Gadre *et al. EURASIP Journal on Wireless Communications and Networking* (2018) 2018:197
https://doi.org/10.1186/s13638-018-1216-0

EURASIP Journal on Wireless Communications and Networking

RESEARCH Open Access

Centralized approaches for virtual network function placement in SDN-enabled networks

Akshay Gadre, Anix Anbiah* and Krishna M. Sivalingam

Abstract

Software-defined networking (SDN) and network function virtualization (NFV) represent significant changes to the architecture of data networks. SDN provides clean separation of the control plane from the data plane while NFV helps virtualize functions typically implemented using *middleboxes* into virtual network functions (VNFs). The network function placement (NFP) problem involves placing VNFs to satisfy the service function chaining (SFC) requests of the flows in the network. Current solutions to this problem are slow and cannot handle real-time requests. In this paper, a static NFP solution that uses a divide-and-conquer approach is discussed first, with complexity similar to that of existing solutions. It is shown that the solution is complete and sound. Next, customization of this solution to obtain an agile version that trades off precision for time complexity is discussed. A combination of this divide-and-conquer solution with a modified version of Dijkstra's algorithm is used to solve the problem for dynamic requests. Finally, a centralized, online SDN-enabled approach to solving the problem is proposed. The proposed architecture and algorithms are simulated and analyzed with various system parameters and are shown to scale to large data center network (DCN) topologies.

Keywords: VNF, NFV, SDN, NFP, Divide-and-Conquer, MANO, DCN

1 Introduction

Current data networks are built using massive infrastructure and serve diverse functions in commerce, education, research, and social networks. The network itself is built in layers in accordance with the open systems interconnection (OSI) model. The layers can be broadly classified into the following: (i) L1–L3 that handle the physical and network layers responsible for connectivity and routing and (ii) L4–L7 that handle transport over the network, security and applications. Traditionally, L1–L3 (lower layers) were implemented within the network, while L4–L7 (upper layers) were implemented at the edge of the network. Over time, data networks implemented service functions such as firewalls, intrusion detection, proxy services, and video compression. These service functions primarily operate at the upper layers and have been termed as L4–L7 services. These services are complex and have been encapsulated in specialized hardware known as *middleboxes*. The middleboxes are usually proprietary with a closed architecture. Thus, they impede the cycle of innovation and entail expensive upgrade cycles when the network as a whole must be upgraded to a larger scale. In recent times, middleboxes have been virtualized and replaced by virtual network functions (VNFs) that are more flexible and can scale on demand.

Although virtualizing the network functions improves flexibility and accelerates innovation, this abstraction introduces new shortcomings that need to be addressed to realize such functions in actual deployments. In this paper, we focus specifically on the problem of network function placement (NFP). For every data flow entering the network, we are required to route them through the appropriate network functions and in a pre-defined sequential order. For example, a firewall function may be followed by a deep packet inspection function in the order. However, we also need to adhere to bandwidth and capacity requirements of the links along the paths between these VNFs while routing the data flow through a chain

*Correspondence: anix@cse.iitm.ac.in
Department of Computer Science and Engineering, Indian Institute of Technology Madras, Chennai, India

Springer Open

© The Author(s). 2018 **Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

of such VNFs. Further, this problem is complicated by the fact that the allocation of these VNFs at the nodes must be simultaneously solved. Therefore, trivial shortest path routes might be inefficient for the purpose of NFP since inefficient routing and function placement solutions may lead to congestion in the network. Figure 1 presents a network as an example depicting two flows through the switches and the placement of network functions to service the flows. For instance, Flow 1 utilizes three network functions, one each at nodes 1, 2, and 4.

The chain of VNFs that are required to process a flow is called its service function chain (SFC). We formalize NFP problem as the problem of finding the optimal placement of VNFs for a set of flows subject to their SFC requirements, bandwidth, and resource constraints with the objective of minimizing the overall use of the resources. The NFP problem can be handled either *statically* or *dynamically*.

The *static NFP* problem deals with the problem of servicing a static set of incoming flows, each with an SFC requirement, where the routes are provided by an oracle. This reduces the NFP problem to that of allocating the VNFs appropriately along the routes to optimize the use of available resources. Even this static version of the NFP problem has been proven to be NP-complete [1]. The *dynamic NFP* problem involves handling dynamic requests for flows with SFC requirements, where even the routes need to be decided by us.

The NFP problem has two main components:

- *Routing:* This component deals with choosing an appropriate route through the network topology to reach the destination considering some path computation objective such as latency, prior placement of VNFs and the bandwidth constraints. Thus, depending on the previous set of VNF allocations and available bandwidth, we need to update the routing metrics to minimize congestion and resource usage in the network. We assume that this component is solved for the *static NFP* by an oracle.
- *Allocation:* This component deals with allocating the optimal set of resources for the flows over the paths selected by the routing component to provision the SFCs of the flows, considering the resource availability and previous placements. Given the set of routes provided by the routing component, VNFs are allocated on the network elements or switches along the routes to fulfill the SFC requirements while aiming to optimize the use of the compute resources available in the network.

Even though attempts [2, 3] have been made to develop fast and optimal solutions for both of the above components individually, they remain impractical for real-world implementation. Also, optimality of any one of the above components does not guarantee the optimality of the overall solution since the output of one feeds into the other and thus constraining the overall solution. Instead, we aim to solve this problem considering both components simultaneously, for developing real-world online NFP solvers. Additionally, we propose a consolidated SDN-enabled architecture and protocol to perform real-time solutions for SFC requests.

The main contributions in this work are the following: (i) A new divide-and-conquer approach for the problem of static NFP which is theoretically proven to be sound and complete; (ii) A heuristic version of the above solution

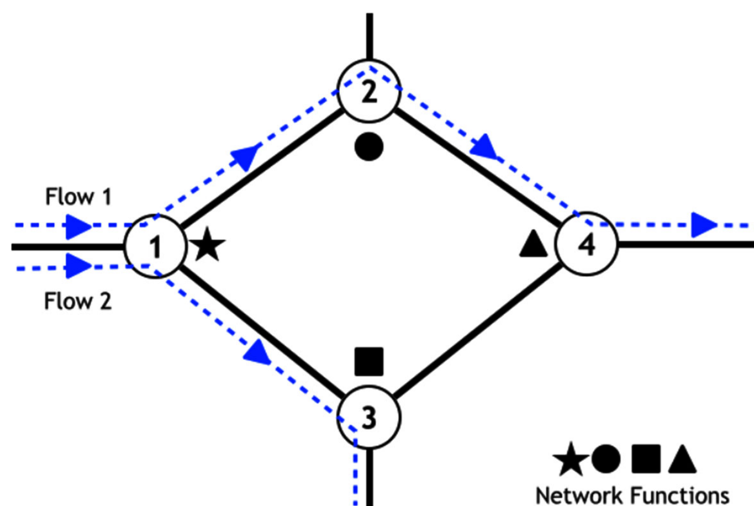

Fig. 1 A sample configuration of flows through a network. A sample network with two flows (Flow 1 and Flow 2) is shown. Flow 1 has three VNFs, while Flow 2 has two VNFs in its SFC

which allows users to trade off accuracy for fast computation times; (iii) An extended version of this heuristic for dynamic NFP along with a complementary and compatible routing algorithm; (iv) A framework for using the above heuristic for real-time online service of SFC requests; (v) A detailed theoretical and empirical analysis of time complexity and the trade-off between the quality of solution and time complexity of the algorithm.

The rest of this paper is organized as follows: first, the previous work done by researchers on this problem and the need for better algorithms are described. Next, the proposed algorithm for solving static NFP is described. In the subsequent section, the above solution is extended with a complementary routing algorithm for dynamically servicing SFC requests. Later, a framework is proposed to harness the power of SDN to use the above algorithm for real-time service of such requests. Simulation results with appropriate inferences and analysis to highlight the major advantages are then discussed. Finally, in conclusion, the proposed protocol, empirical results, and future problems to be solved are summarized.

2 Related work

In this section, brief background information on NFV and SDN are presented and the previous approaches to solving the NFP problem are discussed.

2.1 Network function virtualization (NFV)

NFV architecture aims to harness the power of state-of-the-art virtualization technologies to virtualize the compute, network, and storage resources of commodity network elements to host VNFs that perform the *middlebox* functionality. This has the advantage of avoiding investments in specialized hardware, while allowing elastic scaling of the capacity of the functions [4]. Moreover, switches and routers can use their spare computational resources to implement these VNFs [5]. The VNFs perform the same functions as the conventional *middleboxes* except for the fact that these functions are now implemented in software. This architecture accelerates innovation of the capabilities of the network functions and supports real-time scaling of the virtualized resources as the number of data flows through the network and their service requirements change dynamically [6].

2.2 Software-defined networking (SDN)

Another next-generation network technology, software-defined networking (SDN) [7], is a paradigm of networking that separates the control and the data planes. In this architecture, the data plane is primarily responsible for forwarding the data packets through the network based on flow rules. A centralized controller (or a set of synchronized centralized controllers) is responsible for computing the flow rules based on the state of the network to be pushed to the data plane. A protocol is required for communication between the centralized controller and the switches that implement the forwarding, with *OpenFlow* [8] being a leading example. SDN allows commodity switches in the data plane to be controlled by control software through standard interfaces. Moreover, in this architecture, the controllers can be modified and upgraded easily to effect changes to the behavior of the network as a whole [9].

SDN and NFV are complementary [10] in nature as illustrated in Fig. 2. They have similar objectives of replacing proprietary and closed network elements (switches)

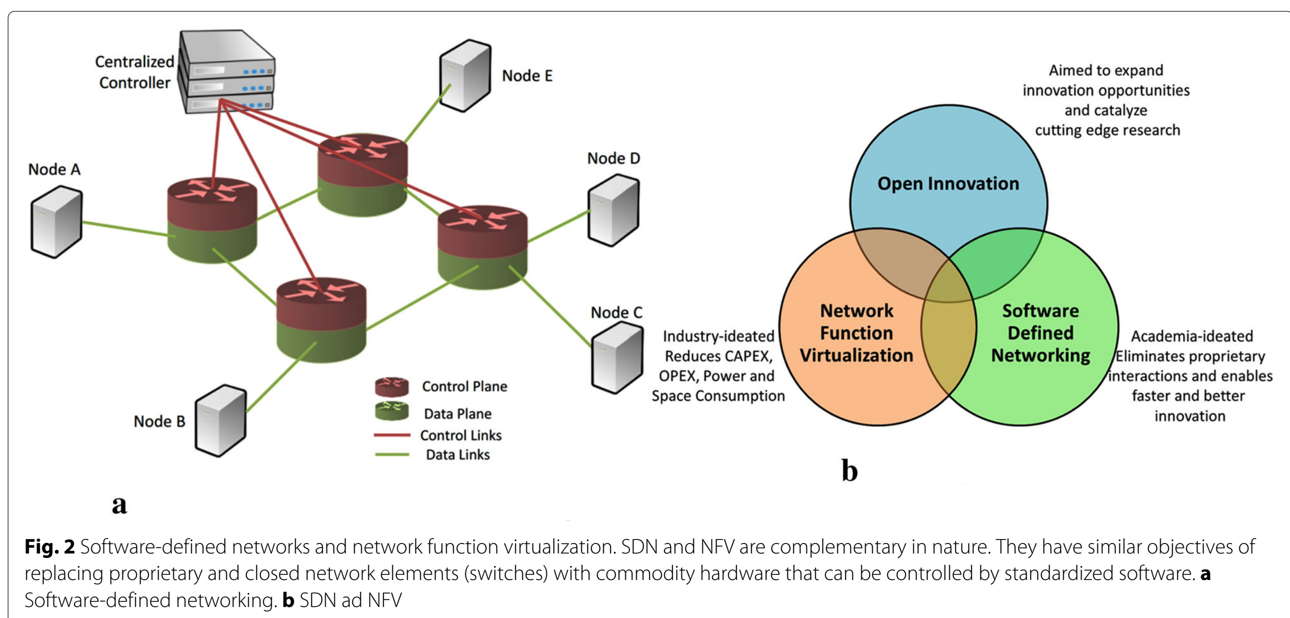

Fig. 2 Software-defined networks and network function virtualization. SDN and NFV are complementary in nature. They have similar objectives of replacing proprietary and closed network elements (switches) with commodity hardware that can be controlled by standardized software. **a** Software-defined networking. **b** SDN ad NFV

with commodity hardware that can be controlled by standardized software.

