Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Media Informatics

Németh Balázs

Flexible mapping of virtual services to computing and network resources

KONZULENS

Dr. Sonkoly Balázs,
(BME – TMIT)

BUDAPEST, 2014

Hungarian abstract

A mai hozzáférési, aggregációs és szolgáltatói maghálózatok többnyire merev vezérlési síkkal rendelkeznek, ami nem teszi lehetővé a gyors innovációt és a változó igények követését. Az eddigi protokoll foltozás és túlméretezés helyett egy újabb megközelítésre van szükségünk a hálózatok kapacitásának és kihasználásának a növeléséhez. További cél összetett szolgáltatások gyors és rugalmas létrehozása és üzembe helyezése hatalmas szolgáltatói vagy céges hálózatokon.

Ezekre a problémákra kínálnak megoldást a ma is aktív kutatás alatt lévő Software Defined Networking (SDN) és Network Function Virtualization (NFV) eszközök és irányelvek. Az SDN célja a hálózati eszközök szoftveres programozhatóságának biztosítása, mellyel a hálózat viselkedése dinamikusan irányítható. Az NFV célkitűzése pedig a hagyományosan speciális hardver eszközökben megvalósított hálózati funkciók szoftver komponensekben való implementálása, aminek eredményeként a virtuális funkciók általános célú hardveren, az aktuális forgalmi viszonyok alapján kiválasztott helyen futtathatók. Ezen komponensek láncolásával összetett szolgáltatások kialakítására nyílik lehetőség.

A legfrissebb SDN és NFV eredményekre építve, több aktív kutatási projekt tűzte ki célul hálózatok és szolgáltatások teljes virtualizálását egy erre a célra tervezett hálózati architektúra felett. Ezen architektúra segítségével magas absztrakciós szinten írhatjuk le a hálózati működést, melyet a szoftveresen megvalósított szolgáltatás komponensek rugalmasságának köszönhetően dinamikusan tudunk leképezni a rendelkezésre álló erőforrásokra, ezzel biztosítva a hálózat folyamatos optimális működését.

A vázolt rendszer egyik kulcs feladata, hogy az absztrakt leírást alkotó szolgáltatás láncoknak megtalálja valamilyen szempontból a lehető legjobb leképzését a hálózat fizikai erőforrásaira, a láncokra vonatkozó követelmények teljesítése mellett. Dolgozatom célja ezen algoritmuselméleti szempontból meglehetősen bonyolult feladatra megoldást adni, melynek eredményeként a különböző típusú számítási és hálózati erőforrások működése együtt optimalizálható. A probléma bonyolultságából adódóan reális célom az absztrakt leíráson és a hálózati gráfon futó, heurisztikus, közelítő megoldások kidolgozása, melyek futási ideje a legnagyobb hálózatok esetén is másodperces nagyságrendű.

Munkám során megvizsgáltam a rokon problémákra javasolt megoldásokat, megismertem a területen lévő eredményeket és a használatos eszközöket. Ezeket felhasználva tereztem egy saját, hálózati gráfokon futó, polinomiális idejű, heurisztikus, számos paraméterrel hangolható algoritmust. Továbbá definiáltam egy metrikát, mellyel a hálózat erőforrásainak kihasználtságát számszerűsíthetjük, ezzel értékelve, hogy a hálózat mennyire leterhelt, mennyire preferált az adott állapota. A hálózat preferencia értéke arra is használható, hogy két különböző, követelményeknek megfelelő leképzés közül választani tudjunk az hálózat egészére vonatkozó erőforrás kihasználtság alapján. Dolgozatomban ismertetem az algoritmusom tervezésének lépéseit, magát az algoritmus működését, illetve implementálását és tesztelését. Zárásul pedig felvázolom kutatási munkám további lépéseit és terveimet.

English abstract

Today's access, aggregation and core networks mostly have rigid control plane, which does not enable fast innovation and to keep up with the changing requirements. Instead of the overprovisioning and make, do and mend around networking protocols, we need a new approach for increasing network capacity and improving network resource utilization. Furthermore, fast and flexible creation of composite services on enormous carrier or enterprise networks would be a decent goal.

For these problems, solutions are offered by the up-to-date research results of Software Defined Networking (SDN) and Network Function Virtualization (NFV) tools and principles. The goal of SDN is to support the programmability of network devices with arbitrary software, enabling the dynamic control of network behavior. The objective of NFV is the implementation of originally application specific hardware implemented network functions in software components, which could be run on general purpose hardware, thus enabling us to run the virtual functions anywhere in the network according to the actual traffic conditions. Chaining these software components opens the door to flexible creation of complex services.

Several research projects aim at realizing full network and service virtualization using a novel architecture designed for this purpose and building on the newest findings of SDN and NFV. This architecture enables the description of network behavior on a high abstraction level, which can be mapped dynamically to the available network resources thanks to the flexibility of service components implemented in software. The dynamic mapping ensures the continuous optimal operation of the network.

One of the key tasks of the mentioned architecture is to find in some sense the best mapping of the abstract description to physical network resources, satisfying the given requirements to the service chains. The goal of my paper is to give a solution to this algorithmically difficult problem, so that the operation of various computing and network resources could be optimized jointly. Based on the difficulty of the problem, my realistic purpose is to design a heuristic, approximate algorithm operating on the abstract description and the network graph, with running time of the order of a few seconds even on the largest networks.

During my research I examined the suggested solutions to analogous problems and I got acquainted with the applied tools and findings of this field. Using this knowledge, I have

designed my own heuristic, approximate, polynomial algorithm, which runs on network graphs and tunable with many parameters. Furthermore, I have defined a metric for measuring the network resource utilization, which can be used to evaluate the state of the network, as a preference value of its actual load distribution. The preference value could also be used to choose between two requirement satisfying mappings, based on the overall resource utilization. In this paper, I write about the steps of designing the algorithm, the operation of the algorithm itself, and its implementation and testing. Finally, I sketch my future research and plans.

1 Introduction

Nowadays, we have increasingly many and strict expectations from communication networks as users or companies. Network and service providers are familiar with this fact, and they will try to satisfy the needs of their customers and charge them for the supported services. But today's network management and service development is not up to the task yet, so technology research is required in this field.

Network flexibility, automatized and optimized operation of the largest networks, fast time-to-market of new services are going to be basic requirements of every provider in the near future. To achieve these, the general information technology approach of leveraging the abstraction level of development could also be applied to networks. Just like it happened in the second half of the last century with software development on single machines, we could achieve the high abstraction level of programming of a whole network as one entity.

Configuring and programming the entire network at the same time raises many issues, so a new network architecture should be developed to support this task. The long-standing method of designing a layered architecture to divide complex problems to smaller tasks, could result in the following three layers: service, optimization and infrastructure.

Service layer provides the ability to handle the network on a high abstraction level, where the user and the provider can focus on the operation itself, and do not have to deal with low level problems. They could describe the network in a natural way, for example think in service boxes as elementary network functions, which can be configured, and connections can be defined between them.

The optimization layer's task is to interpret the abstract definition to a lower level, which is understandable by the networking devices. Doing all this with optimal network resource utilization, operation verifiability, providing flexible management and observation possibilities in a fully automated manner, so the probability of human errors are minimized.

Infrastructure layer provides a standard interface of the general purposed network capability hosting and forwarding devices for the upper layer. This is the bottom layer of the architecture, the most hardware-close level, where the main approach is virtualization for providing simultaneous behavior capability, which gives another dimension of optimization possibilities.

This operation definition and architecture design approach seems very promising. Many research projects and inter-company organizations work to create, standardize, develop, spread and exploit the capabilities of this architecture. The research presented in this paper aims to participate in such development work, specifically in the interpretation and compile task of the middle optimization layer of the envisioned architecture. A possible service requirement description and a corresponding algorithm are proposed as the main output of this research. More specifically, given an abstract service description as input, my algorithm maps the request to the currently available computing and networking resources; in the followings its implementation and evaluation are presented.

The paper is organized as follows: a deeper insight to the motivation and to the basis of the problem is provided in section 2 with the high level definition of the task; in section 3, various related problems and state-of-the-art research is introduced; section 4 presents the design choices and other important statements about the research and the results; section 5 explains the designed approximation algorithm as the proposed solution, in two phases of preprocessing and core functioning in details, and finally some basic test cases are presented; the last section makes conclusions and summarizes the paper.

2 Background and Motivation

The following three subchapters aim at introducing the reader to the principles which enables us to envision the architecture described above, and to call it feasible. Software Defined Networking, Network Function Virtualization and Service Chaining are strongly coupled notions, and the functioning of one of them implies or requires the others to be present.

2.1 Software Defined Networking

Software Defined Networking (SDN) is a novel approach to every computer network related activity. Controlling the network from a centralized software, the cost of setup, maintenance and management can be dramatically decreased. Furthermore, SDN supports the customization and optimization of network behavior.

The main idea of this new approach is the separation of data and control planes in computer networks. Centralizing the network control, the administrators do not have to deal with configuring all the network devices, they can just program the *controller* to gain the freedom of software development on the whole infrastructure as one entity. SDN is the possibility of network programmability.

The principles of SDN are spreading fast, both in industrial and academical environment, thanks to the Open Networking Foundation (ONF) [1]. ONF is an organization founded by many multinational companies in 2011, it has more than 100 company members by now. Since it was established, the organization is dedicated to the promotion and adaptation of SDN. For this goal, they designed the OpenFlow protocol for public use, which realizes an open interface for networking hardware and gives way to fast innovation.

The OpenFlow is the communication protocol between the controller and the network devices. It defines the interface of the switches or routers, so that any general purposed computer could connect to them as their controller [2].

OpenFlow enabled switches must be able to handle flow tables, which is populated by basic, relatively low level packet header matching rules, produced by the OpenFlow controller. If an incoming packet to the switch matches an OpenFlow rule, the corresponding action is executed (forward to a set of switch ports, drop or forward to controller). If multiple rules are matched by a packet, their priority decides which rule should take effect. In case of an empty flow table, or if a packet is not matched by any rule, the packet is forwarded to the controller,

which decides what to do with that, and can install rules to handle similar traffic exclusively on the datapath. Thanks to this behavior, after some time the flow tables can converge to a consistent state on the whole network, handling all traffic on the datapath.

The need for higher abstraction languages has arisen in the past years, by the spreading SDN applications. Programming directly the behavior of the OpenFlow protocol and writing low level flow table entries can be ineffective and sometimes cumbersome. Several research projects aim at developing higher abstraction languages and compilers for SDN networks to enable as abstract development of networking software as we currently have in the case of ordinary software for single computers. One of them is The Frenetic Project [3], which provides theoretical background to its open source languages and compilers.

2.2 Network Function Virtualization

Network Function Virtualization (NFV) offers a new way to design, deploy and manage networking services [4]. The main goal of NFV is to enable us to run networking services in software, that were formerly implemented in hardware, like DHCP, DNS, network address translation (NAT) and firewall services, content caching or even webservers.

The service realization in software component gives the network providers the freedom of replacing the network function to other site of execution according to the traffic circumstances; and creating new instances to scale with load; and not depending on specific vendors, because all the services could be run on general purposed hardware. These possibilities can lead to significant advantage on the market, and enables faster time-to-market, fast innovation, and can eliminate overprovisioning. All of this done by utilizing standard IT virtualization technologies.

Just like in the case of SDN, the usage and spreading of NFV is also encouraged by organizations and company co-operation. European Telecommunication Standards Institute (ETSI) approved the foundation of NFV Industry Specification Group (NFV ISG) [5], with the objective of industry consensus on business and technical requirements for NFV, and to agree on common approaches to meet the specified requirements. NFV ISG publishes NFV use cases, requirements, architectural framework, proof of concept and terminology in their review paper [5]. All of this activity is required for organizations and researchers to achieve wide applicability and effective realization of NFV tools and principles.

