A Customizable Agile Approach to Network Function Placement

Akshay Gadre, Anix Anbiah and Krishna M. Sivalingam
Dept. of CSE, Indian Institute of Technology Madras, Chennai, India
Emails: {*agadre@cse.iitm.ac.in, anix@cse.iitm.ac.in, skrishnam@iitm.ac.in, krishna.sivalingam@gmail.com*}

***Abstract*—This paper deals with servicing Virtual Network Function (VNF) chaining requests in a Network Function Virtualization (NFV) based system. This is also referred to as the Network Function Placement (NFP) problem. Existing solutions to this problem are slow and not suitable for deployment as dynamic NFP-solvers in networks. In this paper, we first propose an NFP solution that uses a divide-and-conquer approach, with a complexity similar to that of existing solutions. We show that the solution is complete and sound. Next, we propose ways to customize our solution to obtain an agile version that trades off precision for significantly lower time-complexity. The proposed algorithm is analyzed for various system parameters and it is shown to be scalable for large data center network (DCN) topologies.**

I. Introduction

In order to implement networking services such as Firewalling or HTTP Proxy (commonly known as Layer4-Layer7 services), the network layer entity is forced to "peek" into parts of the data packets that is unnecessary for it to perform its primary function. These services are implemented using dedicated hardware, often called as *middleboxes*.

Network Function Virtualization (NFV) refers to an architecture where the middleboxes are replaced with *Virtualized Network Functions* (VNFs) [1]. The idea is to spawn virtual machines or processes when and where needed to perform the middlebox's network functions. This has the advantage of avoiding investments in specialized hardware, while allowing elastic scaling of the capacity of the functions. In the meantime, switches and routers can use their spare computational resources to implement the VNFs in the data plane of the network.

This architecture gives rise to many interesting problems. Primarily, these problems are about the number of instances of the various network functions required and their placement in the network [2]. Each packet flow in the network requires a sequence of functions that must process the flow according to some defined network policy. The functions have a partial order in which they must be applied to the flow. Given that each instance of a function can be applied to several flows, we need to solve a complex optimization problem that takes into account several factors in determining the placement of VNFs.

Figure 1 presents a sample network depicting two flows through the switches and the placement of network functions to service the flows. For instance, Flow 1 utilizes three network functions at nodes 1, 2 and 4.

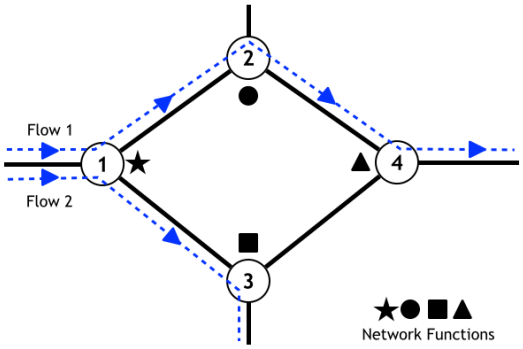


Fig. 1: A sample configuration of flows through a network.

The placement problem is known to be NP-Complete [3] and has primarily been solved by treating it as a constraint satisfaction problem. There are two components to this problem: (i) routing the flows subject to traffic engineering optimization; and (ii) placing the network functions subject to the constraints of resource availability on the switches.

The work published on this problem can be classified into three categories: (i) those that consider network functions implemented on physical appliances and describe an overlay network comprising encapsulated tunnels between instances of these functions; (ii) those that consider virtual network functions but take the approach of routing the flows to, from and between the VNFs; and (iii) those that treat VNFs as being capable of getting broken into relatively simple functions that can be distributed and implemented among the switches themselves by using their spare computational and storage capacity. We are primarily interested in work that belongs to the third category above.

The work presented in [4] attempts to solve path computation and NFP 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 solver to find solutions.

Similarly, the work discussed in [5] attempts a co-ordinated solution that solves both the composition of the network

functions in a chain as well embedding them in the substrate network. They employ a greedy approach with backtracking to keep the run time within acceptable limits. The approach described in [3] also employs an integrated design for NFP and end-to-end demand realization (routing). They demonstrate that this problem is NP-complete.