2.3 Network function placement (NFP)

The NFV architecture gives rise to many interesting problems. One such problem is about the number of instances of the various VNFs required and their placement in the network [2, 3]. Each flow in the network requires a chain (either a strict order or a partial order) of functions that must process the flow according to some defined network policy. Such a chain is termed as a service function chain (SFC) [11]. Each instance of a function can be applied to several flows and takes up resources in the network that are constrained in their availability. Thus the placement of these functions is a complex optimization problem that takes into account the requirements and the constraints and the overall objective of minimizing the consumption of resources [12].

The problem has been widely studied and an extensive survey of the solutions is provided in [2, 3]. The placement problem is known to be NP-Complete [1] and has primarily been solved by treating it as a constraint satisfaction problem.

The work presented in [12] attempts to solve path computation (routing) and function allocation as a unified problem. It combines the path computation constraints and the network function placement constraints into a single instance of a mixed integer quadratically constrained program (MIQCP) and uses a SAT solver to find solutions. Similarly, the work discussed in [13] attempts a co-ordinated solution that solves both the composition of the network functions in a chain as well as embedding them in the substrate network. The approach described in [1] also integrates the solution for VNF allocation and end-to-end demand realization. Some researchers have also attempted to treat VNFs as being capable of getting broken into relatively simple functions that can be distributed and implemented among the switches themselves by using commodity compute, network, and storage resources. The work in [14] proposes a programming model which allows a centralized stateful program to be distributed and deployed using state variables implemented at various switches in the network.

The routing aspect of the NFP problem involves finding the *k-shortest paths* through the network. This problem has been widely studied, and D. Eppstein's solution [15] is the best known solution that allows looping paths. However, if we consider only loopless paths, then Yen's algorithm [16, 17] gives the solution in pseudo-polynomial time. Two variants of these previous algorithms are proposed in this paper as part of the dynamic NFP solution.

The consolidated approach in this paper for *dynamic NFP* is in contrast to the above solutions, since we propose an architecture and a protocol which are consistent with current state-of-the-art NFV and SDN architectures. The paper addresses the interaction of the two technologies with each other and builds on the prior research work on this topic by the authors [18, 19].

3 Static network function placement

In this section, we formally describe the problem statement for static NFP. Then, we present our *sound* and *complete* algorithm to solve the problem. A *customizable*, *agile* heuristic version of the algorithm that is exponentially faster than the state-of-the-art solutions is proposed. We define *completeness* of an NFP algorithm as its ability to consider all possible outcomes and generate a solution if one exists and *soundness* as the property of the output being the optimal solution. Also, a solution is considered more *agile* than another solution if the algorithm has lower time complexity and *customizable* if there are parameters that network administrators can tweak to improve the performance according to their needs or available resources.

3.1 Problem definition

As described in previous sections, for the static NFP problem, the paths of the flows are provided by an oracle. The computation of such paths is left for the dynamic NFP solution. Given the paths, we attempt to address the problem of placement of network functions, subject to the various constraints such as the SFC requirement of each flow and the resources available at each switch to deploy the functions. We consider the chain of functions to be of *linear order*. That is, functions that split or merge the flows are not considered. Similarly, We restrict our discussion to network functions that can be deployed on the network switches [5]. We call these nodes as NFV-enabled nodes. Given the flows through the network and their SFC requirements (SFC_k), we need to place function instances at the switches such that the overall resource cost is minimized.

The resource cost of a virtual network function has two components:

- *Instance cost (I_j):* This is the resource cost to create an instance of the function on any switch.
- *Service cost (S_j):* This is the dynamic resource cost of the function that a switch requires to service one unit of flow data rate.

Modern state-of-the-art switches usually have three types of resources—network, computational, and storage. To simplify our discussion, we will capture the capacity of a switch to host network functions as a single unified resource (Cap_i). It is easy to extend the model to different types of resources. We formally define our problem statement as follows:

Problem statement: Find the placement of network functions among the switches in a network given a set of flows, with each flow taking a pre-computed path through the network and requiring a linear partial order of network functions, each of which has an instance cost and a service cost associated with it, such that the overall cost of placing the network functions for all given flows is minimized.

Let the number of NFV-enabled nodes be N and and number of network functions be M. Let the number of incoming requests be K. A is an allocation matrix where $A_{ij} = 1$ denotes there is an allocation of network function j on node i. F is a routing matrix where F_{ijk}=1 denotes that the flow k is using the network function j allocated on node i. BW is a bandwidth matrix where BW_k denotes the bandwidth utilized by flow k. The problem statement translates directly to:

$$\text{minimize} \quad \sum_j C_j, \qquad i = 1,\ldots,N,\ j = 1,\ldots,M,\ k = 1,\ldots,K$$

$$\text{subject to} \quad C_j = \sum_i \left(I_j A_{ij} + S_j A_{ij} \sum_k \left(F_{ijk} BW_k \right) \right), \qquad \text{Cost Definition}$$

$$Cap_i > \sum_j \left(I_j A_{ij} + S_j A_{ij} \sum_k \left(F_{ijk} BW_k \right) \right), \qquad \text{Capacity Constraint}$$

$$\forall_{ij} F_{ijk} \in SFC_k, \qquad \text{SFC Constraint}$$

3.2 Proposed solution

In this section, we define the system and its parameters that define the problem and the approach to model them. Then, we describe our divide-and-conquer approach to solving the NFP problem. Finally, we present a customizable heuristic version of our algorithm which is more agile.

3.2.1 System description

A VNF is characterized by a three-tuple containing (i) the name of the function, (ii) the instance cost (I_j), and (iii) the service cost per unit data rate of flows (S_j). Although the service cost is taken to be linearly increasing with the data rate, this can be replaced in the algorithm by other functions that are non-linear. A tenant request is defined as a flow taking a given path through the network and having a specific SFC requirement. Thus, a tenant request is described by an ingress node (src_k), an egress node (dst_k), the path taken by the flow through the network (P_{kl}), the data rate of the flow (BW_k), and the sequence of functions that need to service this flow (SFC_k). Often, a flow only needs to be serviced by a set of functions or a partial order rather than a specific sequence of them. The algorithm assumes that such requests have been pre-processed (one such heuristic is presented in [12]) and a specific sequence has been provided as input.

Since the path is also pre-defined for each tenant request, the only information about the network that is required is the resource availability of the network elements. An instance of a network function placed at a specific switch to service a set of one or more tenant requests is called an *allocation*. Note that this can be a subset of the set of all requests passing through this switch with the given VNF in their network function sequence. This may be caused due to the limited availability of resources on the node. The output of the algorithm is a set of *allocations* that satisfy as many input tenant requests as can be fully serviced.

An *NFPSystem* is an instance of the proposed algorithm that solves the NFP problem or a sub-problem. By the definition of the NFP problem, an *NFPSystem* takes as input the list of possible network functions, the allocations that were done a priori, the set of tenant requests that need to be serviced, and the network describing currently available resource capacities of the network elements. It returns with the set of allocations for the given inputs that has the minimum cost among all possible sets of allocations.

3.2.2 Divide-and-conquer algorithm (DCA)

This section presents the divide-and-conquer algorithm (DCA) which solves NFP and is shown to be complete and sound. The pseudo-code of this algorithm is described in Algorithm 1. We create the first instance of *NFPSystem* with the complete available set of network functions, an empty set of allocations, the complete set of tenant requests (in the format described earlier), and the network describing the resource capacities of all the network switches. It starts by finding the set of all possible candidate allocations.

Only maximal allocations are considered initially, i.e., when a function is allocated on a node, it services all requests that pass through the node requesting that function. This will be the optimal allocation strategy if there were infinite resources on each switch. However, in reality, the capacity of the switches is constrained.

For each allocation, the algorithm verifies the cost of servicing the given request. Note that we accrue the instance cost of the network function only if an instance of that function is not already allocated on that node. If an allocation is not feasible, then the tenant request that has the maximum dynamic service cost is removed and the process is repeated. The intuition behind this is to minimize the number of instances of the same type of network function (say, firewall) and, in turn, minimize the instance cost.

If the allocation is left with no requests to serve, then the given request cannot be handled in the current *NFPSystem*, making the *NFPSystem* infeasible for any allocation of the function on the specified node. Hence, no update of the optimum allocation is done.

On the other hand, if an allocation is feasible, we remove the resource capacity needed for that allocation and the available capacity on the corresponding switch is reduced. Then a divide-and-conquer method (line number 6 in Algorithm 1) is used to divide the requests that are being serviced by this allocation of the given function into two parts. For example, consider a request with path A-...-B-C-D-...-E and a VNF chaining request F1-...-F2-F3-F4-...-F5 and let the current allocation place F3 at C for this request. As a result, two new requests are created as <A-...-B-C, F1-...-F2> and <C-D-...-E, F4-...-F5>.

A sample run of the divide-and-conquer algorithm is demonstrated in Fig. 3. Consider the four-node network shown in the figure. The tables list the current capacities of each node, instance, and service costs of each function and the outstanding requests. Initially, the VNF denoted by ★ is allocated on node 2, which results in the capacity of node 2 getting updated and a new reduced set of requests getting generated. This is shown in Fig. 3b. This takes care of both tenant requests getting serviced by this allocation (★ on node 2). Similarly, other allocations are done one after the other until all requests are satisfied. The final state showing all allocations in one path is shown in Fig. 3e.

Algorithm 1: Divide-and-Conquer Algorithm (DCA) for NFP

```
1  NFPSystem (functions, allocs, reqs, net);
   Input  : Set< Function > functions,
            Set< Allocation > allocs, Set< Request >
            reqs, Network net
   Output: Set< Allocation >
2  findAllPossibleAllocations();
3  for each Allocation a do
4      if verifyAllocation(a) then
5          newNet=updateNetwork(net);
6          newReqs=DNCRequests(reqs,a);
7          newAllocs=allocs+a;
8          out=fork(new NFPSystem
           (functions,newAllocs,newReqs,newNet));
9          if out ==NULL then
10             if a.reqList.size>1 then
11                 removeLargestDatarateFlow(a);
12                 goto 6;
13             else
14                 continue ;
15         else
16             calculateCostAndUpdateMin(out+a);
17     else
18         if a.reqList.size>1 then
19             removeLargestDatarateFlow(a);
20             goto 5;
21         else
22             continue ;
23 return minCostSetAllocations;
```

We recursively create parallel *NFPSystem* instances to identify the optimal allocation by exploiting the independence of one allocation from the other. Once a feasible allocation has been identified, this allocation is added to the current list of incoming set of allocations to be passed to a sub-system to handle the new set of requests.