One of the NFV research projects is ClickOS [6], just to name one. It provides tiny virtual machines, which realizes some basic network functionality that can be connected to each other to operate as a composite network service. ClickOS instances boot very quickly, consume limited amount of memory and provides fast packet processing, thus enabling to run hundreds of them concurrently on a single commodity server.

2.3 Service Chaining

The idea of service function chaining is strongly rooted in the soil created by the novel technologies of SDN and NFV. Service function chaining is basically the ability to control a network on a very high abstraction level, where we only have to deal with the behavioral description of the network, and not with all the networking devices one by one. As far as the ability of virtual service creation is granted by the technologies of NFV; and SDN ensures the efficient usage of arbitrary network control logic, it is a natural requirement to elevate the abstraction level of the description of network behavior.

To explore and bring forth the possibilities of this new approach of service delivery and operation, Internet Engineering Task Force (IETF) has agreed to create a working group, called Service Function Chaining Working Group (SFC WG). The core use cases and the document of SFC problem statement was created in early 2014. The WG will produce an architecture for SFC, explore what information shall be gathered from a network for appropriate operation, and define connected terms and protocols if necessary [7].

In one of the recent papers of SFC WG, one can read the definitions of service chain and service function. As these notions are very often used in my document, I would like to quote their definitions, published by SFC WG [8].

"**Service Function (SF):** A function that is responsible for specific treatment of received packets. A Service Function can act at various layers of a protocol stack (e.g., at the network layer or other OSI layers). As a logical component, a Service Function can be realized as a virtual element or be embedded in a physical network element. One or multiple Service Functions can be embedded in the same network element. Multiple occurrences of the Service Function can exist in the same administrative domain. "

Definition 1: Service Function according to SFC WG [8].

As the Definition 1 implies, a service function can be pretty many things that operate on network traffic. To name some examples, it can be firewall, Deep Packet Inspection (DPI), load balancer, NAT and many more. The list of service functions is not definite yet, or maybe never will be.

The definition of service function chaining can be seen on Definition 2.

"**Service Function Chain (SFC):** A service function chain defines a set of abstract service functions and ordering constraints that must be applied to packets and/or frames selected as a result of classification. The implied order may not be a linear progression as the architecture allows for SFCs that copy to more than one branch, and also allows for cases where there is flexibility in the order in which service functions need to be applied. The term service chain is often used as shorthand for service function chain. "

Definition 2: Service Function Chaining according to SFC WG [8].

From the Definition 2, I would like to emphasize that, a service chain can have multiple branches. As far as the definition does not imply that, the graph of services could contain loops (but does not prohibit it either), maybe it would be more illustrative to call it service function tree, but let us stick to the term of IETF. The architecture, that the definition mentions, is the one to be defined by the SFC working group, so the definition also refers to the ability of service chain branching, as a requirement for their architecture.

The UNIFY project is independent from the SFC WG, but there are some similarities in their problem statements and goals, so in my opinion, from the perspective of UNIFY, it could be instructive to keep tags on the results, statements and requirements published by SFC working group.

2.4 European research project

A European research project called UNIFY, activates both academia and industry to develop a unified architecture of carrier networks and data centers. The main goal of the project is to create a reference architecture for fully virtualized networks, allowing flexible service creation using service chaining, and automated optimization of network resource utilization. Another important objective is the development of new, virtualization aware management technologies [9].

UNIFY defines three main architecture layers, as presented in Figure 1; it is a possible realization of the envisioned architecture, which was described on a high level in the Introduction. The architecture users can be enterprises, Over-the-top content (OTT) providers or even end users.

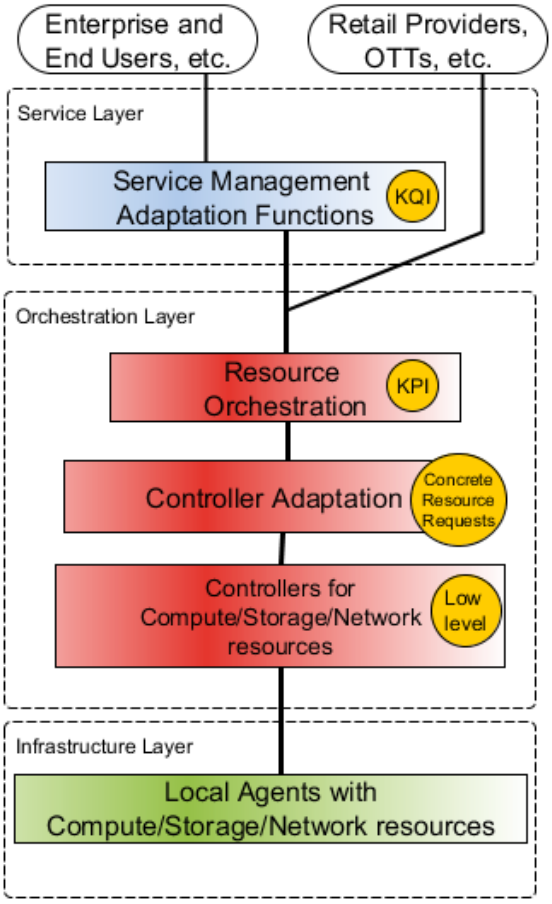

Figure 1: Overview of UNIFY's overarching architecture.

The most difficult problem is raised by the middle (Orchestration) layer, so it should be further divided into sub-layers, as Figure 1 demonstrates. UNIFY defines three level of abstraction of network description. The input of the topmost level comes from the user, who

can define what services and in what order he/she wants to apply on the network traffic. The user can define high level Key Quality Indicators (KQIs), like how many users should the instantiated firewall handle. The input to the first sub-layer of the middle level (called Resource Orchestration) services are decomposed into elementary service functions defining a network function request graph; and KQIs are translated to Key Performance Indicators (KPIs), which defines what technical constraints, like latency, bandwidth and CPU capacity, should be held. This is translated to concrete resource reservations by the Resource Orchestration sub-layer. The next sub-layer distributes the reservations among the controllers of the network infrastructure. This low level network description defines where the network functions should be run, and which paths the traffic should take between them.

As I will demonstrate in the followings, there are many mathematical and technical challenges connected to this part of the UNIFY framework, my paper researches this field of current computer engineering problems, specifically the task of the Resource Orchestration sub-layer.

2.5 Problem statement

During my research, I addressed the problem of the translation between the two bottom abstraction levels, which is the task of the orchestration and optimization algorithms. In this section I only highlight the problem, and do not go into details.

The input from the upper abstraction level is a graph of elementary virtual network functions (VNFs) with their types and resource requirements. There are Service Attachment Points (SAPs), on the border of the request graph[1], defining which network entities (server, gateway, user terminal) should the service graph operate between. End-to-end (SAP-to-SAP) or inter-VNF KPIs (latency, bandwidth, etc.) can also be given on specific paths of the service graph.

We could consider the service graph as a generalization of the IETF definition of service function chain on Definition 2, with the extension of loops. The meaning of loops in the service graphs could be questioned[2], but I do not want to make the constraint of looplessness, because the list of network functions is not clear in the state-of-the-art, and I think loops in the request graph could easily gain sense by inventing new VNFs.

[1] The terms *request graph* and *service graph* are used interchangeably, both of them mean the graph of VNFs.
[2] One could ask: Does it make any sense? Doesn't it make the request graph ambiguous?

The substrate network consists of switches and general purposed nodes, which could also be capable of forwarding. The nodes can run a specific set of VNF types, in as many instances as they can satisfy with the available resources. The nodes and switches are connected with links, which have limited bandwidth capacities and accumulate latency, just like networking devices. SAPs are connected to the network.

The mapping procedure is demonstrated in Figure 2, SAPs are represented by ovals, VNFs are rectangles.

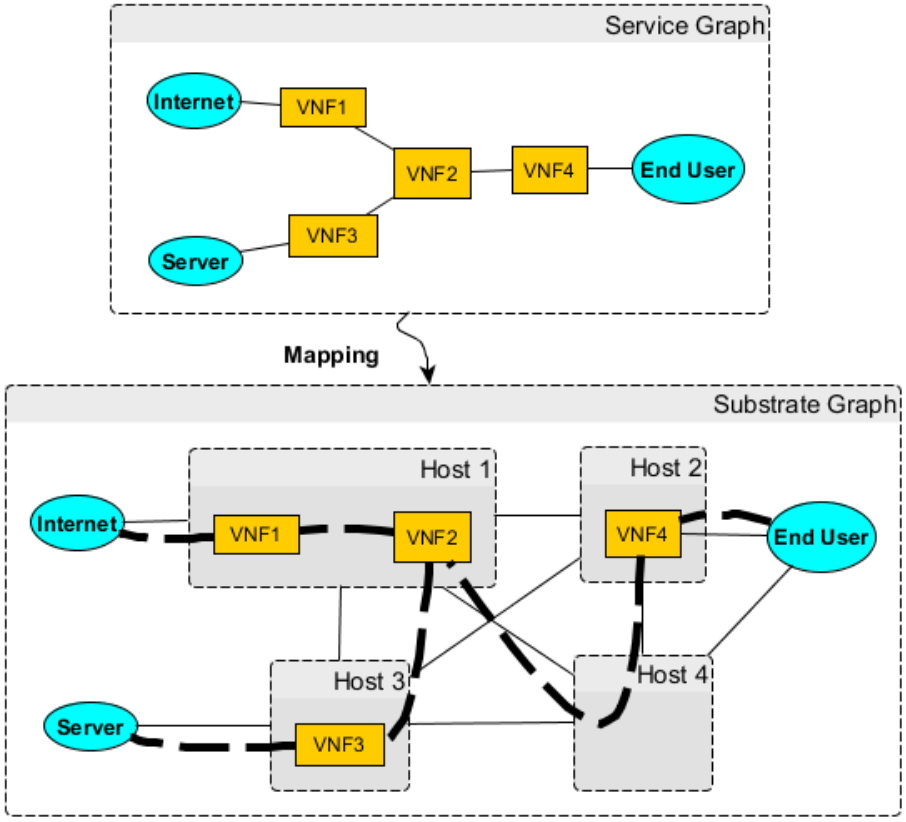

Figure 2: An example mapping of service graph to substrate graph

The first part of the output should be a mapping of all VNFs to one specific substrate node each, which can satisfy their resource request (one node can host multiple VNFs). Secondly, the output should contain the mapping of all request links to paths of the substrate network. All the given KPIs must be satisfied strictly. The mapping must be transparent to the upper layer, the VNFs must be applied in the given order.

The mapping shall be optimized in a sense, that as many service requests should be hosted on the same substrate network as possible.

I place out of the scope of research the problem of traffic steering. Traffic steering is about how and where the substrate network should classify the traffic and determine that which sequence of VNFs shall be applied to a specific subset of the traffic. This problem is orthogonal, and can be handled separately.

As I will explain it later, there are many approaches to similar problems with the utilization of linear programming, but today, the IT world is short on solutions to the problem of VNF mapping utilizing other approaches, dispensing their advantages. So I place out of scope the linear programming, as a general method of optimization problem solution.

3 Related Problems

Due to the recent results of research projects operating on the fields of NFV and SDN, there are many findings to consider if one would like to solve the problem defined in the previous chapter. The papers of these state-of-the-art results are necessary to get familiar with, because these could be a very important source of ideas, or could even already provide solution to a generalization of the problem or the problem itself. Thanks to the much effort invested in this field by the scientists of the world, there are abundance of publications and research result available. Only the most relevant ones are presented here, further important papers can be found in [3] [10] [11] [12] [13] [14] [15].