In this paper, we assume that the flows have been routed along optimal paths and thus attempt to find a feasible solution for the second component. For the purpose of this paper, we term this component as the Network Function Placement (NFP) problem. We propose an algorithmic approach that is agile and solves the problem faster than currently available solutions. Such an approach is more suitable for a network whose state is prone to changes and where the flows are frequently being initiated and terminated. Our proposed approach assumes that the path computation of flows is done independently and optimally. The placement of the network functions is handled separately using a divide-and-conquer approach with instances of smaller sub-problems that are solved in parallel.

II. Problem Definition

The Network Function Placement (NFP) problem has been widely studied as described in the previous sections. In this paper, we separate the placement of network functions from path computation. We assume that the flows are routed along the optimal paths from ingress to egress, using a suitable cost metric. Given the optimal paths, we address the problem of having to place the network functions along these paths, subject to the various constraints such as the chain of functions required by each flow and the resources available to deploy the functions. We also consider the chain of functions to be of *linear order*. That is, we do not consider functions that split or merge the flows that they process.

Innovation in network switches has improved their ability to host stateless and stateful network functions significantly. They have spare memory and computational resources which enable them to host these functions. We will restrict our discussion to network functions that can be deployed on these switches.

Modern state-of-the-art switches usually have 3 types of resources: network, computational and storage. To simplify our discussion, we will represent the capacity of a switch to host network functions as a single unified resource. This can be extended the model to different types of resources.

Problem Statement: Given the flows through the network and their requirements for network functions, the objective to place them in the switches such that the maximum resource cost on any single switch is minimized.

The resource cost for the virtual network functions has two components:

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

Thus, this work's objective is to find the placement of network functions among the switches in a network given a set of flows, with each flow taking an optimal 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 is minimized.

III. Proposed Solutions

This section presents the system definitions, the proposed Divide-and-Conquer algorithm and a customizable agile version of this algorithm.

A. System Description

The following definitions are used in the rest of the paper.

A virtual network function (VNF) can have many characteristics such as the input data rate that it needs to handle, its output data rate and the distribution of the output over a set of output interfaces, based on the forwarding rules. In this paper, we only consider linear network functions that do not affect the data rate of the flows being processed by them. Hence, a VNF is defined by a 3-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). The number of different network functions is denoted by M.

A *tenant request* is defined by a source node, a destination node, the bandwidth of the flow (BW_k), the path that the flow will take and a sequence of network functions that need to service this flow before egress. The number of incoming tenant requests is denoted by 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. In our case, we assume that such requests have been pre-processed (one such heuristic is presented in [4]) 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. The number of network elements in the network is denoted by N.

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, due to the limited availability of resources on the node. The output of our algorithm is a set of *allocations* that satisfy all the tenant requests provided as input.

An *NFPSystem* is an instance of our algorithm that solves the NFP problem or a sub-problem. By our 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.