A parallel *NFPSystem* is created as a separate thread to solve a sub-problem and return the set of optimal allocations satisfying that new *NFPSystem*. If the sub-system that handles the new requests is infeasible with the current allocation then the tenant request with the highest data rate is removed. We repeat this process until there is one request left in the allocation. If the allocation has only one request, then there are no further possibilities for reducing the allocation. We deem such a *NFPSystem* infeasible which does not yield a candidate for optimum allocation.

Note that the function chains and the network paths get smaller for the newly created instances of the *NFPSystem*. Eventually, the new set of requests will contain requests with a trivial path that has only a single node. At this point, a simple feasibility check is used to check if that node can support all the functions and, if so, they are allocated on that node. Otherwise, it is deemed infeasible. Another trivial case occurs when a request has an empty set of functions to be serviced along a path. In other words, there are no functions required to be allocated on a given path. These requests are discarded since they are trivially satisfied.

Given the set of sub-systems, each *NFPSystem* compares the cost of allocations from each sub-system and keeps track of the one with the least cost. Similarly, it compares the cost among all such allocations by waiting for all the threads to join and the overall minimal allocation is returned to the parent system which spawned that system originally. This algorithm, thus, performs exhaustive search and is therefore as complex as the current solutions based on MIQCP. Thus, it has a time complexity of $\mathcal{O}\left(e^{poly \log (poly)}\right)$. We provide the detailed analysis in Section 6. Therefore, a more agile solution that can act as a dynamic NFP solver is required, which is presented in the next section.

3.2.3 Fast heuristic algorithm (DCA-H)

Algorithm 1 searches for optimal allocation of all tenant requests exhaustively. There are many ways the algorithm can be made more agile by sacrificing the completeness and the soundness of the algorithm. In this section, we

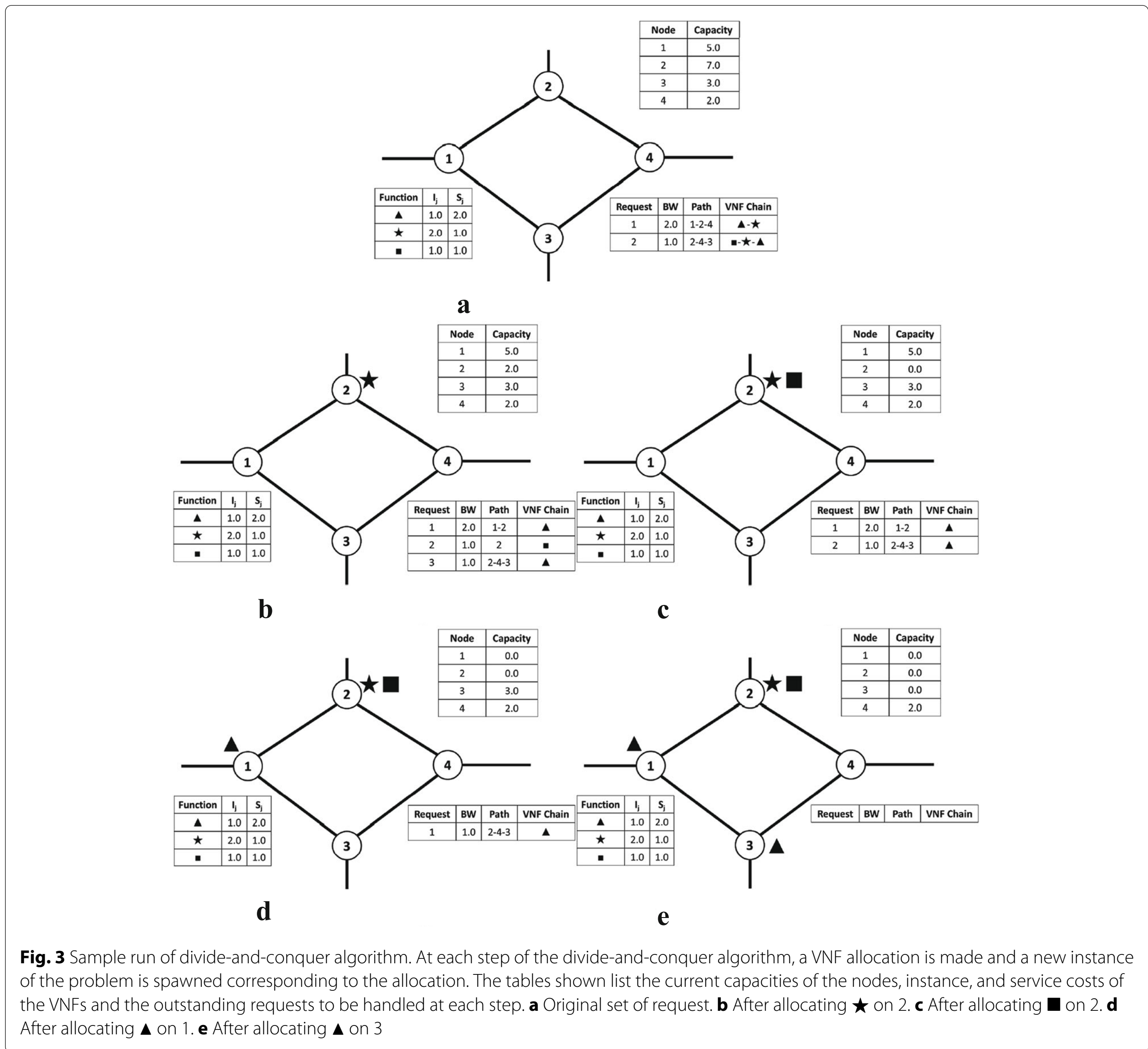

Fig. 3 Sample run of divide-and-conquer algorithm. At each step of the divide-and-conquer algorithm, a VNF allocation is made and a new instance of the problem is spawned corresponding to the allocation. The tables shown list the current capacities of the nodes, instance, and service costs of the VNFs and the outstanding requests to be handled at each step. **a** Original set of request. **b** After allocating ★ on 2. **c** After allocating ■ on 2. **d** After allocating ▲ on 1. **e** After allocating ▲ on 3

present DCA-H, which is a heuristic version of the algorithm and is more customizable and agile. Note that this algorithm is called *agile* since it is exponentially faster than all previous solutions and *customizable* since there are multiple parameters that network administrators can tweak to improve the performance according to their available compute resources.

The first and most effective way to improve agility is to bound the branching at every *NFPSystem* instance. This can be enforced by only looking at top T of the allocations possible at every step for the optimal cost solution. Once the branching is curtailed, the algorithm becomes significantly faster. Therefore, to achieve this optimization, only the top T allocations in line 4 of Algorithm 2 are considered. This requires sorting the allocations in a particular order beforehand such that the top T allocations will reasonably represent the overall system. Some of the possible sorting orders are the decreasing order of number of requests each allocation serves, decreasing order of data rate of the flows that each allocation serves, or the decreasing order of cost to the system due to the allocation.

We introduce another heuristic by reducing the backtracking upon discovering the subsystem created by a verified allocation is infeasible to solve. This can be accomplished by making lines 11–17 optional in Algorithm 2. Note that this will have significant performance impact since this polynomially reduces the upper bound on the depth of the *NFPSystem* visitor tree. The same is true for removing backtracking unverified

Algorithm 2: Heuristic Algorithm(DCA-H) for NFP

```
1  HeuristicNFPSystem (functions, allocs, reqs, net);
   Input  : Set< Function > functions,
            Set< Allocation > allocs, Set< Request >
            reqs, Network net
   Output: Set< Allocation >
2  findAllPossibleAllocations();
3  sortAllocations();
4  for a in top T allocations do
5      if verifyAllocation(a) then
6          newNet=updateNetwork(net);
7          newReqs=DNCRequests(reqs,a);
8          newAllocs=allocs+a;
9          out=fork(new HeuristicNFPSystem
           (functions,newAllocs,newReqs,newNet));
10         if out ==NULL then
11             OPTIONAL[ if a.reqList.size>1 then
12                 removeLargestBandwidthFlow(a);
13                 goto 6;
14             else
15                 continue ;
16             ]
17         else
18             calculateCostAndUpdateMin(out+a);
19     else
20         OPTIONAL[if a.reqList.size>1 then
21             removeLargestBandwidthFlow(a);
22             goto 5;
23         else
24             continue ;
25         ]
26 return minCostSetAllocations;
```

allocations (allocations that will not meet the resource constraints), which can be done by making lines 22–28 optional in Algorithm 2. This will not drastically improve the time complexity but will significantly reduce the soundness of the algorithm. Thus, it is recommended that this part is left unmodified in DCA-H.

To summarize, we proposed a complete and sound algorithm for static NFP in this section and provided a customizable agile heuristic version of the same, which allows tenants to trade off precision for better computation time.

4 Dynamic network function placement

In this next section, we define the problem of dynamic NFP formally and describe a modular approach which harnesses the power of independent optimal solutions of each of the individual components, namely, routing and allocation. While the problem of placement of virtual network functions (VNFs) has been widely studied, most solutions either look at routing of flows and optimal allocation of VNFs on the route individually or are too slow for practical implementation as online VNF solvers.

Note that each VNF has a specific migration delay (Mig_l), a start up delay (Stp_l), and a tear down time (Trp_l). Static values are assumed for migration delay for a function, since usually the migration happens through an overlay network built in the NFV Management and Orchestration (MANO) architecture of the ETSI (European Telecommunications Standards Institute). Also, homogeneity across all commodity switches for static start up and tear down delay is assumed.

The allocation component of the problem deals with the least cost allocation of network functions given a static set of requests and the corresponding routes that they will take. Note that the allocation optimization problem remains the same as in dynamic NFP as static NFP. The routing part of the problem tries to look for the best possible route for a VNF chaining request where a set of VNFs are statically allocated and routes may loop to satisfy function chains. Our approach ignores the possibility of dynamic VNF migration and may end up using inefficient routes to meet the SFC requirements. However, even the fastest VNF migration [20] is too slow for migration during a flow, hence justifying our assumption.