In this chapter I am going to present related problems, occasionally from other field of informatics, which gave me guidelines, inspiration and ideas to plan my own mapping algorithm. The main terms of this chapter are Virtual Network Embedding, Virtual Datacenter mapping, graph pattern matching, graph partitioning.

3.1 Virtual Network Embedding

The demand of hosting multiple logical topologies on one physical network has arisen in the past few years, first in datacenters, nowadays in various kinds of networks. In this case we want the substrate network to provide resource for the requirements of the virtual topology consisting of virtual nodes and virtual links. Finding an appropriate mapping, a solution for the problem of Virtual Network Embedding (VNE), is an NP-hard problem [16].

3.1.1 A useful objective function

In the paper of Chowdhury *et al* [16] the requirements are CPU capacity in the case of virtual nodes and bandwidth capacity in the case of virtual links. The VNE problem is very similar to the problem proposed in the previous chapter. In both problems there is a graph of logical entities with computing capacity as demands, and logical links connecting them with required minimal bandwidth on the paths connecting the mapped nodes. Chowdhury et al gave two MIP (mixed integer program) based solutions, which method I placed out of scope in the beginning.

The first main cause, Chowdhury's algorithm does not fit our needs, it does not support request node collocation, or in other words it maps every request node to a different substrate node. I do not want to make such constraint in the case of VNF mapping. Secondly, this

algorithm handles the mapping of bandwidth request in a splittable manner, thus enabling traffic to be splat to multiple paths between two request nodes. (This is caused by the usage of multi-commodity flow problem for finding paths.) It could result in an unpleasant mapping, when we do not have the full control of latency between the VNF-s, because one traffic flow can be divided between multiple paths. And thirdly, the algorithm provides only an approximate solution despite the sophisticated tools used.

The MIP formulation obviously includes an objective function for selecting the best mapping. So I am not going into the details of the MIP formulation, just discuss and evaluate the objective function which still can be useful to define my own.

The objective function tries to describe the overall state of the loaded network by one scalar. The sum of reserved CPU capacity on all substrate nodes is weighted by the reciprocal of the residual capacity. Then, this sum is added to the sum of all reserved bandwidth of every links, weighted by the reciprocal of residual link capacity. Each node and link has a constant multiplier to control their importance. This composite objective function is to be minimized in the linear program, so that it would minimize the cost of request embedding, as well as balance the load [16]. This kind of multi-purposed objective function is definitely the path to follow.

My algorithm's objective function is the generalization of the above explained one, taking more resource types and parameters into consideration. This will be discussed in more details later.

3.1.2 Approximation algorithm for the first sight

The paper of Fürst *et al* deals with the problem of request node collocation as a part of the Virtual Network Embedding problem [17]. The main goal of the paper is to give an optimal grouping of the virtual nodes in the sense of minimizing the amount of link resources, by mapping the virtual nodes of a group to the same substrate node. The grouping must be done before running the mapping algorithm, this method is called pre-clustering. The results of Fürst et al gave an optimal pre-clustering of an arbitrary virtual network, which can be used as the input of any other virtual network embedding algorithm. The paper states that, the lack of collocation is a general flaw of state-of-the-art VNE algorithms. The pre-clustering is done by a linear programming algorithm, just like the above mentioned embedding algorithm.

Fürst's paper was more interested for me because of its approximation algorithm for mapping, and a referenced algorithm from another paper, both of them were used as test algorithms to study the effect of pre-clustering.

LoCo stands for *location correlation,* which is the name of the own approximation algorithm of Fürst *et al* for Virtual Network Embedding, which directly supports virtual node collocation. The algorithm of LoCo can be seen in Figure 3, which is explained below. Graph *G* is the virtual network to map, *M* is the set of already mapped virtual nodes and *P* is the set of pending virtual nodes, which are to be mapped next. *P* is initially the neighbors of the starting virtual node, *s*.

Algorithm 1 The LoCo Algorithm

Require: VNet $G = (V, E)$, $M = \{s\}$ for some $s \in V(G)$, $P = (\Gamma(s))$
while $|P| > 0$ **do**
sort P (* decreasing link capacities *)
choose $u = P[0]$ (* next node to map *)
map u (* forward checking *)
map $\{u, v\}$ $\forall$ $v \in M$, **where** $\{u, v\} \in E(G)$
$M = M \cup \{u\}$ **and** $P = P \setminus \{u\}$
end while
if (embedding failed), **backtrack** on s

Figure 3: Pseudo-code of the LoCo algorithm [17].

LoCo is a greedy algorithm starting from *s* and then checking whether one of the neighboring virtual nodes can be mapped to a substrate node, where either the other end of the request link or another virtual node was mapped. After a successful mapping, the algorithm checks if there are virtual links, whose both ends are already mapped somewhere (this checking happens on the second *map* line). In continuation, the embedding process steps to the virtual node which is connected with the highest bandwidth requirement to the already mapped ones.

If a virtual node cannot be mapped anywhere, the algorithm can also backtrack the mapping, by removing the previously mapped virtual node from the substrate network, and trying another possible host for the requested computing capacity. Hereby, LoCo is an effective, fast and simple suboptimal algorithm to solve the problem of VNE.

The paper also publishes the measurement results on how much does the authors' pre-clustering algorithm (as far as, it is their main contribution) influence the substrate network utilization in co-operation with various embedding algorithms. LoCo performs pretty well on the tests, with and without pre-clustering, which also demonstrates, it is worth investigating the possibilities of approximation algorithms for VNE.

3.1.3 Meta information

Furthermore, Fürst's paper presents a general idea, a framework for virtual network embedding algorithms, which can be used to speed up or give guidelines to any mapping algorithms. The idea is referred as MetaTree.

MetaTree is based on a hierarchical partition of the hosting network, thus forming a tree, which's every node is an induced subgraph of the substrate network graph. The edges of MetaTree indicate the containment relationship between subsets [17].

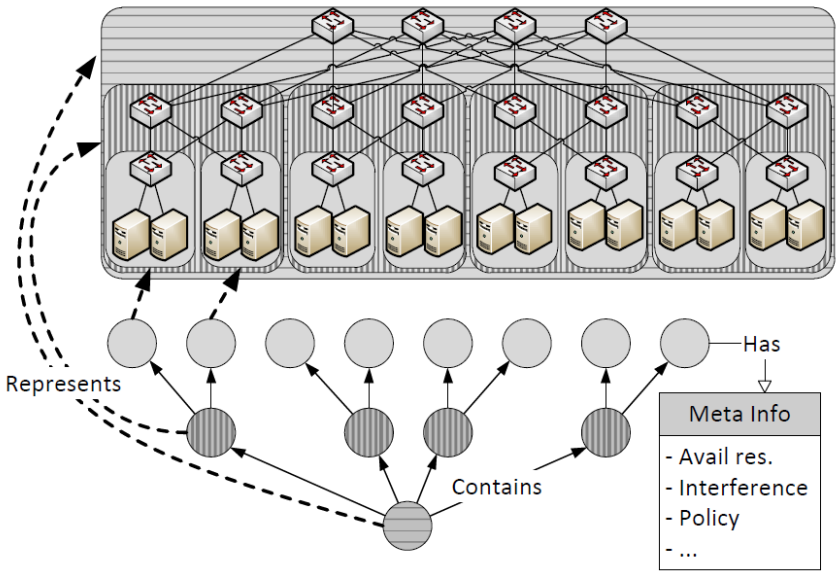

Figure 4:MetaTree partitions of a substrate network graph [17].

In Figure 4 we can see a picture demonstrating the structure of MetaTree. Every MetaTree node can have meta-information about the subgraph it refers to. The framework does not give any restriction on what can be a meta-information, it can include any useful data for the embedding process. For example, the meta-data can give the total amount of available resources, or the list of hostable network functions in the corresponding subgraph. But we can also imagine more theoretical attributes as meta-data, like the maximal distance of substrate nodes calculated in latency.

The paper itself, does not give any specific method on how to construct a MetaTree, but it refers to other publications dealing with the problem of graph partitioning. For example, Farhat's greedy algorithm [18], which partitions graphs into a given number of pieces based on node distances, achieving this by a simple breadth-fisrt searching (BFS) manner. The algorithm starts from a random node, and continues the BFS until the required number of nodes for one partition is reached, and handles the induced graph of the reached nodes as one partition.

After this, the finding of the next subgraph is started from another node, until the whole graph is covered. We could replace the BFS method with shortest path search on a network graph weighted by link and node latency. Thus, achieving a MetaTree with latency information of the corresponding subgraphs.

The combination of MetaTree and distance based graph partitioning are very useful ideas that could be used for planning a custom algorithm, which will be explained in detail in the appropriate chapter.

3.2 Virtual Data Center Allocation

In the previous section, I have mentioned that Fürst et al used existing mapping algorithms to test the effect of their pre-clustering algorithm. One of those was SecondNet [19], which is a data center network virtualization architecture, with allocation algorithm capable of mapping virtual networks. Therefore, the problem of Virtual Data Center Allocation (VDCA) is also similar to the problem of my study, just like Virtual Network Embedding. In the paper of SecondNet, a formal proof on the NP-hardness of VDCA problem is given, which also demonstrates the difficulty of the field of network resource mapping.

In VDCA, a connected set of virtual machines, with given resource requirements, shall be mapped to physical servers. The problem statement of VDCA diverges from VNE in that, the authors make difference between switches and servers in the substrate network, but this is not an issue with the applicability of the algorithm. Furthermore, they define two types of the VDCA problems, according to the strictness of bandwidth requirement.

Firstly, type-0 needs to allocate paths with guaranteed bandwidth between two virtual machines (VM-s). Secondly, type-1 is between best-effort and type-0, so that it only guarantees local ingress/egress bandwidth reservation for virtual machines, but not on the entire path connecting them. Obviously, we want to deal with type-0, because the other two types can be seen as the special case of type-0 and we also defined service chain requirements to be strict.

Initially, SecondNet performs a clustering[3] based on hop-count, the number of hops from one server to another. The clusters are not disjoint on the servers (a server can be in multiple clusters), and the pre-clustering of Fürst et al, can still be used despite the clustering

[3] The two terms, cluster and pre-cluster, can cause understandability issues here, but these are the terms used by the authors, so I would like to stick to them. To make clear: cluster belongs to SecondNet, pre-cluster belongs to the algorithm of Fürst et al.

of SecondNet. (The algorithm of Fürst et al pre-clusters to collocate virtual nodes connected with high bandwidth requirements between them.) SecondNet uses its clusters as subgroups of physical servers for the allocation of all the virtual machines (or the whole virtual network request).

In the first step, the VDCA algorithm chooses a cluster which is big enough for mapping the entire request. Then, it builds a bipartite graph, with the virtual machines at one of the color classes and the physical servers at the other color class. The edges are drawn between a pair of VM and server if, and only if that mapping is feasible according to the requirements of the virtual machine. Two nodes are also added to the bipartite graph, connecting one of them to all VM-s, and the other one connected to all servers, referring to the nodes as source and destination respectively. With appropriate weight and cost assignment to the edges, the problem transforms to a min-cost flow problem, which can be solved efficiently.

After successful VM allocation, in the third step, the algorithm finds paths to the VM pairs, in descending order of required bandwidth (to fail earlier if a bandwidth need cannot be satisfied with the actual VM allocation). If the path allocation fails, the algorithm moves to the next cluster to try again the search, otherwise, the mapping is finished.