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             end
16         else
17             calculateCostAndUpdateMin(out+a);
18         end
19     else
20         if a.reqList.size>1 then
21             removeLargestDatarateFlow(a);
22             goto 5;
23         else
24             continue ;
25         end
26     end
27 end
28 return minCostSetAllocations;
```

B. 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 presented in Algorithm 1. The first instance of *NFPSystem* is created with the complete set of functions, 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. This is required to guarantee the soundness property which will be discussed in more detail in Section IV-A.

We first consider only maximal allocations, i.e. when a function is allocated on a node, it services all requests that pass through the node requesting that function. For each allocation, we first verify if it is feasible. Here, we check whether the cost of the servicing the given requests is less than the resource capacity of the node in the current system. 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 we remove the tenant request that has the maximum dynamic service cost. After that, we again start with the verification of the resultant *allocation*. If the allocation is left with no requests to serve, then it is discarded making the *NFPSystem* infeasible for any allocation of the function on the specified node. Hence, no update of the optimum allocation is done.

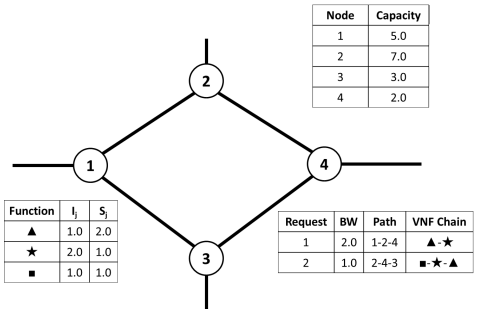
If an allocation is feasible, we first account for the resource capacity needed for that allocation by updating the available capacity on the corresponding switch. Then we use a divide-and-conquer method (line number 6 in Algorithm 1) to distribute the requests that are being serviced by this allocation of the given function. For example, let us say that we have a request with path A-...-B-C-D-...-E and a VNF chaining request F1-...-F2-F3-F4-...-F5 and the current allocation places F3 at C for this request. Then, we create two new requests for this as <A-...-B-C,F1-...-F2> and <C-D-...-E,F4-...-F5>.

A sample run of the divide-and-conquer algorithm is demonstrated in Figure 2. Consider the 4-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, we allocate ★ on node 2. This results in the node 2 capacity getting updated and a new set of requests getting generated. This is shown in Figure 2b. 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 Figure 2e.

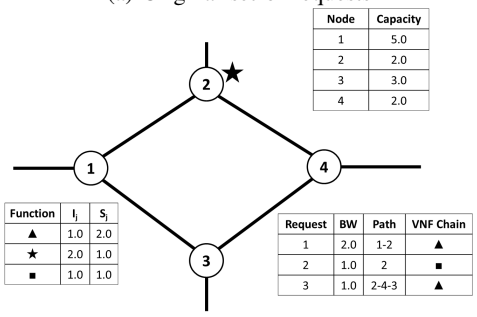
Once a feasible allocation has been identified, we add this allocation to the current list of incoming allocations to be passed to a sub-system to handle the new set of requests. We create a parallel *NFPSystem* in a separate thread which will solve this new sub-system and return the set of optimal allocations satisfying the new set.

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. The process is repeated again after updating the capacity of the switch corresponding to the allocation. If the allocation has only one request, then there are no further possibilities for reducing the allocation. The allocation is now deemed infeasible and does not yield a candidate for optimum allocation. In this case, we continue without any update of the optimum allocation.

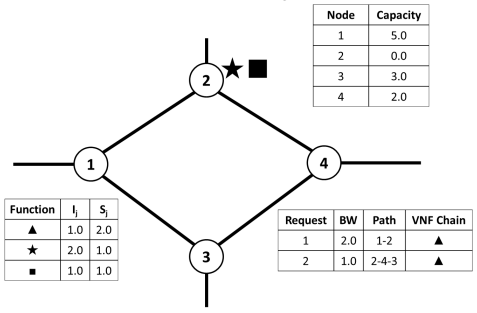
Note that the function chains and the network paths get smaller for the newly created instances of the *NFPSystem*. Eventually, the new set of requests will contain requests with a trivial path that has only a single node. At this point, a simple feasibility check is made to see if that node can support 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.



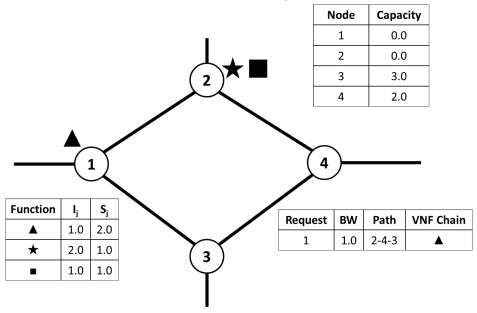
(a) Original set of requests



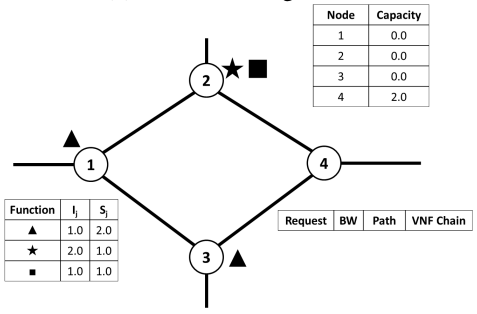
(b) After allocating ★ on 2



(c) After allocating ■ on 2



(d) After allocating ▲ on 1



(e) After allocating ▲ on 3

Fig. 2: Sample run of Divide-and-Conquer Algorithm.

Given an allocation, 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 considers all feasible solutions and is therefore as complex as the current solutions based on MIQCP and has a time complexity of $\mathcal{O}(e^{poly\log(poly)})$, as shown in Section IV-B. This motivates the need for a agile solution that can act as a dynamic NFP solver in virtualized networks.