Let the number of incoming requests be K and the number of links in the network be L. P is a path matrix where P_{kl}=1 denotes link l used in route for flow k. BW is a bandwidth matrix where $\mathrm{BW}_{k/l}$ denotes the bandwidth utilized by flow k or maximum bandwidth of a link l. Lat is a latency matrix where $\mathrm{Lat}_{k/l}$ denotes the latency constraint for a flow k or latency of a link l. Formally, the routing component can be described as the following optimization problem:

$$\begin{aligned}
\text{minimize} \quad & \sum_l \left(\sum_k P_{kl} Lat_l + (\exists_k P_{kl} == 1)\,(Mig_l + Stp_l) \right) \\
& \qquad\qquad k = 1, \ldots, K,\; l = 1, \ldots, L \\
\text{subject to} \quad & BW_l \geq \sum_k P_{kl} BW_k, \qquad \text{Bandwidth Constraint} \\
& P_{ksrc_k} = 1; P_{kdst_k} = 1 \qquad \text{Source and Destination Constraint}
\end{aligned}$$

The combination of the routing and allocation aspects of the NFP problem makes it intractable and most of the complete and sound solutions of the above problems are too slow for deployment in real networks. However, there exist fast heuristics which solve both of the above problems that significantly improve the time complexity. In this section, we propose a consolidated solution which combines such heuristic approaches. However, since the

solution is heuristic, neither the soundness nor the completeness of such a solution is guaranteed. To summarize, the proposed solution is a consolidated approach for VNF placement, which combines the objectives of optimal routing as well as least-cost placement VNFs, to provide a real-world online solution for NFP.

4.1 Hybrid Dijkstra's algorithm

In this section, we discuss an approach for the routing aspect of the dynamic NFP problem that also considers the importance of the re-utilizing previously allocated functions which are servicing existing flow requests already. Traditionally, link state routing protocols such as the Open Shortest Path First (OSPF) protocol are used for (i) discovering and maintaining the link state of the network and (ii) using algorithm such as Dijkstra's to compute shortest paths based on administrative weight or path attributes such as latency. We use the overall latency of allocation as the optimization criterion [21].

However, such path computation is completely oblivious to the already allocated network functions in the network and conceivably, a longer path with network functions which can be reused for the request being serviced, can provide lower latency by avoiding the migration and setup latency of the network functions. Algorithm 3 is a hybrid Dijkstra's algorithm that addresses this problem by considering the presence of network functions on the commodity switches to compute a path from source to destination.

Algorithm 3: Hybrid Algorithm for routing tenant requests

```
1  HybridRouting(net, req, allocs);
   Input  : Network net, Request req,
            Set< Allocation > allocs
   Output: Path p
2  allPaths=Set< Path >;
3  shortest=Dijkstra(net,req.src,req.dst);
4  for each edge e in shortest do
5  |   newShortest=Dijkstra(net-e,req.src,req.dst);
6  |   allPaths.add(newShortest);
7  minCost=Cost(shortest);
8  for each Path q in allPaths do
9  |   cost=Cost(q);
10 |   if cost<minCost then
11 |   |   shortest=q;
12 |   |   minCost=cost;
13 |   end
14 return shortest;
```

The algorithm proceeds as follows. The shortest path between the source and the destination using the traditional Dijkstra's algorithm (line 3) is first identified. Then, to provide alternatives to the shortest path, k-shortest paths are computed between the source and destination. These paths are retrieved by excluding individual links of the shortest path. This approach is similar to the ones taken by Yen [16, 17] to find loopless k-shortest paths. Thus, the path computed upon removal of each edge is computed and added to the set of paths to be checked (lines 4–6).

Algorithm 4: Heuristic Algorithm for NFP

```
1  HeuristicNFPSystem(functions, allocs, p, fs, net);
   Input  : Set< Function > functions,
            Set< Allocation > allocs, Path p,
            FunctionSeq fs, Network net
   Output: Set< Allocation >
2  findAllPossibleAllocations(s, p);
3  sortAllocations();
4  for a in top T allocations do
5  |   if verifyAllocation(a) then
6  |   |   newNet=updateNetwork(net,a);
7  |   |   {newFS1,newFS2,newP1,newP2}=
   |   |   DNCNewSystems(fs,a);
8  |   |   newAllocs=allocs+a;
9  |   |   out1=fork(new HeuristicNFPSystem
   |   |   (functions,newAllocs,newFS1,newP1,newNet));
10 |   |   out2=fork(new HeuristicNFPSystem
   |   |   (functions,newAllocs,newFS2,newP2,newNet));
11 |   |   if out1 ≠NULL and out2 ≠NULL then
12 |   |   |   calculateCostAndUpdateMin(out+a);
13 |   |   end
14 |   end
15 return minCostSetAllocations;
```

An illustration of the above algorithm is shown in Fig. 4. The traditional Dijkstra's algorithm is used to find the path with the least latency (1-2-5-7) from the source to the destination node (Fig. 4b). Next, for each edge e in this path, the shortest path is computed assuming e is unavailable or has ∞ latency. As illustrated, there are three candidate paths, namely, "1-2-5-7", "1-3-6-7", and "1-2-4-5-7", each found after excluding one of the edges in the shortest path.

The cost of each path is calculated as follows:

$$c_i = \sum_{j \in \text{edges}} \text{latency}_j \times \left(1 - \alpha_r \frac{\text{matchingAllocs}}{\text{totalFuncs}}\right)$$

where *matchingAllocs* refers to the number of allocations on the path, where the VNF of the allocation occurs in the function sequence of the request and *totalFuncs*

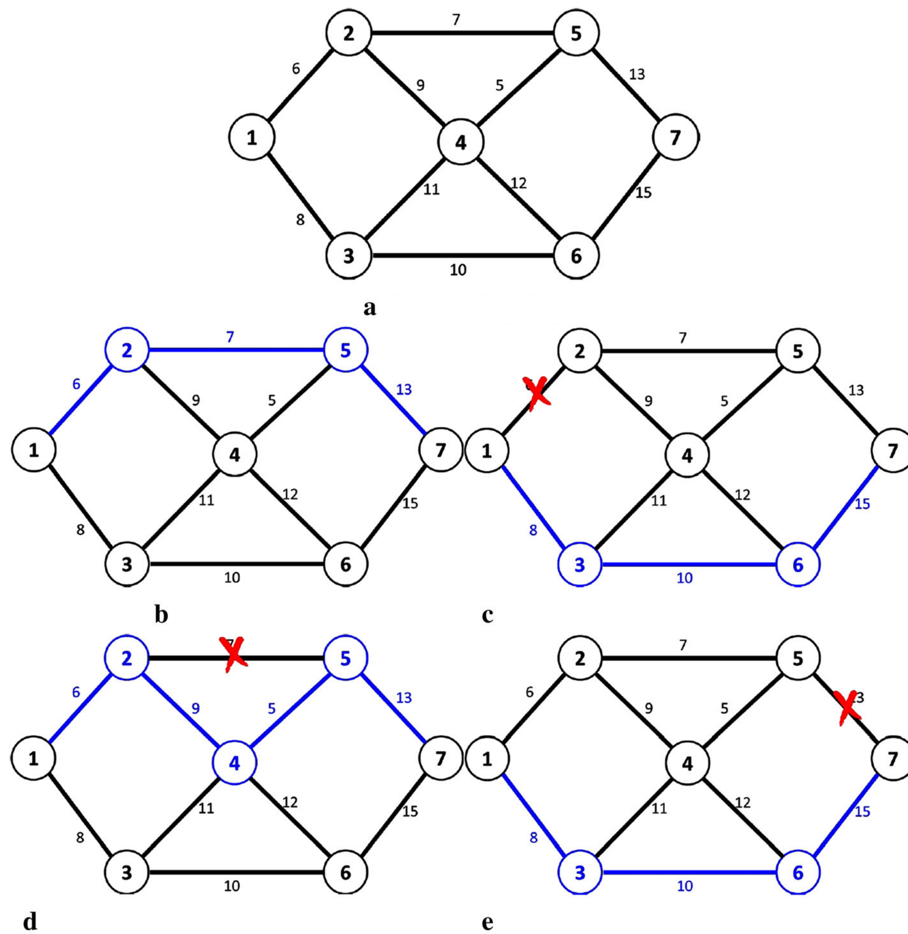

Fig. 4 Sample run of the hybrid Dijkstra's algorithm to generate candidate paths. The shortest path between the source and the destination using the traditional Dijkstra's algorithm is first identified. Then, to provide alternatives to the shortest path, k-shortest paths are computed between the source and destination. These paths are retrieved by excluding individual links of the shortest path. **a** Sample Topology. **b** Shortest path. **c** Shortest path assuming 1-2 is inactive. **d** Shortest path assuming 2-5 is inactive. **e** Shortest path assuming 5-7 is inactive

denotes the total number of functions in the SFC of the request. Note that in the above formula, the coefficient of $\frac{\text{matchingAllocs}}{\text{totalFuncs}}$ (α_r) represents the ratio of importance given to previous allocations versus the actual latency/cost of the path. We use $\alpha_r = 0.25$ for our results but it can be replaced with any other number depending on the ratio of importance you want to give the existing allocated network function. Using the above cost heuristic, the cost of the shortest path (the one with the lowest latency) is found (line 8). Then for all candidate paths in *allPaths*, we calculate that cost and if it is less than the cost of the currently known lowest cost path, then it is used to replace the shortest path (lines 9-15). Finally, we return the path with the lowest cost (line 16).

4.2 Fast Centralized NFP Solution (FDCA-H)

In this section, a heuristic approach is discussed for finding the optimal allocations of functions to resolve the SFC requirements of a tenant request using the path computed using the abovementioned algorithm. When a request for finding new allocations is handled, there may be allocations already done on the given path. Thus, either existing allocated functions can be used to serve this flow or due to resource constraints, new virtual network functions must be spawned. Using existing allocated functions has the advantage of not waiting for the migration and setup of the VNF instances to be complete, while instantiating new functions will allow load balancing, which may facilitate better service of future NFV requests.

This fast heuristic approach is described to solve the allocation problem based on the paths computed by Algorithm 3 builds upon the solution described in Algorithm 1. Note that, in this approach, the problem always reduces at every computation depth and thus guarantees completion. We will show in Section 6.1.1 that this algorithm is neither complete nor sound, but the deficit in

performance is compensated by the immense decrease in time complexity.

Figure 5 shows a sample run of the algorithm for the path shown in Fig. 4d assuming that ★ is allocated on node 4. The static and dynamic costs of each function are displayed along with the remaining capacity of each node in the path. We find that the next best possible allocation is to allocate the node 4 to also serve the incoming request 1. After allocating ★ on node 4 (Fig. 5b), the request is divided into two sub-requests as described above. Note that, since ★ was already allocated, only the service cost $\times$ BW_j units of resources were used. Next, if ■ is allocated on node 2 (Fig. 5c), then the first sub-request is fulfilled and only the second sub-request remains. Finally node 7 being the only possible node to support ▲ with bandwidth 2.0, ▲ is allocated on node 7 (Fig. 5d) and a potential set of allocations to service request 1 is arrived at. Thus, the above online NFP solver uses the information of previous allocations on the path along with current state of the system to look for an optimal set of allocations to service a request.

5 SDN-enabled framework for online NFP

In this section, we describe a centralized SDN-enabled consolidated approach to solve the problem described above with individual components, as illustrated in Fig. 6. In a traditional network, a set of hardware-based *middle-boxes* and other commercial switches are used to implement the necessary network functions. For the purpose of our discussion, we will assume that the network is made of NFV-enabled commodity switches that are capable of being controlled by an SDN controller connected directly to each other. Let the topology of such a network be termed as Network Function Topology (NFT). This allows the simplification of discussion by avoiding discussion about conventional switches in the network.

