on $\mathcal{F}$. For each set $A_i \in \mathcal{F}$, define a node n_i in the graph. Also, for each $s_j - t_j$ pair, define l nodes n_i^j, where $1 \leq i \leq l$.

Each path p_j in $\mathcal{G}$ has its end points at s_j and t_j and has l intermediate nodes. Intermediate node i of the path is either n_i if $j \in A_i$, or n_i^j otherwise. Essentially, all paths corresponding to elements of A_i converge at n_i as their i^{th} intermediate node. An example of a Set Cover problem and

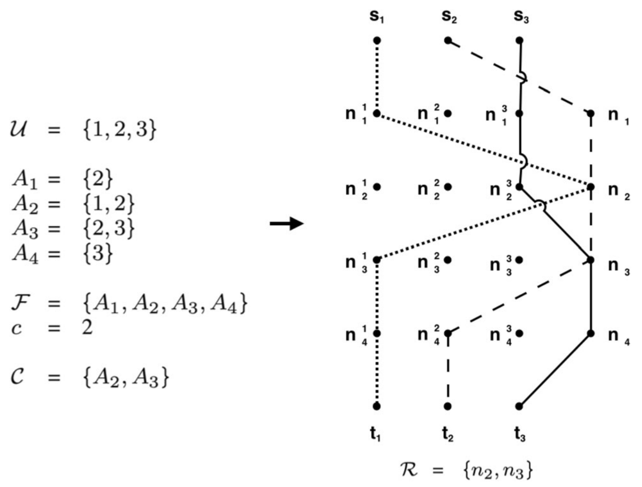

$\mathcal{U} = \{1,2,3\}$
$A_1 = \{2\}$
$A_2 = \{1,2\}$
$A_3 = \{2,3\}$
$A_4 = \{3\}$
$\mathcal{F} = \{A_1, A_2, A_3, A_4\}$
$c = 2$
$\mathcal{C} = \{A_2, A_3\}$
s_1 s_2 s_3
n_1^1 n_1^2 n_1^3 n_1
n_2^1 n_2^2 n_2^3 n_2
n_3^1 n_3^2 n_3^3 n_3
n_4^1 n_4^2 n_4^3 n_4
t_1 t_2 t_3
$\mathcal{R} = \{n_2, n_3\}$

Figure 22. Illustration of the reduction of Set Cover problem to S-NFP problem.

its corresponding instance of the S-NFP problem are illustrated in figure 22.

The NP-completeness of the S-NFP problem can be proved by proving the following *lemma*, thereby proving the equivalence of an instance of the Set Cover problem with the corresponding instance of the S-NFP problem.

Lemma *A solution to a feasible instance of the Set Cover problem with j elements in $\mathcal{U}$ and c as the bound on the cover set exists iff a solution exists to the corresponding S-NFP problem with j paths and c as the constraint on the number of network function instances.*

Proof Consider an instance of the Set Cover problem and its corresponding instance of S-NFP problem constructed as described above. Let a solution exist for the S-NFP problem, with $\mathcal{R}$ being the solution. The solution for the Set Cover problem instance can be constructed as follows. For each $r \in \mathcal{R}$, if r is node n_i, then include A_i in the solution set $\mathcal{C}$. If the r is a node of the form n_i^j, then include any set A_k s.t $j \in A_k$ (it must exist if there is a feasible solution). Since $\mathcal{R}$ has at least one node on each of the m paths, it follows that $\mathcal{C}$ has at least one set which contains each one of the m elements in $\mathcal{U}$. Also, $\mid \mathcal{C} \mid = \mid \mathcal{R} \mid$.

Conversely, let a solution set $\mathcal{C}$ exist for the Set Cover problem, solution set $\mathcal{R}$ for the S-NFP problem can be constructed by including in $\mathcal{R}$ vertex n_i which corresponds to each $A_i \in \mathcal{C}$. Thus, above lemma is proved. It follows that S-NFP problem, and therefore the more general NFP problem, are NP-Complete. □

References

[1] Carpenter B and Brim S 2002 *Middleboxes: taxonomy and issues*. Technical Report IETF, RFC 3234, February

[2] Han B, Gopalakrishnan V, Ji L and Lee S 2015 Network function virtualization: challenges and opportunities for innovations. *IEEE Commun. Mag.* 53(2): 90–97