C. Fast Heuristic Algorithm (DCA-H)

Algorithm 1 searches for optimal allocation of all tenant requests exhaustively. There are many ways our algorithm can be made more agile by sacrificing the completeness and the soundness of the algorithm. We call this heuristic version of our algorithm as DCA-H.

The first and most effective way is to bound the branching at every NFPSystem instance. This can be enforced by only looking at the top T allocations possible, at every step for the optimal cost solution. Once the branching is curtailed, the algorithm becomes significantly faster. This is done in line 3 of Algorithm 1.

It is important to sort 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 of each allocation in the current situation. Note that each of these sorting orders can be countered with corresponding counter-examples, but the choice depends on the type of requests expected. The administrator of a network can use a custom sorting method that works reasonably well for that network. We thus add a function *sortAllocations();* before line 3 of Algorithm 1.

Another heuristic is to reduce the backtracking done when the subsystem created by a verified allocation is infeasible to solve. This can be accomplished by making lines 10–13 optional in Algorithm 1. 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 allocations. This can be done by making lines 20–23 optional in Algorithm 1. This will not logically 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.

IV. Algorithm Analysis

In this section, we show that Algorithm 1 (DCA) is complete and sound. Then, we discuss the time complexity of the DCA-H algorithm to justify that it is polynomial when all the subsystems of a given system are run in parallel. We then analyze our solution with various parameters and finally

show that it scales to even large scale state-of-the-art DCN topologies.

A. 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. When our algorithm is extended to even allow arbitrary sequence of functions, it will require exponential time for this property to remain true. Hence, current solutions such as [4] try to limit the combinatorial increase using heuristic algorithms. Thus, we can argue that DCA is complete.

Soundness of an NFP algorithm is the property of all the output solutions 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, we know that at each stage it has found the minimum cost subsystem after each allocation. Also, at each stage we find the allocation which gives us the minimum cost. After each system and its subsystems return the possible allocations, we return the minimum cost among all possibilities to the parent system. This guarantees that given a feasible *NFPSystem*, our algorithm finds the most optimum cost solution for all subsystems that are visited. The guarantee of visiting all possible nodes is the other aspect of the soundness property.

If the algorithm does not give a solution that means no system where at least one function was allocated to some node led to any *NFPSystem* subsystem which was feasible. This basically means that the current system is infeasible. This is because servicing of the VNF chains requires at least one of the network functions required by one of the requests on one of the nodes to lead to a feasible *NFPSystem*. Using this argument recursively and the optimality argument, we claim that DCA is sound.

B. Theoretical Complexity Analysis

To analyze the complexity of the algorithm, we determine the upper bound on the time complexity of a single NFP system. Then, we determine the maximum depth that can be reached by each of parallel chains created. The overall time complexity of the parallel algorithm will be the value of one instance of NFP system multiplied by the number of branches that can be taken to the power of the maximum depth of each branch.

As defined earlier, N denotes the number of network elements; M the number of network functions, and K, the number of tenant requests.