As in all SDN-enabled technologies, only the centralized controller will have the full view of the NFT and

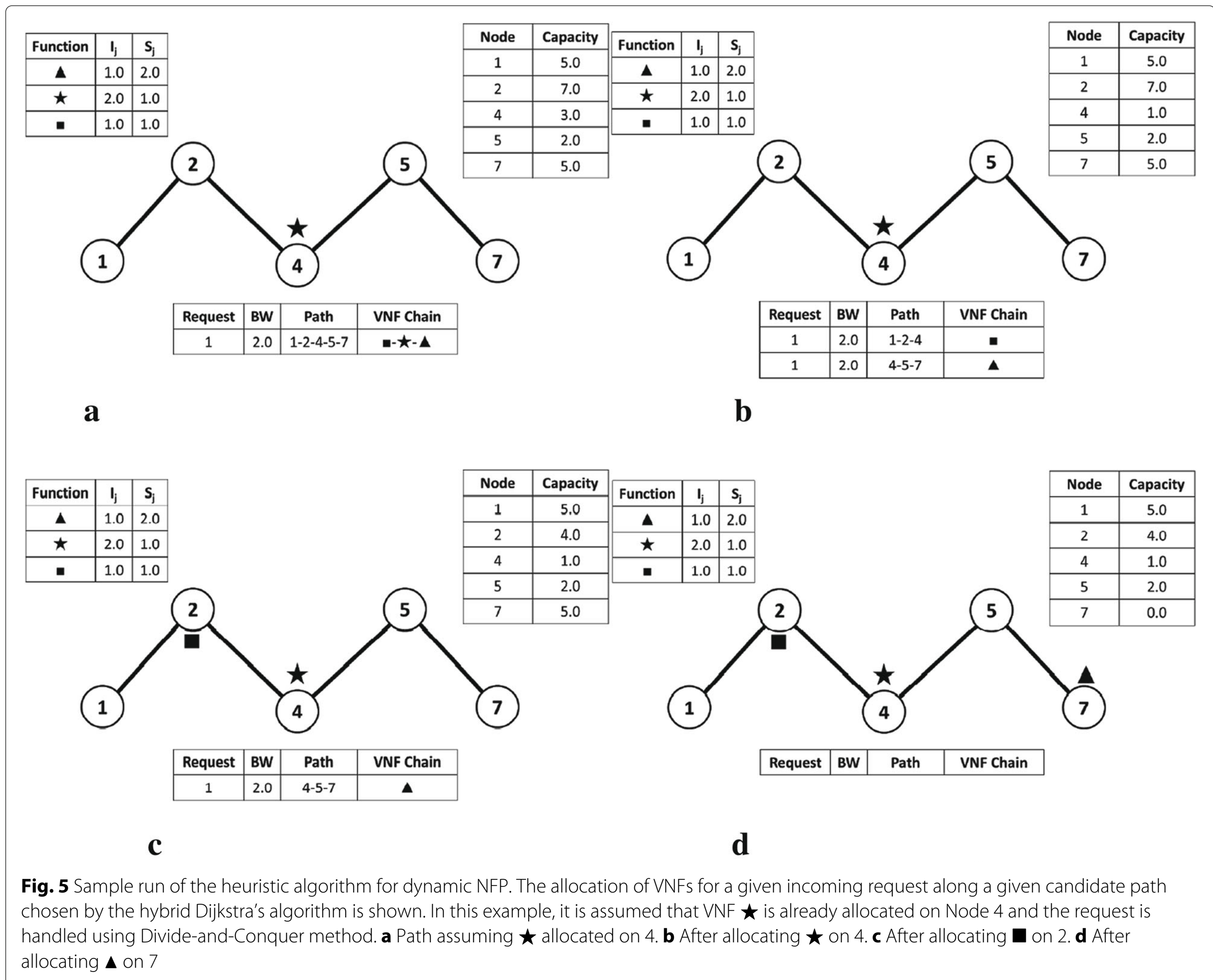

Fig. 5 Sample run of the heuristic algorithm for dynamic NFP. The allocation of VNFs for a given incoming request along a given candidate path chosen by the hybrid Dijkstra's algorithm is shown. In this example, it is assumed that VNF ★ is already allocated on Node 4 and the request is handled using Divide-and-Conquer method. **a** Path assuming ★ allocated on 4. **b** After allocating ★ on 4. **c** After allocating ■ on 2. **d** After allocating ▲ on 7

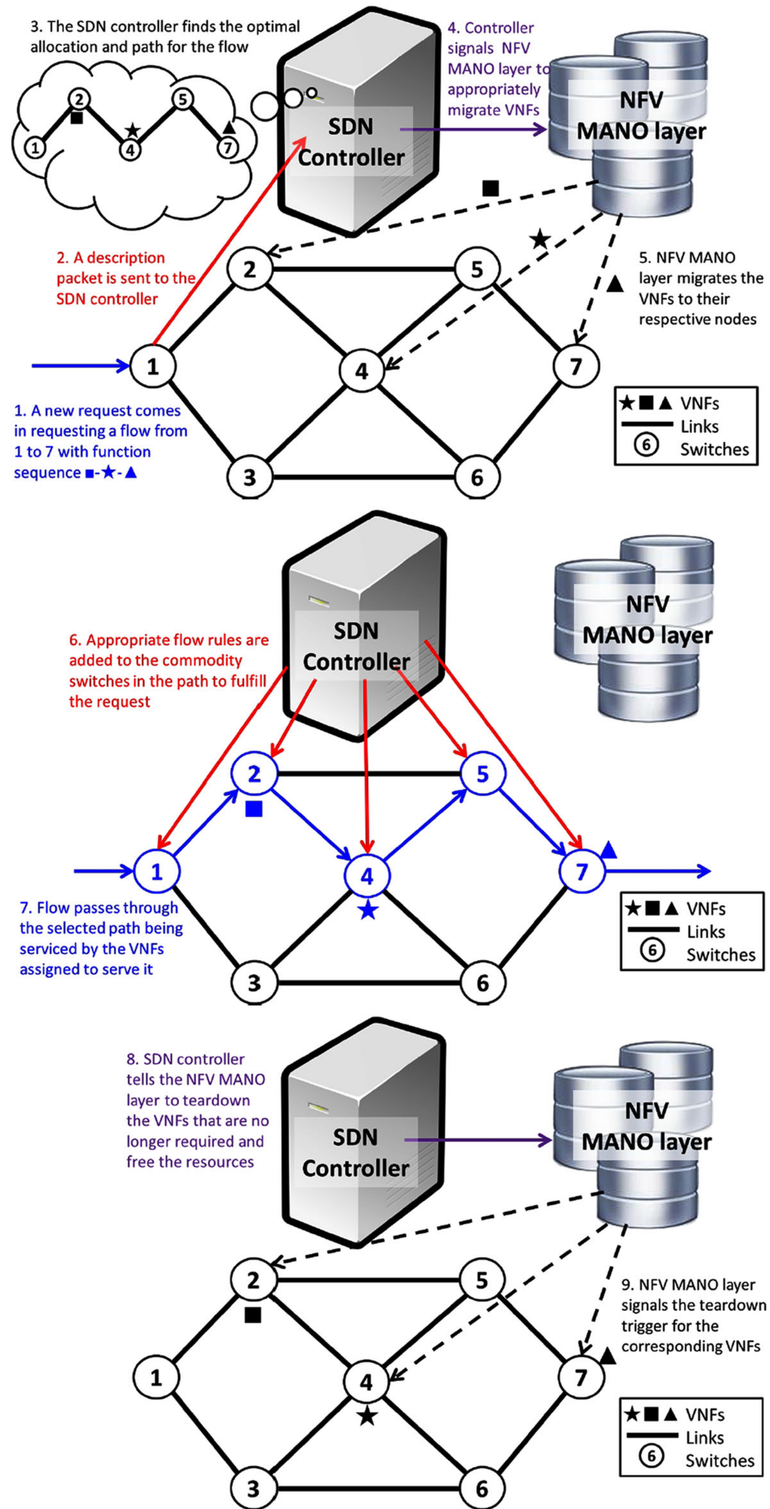

Fig. 6 Proposed protocol for solving NFP. A centralized SDN-enabled approach to NFP is illustrated in this figure. An SDN controller uses the optimal path allocation algorithm and leverages the NFV MANO layer to accomplish NFP for incoming requests

be able to query the remaining bandwidth or resources of its elements. Also the controller will be a reactive agent, which will respond based on the packets or requests that it receives. We also assume that the MANO layer of the NFV architecture can communicate with the SDN controller, as is prevalent in state-of-the-art SDN-NFV combined approaches. Whenever a new flow enters the NFT, the first switch receives a description packet, which is forwarded to the SDN controller. This description packet contains the description of the VNF sequence that the incoming flow requires, the destination node, the bandwidth and the time for which the flow of packets will occur. While the controller is finding the optimal route and allocation for the flow, the packets are routed through the controller to the destination via a high latency path. The controller is assumed to have sufficient resources to run all possible VNFs always to service the chaining request of this flow while the dynamic NFP solution is being calculated [22].

The controller first looks for a set of candidate paths that could possibly be optimal routes for this flow. The main constraining resource for routing is the bandwidth of the links in the NFT. Using appropriate cost functions, the controller finds the best possible route of the incoming request. Once the appropriate route is found, this route along with the sequence of VNFs that need to be serviced for this flow is supplied to the online NFP solver. The NFP solver uses the divide-and-conquer approach to divide the request into smaller requests and find the optimal NFP solution.

It may so happen that there is no possible route or that for the chosen route there is no feasible network function placement found. In such cases, the solution for the request is delayed by a certain amount described as the *backoff delay* of the system. The solution is reattempted periodically until a threshold upper bound value denoted as the *denial cutoff* above which the request is denied and the high latency path through the controller is used for the duration of the flow. However, if the solution is found, then the SDN controller removes the bandwidth and resource capacities of the links and nodes respectively according to the allocation in its internal view of the NFT. It then invokes the MANO layer of the NFV architecture to migrate the required VNF containers to the required nodes and to scale each of the VNFs to the updated bandwidth. Thus, the *migration delay* for a request is defined as the maximum migration delay out of all the migrations of VNFs required for the request. We also define *startup delay* as the maximum sum of migration and startup delays of the VNFs required to meet the needs of the specified allocations.

Next, the flow rules for the request are pushed by the controller in the respective switches. At every switch, the rule will be as follows:

```
match<VLAN ID=reqID; input_port=4>
                =>action<output_port:=3>
```

The rules are added in sequence starting from the destination switch and the last rule is added at the source switch in the path. After this, the incoming flow packets are switched across the data plane. Once the time defined in the description packet is elapsed, the flow is said to have *completed*. Once the flow is completed, the request ID is removed from all allocations stored by the SDN controller. At this point, the bandwidth capacity is freed in the links and dynamic cost for the given request for all relevant allocations in the SDN controller. This may also lead to some allocations serving no active requests in the NFP system. The controller triggers a *teardown* for all such allocations—i.e., it tells the MANO layer that the VNF container is no longer required and can be vacated to free the resources. After a corresponding *teardown delay*, for each such null allocation, the *instantiation cost* for the VNF container is freed at the switch. This SDN-enabled framework harnesses the power of individual heuristics for each of the components and allows a consolidated feedback-based approach for optimal online service of VNF chaining requests.