SecondNet is a very nice algorithm, because it could be used to solve a different task than it was originally planned to, as we can see this from the results of Fürst et al [17]. So naturally, it is not optimized for the task of VNE, but it can give a starting point for planning my own algorithm, and can reveal issues to pay attention to. Originally, it does not support virtual machine (virtual node) collocation by itself, but gained significant improvement in resource utilization by co-operating with pre-clustering mechanisms. So native collocation support should be a compulsory feature for a novel request mapping and resource allocation algorithm.

Another weakness of SecondNet could be its two phased allocation procedure. First, it finds hosts for the VM-s, and does not perform any checking of link resources, and then it tries to find links with enough resource, which can cause many dead ends during execution. One of my main goals in this paper is to design an algorithm that has no need to run in two phases like that.

3.3 Graph pattern matching

3.3.1 Definitions

Graph pattern matching could be the mathematical basis of mapping a set of service chains to the physical resources of a substrate network, with the mathematical notions of subgraph isomorphism and graph simulation.

Unlike isomorphism, graph simulation is not a commonly used notion, so I would like to devote some lines to demonstrate the difference between the two definitions. First, I quote the definition of graph simulation form the paper of Fan et al [20], which can be seen in Definition 3.

> ***Graph simulation***: to find a binary *relation* $S \subseteq V_P \times V$, where V_P and V are the set of nodes in P and G, respectively, such that
> (a) for each node u in V_P, there exists a node v in V such that $(u; v) \in S$, and u and v have the same label, and moreover,
> (b) for each $(u; v) \in S$ and each edge $(u; u')$ in P, there is an edge $(v; v')$ in G such that $(u'; v') \in S$

Definition 3: Definition of graph simulation quoted from paper of Fan et al [20].

Graph isomorphism requires to have a bijective function between the nodes of the pattern and the target graph. Despite this, graph simulation needs only a binary relation between the nodes of the two graphs (which allows many-to-one mapping), as it is in Definition 3. Furthermore, graph simulation introduces the notion of node label, and requires the matching of the labels of the appropriate nodes to fit the definition. Node label can be any kind of data stored on a node, for example the list of hostable network functions or the available resources on a node. Finally, isomorphism requires to have a bijective function between the edges too, despite the definition of graph simulation, which only says, if there is an edge in the pattern, there should be an edge between the appropriate nodes as well in the target graph, but the implication does not stand in the other way. This can be checked in the (b) point of Definition 3.

All in all, graph simulation is much less strict in defining the pattern, but still requires for example pattern edges to be mapped to one edges, and not to paths, as we obviously need it.

So we can state that, in conjunction with Fan et al [20], graph simulation and isomorphism are not quite up to the task of service chain mapping, because they are way too restrictive in defining the graph pattern. Thus, limiting the applicability of existing graph pattern matching algorithms to the problem.

Fan et al study that, what kind of graph pattern definition would be more applicable to new tasks, like social network analysis, so they defined the notion of *bounded graph simulation*. And they show a cubic time algorithm to find such newly defined graph patterns. Their fast algorithm could be scalable to graph sizes of even one billion nodes. In the second part of their paper, they also publish an incremental algorithm to find all matching patterns, without the need to run the whole algorithm from the beginning, if the graph or the pattern changes a little. I will only focus on the first part of the paper of Fan et al presenting the prior algorithm.

To understand the definition of bounded (graph) simulation, we have to look at their definition of the pattern graph, which is called by the authors a *b-pattern*. A b-pattern consists of a set of nodes and edges, as an ordinary graph. In addition, there is a function on the edges of the b-pattern, which gives the upper bound of the length of the path, to which the edges are to be mapped. The length of a path can also be unbounded, and it is denoted by *, instead of the constant integer number. Furthermore, the b-pattern consists of a function on the domain of nodes, which defines the search condition on the nodes as the conjunction of a set of atomic formulas of the form $X\ op\ x$, where X denotes an attribute, x a constant and op can be $<, \leq, =, \neq, >, \geq$ [20]. Naturally, the target graph (also called data graph) also has a function on the nodes, similar to the latter pattern function (but with attributes being always a constant value) to match the pattern attributes to.

By now we are ready for the definition of bounded simulation, which can be found in Definition 4, quoted from the paper of Fan et al. Both G and P are directed graphs, denoting the direction of edges by the order of the node 2-tuple.

Bounded simulation. Consider a data graph $G = (V; E; f_A)$ and a b-pattern $P = (V_p; E_p; f_V; f_E)$. We say that graph G *matches* pattern P via *bounded simulation*, […], if and only if there exists a binary relation $S \subseteq V_p \times V$ such that

(1) for each node u in V_p, there exists a node $v \in V$ such that $(u; v) \in S$;

(2) for each pair $(u; v) \in S$, $v \sim u$; and

(3) for each edge $(u; u)$ in E_p, there exists a nonempty *path* $p = (v; ...; v')$ from v to v' in G such that (a) $(u'; v') \in S$, and (b) $\text{len}(p) \leq k$ if $f_E(u; u)$ is a constant k.

Definition 4: Definition of bounded graph simulation [20].

In Definition 4, f_A and f_V are the attribute functions of target and pattern graphs respectively, defining the values and the search conditions on the node attributes. The path length bounds are given by f_E on the edges of the pattern graph. The matching of a pattern node to a data node, satisfying the search conditions, is denoted by ~ (swung dash), in (2) of Definition 4.

The set S in the definition is called a match, obviously, there could be multiple matches of the pattern P (where one pattern node can fit to more data nodes) in the data graph G. Fan et al prove that, there exists a unique maximum match for every input (or an empty set S, also called a match). Its formal proof can be found in [20]. Their polynomial algorithm finds the maximum match in a data graph for a given b-pattern.

3.3.2 The algorithm of Fan et al

Now we are ready to understand the algorithm itself. In a few words, for every node in the pattern, the algorithm finds a set of potentially matching data nodes, which satisfy the search condition (initially not dealing with the length constraints). And then, it iteratively removes the nodes that violate the distance and connectivity constraints, until no further changes can be made, and the sets contain the maximal set of data nodes, that match the corresponding pattern node.

In Figure 5, the pseudo-code of the algorithm can be seen. The previously explained notations in the definitions are still valid, the new ones will be explained in the followings, line by line.

The output of the algorithm is the unique maximum match of the b-pattern *P* to the data graph *G* via bounded simulation, or the empty set, if there is no match via bounded simulation at all.

Input: A b-pattern $P = (V_p, E_p, f_V, f_E)$ and data graph $G = (V, E, f_A)$.
Output: The maximum match $M_{\mathsf{ksim}}(P, G)$ if $P \trianglelefteq_{\mathsf{bsim}} G$, and $\emptyset$ otherwise.

1. set $S := \emptyset$; compute the distance matrix $\mathcal{X}$ of G;
2. **for each** $(u', u) \in E_p$ and **each** $v \in V$ **do**
3. **if** $f_A(v)$ satisfies $f_V(u)$ **then** compute $\mathsf{anc}(f_E(u', u), f_V(u'), v)$;
4. **if** $f_A(v)$ satisfies $f_V(u')$ **then** compute $\mathsf{desc}(f_E(u', u), f_V(u), v)$;
5. **for each** $u \in V_p$ **do**
6. $\mathsf{mat}(u) := \{v \mid v \in V, f_A(v)$ *satisfies* $f_V(u)$, and *out-degree*$(v) \neq 0$ *if out-degree*$(u) \neq 0\}$;
7. $\mathsf{premv}(u) := \{v' \mid v' \in V,$ *out-degree*$(v') \neq 0$, and $\nexists (u', u) \in E_p$ $(v \in \mathsf{mat}(u), f_A(v')$ *satisfies* $f_V(u')$, and $\mathsf{len}(v', \cdots, v) \leq f_E(u', u))\}$;
8. **while** (there exists a node $u \in V_p$ with $\mathsf{premv}(u) \neq \emptyset$) **do**
9. **for each** $(u', u) \in E_p$ and **each** $v_1 \in \mathsf{premv}(u)$ **do**
10. **if** $v_1 \in \mathsf{mat}(u')$ **then**
11. $\mathsf{mat}(u') := \mathsf{mat}(u') \setminus \{v_1\}$;
12. **if** $\mathsf{mat}(u') = \emptyset$ **then return** $\emptyset$;
13. **for each** u'' with $(u'', u') \in E_p$ **do**
14. **for each** $v_1' \in (\mathsf{anc}(f_E(u'', u'), f_V(u''), v_1) \setminus \mathsf{premv}(u'))$ **do**
15. **if** $(\mathsf{desc}(f_E(u'', u'), f_V(u'), v_1') \cap \mathsf{mat}(u') = \emptyset)$ **then**
16. $\mathsf{premv}(u') := \mathsf{premv}(u') \cup \{v_1'\}$;
17. $\mathsf{premv}(u) := \emptyset$;
18. **for each** $u \in V_p$ and **each** $v \in \mathsf{mat}(u)$ **do** $S := S \cup \{(u, v)\}$;
19. **return** S as $M_{\mathsf{ksim}}(P, G)$;

Figure 5: The maximum match searcher algorithm based on the notion of bounded graph simulation. [20].

First, the algorithm computes the distances of all the nodes in the unweighted data graph, by the utilization of BFS starting from every node. This can be done in cubic time.

In the 2 – 4 line on Figure 5, for every pattern edge the algorithm computes the set of ancestors[4] *v'* in *G* (*anc*), that can match any parent (one hop ancestor) *u'* of *u*, satisfying the data label search condition and the distance constraint. Naturally, the distance of *v'* is counted from the matching data node *v* of the pattern node *u*. So, for example *anc(4; f_V (u'); v)* contains ancestors of *v* in *G*, that are no further than 4 hops, and satisfies the search condition on *u'*. And the set of descendants[5] (*desc*) is calculated similarly for every data node *v* in *G*.

[4] Ancestors of a node *v* in a directed graph are all the nodes that are reachable from *v*, if we are stepping *backward* (opposite direction of the edge direction) on the edges.
[5] Similarly, descendants of a node v are all the reachable nodes, stepping in the direction of edges from *v*.

In the line 6, the *mat* sets are calculated for all pattern nodes that satisfies their search condition (but the distance constraint is not checked). Here we can see the clever usage of node degrees, intuitively: if *u* has at least one child, *v* should have one too, otherwise *v* is not a possible match for *u*. Next, the data nodes, which *cannot* match *any* parent of *u*, are gathered in *premv(u)*.

After the initialization processes, in line 9.-11. for each *u'*, the parents of *u*, the cycle eliminates data nodes from *mat(u')*, by the utilization of *premv(u)*. If any of the *mat* sets becomes empty, there cannot be a match that can cover all pattern nodes, because *mat* sets cannot increase during execution (line 12.).

Furthermore, the elements of *premv(u)*, denoted by v_1 on the Figure 5, are used to rule out some more possible matches of *u'* (note that, v_1 cannot match *u'* because of the distance constraint – see the conjunction of line 9. and 10). By looking at the parents of *u'*, denoted by *u''* (see line 13.), v_1' is an ancestor of v_1 (see line 14.), that is a possible match of *u''*, but only if there is at least one data node among its descendants, which can be a match to *u'* (see line 15.). Otherwise it is added to *premv(u')*. This whole refining process continues until there is a chance for decreasing any *mat* set, by being *premv* not empty (see line 8.)

Finally, the maximal match is produced from the *mat* sets of every pattern node (line 18.).

Fan et al also gives formal proof to that, the algorithm always terminates and it gives the maximal match of a b-pattern in a data graph in terms of bounded simulation (correctness proof). Furthermore, they prove the cubic runtime complexity of the algorithm.