DCA Complexity: We first start an *NFPSystem* by finding all possible allocations. The total number of such allocations is $\mathcal{O}(NM)$. Then for each allocation we first verify the allocation, which takes $\mathcal{O}(1)$. Then, we update the network with complexity $\mathcal{O}(1)$ since it involves just the update of the capacity of a network node. We then execute the divide-and-conquer algorithm 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. However, the number of possible allocations definitely decreases since it is a subset of the possible allocations already done. These steps can happen at most the number of times this loop executes, which is $\mathcal{O}(K)$, the maximum size of an allocation. Thus the complexity of each run of NFPSystem is $\mathcal{O}(NM + K)$.

We next analyze the depth that a tree can grow to, 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 complexity in time domain is $\mathcal{O}((NM + K) \cdot (NM^{(NMK)}))$ which is $\mathcal{O}(e^{poly \log (poly)})$. This complexity will decrease when run in parallel. Thus, Algorithm DCA is exponential in worst-case time complexity.

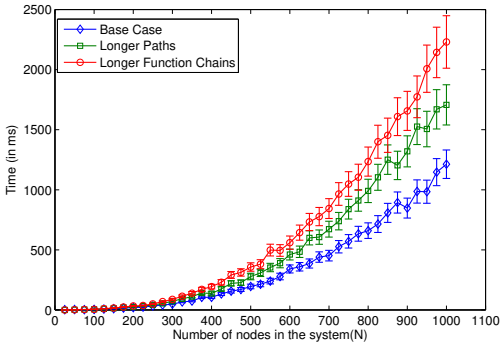
Heuristic Complexity: In the heuristic, we sort the allocations, which is done in $\mathcal{O}(NM \log (NM))$ time. Thus, the worse-case complexity without any heuristic optimizations becomes $\mathcal{O}((NM \log (NM) + K) \cdot (NM^{(NMK)}))$. Looking at only the top T allocations will bound the number of branches at each step by T (a constant). This will change the multiplicative factor to $\mathcal{O}(\ldots \cdot (T^{(NMK)}))$. This leads to the overall complexity of $\mathcal{O}(e^{poly})$, 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 was to reduce the backtracking that happens when we find that a verified allocation leads to an infeasible subsystem. This reduces the first term of the complexity to $\mathcal{O}((NM \log (NM)) \cdot \ldots)$. The change of this term appears to yield negligible improvement. However, the actual time taken by the algorithm is significantly lower. This is because, for a given network and set of network functions N and M are fixed while K is dynamic and having the complexity bound by a static factor is much better. 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 which is very important to give certain guarantees in a network system.

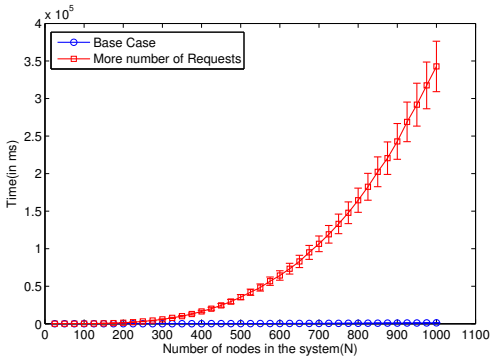
The last optimization of reducing backtracking when an allocation is not verified does not give any benefit in bounding the complexity since it does not actually reduce the number of times the NFP system is run. Hence, it is recommended to keep it in place to reduce the loss in precision.

V. Empirical Time Complexity

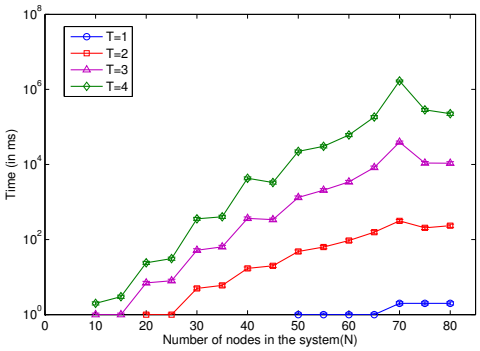
In this section, we compare the time taken (in milliseconds) by the proposed heuristic algorithm, varying various parameters.



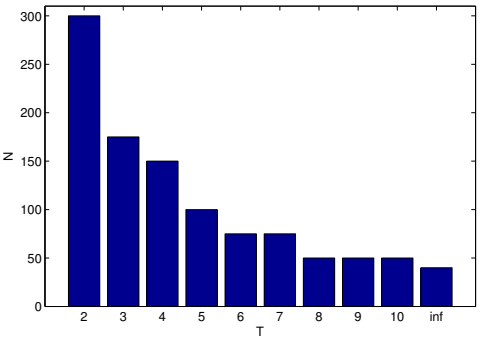
(a) Comparison with longer paths and function sequences



(b) Comparison with more number of requests



(c) Comparison with various values of T



(d) Maximum practically feasible N for various values of T

Fig. 3: Comparison of time taken with various parameters varied

Base Case Parameters: In the base case, given a set of nodes N, we generate $\sqrt{N}$ requests. Each node has a resource capacity of $N^{0.8}$ (other values are possible too). 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. Each of the nodes on the path is randomly chosen from 1 to N. The number of functions requested in 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. This experiment is run from N = 1 to 1000. For each N, the experiment is run 25 times and the 90% confidence interval is found.

Then, we run the same experiment for changes in various parameters and compare it with the base case to find the contribution to the complexity for each of those parameters.

A. *Varying different parameters*

First, we compare our base case with requests having longer paths and longer function sequences. For longer paths, we vary the path length from $\sqrt{N}$ to $N^{0.6}$ instead, while for long function sequences we vary the function sequence length from 1 to $\sqrt[3]{N}$. The effect of increasing the path size and having longer function chains per request is shown in Figure 3a. It is seen 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. This is because the amount of overlap increases with the former and there are chances that backtracking will be required.