6 Experimentation and results

In this section, details of the experimentation and theoretical and empirical analysis of various approaches are presented and it is shown that these approaches are significantly better than prior solutions to the NFP problem.

6.1 Analysis of static NFP

In this section, Algorithm 1 (DCA) is shown to be complete and sound. Then, the time complexity of the DCA-H algorithm is discussed to justify that it is polynomial when all the subsystems of a given system are run in parallel. Further, the solution is empirically studied with various parameters and finally shown to scale to even large state-of-the-art DCN topologies.

6.1.1 Completeness and soundness of DCA

Completeness of an NFP algorithm is its ability to consider all possible inputs and generate a solution if one exists. This is automatically taken care of for each sub-problem in our case by the definition of the inputs. This is guaranteed for a given linear function chain and a given path for every request and given resource capacity of a network node. Thus, DCA is proven to be *complete*.

Soundness of an NFP algorithm is the property of the output solution being of optimum cost and if the algorithm does not find a solution, then there does not exist a solution. If the algorithm produces a solution, it is

clear that at each stage it has found the minimum cost subsystem after each allocation. Also, at each stage, we find and use the allocation which gives us the minimum cost. After each system and its subsystems return the possible allocations, the minimum cost among all possibilities is returned to the parent system. This guarantees that given a feasible *NFPSystem*, the algorithm finds the optimal solution for all subsystems that are visited. Using this argument recursively, DCA is proven to be *sound*.

6.1.2 Theoretical complexity analysis

For this analysis, N denotes the number of network elements, M the number of network functions, and K the number of tenant requests.

DCA complexity: A given *NFPSystem* is started by finding all possible allocations. The total number of such allocations is $\mathcal{O}(NM)$. Then for each allocation, the allocation is verified and the network is updated, both of which takes $\mathcal{O}(1)$. The divide-and-conquer algorithm is then executed to find a new set of requests, which is $\mathcal{O}(K)$. In the worst case, the number of new requests is twice the number of current requests. Thus the complexity of each run of NFPSystem is $\mathcal{O}(NM + K)$. Then, the depth that the search tree can grow to is analyzed, which has an upper bound of the number of possible allocations $\mathcal{O}(NM)$ multiplied by the maximum number of times a function can be allocated on the node, which is $\mathcal{O}(K)$. Thus, the upper bound of the depth is $\mathcal{O}(NMK)$. The branching at each step is determined by the possible number of allocations, which is $\mathcal{O}(NM)$. Thus, the worst case time complexity is $\mathcal{O}\left((NM + K) \cdot \left(NM^{(NMK)}\right)\right)$ which is $\mathcal{O}\left(e^{poly \log (poly)}\right)$.

Heuristic complexity: In the heuristic version, the allocations are sorted, which is done in $\mathcal{O}(NM \log (NM))$ time. Thus, the worse-case complexity with the sorting, but without any heuristic bounding becomes $\mathcal{O}\left((NM \log (NM) + K) \cdot \left(NM^{(NMK)}\right)\right)$. Looking at only the top T allocations will bound the number of branches at each step by T (a constant). This will change the multiplicative factor to $\mathcal{O}\left(\ldots \cdot \left(T^{(NMK)}\right)\right)$. This leads to the overall complexity of $\mathcal{O}\left(e^{poly}\right)$, where the exponent is actually quite small in most of the typical scenarios. For $T = 1$, the algorithm becomes polynomial time with complexity $\mathcal{O}(NM \log (NM) + K)$. The next optimization step is to reduce the backtracking that happens when we find an infeasible subsystem which reduces the first term of the complexity to $\mathcal{O}((NM \log (NM)) \cdot \ldots)$. Moreover, for $T = 1$, the complexity reduces to $\mathcal{O}(NM \log (NM))$. Note that this complexity is independent of K and is only a function of the network topology and set of possible network functions that can be allocated. This is very important to give performance guarantees in a network system.

6.1.3 Empirical time complexity analysis

Base case parameters: Given a set of nodes N, we generate $\sqrt{N}$ requests. Each node has a resource capacity of $N^{0.8}$ and each request has a random number of nodes in its path size, chosen uniformly from the range $\sqrt[3]{N}$ to $\sqrt{N}$. Each request has a data rate of one unit and the nodes on the path are randomly chosen from 1 to N. The number of functions requested for each path are chosen randomly from 1 to $\sqrt[4]{N}$. Each of those functions are chosen from a pre-decided set of 10 functions. The value of T chosen for the base case is 1.

The base case is compared with requests having longer paths and longer function sequences. For longer paths, path length is varied from $\sqrt{N}$ to $N^{0.6}$ instead, while for long function sequences, the function sequence length is varied from from 1 to $\sqrt[3]{N}$. The effect of increasing the path size and having longer function chains per request is shown in Fig. 7a which shows that the computation times is of the order of seconds in the scenarios considered. It is seen that having longer function chains affects performance more adversely than having longer paths.

Next, the base case is compared with a system having N requests instead of $\sqrt{N}$. Increasing the number of requests increases computation time exponentially, as seen in Fig. 7b. This is because increase in the number of requests increases the number of possible allocations exponentially. This reaffirms the results of our complexity analysis. Then, T is varied from 1 to 4 to find the complexity of increasing the number of top T allocations considered. As expected, increasing T results in an exponential increase in the time taken for running the system. This is shown in Fig. 7c, where time is plotted on a log scale.

Another important dimension to analyze the solution is to find at what values of N does it become impractical to be run for different values of T. For this analysis, the base case is checked with values of T varying from 2 to 10 and values of N increasing in jumps of 25 each. The cut-off time for the simulation is taken as 30 min beyond which the solution becomes impractical. The same experiment is repeated for DCA as well with N increasing in jumps of 5 each. The results are shown in Fig. 7d. It is seen that to make this algorithm useful for practical purpose for values of T beyond 2, significant additional algorithm optimization is required. Also, with increase in values of T, the maximum value of N, for which it is reasonable, decreases exponentially. It should also be noted that the DCA did not finish in 48+ h for $N = 45$.

6.1.4 Precision time trade-off analysis

Precision is traded off for time complexity when a smaller value of T is chosen (Table 1). Hence, it must be verified whether the performance with smaller values of T is comparable to the larger values of T. Three topologies of 25,

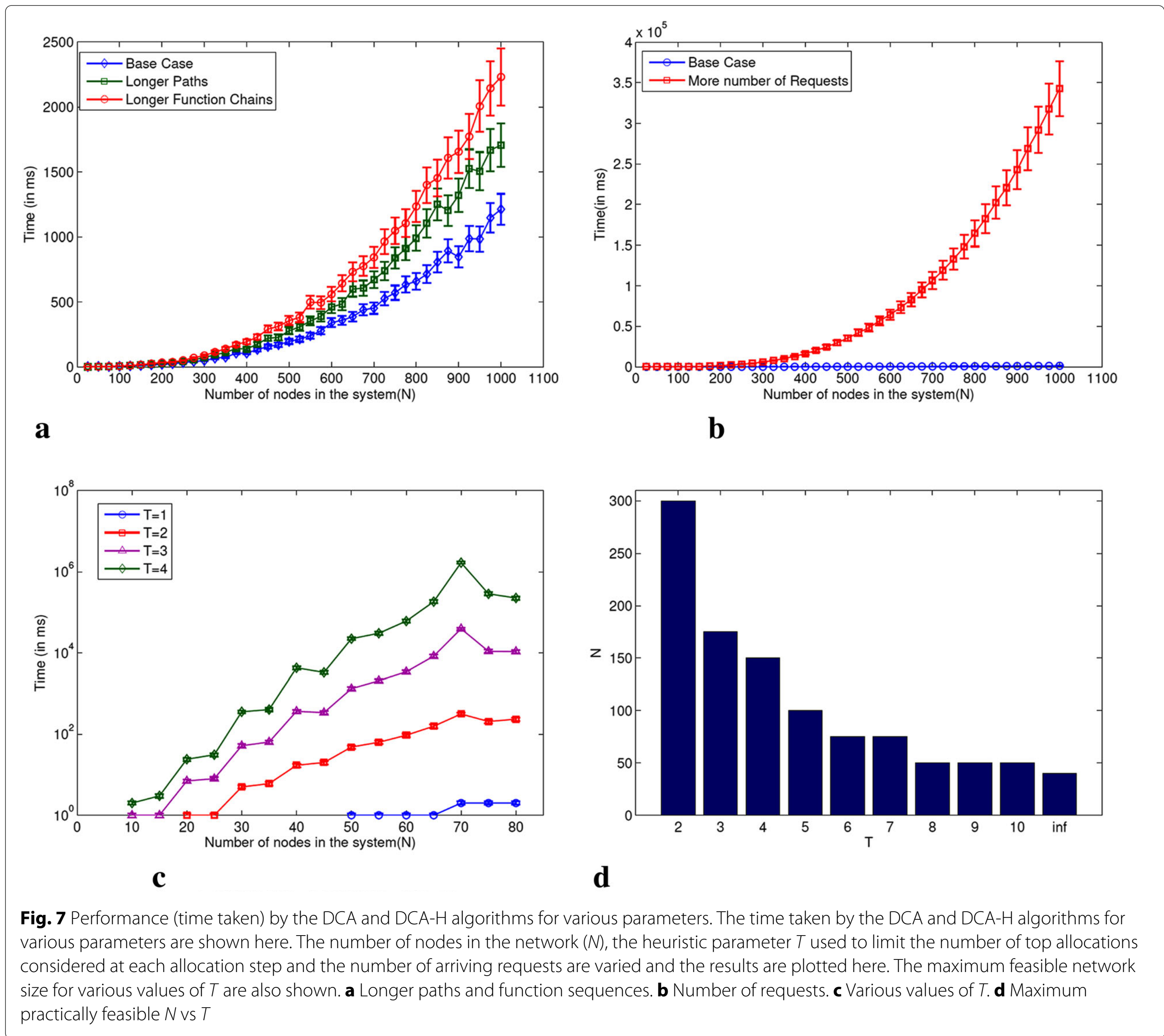

Fig. 7 Performance (time taken) by the DCA and DCA-H algorithms for various parameters. The time taken by the DCA and DCA-H algorithms for various parameters are shown here. The number of nodes in the network (*N*), the heuristic parameter *T* used to limit the number of top allocations considered at each allocation step and the number of arriving requests are varied and the results are plotted here. The maximum feasible network size for various values of *T* are also shown. **a** Longer paths and function sequences. **b** Number of requests. **c** Various values of *T*. **d** Maximum practically feasible *N* vs *T*

50, and 100 nodes each were tested with the base case criteria and various values of T. For 50 nodes, the time taken by $T = 1$ was 2 ms, $T = 3$ was 1.8×10^3 ms, and $T = 5$ was 1.2×10^5. The output minimum cost values of 7.2 resource units matched for both. The simulation for T=3 and T=5 took 15+ h and thus results have not been compared. This shows that the deficit in performance by choosing lower values of T can be reduced substantially by using an appropriate sorting mechanism.

Table 1 Precision Time Trade-Off

Resource units	T=1	T=3	T=5	T=∞
N=25	7.2	7.2	7.2	7.2
N=50	14	14	14	14
N=100	17.6	17.6	17.6	17.6

6.1.5 Scalability in DCNs

Since NFV is well suited for data center networks (DCN), a DCN scenario is also considered. Here, the flows are primarily between leaf nodes or between core nodes and leaf nodes. The topology used is the fat-tree topology described in [23].