[3] Bec M T and Botero J F 2015 Coordinated allocation of Service Function Chains. In: *Proceedings of IEEE GLOBECOM*, pp. 1–6

[4] Anbiah A and Sivalingam K M 2017 Funplace: a protocol for Network Function Placement. In: *Proceedings of the IEEE Conference on Local Computer Networks (LCN)*, Singapore, October, pp. 587–590

[5] Leiserson C E 1985 Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.* 100(10): 892–901

[6] Guo C, Lu G, Li D, Wu H, Zhang X, Shi Y, Tian C, Zhang Y and Lu S 2009 BCube: a high performance, server-centric network architecture for modular data centers. *ACM SIGCOMM Comput. Commun. Rev.* 39(4): 63–74

[7] Charikar M, Naamad Y, Rexford J and Zou X K 2018 Multi-commodity flow with in-network processing. In: *Proceedings of the International Symposium on Algorithmic Aspects of Cloud Computing*, pp. 73–101

[8] Anwer B, Benson T, Feamster N and Levin D 2015 Programming slick network functions. In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, p. 14

[9] Cho D, Taheri J, Zomaya A Y and Bouvry P 2017 Real-time virtual network function (VNF) migration toward low network latency in cloud environments. In: *Proceedings of the 10th IEEE International Conference on Cloud Computing (CLOUD)*, pp. 798–801

[10] Baumgartner A, Reddy V S and Bauschert T 2015 Mobile core network virtualization: a model for combined virtual core network function placement and topology optimization. In: *Proceedings of the 1st Conference on Network Softwarization (NetSoft)*, pp. 1–9

[11] Mehraghdam S, Keller M and Karl H 2014 Specifying and placing chains of virtual network functions. In: *Proceedings of IEEE CloudNet*, pp. 7–13

[12] Kodirov N, Bayless S, Ruffy F, Beschastnikh I, Hoos H H and Hu A J 2018 VNF chain allocation and management at data center scale. In: *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, pp. 125–140

[13] Moens H and De Turck F 2014 VNF-P: a model for efficient placement of virtualized network functions. In: *Proceedings of the 10th International Conference on Network and Service Management (CNSM)*, pp. 418–423

[14] Mohammadkhan A, Ghapani S, Liu G, Zhang W, Ramakrishnan K K and Wood T 2015 Virtual function placement and traffic steering in flexible and dynamic software defined networks. In: *Proceedings of the International Workshop on Local and Metropolitan Area Networks (LANMAN)*, pp. 1–6

[15] Lin T, Zhou Z, Tornatore M and Mukherjee B 2016 Demand-aware network function placement. *J. Lightwave Technol.* 34(11): 2590–2600

[16] Addis B, Belabed D, Bouet M and Secci S 2015 Virtual network functions placement and routing optimization. In: *Proceedings of IEEE CloudNet*, pp. 171–177

[17] Ghaznavi M, Khan A, Shahriar N, Alsubhi K, Ahmed R and Boutaba R 2015 Elastic virtual network function placement. In: *Proceedings of the 4th International Conference on Cloud Networking (CloudNet)*, pp. 255–260

[18] Li X and Qian C 2016 A survey of network function placement. In: *Proceedings of the 2016 13th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, pp. 948–953

[19] Harutyunyan D, Nashid S, Raouf B and Riggio R 2019 Latency-aware service function chain placement in 5G mobile networks. In: *Proceedings of the IEEE Conference on Network Softwarization (NetSoft 2019)*

[20] Zheng D, Peng C, Liao X, Luo G, Tian L and Cao X 2019 Service function chaining and embedding with spanning closed walk. In: *Proceedings of the 2019 20th IEEE International Conference on High Performance Switching and Routing (HPSR)*, pp. 1–5

[21] Gadre A, Anbiah A and Sivalingam K M 2017 A customizable agile approach to network function placement. In: *Proceedings of the European Conference on Networks and Communications (EuCNC)*, pp. 1–6

[22] Gadre A, Anbiah A and Sivalingam K M 2018 Centralized approaches for virtual network function placement in SDN-enabled networks. *EURASIP J. Wirel. Commun. Network.* 2018(1): 197

[23] Eramo V, Miucci E, Ammar M and Lavacca F G 2017 An approach for service function chain routing and virtual function network instance migration in network function virtualization architectures. *IEEE/ACM Trans. Netw.* 25(4): 2008–2025