4 Specifying the input and output

The goal of this section is to appoint and further specify the problem of VNF mapping, because it is not yet specific enough to start the discussion of the algorithm. The design choices, that were required to make during my research, are also presented here.

First, let us start with specifying the request graph and the service chains. The service graph is basically a network of VNF instances and SAPs, connected by request links. We can think about the links as undirected graph edges, because directing the traffic in the substrate network after the mapping, could also be the task of traffic steering, which I have placed out of the scope of the problem earlier.

I am going to handle the resource requirements of the VNFs in the two components of memory and CPU computing capacity. Every VNF in the service graph must specify what kind of network function it should operate (only one type). I denote the VNF type with capital letters of the English alphabet. All VNFs shall have a unique name; it is recommended to be a readable character string, which is understandable by the user, because I plan to use these only for graphical representations. In the algorithm, I use unique integer identifiers to refer to the VNFs. More than one instances of a VNF type can be in the service graph at the same time. The undirected links in the request graph mean reachability constraints between the VNFs.

One service chain consists of the maximal latency permitted, given in milliseconds, that the mapped paths shall not exceed even after applying all of the VNFs of the chain. Similarly, bandwidth requirement can be given to every service chain in megabit per seconds that the user shall be able to measure between the ends of the service chain. The chain itself shall be given as an ordered list of the integer identifiers of the VNFs, and (in case of SAP-to-SAP chains) SAP identifiers at both ends of the lists, indicating where the chain should be attached to. Not SAP-to-SAP chains can also be given (either or both ends starting from a VNF), with that constraint, it must be a part of a SAP-to-SAP chain[6]. The bandwidth requirement of a not SAP-to-SAP chain shall determine how much bandwidth capacity is needed in addition to the requirement of the containing SAP-to-SAP chain.

Arbitrary number of service chains can be given as part of the input, which are represented in an unordered list of service chains. As far as a VNF instance can only be mapped

[6] I explain it in a later section why I had to make such constraint, and how it could be resolved. But the algorithm currently can only handle input with the given constraint.

to one hosting node in the substrate graph[7], and if multiple service chains want to use the same VNF instance, the request paths leading to the host of that VNF must be mapped accordingly in the substrate network. The request links' bandwidth requirement is given indirectly by the sum of bandwidth requirements of the service chains, which contain that specific request link.

SAPs shall have no resource requirements, they should possess a unique name, which is used in both (substrate and request) graphs to identify them. If there exist a SAP in the request graph, but a SAP with the same name does not exist in the substrate graph, an error should be raised. A SAP name could be anything, which contains enough information for the surrounding architecture layers or the network management system to identify which physical entity it refers to. For example it could be the IP address of a server. SAPs should also have a unique integer identifier in both graphs, but these identifiers do not have to be identical in the two graphs for same SAP instances, because they are not used for the mapping (only their name is used). But the integer identifier of the SAPs in the service graph is used in service chain definition for telling where the services shall be attached.

The substrate graph or the physical topology contains hosts, switches and SAPs. SAPs do not need to have any additional parameters, as they have in the request graph.

Hosts have resources of available CPU capacity and memory, which is used by the hosted VNFs. Maximal resources of each resource component should also be stored; in an unloaded network, available and maximal resources should obviously be equal. Every host should be able to run a set of VNF types, and they should be able to host multiple instances of one type of VNF, if they have enough resource. The set of hostable VNFs is represented by a (unordered) list of capital letters of the English alphabet. Furthermore, the hosts should have two unique identifiers, one easily readable for graphical representation and one for inner representation.

Switches can be added to the substrate graph, but they can be modeled by hosts with zero available resources. So from the perspective of the algorithm design, if we connect hosts directly or through switches, are identical cases. But naturally the operation of the algorithm will be influenced by the two cases, because if we model all switches with zero resourced VNFs, an additional resource checking would be executed on every modeled switch.

[7] Our problem does not include optimizations with the tool of splitting or doubling VNF instances, it should be done by the upper layer.

The substrate links should have attributes indicating how much more bandwidth they can foster currently, and also the maximal bandwidth capacities should be stored for every link. Furthermore, if both ends of a request link is mapped to the same substrate node, the hosts should have a maximal and available inner bandwidth parameter, indicating how much inner traffic they can handle between the collocated VNFs.

In both of the graphs every node should store information about its type, whether it is a switch, a VNF, a SAP or a host, so the algorithm could decide what it could do with the actual entity.

For fast prototype creation and moderating the task's complexity, it is a general approach to keep some input parameter constant. In my case, I did not want to decrease the complexity of the problem, so I made the trade-off with the time variable latency. In other words, as an input parameter of the substrate network, I made latency time invariant. So I made up three latency values, one for link latency, one for forwarding latency of hosts or switches, and finally one for VNF processing latency, and I use them as constants. This restriction does not change the task's complexity, because latency still must be handled.

In the future, it would be easy to extend the algorithm with handling dynamic latency, but this trade-off made the algorithm implementation much faster and proceeding to the testing could be done earlier. When I will discuss the algorithm implementation in a later section, I will refer to this condition and note which parts and how should be modified for dynamic latency handling, and I also explain in detail why it helped.

5 Proposed solution

This chapter is organized as follows: first, I present the pseudo-code of my VNF mapping algorithm in two parts. Secondly, I show some basic test cases and propose an objective function, which could be used to evaluate the state of a network in general, according to the actual resource reservations. I use this function to evaluate the algorithm, comparing to the calculated optimal mapping in the sense of the proposed objective function, by an alternative (not practical) algorithm.

5.1 My service chain mapping algorithm

The algorithm operates in two phases, during its presentation, I am going to follow the order of execution, starting with the input preprocessing part, and finishing with the core part, which does the actual mapping procedure.

Unluckily, there is no universally accepted pseudo-code format standard, which I could follow, so on the subsequent figures of the algorithm presentation, I will use the following conventions. Italic font style represents variables, or functions which return some attribute of an entity. Bold font style indicates general programming terms or control structures, for example a *for* loop or *if-then-else* structure, which I am not going to explain. Plain text indicates function names, tags or phrases, which needs further explanation, and can be found in the text, commenting on the specific lines of the pseudo-code.

5.1.1 Preprocessing

In Figure 6, we can read the pseudo-code of the preprocessing required to run on the input structures of the algorithm, before the actual VNF mapping could be executed. The goal of the preprocessing is to ease the implementation of the core part and to decrease its runtime complexity. It is achieved (besides some other minor tasks) by dividing the service graph to "subchains", which are defined by the intersections of the input service chains, and are disjoint on the set of VNF instances. Furthermore, a subgraph of the substrate network is returned for every subchain, which is used to map the entire subchain. This process will be explained with the pseudo-code of the *Preprocessor_Algorithm*

Subgraph finding

The input of the *Preprocessor_Algorithm* is given directly by the upper layer of the service chaining architecture, its format is specified in the earlier sections, which I have formalized here, in Figure 6.

The substrate network is an undirected graph with vertex set denoted by *V*, and edge set denoted by *E*. The domain of the functions *fres* and *fbw* are *V* and *E*, and returns the available resources of a substrate node and the available bandwidth of a substrate link respectively. The set of VNFs is represented by *Vp* and the set of request links is *Ep*. Function *freq* returns the resource requirement of a VNF instance, and *fvnf* returns the type of a VNF instance.

The list of service chains is given by an unordered list of 3-tuples, it can contain arbitrary number of service chains. Every service chain should consist of an ordered list of integer identifiers of VNF and SAP instances from *Vp*. The other two parameters are *lat* and *bw*, which denote the maximal permitted latency of the service chain given in milliseconds, and the minimal required bandwidth given in megabit per seconds respectively. This structure is denoted by the *service_chains* variable in Figure 6.

Preprocessor_Algorithm

Input: *substrate_network* = (*V*, *E*, *fres*, *fbw*), *service_graph* = (*Vp*, *Ep*, *freq*, *fvnf*), *service_chains* = [(*chain*, *lat*, *bw*)*].

Output: Preprocessed *substrate_network* and *service_graph*, and also *subchains* with corresponding *subgraphs*.

```
1. processNetwork:
2. for each host in substrate_network do
3.    Make loop in host.
4. endfor
5. for each node1, node2 in substrate_network do
6.    Calculate shortest path between node1 and node2.
7. endfor
8. output.append (substrate_network)
9. processRequest:
10.  sort_ascending (service_chains by lat)
11.  for each chain, lat, bw in service_chains do
12.     maxhop = transform (lat, chain )
13.     for each link in chain do
14.        fbwreq (link) += bw
15.     endfor
16.     if chain is end-to-end then
17.        e2e_subgraphs.append (findSubgraph (chain, maxhop))
18.        e2e_chains.append (chain, lat, bw, maxhop)
19.     else
20.        not_e2e_chains.append (chain, lat, bw, maxhop)
21.     endelse
22.  endfor
23.  service_graph.add (fbwreq), output.append (service_graph)
```

```
24. divideIntoDisjointSubchains:
25.   fcolor = colorLinksAndVNFs (service_graph, e2e_chains, \
       not_e2e_chains)
26.   colored_service_graph = copy (service_graph)
27.   colored_service_graph.add (fcolor)
28.   while colored_service_graph has edge do
29.     e2e_chain = e2e_chains.myiterator()
30.     for each reqnode in \
         custom_order (colored_service_graph.nodes()) do
31.
32.       if any reqnode_neighbor in reqnode.neighbors() and \
           fcolor(reqnode_neighbor) contains fcolor(e2e_chain) then
33.           subchain, subgraph_of_subchain = findOneSubchain \
              (fcolor, reqnode, reqnode_neighbor, e2e_subgraphs)
34.           subchains.append (subchain)
35.           subgraphs.append (subgraph_of_subchain)
36.           for each edge in subchain do
37.              colored_service_graph.remove(edge)
38.           endfor
39.           break
40.       endif
41.
42.     endfor
43.   endwhile
44.   output.append (subchains, subgraphs)
45. return output
```

Figure 6: Pseudo-code of the input preprocessing for the VNF mapping algorithm of my own design.

The first part of the Preprocessor_Algorithm is *processNetwork* (see lines 1.-8. of Figure 6). As I have foreshadowed in the earlier sections, collocating two ends of a request link has the cost of inner bandwidth[8] utilization, at the amount of the bandwidth requirement of the request link between the two VNF instances. I have added a loop link in every host of the substrate network for storing the inner transmission bandwidth limit, which is generally much[9] higher than the bandwidth capacity of a substrate link, but still limited.

Next, for every pair of substrate node, the shortest path (e.g. their distance in hop-count) is calculated and stored; this information will be used later in the preprocessing. This can be done by the Floyd-Warshall algorithm [21], with constant edge weights, in worst case complexity of $O(Vp^3)$. If we want to handle dynamic latencies, distances should be calculated in latency values of forwarding, processing and links.

A link weight is calculated for every substrate link, and even for the loops (not indicated in the pseudo-code of Figure 6). Weight is a neutral bandwidth value divided by the maximal (the currently free) bandwidth capacity of the substrate link. Thus, links with higher bandwidth than the neutral, are preferred (have smaller weight) over other links.

[8] Physically, it could be implemented as in-memory transmission on the hosts.
[9] Especially, if the host would be configured to use *netmap* [24] for example.

After this, the preprocessing of the substrate network is finished, *substrate_network* variable is added to the output, with the additional loop edges, as indicated in line 8 of Figure 6.

The *processRequest* involves the rest of the preprocessing algorithm (lines 9 – 44). In line 10, the function *sort_ascending* sorts the service chains (modifying the *service_chains* variable in place) in ascending order, based on latency requirement. The comparison between two service chains is made exclusively on their latencies, so the service chain with the strictest latency requirement comes to the beginning of the *service_chains* list.