The capacity of each leaf node was kept at $\frac{N}{2}$, each edge and aggregate node at $\frac{N^2}{4}$, and each core node at $\frac{N^3}{8}$ in a N-pod FatTree topology. The data rate of each flow was kept constant at 1.0 unit. This was done so as to not affect the symmetry of flow requests from each node. The number of requests generated for an N-pod FatTree was $\sqrt{\mathcal{N}}$, where $\mathcal{N}$ denotes is the number of nodes is an N-pod FatTree. The number of functions in the function chain were specified to be randomly chosen between 1 to 3 for core-to-end paths and 3 to 5 for end-to-end paths proportional

to the worst case number of nodes in the paths. N was varied from 2 to 48 with jumps of 2 each.

The results obtained are shown in Fig. 8. Even for large 48-pod FatTrees containing 30,528 nodes, the time taken for end-to-end requests was around 10 min. For core-to-end requests, the time taken was 2.5 min, which is relatively acceptable.

6.2 Analysis of dynamic NFP

In this section, the execution time (in ms) of the proposed algorithm is studied with varying parameters. The execution times for randomly generated connected topologies are studied first, followed by DCN topologies of K-pod Fat Trees.

6.2.1 Non-symmetric topologies

In this section, how the algorithm scales to bigger topologies with a large number of NFV-enabled switches/nodes is analyzed. For each N-node topology, following arrival rates of requests are studied:

- *Heavy:* N requests arriving in N ms.
- *Medium:* $N^{\frac{4}{5}}$ requests arriving in N ms.
- *Sparse:* $\sqrt{N}$ requests in arriving in N ms.

A *backoff delay* of $0.1N$ ms and a *denial delay* as $2N$ ms are used. Each request is generated within these parameters with random ingress and egress nodes. The number of functions in the functionSequence (chosen randomly) are Min($\sqrt{N}$,Random(5,...,15)). Each NFV-enabled node is provided with a capacity support approximately Min($\sqrt{N}$,10) flows on it. Each link is given enough capacity to support an average of five requests (chosen randomly). The whole experiment is repeated 10 times to discard results due to anomalous behavior.

Following parameters are tracked:

- *Path computation time/delay:* This is the time taken by the hybrid Dijkstra's algorithm at the SDN controller to find the candidate paths and choose one out of them for a given request.
- *NFP computation time/delay:* This is the time taken by the heuristic NFP algorithm at the SDN controller to find the optimal allocation for the given request on the chosen path.
- *Feasibility delay:* This is the delay of *backoff* and *denial* caused due to lack of feasible solutions at a given time for the incoming request.
- *Migration time/delay:* This is the time taken by the VNF containers to be migrated to the appropriate NFV-enabled switches.

The results have been shown in Fig. 9. The path computation time (Fig. 9a) seems to be a quadratic function of the number of nodes (N) which is aligned to the expectation since the algorithm looks at approximately N candidate paths of approximately N length. Although, for 1000 nodes and heavy requests, the path computation takes 1.6 h, if we consider only the NFV enabled nodes on a topology, the number are not expected to exceed 500 nodes even for a 10,000-node topology where the time taken is 5.4 min. Also, faster and better routing algorithms can definitely reduce this time overhead.

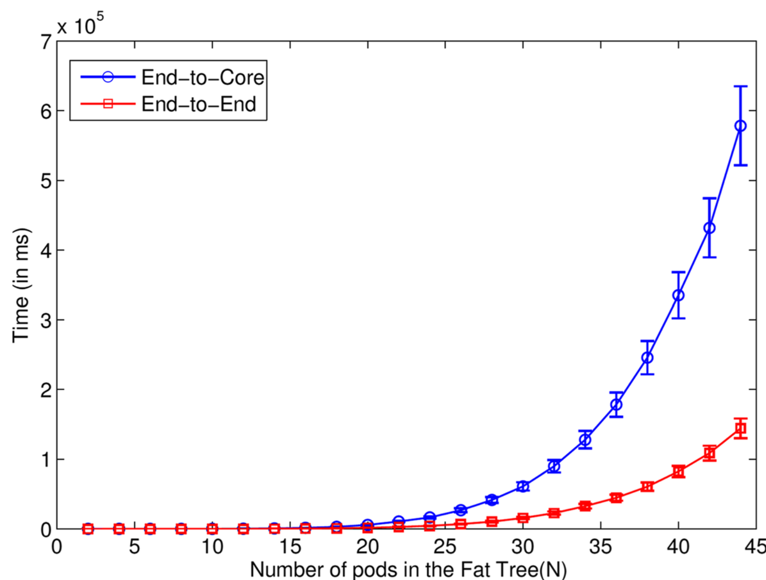

Fig. 8 Performance of DCA on FatTree-based DCN topologies. The performance of DCA in Data Center Networks with FatTree topologies with various number of pods (N) is shown

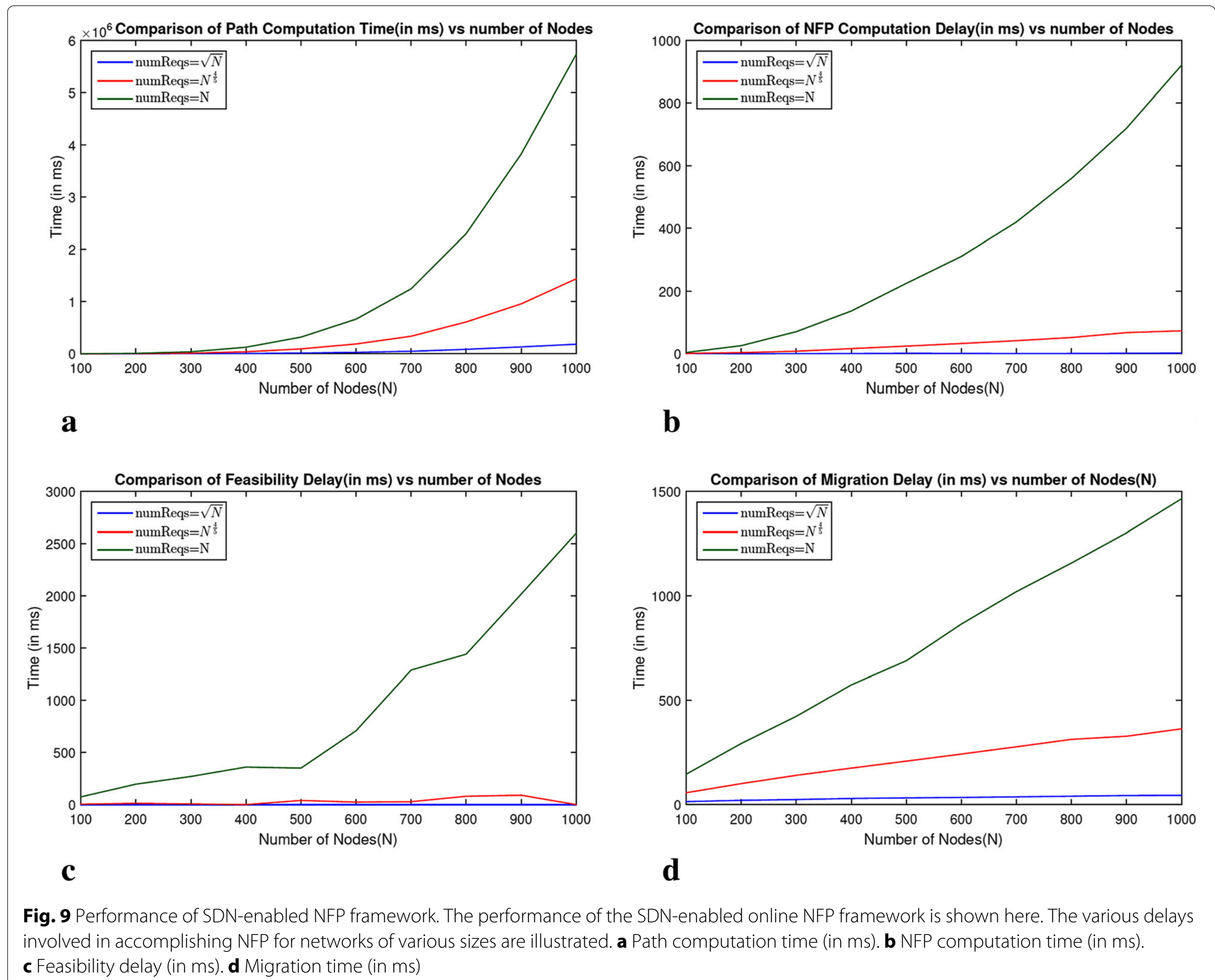

Fig. 9 Performance of SDN-enabled NFP framework. The performance of the SDN-enabled online NFP framework is shown here. The various delays involved in accomplishing NFP for networks of various sizes are illustrated. **a** Path computation time (in ms). **b** NFP computation time (in ms). **c** Feasibility delay (in ms). **d** Migration time (in ms)

The NFP computation time (Fig. 9b) for 1000 nodes and heavy requests is 920 ms at an average and is only a function of the size of the path and the function sequence. Since the path size is constrained to 1000 in the worst case and the function sequence to 15, this result is pretty reasonable to characterize real-world situations. The feasibility delay (Fig. 9c) of 2600 ms for 1000 nodes and heavy requests is actually more than the NFP computation delay which suggests that, either better NFP solutions can be considered until the feasibility delay becomes less than the NFP computation delay, or the *backoff delay* and the limit for *denial delay* can be reduced. The migration delay (Fig. 9d) for 1000 nodes and heavy requests is 1463 ms though real-world values of migration delays will give a better view on this particular factor.

6.2.2 DCN topologies: K-pod Fat Trees

Data center networks are some of the most important places where our proposed protocol can be primarily used. In DCNs, there are two types of flows:

- *End-to-end flows:* These are primarily between leaf nodes (such as for Map-Reduce tasks). The path computation just has to look at all cores and once a core is fixed, the path is automatically defined (direct path).
- *Core-to-end flows:* These are primarily between core nodes and leaf nodes (such as for content delivery).

The topology to be used for experimentation is the fat-tree topology described in [23]. For a K-pod fat tree, K flows are initiated every K^3 ms and K^2 flows every K^3 ms. The *backoff delay* is taken as K ms and the *denial delay* as K^3 ms. Each request is generated as defined above between random core and leaf node or random leaf and leaf node. The number of functions in the function sequence (chosen randomly) Random (1,...,3) for Core to Leaf Node flows and Random(1,...,5) nodes for Leaf to Leaf Node flows. The whole experiment is repeated 10 times to discard data from anomalous behavior.

The results have been shown in Fig. 10. As is evident from the results, the path computation for end-to-end nodes for K^2 flows takes on 70-pod fat trees 28 s, the NFP computation time is 11 s and the migration time is 7 s. Thus, when a new request comes, it takes about 1 min for the flow to shift to the data plane for a 91,875-node topology. Most of the current large-scale data center networks are 48-pod fat-trees (30,528 nodes) for which the combined delay in worst case is just 8 s. This shows that our protocol can easily scale to the largest of data center networks.