Next, we compare our base case with a system having N requests instead of $\sqrt{N}$. Increasing the number of requests increases computation time exponentially, as seen in Figure 3b. This is because increase in the number of requests increases the number of possible allocations exponentially. This reaffirms our complexity analysis which shows that increase in the number of requests will have a more adverse effect than increasing the function chain or path length.

Then, we vary T 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 Figure 3c, where time is plotted on a log scale. The time taken for even 80 node graphs increases by an order of two.

However, we lose precision for time complexity when we choose a smaller value of T. Hence, we need to verify whether the performance with smaller values of T is comparable to the larger values of T. We ran 3 topologies of 100, 200 and 500 nodes each with the base case criteria and values of T as 1, 4 and 10. For 100 nodes, the time taken by T=1 was 46 ms, T=4 was 5.45×10^5 ms and T=10 did not finish in 15+ hours. The output minimum cost values of 17.6 resource units matched for both. For N=200 and 500, the simulation for T=4 and T=10 took 15+ hours 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.

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, we run the base case with values of T varying from 2 to 10 and and values of N increasing in jumps of 25 each. We assume 30 minutes to be the cut-off time for the simulation beyond which the solution becomes impractical. We repeat the same experiment for DCA as well with N increasing in jumps of 5 each.

The results are shown in Figure 3d. It is seen that to make this algorithm useful for practical purpose for values of T beyond 2, significantly 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+ hours for $N = 45$.

B. Scalability in DCNs

Since NFV is well suited for data center networks (DCN), we consider a DCN scenario. Here, the flows are primarily between leaf nodes (e.g., Hadoop Map-Reduce Tasks) or between core nodes and leaf nodes (e.g., Content Delivery Networks). Thus, we will restrict our discussion to these types of flows. The topology we will be considering is the fat-tree topology described in [6].

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 were $\sqrt{\mathcal{N}}$, 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.

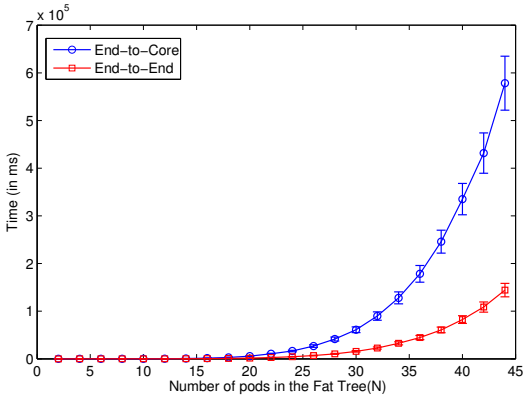


Fig. 4: Performance in a FatTree based DCN

The results obtained are described in Figure 4. Even for large 48-pod FatTrees containing 30,528 nodes, the time taken for end-to-end requests was around 10 minutes. For core-to-end requests, the time taken was only 2.5 minutes, which is within acceptable range.

VI. Conclusions

Virtual network function chaining implemented in software-defined networks, remains one of the critical problems in the field of NFV. This paper proposes a model equivalent to the MIQCP for the NFP problem and shows that it is complete and sound. We further show that, this model can be customized to become a heuristic algorithm which trades off precision for time complexity using various parameters. We theoretically prove that there exists a set of parameters for which our algorithm has a complexity which is just a function of the topology and not dependent on the number of requests coming in. We have analyzed the algorithm for various parameter values to show their influence on the time complexity of the heuristic algorithm. We further show that our algorithm scales to large state-of-the-art DCN topologies with time in order of few minutes.

References

[1] ETSI, "Network Functions Virtualisation," http://www.etsi.org/technologies-clusters/technologies/nfv, oct 2016.

[2] X. Li and C. Qian, "A survey of network function placement," in *Proc. of IEEE CCNC*, Las Vegas, NV, 2016, pp. 948–953.

[3] T. Lin, Z. Zhou, M. Tornatore, and B. Mukherjee, "Demand-Aware Network Function Placement," *Journal of Lightwave Technology*, vol. 34, no. 11, pp. 2590–2600, 2016.

[4] S. Mehraghdam, M. Keller, and H. Karl, "Specifying and Placing Chains of Virtual Network Functions," in *Proc. of IEEE Intl. Conf. on Cloud Networking (CloudNet)*, 2014, pp. 7–13.

[5] M. T. Beck and J. F. Botero, "Coordinated allocation of service function chains," in *Proc. IEEE GLOBECOM*, 2015, pp. 1–6.

[6] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. of ACM SIGCOMM*, 2008, pp. 63–74.