The *for* loop of lines 11 – 22 iterates over on all the input service chains. Line 12 uses the trade-off I have made with keeping the latency values constant. The *transform* function subtracts the number of VNF instances required to go through in the actual *chain*, multiplied by the constant value of latency of VNF application, acquiring the available latency which can be used for forwarding and link latency. The service chain will be mapped to a path in the physical network, which path will consume one time link latency value and one times forwarding latency in every hop. Thus, if we divide the available latency by the sum of the constant link latency and constant forwarding latency, we acquire how many hops long can the path be at most, which hosts the VNFs of the service chain.

From now on, the *maxhop* variable will act as the limit, instead of maximal latency. As I have mentioned earlier, the adaptation of the algorithm to dynamic latency handling will be easy to implement, because instead of the hop-count, we would have to calculate distances with the actual latency values of links, hosts and switches, and use *lat* as the distance limit[10].

Next, in line 13-15 of Figure 6, the algorithm adds the bandwidth requirement of the actual service chain to all the links of *Ep* which are contained in *chain* (the actual service chain). By doing this and iterating over all the service chains, from now on, we can speak about bandwidth requirements of individual request links, its vales can be acquired by the *fbwreq* function, which is added to the service graph structure. Furthermore, the preprocessed *service_graph* is also appended to the output, with bandwidth requirements given on its request links.

Furthermore, in lines 16-21 of Figure 6, if the actual chain is a SAP-to-SAP chain, the *findSubgraph* function determines an induced subgraph of the substrate network topology,

[10] One-way latency measurement is not trivial in general, but there are findings in this problem, and furthermore, here we need only one-way latencies of individual links, which is much easier to measure.

which shall be used for the mapping of the entire chain (and not the other hosts and links of the substrate network, outside of the induced subgraph). The subgraph returned by the function includes all the nodes (hosts and switches), which's sum of distances from the first and last element of the *chain* is less than or equal to *maxhop*. In case of dynamic latency handling, this comparison would be done directly to the upper limit of the latency instead of *maxhop*, and distance would be calculated measured in the actual latency and not in hop-count. Here, the distances are not calculated, because the lines 5 – 7 of the Preprocessor_Algorithm have already stored it.

The subgraph finding is realized by a traversal from a SAP in the substrate network, which is one of the ends of *chain*. (Subgraph finding is only done on SAP-to-SAP chains, and SAPs of the service graph can be mapped unambiguously to the SAPs of the substrate network by their names, as I have conditioned earlier.) The inclusion criteria is checked for every node, until the process can be continued. After completion, if the other end of the service chain is not in the induced subgraph, an error is raised, because it would mean that the distance of the two SAPs is more than the permissible end-to-end latency on the service chain.

Subgraphs are gathered in *e2e_subgraphs* and SAP-to-SAP chains in *e2e_chains*, and finally not SAP-to-SAP chains in *not_e2e_chains*, now with maximal hop-count included. These lists are used later in the preprocessing. Given an SAP-to-SAP chain, the algorithm can determine the corresponding stored subgraph (this fact is not reflected well by the pseudo-code).

Disjoint subchains

The *divideIntoDisjointSubchains* section lasts from line 24 to line 43 of Figure 6. Its task is to divide the request graph into disjoint subsets (subchains) based on the intersection of service chains.

First, let us imagine different colors for all service chains, and use them to color all the edges and vertices of the request graph contained by any of the chains. If an edge or node is contained by multiple service chains, color it with all the corresponding chain colors, gaining a new color (suppose that we can determine unambiguously every color's components). An example of such coloring is given in Figure 7, where nodes number 6 and 7, and the link between them has the color of both service chains, creating a separate color from *Chain1* and *Chain2*.

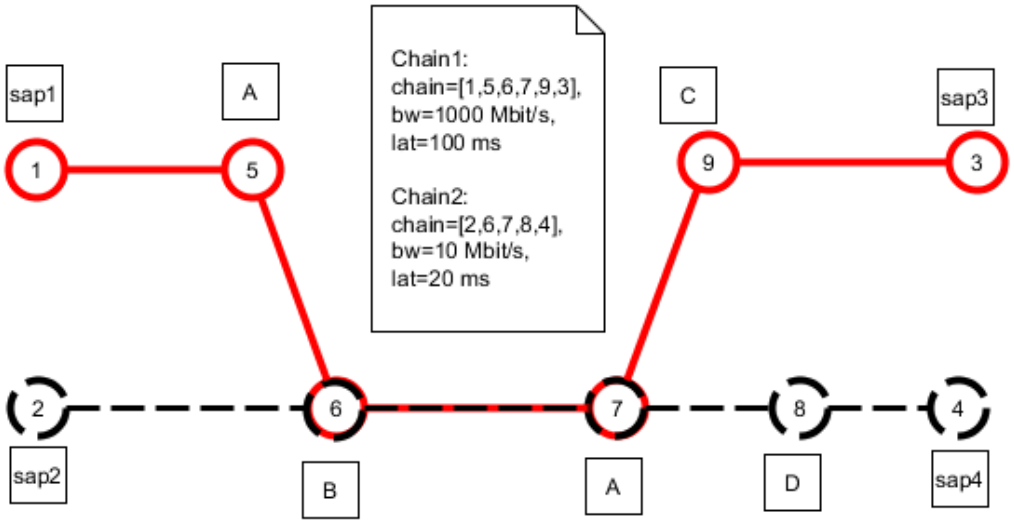

Figure 7: An example request graph and its coloring based on chain colors.

In the 25 line of Figure 6, the *colorLinksAndVNFs* function creates the explained and demonstrated coloring of the service graph. As indicated in the pseudo-code, not SAP-to-SAP chains have their own colors too. The *fcolor* function can determine the set of service chains[11] that includes the given link or VNF instance. Furthermore, *fcolor* can return the color of a single service chain. The *fcolor* function is appended to the structure of *colored_service_graph*, which is a copy of the *service_graph* variable. I will also refer to this new graph as colored request graph.

The *while* loop of lines 28-43 on the Figure 6 iterates until all of its edges and nodes are part of any of the subchains. A SAP-to-SAP service chain is selected by the *myiterator* function in every iteration of the *while* loop. *myiterator* returns every service chain twice in ascending order of latency requirements (chain with the strictest requirement comes first two times, and then the next strictest two times, etc.).

The *for* loop starting from line 30 iterates on the nodes of the colored request graph. The *custom_order* here means that, the iteration starts with SAPs in ascending order of degree, and if all SAPs are done, it continues on VNFs in the same ordering as in the case of SAPs.

Line 32 has a bit uncommon *if* condition with the *any* keyword. In a few words, this line checks if the current VNF instance has any neighbor, whose color contains the color of the actual service chain, stored in *e2e_chain*. The *contains* operator checks the inclusion of the

[11] In the implementation this set is realized by a hashable constant set of integer identifiers of service chains. For the comparation of the link/edge colores, I used hash comparation for performance reasons.

one-element set of *fcolor (e2e_chain)* in the set of *fcolor (reqnode_neighbor)* [12]. By iterating on the nodes of colored request in ascending order of degree (see *custom_order*), the *if* condition evaluates first to *true* on the node with the smallest number of degree.

The *findOneSubchain* function starts from the actual node (SAP or VNF instance), in the direction of the neighbor which satisfied the condition. It uses the color of the link between *reqnode* and *reqnode_neighbor* to traverse as much edge of the colored request with the same color as it can. Only the color of the edges (not the nodes!) is inspected. This color is not necessarily the same color which *reqnode* has, because for instance, if *reqnode* is a SAP and there is another SAP-to-SAP service chain starting from here and continuing on another request link than the selected one, then the color of the link connecting to *reqnode_neighbor* is used. See Figure 8, and bind the variables *reqnode=sap2* and *reqnode_neighbor* = VNF number 5.

The definition of the coloring implies that, the traversal cannot have branches, it can only return a sequence of nodes and edges. An almost complete example can be seen on Figure 8, only the traversal of links 5-6, 6-7 and 7-9 are left. All of these link would be in separate subchains each.

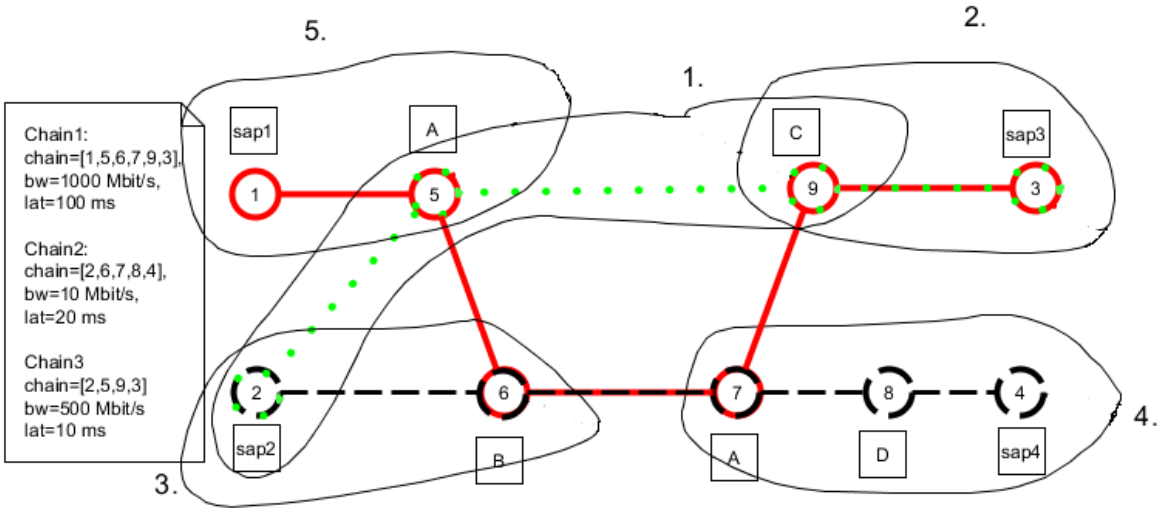

Figure 8: The demonstration of the (almost finished) output of divideIntoDisjointSubchains *section.*

The *findOneSubchain* function also utilizes the subgraphs found for the SAP-to-SAP chains and stored in *e2e_subgraphs* variable. In the case of 1st subchain on Figure 8, as the traversal on the color of *Chain3* reached VNF number 5, a placement criteria is set for that VNF, because this instance is used by another SAP-to-SAP chain. The algorithm can know

[12] This condition checking is realized by a list comprehension on the neighbors using the containment testing, and the folding of the list to one boolean value using the disjunction operator.

this, because the color of VNF number 5 is different from the color it actually used for the traversal. The placement criteria can be given to VNF instances, and it is a set of substrate nodes, where that VNF can be mapped. This set is determined by the intersection of the subgraphs of the service chains, which utilize that VNF instance.

The subchain finding finishes when the traversal cannot proceed on links with the selected color, or it encounters a VNF instance, which is already in another subchain with stricter latency requirement (or in other words, already have placement criteria given).

The other return value of the *findOneSubchain* function is a subgraph of the substrate topology, where all the VNFs of the corresponding subchain shall be mapped. This subgraph is determined by the intersection of the subgraphs of service chains, whose color is contained by the color of the traversal. For example, the subgraph of the 2nd subchain on Figure 8, is the intersection of the induced subgraphs found for *Chain1* and *Chain3*, and as for another example, the subgraph of the 1st subchain is equal to the subgraph of *Chain3* itself.