7 Conclusions

The network function placement (NFP) problem that requires meeting the service function chaining (SFC) requirements of data flows in software-defined networks is a critical problem in the field of network function virtualization. This paper proposes a solution based on the divide-and-conquer strategy for the NFP problem and shows that it is complete and sound. The solution is applied to both static NFP requests and dynamic NFP requests. It is shown that this model can be customized to become a heuristic algorithm which trades off precision for time complexity using various parameters. It is theoretically shown that there exists a set of parameters for which our algorithm has a complexity that is just a function of the topology and not dependent on the number of requests received. The algorithm was studied for various parameter values to show their influence on the time complexity of the heuristic algorithm. Based on the results, the algorithm scales to large state-of-the-art DCN topologies. In the case of dynamic NFP, existing solutions to optimize individual components of this complex problem are too slow for being used as online NFP solvers. This paper combines the divide-and-conquer approach to NFP with a modified Dijkstra's algorithm to propose an

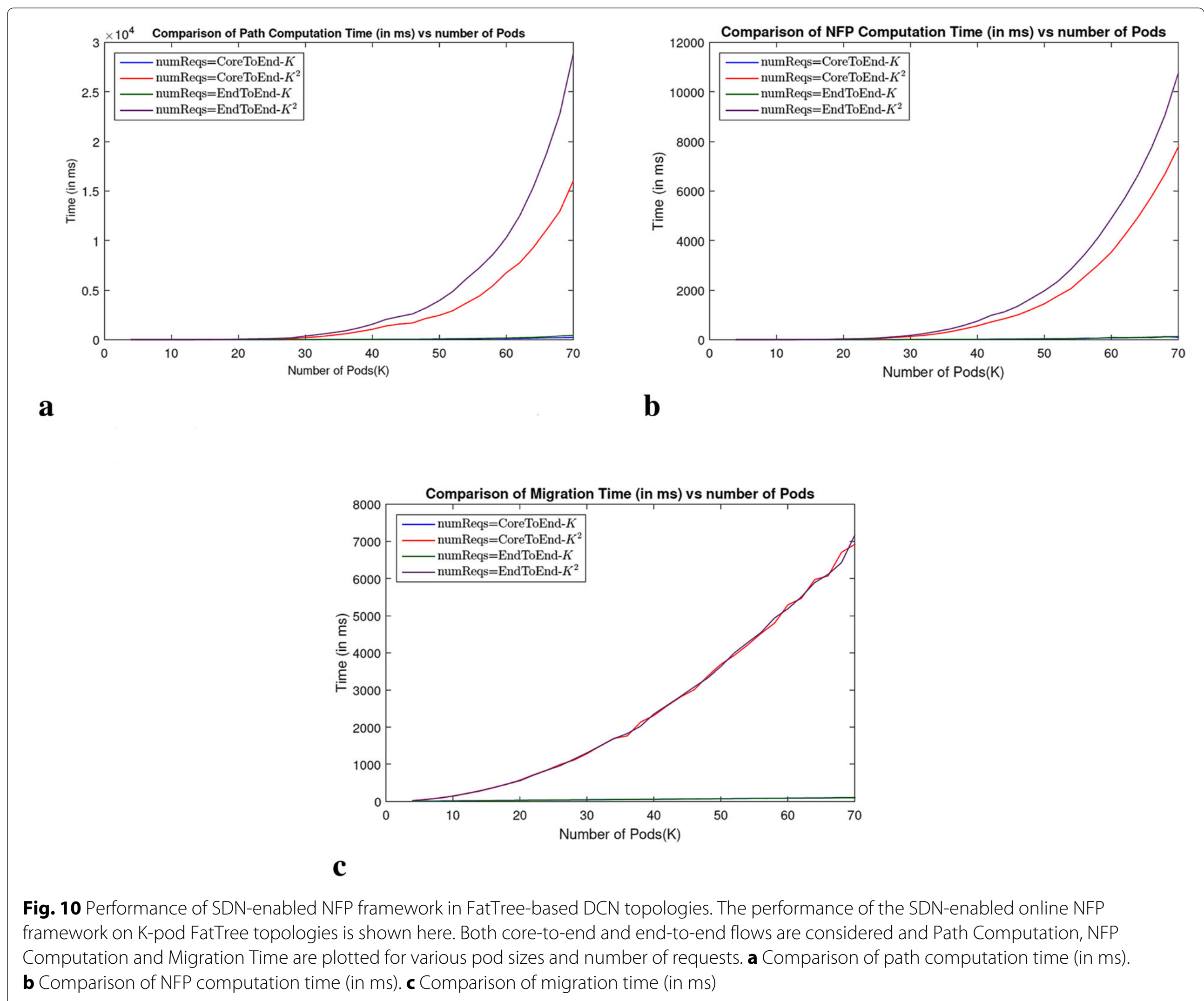

Fig. 10 Performance of SDN-enabled NFP framework in FatTree-based DCN topologies. The performance of the SDN-enabled online NFP framework on K-pod FatTree topologies is shown here. Both core-to-end and end-to-end flows are considered and Path Computation, NFP Computation and Migration Time are plotted for various pod sizes and number of requests. **a** Comparison of path computation time (in ms). **b** Comparison of NFP computation time (in ms). **c** Comparison of migration time (in ms)

online NFP solver. Moreover, a framework based on SDN controllers and the ETSI MANO architecture is presented for deploying flows with corresponding VNF chains. The proposed protocol is simulated on non-symmetric as well as DCN topologies and it is shown that even for DCNs beyond the scale of current use, the algorithm takes time in the order of seconds. With further work, these heuristic components can mature into fast and more optimal algorithms meeting the constraint of solving the problem in real time. There is also a possibility of adding fault tolerance to the proposed protocol to handle failures of links, nodes, or VNFs.

Acknowledgements
The authors would like to acknowledge their fellow researchers at the DonLab Research Group of the Department of Computer Science and Engineering, IIT Madras for their valuable input and feedback.

Funding
There are no external sources of funding for this research work.

Authors' contributions
AG is the main researcher for this work and the first author of this paper. The simulation framework was programmed by AG and used to generate the results. A major portion of the work was completed while AG was a B.Tech-M.Tech Dual Degree student at Indian Institute of Technology Madras. He is currently a Ph.D. scholar at Carnegie Mellon University.
AA is a co-researcher in this work. He has been working on network function placement-related problems, analyzing various aspects of both static and dynamic versions of the problem as well as comparison of centralized and distributed approaches. AA is a Ph.D. scholar at the Indian Institute of Technology Madras. He is responsible for final editing of the manuscript.
KMS provided key input to defining the problem and various approaches to solution as well as analysis and verification of the results. He is currently the head of the Department of Computer Science and Engineering at Indian Institute of Technology Madras. He also leads the DonLab Research Group, which, among other things, is researching network function virtualization. He is an IEEE Fellow, ACM Distinguished Scientist and INAE Fellow.
All authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Received: 30 October 2017 Accepted: 30 July 2018
Published online: 10 August 2018

References

1. T Lin, Z Zhou, M Tornatore, B Mukherjee, Demand-Aware Network Function Placement. J. Light. Technol. **34**(11), 2590–2600 (2016)
2. X Li, C Qian, in *Proceedings of IEEE CCNC*. A survey of network function placement (IEEE, Las Vegas, 2016), pp. 948–953
3. J Gil Herrera, J Felipe Botero, Resource Allocation in NFV: A Comprehensive Survey. IEEE Trans. Netw. Serv. Manag. **13**(3), 518–532 (2016)
4. B Han, V Gopalakrishnan, L Ji, S Lee, Network function virtualization: Challenges and opportunities for innovations. IEEE Commun. Mag. **53**(2), 90–97 (2015)
5. R Bonafiglia, I Cerrato, F Ciaccia, M Nemirovsky, F Risso, in *Fourth European Workshop on Software Defined Networks (EWSDN)*. Assessing the performance of virtualization technologies for NFV: a preliminary benchmarking, (2015), pp. 67–72
6. J Hwang, KK Ramakrishnan, T Wood, NetVM: High Performance and Flexible Networking using Virtualization on Commodity Platforms. IEEE Trans. Netw. Serv. Manag. **12**(1), 34–47 (2015)
7. N McKeown, Software-Defined Networking. INFOCOM Keynote Talk. **17**(2), 30–32 (2009)
8. N McKeown, T Anderson, H Balakrishnan, G Parulkar, L Peterson, J Rexford, S Shenker, J Turner, Openflow: enabling innovation in campus networks. ACM SIGCOMM Comput. Commun. Rev. **38**(2), 69–74 (2008)
9. L Vanbever, J Reich, T Benson, N Foster, J Rexford, in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. Hotswap: correct and efficient controller upgrades for software-defined networks, (2013), pp. 133–138
10. C Bu, X Wang, M Huang, K Li, SDNFV-based dynamic network function deployment: model and mechanism. IEEE Commun. Lett. **22**(1), 93–96 (2018)
11. P Quinn, T Nadeau, *Problem statement for Service Function Chaining*, (2015)
12. S Mehraghdam, M Keller, H Karl, in *Proceedings of IEEE International Conference on Cloud Networking (CloudNet)*. Specifying and Placing Chains of Virtual Network Functions, (2014), pp. 7–13
13. MT Beck, JF Botero, in *Proceedings of IEEE GLOBECOM*. Coordinated allocation of service function chains, (2015), pp. 1–6
14. MT Arashloo, Y Koral, M Greenberg, J Rexford, D Walker, in *Proceedings of the 2016 ACM SIGCOMM Conference*. SNAP: Stateful network-wide abstractions for packet processing (ACM, 2015), pp. 29–43
15. D Eppstein, in *IEEE 35th Symposium on Foundations of Computer Science, Santa Fe*. Finding the k shortest paths, (1994), pp. 154–165
16. JY Yen, Finding the K-shortest loopless paths in a network. Manag. Sci. **17**, 712–716 (1971)
17. JY Yen. vol. 20, in *41st Meeting of Operations Research Society of America*. Another algorithm for finding the K-shortest loopless network paths, (1972)
18. A Anbiah, KM Sivalingam, in *42nd Annual IEEE Conference on Local Computer Networks (LCN)*. Funplace: A Protocol for Network Function Placement (IEEE, Singapore, 2017)
19. A Gadre, A Anbiah, KM Sivalingam, in *Proceedings of European Conference on Networks and Communications (EuCNC)*. A customizable agile approach to Network Function Placement, (2017), pp. 1–6
20. D Cho, J Taheri, AY Zomaya, P Bouvry, in *IEEE 10th International Conference on Cloud Computing (CLOUD)*. Real-time virtual network function (vnf) migration toward low network latency in cloud environments, (2017), pp. 798–801
21. B Heller, R Sherwood, N McKeown, in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks, ACM*. The controller placement problem, (2012), pp. 7–12
22. A Dixit, F Hao, S Mukherjee, T Lakshman, RR Kompella, in *IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. Elasticon; an elastic distributed sdn controller, (2014), pp. 17–27
23. M Al-Fares, A Loukissas, A Vahdat, in *Proceedings of ACM SIGCOMM*. A scalable, commodity data center network architecture, (2008), pp. 63–74

Submit your manuscript to a SpringerOpen journal and benefit from:

► Convenient online submission
► Rigorous peer review
► Open access: articles freely available online
► High visibility within the field
► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com