In the process of dividing the request graph, every subchain and their corresponding subgraphs are stored in the *subchains* and *subgraphs* variables, as lines 34-35 of Figure 6 indicate. Every link of the just found subchain is removed from the colored request graph. The ending nodes of the link are only removed from the graph, if their degree would decrease to zero by the link removal. For example, after running five iterations of the subchain finding process, on the request graph of Figure 8, there would only be three edges of 5-6, 6-7 and 7-9.

After all of the SAPs are removed from *colored_service_graph*, the search for one subchain can only start from a VNF instance, which is already part of one of the subchains. This statement is not reflected from the pseudo-code, because I did not want to overload it with information which is not crucial for understanding the algorithm.

As far as a subchain is always at least one edge long, in every iteration of the *while* loop, the number of edges of *colored_service_graph* is decreased at least by one. Edges are never added to the graph, so the subchain finding process always terminates after *Ep* iterations at most.

As I have mentioned earlier, the algorithm supports the mapping of not SAP-to-SAP chains, but only if they are contained by the union of SAP-to-SAP chains. If it is true, after dividing the request graph to subchains, all the not SAP-to-SAP chains are also part of some subchain. Their bandwidth requirement is already added to the bandwidth requirements of the request links, but using subgraphs for their mapping (which is one of the main components of

heuristic insurance of latency requirement[13]) is not yet supported. For supporting not SAP-to-SAP chain mapping, which do not obey to the above criteria is also possible with slight modifications of the subgraph finding process.

Finally, in line 44 of Figure 6, all the found subchains and their corresponding subgraphs are appended to the output of the Preprocessor_Algorithm.

The complexity of the algorithm is determined by the complexity of three components: Floyd-Warshall algorithm, substrate network initialization, subchain finding (roughly overestimated). This can be seen on Figure 9, where among other, previously mentioned symbols, *c* indicates the number of service chains.

$$O\left(|V|^3 + c(|V| + |E|) + \left|V_p\right|^4\right)$$

Figure 9: The complexity of Preprocesspor_Algorithm.

By finding subgraphs and subgraph intersections for the subchains with placement criteria on the appropriate VNF instances, we acquired a strong heuristic for making insurance to the latency (hop-count) requirement of service chains easier. Finding solution for the rest of the mapping problem is the task of the Core_Algorithm.

5.1.2 Core algorithm

After the input preprocessing is done, the algorithm acquires a list of subchains in ascending order of latency requirement. The edge sets of subchains are disjoint on the edge set of the input service graph.

In a nutshell, the algorithm starts the mapping from the end of a subchain, and calculates a composite objective function value for all the hosts of its subgraph, and chooses the best one. The objective function value calculation for a host incorporates a composite preference value of the load state of that host, and the sum of the link weights of the path leading to the actual host. The importance between the two components can also be weighted. The algorithm greedily continues these calculations on the other nodes of the subchain and then the next subchain, while respecting feasibility conditions and placement criteria of VNFs.

In the first line of the pseudo-code of Figure 10, the preprocessing algorithm of Figure 6 is called, which returns the structures explained in the previous section.

Core_Algorithm

[13] Other components and detailed explanation is given in the next chapter.

```
Input: Substrate network topology, service graph, service chains with
requirements.
Output: VNF to substrate node mapping, link to path mapping.
   1. substrate_network, service_graph, subchains, subgraphs = \
        Preprocessor_Algorithm (input)
   2. for each node in service_graph do
   3.   if node is SAP then
   4.     mapped (node) = (true, id_in_network (node))
   5.   else mapped (node) = false
   6. endfor
   7. for each link in service_graph do
   8.   mapped_link (link) = false
   9. endfor
   10.    for each subc, subg in subchains, subgraphs do
   11.      for each node1, node2 in subc.edges() do
   12.        if not mapped (node2) then
   13.
   14.        mapOneVNF:
   15.          best_host = (infinite, null)
   16.          for each n, path_to_n in \
                 all_paths (id_in_network (node1), subg.nodes()) do
   17.            if n is a host and n can host fvnf(node2) and \
                    n in placement_criteria_of (node2) and \
                    fres(n) satisfies freq(node2) then
   18.              value = objectiveFunction (node2, n, path_to_n)
   19.              if value < best_host[0] and value >= 0 then
   20.                best_host = (value, n)
   21.                mapped_link (node1, node2) = path_to_n
   22.              endif
   23.            endif
   24.          endfor
   25.          if best_host[0] != null then
   26.            mapped (node2) = best_host[1]
   27.            updateNetworkResources (mapped(node2), \
                    mapped_link (node1, node2))
   28.          else
   29.            Error, VNF cannot be mapped anywhere!
   30.          endelse
   31.
   32.        else
   33.          mapOneRequestLink (node1, node2, fbwreq (node1, node2))
   34.        endelse
   35.      endfor
   36.    endfor
   37.  return mapped, mapped_link
```

Figure 10: The core part of the VNF mapping algorithm of my own design.

In lines 2-6 of Figure 10, for every VNF instance in the service graph, the mapped variable is set to *false*. If the *node* variable points to a SAP of the request graph, then the integer identifier of its appropriate SAP node in the substrate network can be unambiguously determined by their matching names, as I have already demanded this from the algorithm input in a previous section. The *id_in_network* function returns the integer identifier of a request node's host or matching SAP. In the pseudo-code we can assume dynamic typing of the variables, so assigning a 2-tuple or a boolean value does not make any problem.

In the next three lines, link mapping is initialized similarly, but here we can handle them homogenously. The *mapped_link (link)* function[14] will return the substrate path where the *link* variable will be mapped, after the algorithm have finished.

Mapping one VNF

The main *for* loop lasts from line 10 to line 36. It iterates over all the subchain – subgraph pairs returned by the preprocessing algorithm. The next embedded *for* loop iterates on the edges of the actual subchain, indicated by *subc* in Figure 10. The two ends of the current request link are stored in *node1* and *node2*. As far as the preprocessing prioritized SAP-to-SAP chains in the subchain division process, the first *subc* values will start from SAPs. Later on, if everything goes well in the mapping procedure (every VNF will be able to be mapped somewhere), the following subchains will also start from either a SAP, or an already mapped VNF instance. In a few words, emphasizing the important consequence, the *id_in_network* function will always return a valid integer identifier value of a substrate node for every request node stored in *node1*.

The *node2* variable will usually point to a VNF instance, which is not yet mapped. So now we are ready for the *mapOneVNF* section, which lasts from line 14 to line 30 on Figure 10. The mapping process stores the best host seen before, for the actual VNF instance in the *best_host* variable, with the objective function value calculated for that VNF – host pair. *best_host* is initialized to infinite objective function value and an invalid host identifier.

In the *for* loop of lines 16-24, the algorithm iterates over all the nodes of the corresponding subgraph, indicated by *subg.nodes()* in line 16. The *all_paths* function returns a list of 2-tuples of substrate node integer identifier (variable *n*) and a substrate path leading to that substrate node (variable *path_to_n*). The paths calculated start from the *id_in_network (node1)* substrate node to every node of *subg*. I used the basic Dijkstra's algorithm with the edged weights, which were calculated by the Preprocessor_Algorithm (but not indicated in its pseudo-code on Figure 6). The edge weights are the quotient for a neutral bandwidth value and the actual available bandwidth capacity of the edge.

Line 17 formalizes what is required from a substrate node to be a candidate host for a specific VNF. First, *n* have to be a host, capable of running the VNF type of *node2*, and *n* must be inside the placement criteria of *node2* (which can be stricter than the nodes of the current

[14] In Python terms, we could also call it a dictionary.

subgraph, see previous section for explanation), and finally it shall have enough resource to foster the all the resource requirement components of *node2* (which are currently CPU and memory capacity).

I would like to draw the reader's attention to the collocation support here, because *node1* is always the first element of the *path_to_n* variable, and if *node1* is a host, it is also checked for hosting capability in the *for* loop.

Mapping one VNF is done by selecting the best host for that particular VNF instance, according to a composite preference value calculation, which is discussed in the followings.

Preference value calculation

The *objectiveFunction* in line 18, calculates a numeric value for the *node2 – n* mapping with a path of *path_to_n* from *id_in_network (node1)*. If the mapping is not available because not every link of *path_to_n* have enough bandwidth to host *fbwreq(node1, node2)* megabit per seconds, then the *objectiveFunction* returnes a negative value.

The return value of *objectiveFunction* is the weighted sum of two components. It determines the cost of the potential mapping, so its value shall be minimized. The first component is the sum of the link weights in *path_to_n* multiplied by the required bandwidth of the request link. The second component is the sum of preference function values of the actual load state for every resource component of the substrate node *n*. The less loaded a host is, the more likely the algorithm wants to choose it as the home of the VNF instance.

Splitting the traffic of the request link, which generates the bandwidth requirement, is not allowed. Sometimes a network traffic flow can be handled on multiple paths at the same time, but some flows not. I suppose that, if a flow could be split, multiple service chains could be given to achieve this, so my algorithm handles every flow unsplittable.

The general form of the preference value function, used in the composite calculation of *objectiveFunction*, can be seen on Figure 11. The function's domain is the real numbers in the interval of [0, 1], which determines the ratio (percentage, if multiplied by 100) of occupied and maximal capacity of a resource component of a host (e.g. 0.7 in a host for CPU, means that the host's 70% of CPU capacity is used).

$$f(x) = \begin{cases} (e+1)^{\frac{x-c}{1-c}} - 1\,, & if\ x \geq c \\ 0\,, & otherwise \end{cases}$$

Figure 11: The preference value function and its parameters.

The parameter *c* on Figure 11 determines what utilization value is the maximal, when we like load states equally preferable. The *e* parameter means what preference value a fully loaded host resource state should return to the algorithm. The two points on the function graph is connected by an exponential curve. A separate parameter pair of *e* and *c* can be given for every resource component (in our problem for CPU and memory). An example parameterization can be seen on Figure 12, with values *e=2.5* and *c=0.4*. I have used these specific parameters for both of the preference functions.

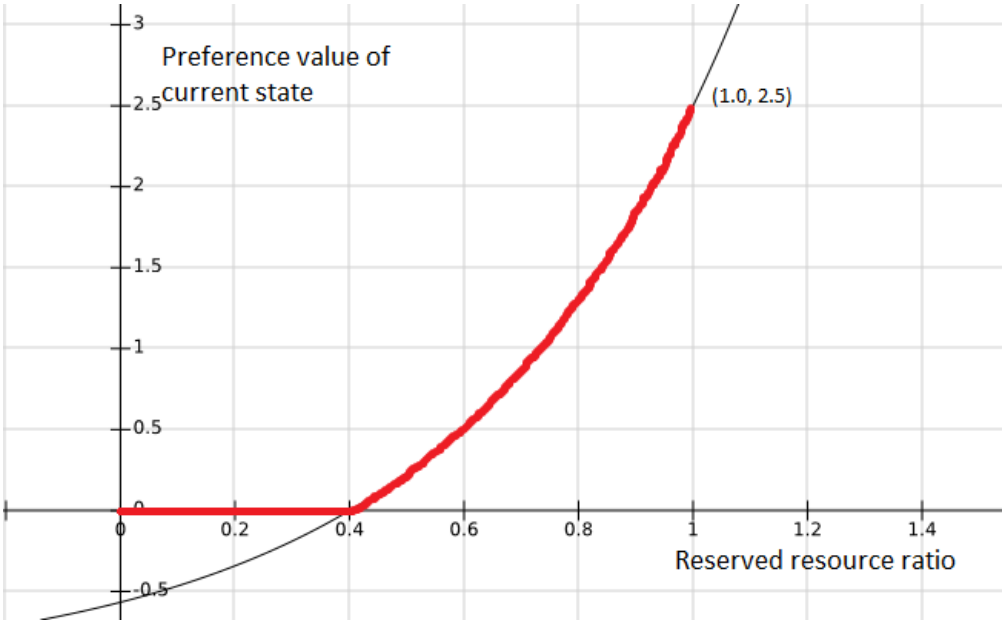

Figure 12: An example preference value function parameterization.

The preference value function is only C^1 continuous, which can cause a big "jump" in the values after the *c* parameter. This can be eliminated by well-proportioned function parameters or by replacing the function to some polynomial, which can operate on the whole interval. Moreover, a possible future work could be finding the optimal parameters, and maybe making them dependent of the topology or the resource conditions.

All in all, the maximally preferred host is chosen in every step of the mapping process. This is realized by storing all the significant data of a VNF – host mapping pair in the *best_host* variable, demonstrated by the *if-then-else* structure of lines 25-30 of Figure 10. After trying all hosts in the actual subgraph, the mapping is stored in *mapped* for the output, and the available substrate resources are updated with the just found mapping (see line 27). The *updateNetworkResources* function also updates the edge weights, as far as the available bandwidth capacities have decreased, even in the case of VNF collocation, the logical loop link's bandwidth and weight should be updated. If the algorithm could not find any possible

host for a VNF (see lines 28-29), then an error is raised. Backtracking on the possible mappings in this case would be an obvious and desirable improvement of the algorithm, which I plan to implement in the near future.

Lines 32-34 are executed when the algorithm reaches the end of a subchain, or in other words, *node2*, the actual VNF instance is already mapped or it is a SAP. In this case, only the link connecting the last two request nodes must be mapped to a path between their hosts. The *mapOneRequestLink* function finds such a path, and updates the data of its links.

Finally, the link and VNF mappings are forwarded to the lower architecture layer or returned to the network management system.

The termination of the algorithm is trivial in this part, because only for loop iterations are used with specific bounds.

The complexity of Core_Algorithm is composed of initialization and the mapping-objective function calculation, its rough overestimation can be seen on Figure 13, with the previously used notations.

$$O\left(\left|V_p\right| + \left|E_p\right| + \left|E_p\right|^2 |V|\,|E|\right)$$

Figure 13: The complexity of Core_Algorithm.

5.2 Evaluation and Testing

I have implemented my algorithm in Python 2.7.4 language and used the NetworkX Python library [22] for basic graph processing. The focus of this early stage of my research work was on getting acquainted with state-of-the-art solutions and designing and implementing my own algorithm. So thorough testing and parameter fine tuning still lay ahead. But still I present some basic test cases, and I propose an objective function for evaluating the actual load of all the resource components of the entire network, which could be used to compare specific mappings to each other.

5.2.1 An objective function

When I have discussed the publication of Chowdhury et al [16], I wrote that the problem with their objective function was that, it does not use every resource parameter of the network. So I have generalized (and a bit modified) it to design a function that fits better to our problem

definition. A generalized objective function for *R* types of resources and one link attribute can be seen on Figure 14.

$$
\begin{aligned}
&substrate\_graph(V, E),\\
&f_{r_i} : [0,1] \to \mathbb{R}^+, i = 1..(R+1)\\
&r_i : V \to [0,1], i = 1..R\\
&r_{R+1} : E \to [0,1]\\
&maximize \left\{ \sum_{i=1}^{R} \left(a_i * \sum_{u \in V} f_{r_i}(r_i(u)) \right) + a_{R+1} * \sum_{(u,v) \in E} f_{r_{R+1}}(r_{R+1}(u,v)) \right\},\\
&where \sum_{i=1}^{R+1} a_i = 1
\end{aligned}
$$

Figure 14: The general form of the objective function used for characterizing the network state.

The node and edge sets of the substrate graph are *V* and *E* respectively, represented on Figure 14. The r_i functions return the free resource ratio of the i-th resource type, this works the same way for the links, which are denoted as the *R+1*-th resource type. The f_{ri} functions return a preference value of the current utilization state of the given node or link. The preference values are summed for all network nodes and edges, and weighted sum of these sums gives the composed scalar value of the network state.

The a_i scalars denote the weights of the resource types, which can be used to control the importance between the different components. Their sum is optionally 1, but it could be used to scale the codomain of the objective function. In the case of weight sum of 1, the codomain depends on the size (node and link count) of the network. If we choose network size dependent value instead of 1, we can scale the objective function codomain to a normalized interval.

In our case, I parameterized this objective function with *R*=2, CPU and memory capacity, and bandwidth capacity as link resource. For the preference value function, I used the same function that the algorithm used for the mapping (they not necessarily should be the same), this can be seen on Figure 12. Naturally, the input of that preference function shall be transformed, because that function uses the occupied resource ratio, but here in the objective

function we need it to use free resource ratio[15]. Luckily, the transformation between them is easy: if *x* is the occupied resource ratio, then *1-x* is the free resource ratio.

In conclusion, this objective function measures how well the load is distributed, and generally also characterizes the total amount of load, because if we load the network more, it is likely that the objective function value will also increase.

5.2.2 Optimal searcher algorithm

I have also implemented a naïve and impractical (slow) algorithm, which is not interesting algorithmically, but my goal with it was to determine the optimal mapping in the sense of the objective function presented in the previous section. In other words, it finds a VNF mapping and a link mapping so that, the objective function value for the load state of the substrate network would be as high as possible.

In the first phase of the algorithm, it generates all VNF – host mappings, and for every mapping, it finds the optimal paths between the VNF instances, where a link is requested. After a possible full mapping is found, it calculates the objective function value, and continues to search for other full mappings. VNF and link mapping generation is realized by a naïve backtrack algorithm[16]. (It starts placing requests on the first available host/link, and updates resources; if it cannot place the next request, it frees up the previous request placing and tries on the next available host/link.).

This algorithm is very slow, and cannot be applied on real network topologies, which are generally very large, and we cannot expect this algorithm to finish its execution in a short period of time. So I have planned a small network topology, which can be handled even by this naïve algorithm to compare the objective function values of its mapping and the mapping returned by my algorithm. I used the input request graph, which I have presented earlier on Figure 7. The network topology can be seen on Figure 15. The graph visualization was generated by an algorithm of *matplotlib* [23].

[15] I had to use free resource ratio in the algorithm, because the preference value had to be minimized in conjunction with path cost leading to the given node. Achieving that, a long and fast path is always less preferred than a short and fast path.

[16] After a VNF mapping is found, link mapping could be implemented more effectively using integer linear programming by defining that request must not be split.

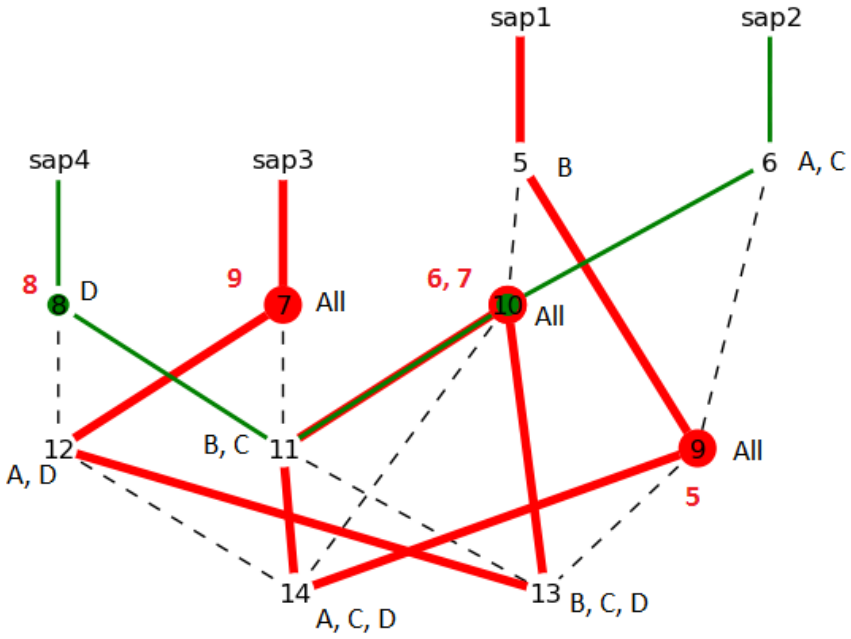

Figure 15: An example network topology
with the full mapping of the request of Figure 7, found by my algorithm.

The mapping of *Chain1* of Figure 7 is represented by the red (thicker) lines, *Chain2* is mapped to the green (thinner) links, and unloaded substrate links are represented by the dashed lines on Figure 15. The integer VNF instance identifiers of the request graph are written with red bold numbers next to the nodes. Only the list of the available VNFs are shown on the figure[17].

The subgraph found for *Chain2* could be 5-hops wide, *Chain1*'s subgraph was 36-hops wide, according to the actual parameterization. As we can see the algorithm utilized the freedom provided by the high latency requirement of *Chain1*. The request link between number 6 and 7 VNF instances is mapped inside the number 10 substrate node, because of the collocation of the VNFs.

The objective function (which was discussed in the explanation of Figure 14), after scaling its codomain to [0, 1000] by setting the sum of a_i, returned a value of 620.24. The naïve optimal searcher algorithm returned a bit better value of 670.24 and found the mapping of Figure 16, which we can consider the optimal mapping in the sense of the proposed objective function.

[17] I think all the VNF resource requirements and the node and substrate link resources would be too much data on the figure. I have verified the operation corresponding to resource checking.

```
VNF mapping:
 [(1, 1), (2, 2), (3, 3), (4, 4), (5, 14), (6, 10), (7, 10), (8, 10), (9, 7)]
Link mapping:
(1, 5, {'mapped_to': [1, 5, 9, 14]}),
(2, 6, {'mapped_to': [2, 6, 10]}),
(3, 9, {'mapped_to': [3, 7]}),
(4, 8, {'mapped_to': [4, 8, 11, 10]}),
(5, 6, {'mapped_to': [14, 11, 10]}),
(6, 7, {'mapped_to': [10, 10]}),
(7, 8, {'mapped_to': [10, 10]}),
(7, 9, {'mapped_to': [10, 13, 12, 7]})]
```

Figure 16: The direct output of the naive optimal searcher algorithm.

On Figure 16 VNF mapping is interpreted in "(VNF identifier, substrate node identifier)" format. I have selected integer identifiers of SAPs according to their names in both graphs (e.g. sap2 is identified by 2 in both graphs), but they are not necessarily have to be like this. As an example for decoding the output shown on Figure 16, VNF instance 5 was mapped to substrate node 14, and the link between *sap1* and VNF 5 was mapped to path 1, 5, 9, 14.

The optimal searcher algorithm gives a similar mapping to what my algorithm found, the objective function values do not differ too much. Both of them satisfies the requirements and balance the load, both of them agrees on the collocation of the VNF instances 6 and 7. Some paths are identical in both outputs; for example the request link between 7 and 9 are mapped to 10, 13, 12, 7 path. But the runtime of the naïve algorithm was a few hours even on such a small substrate graph, while my algorithm was executed in a blink of an eye.

One could ask, how my algorithm would behave on dense (redundant) graphs. The optimal searcher inspects all the paths between mapped VNF instances, which procedure in this case can go up to factorial complexity. To demonstrate the effectiveness of my algorithm, I have run it on 60 node full mesh substrate graph with generated request graphs with more than 70 VNF instances, the execution finished in a few seconds, while the naive algorithm would fail because it would take too much time to finish.

More thorough testing and planning other test cases are still wait ahead. I plan to model the inner network of a telecom provider with its real life parameters, to examine the execution, the returned mapping and the acceptance ratio (how many request graphs could it map) of my algorithm.

6 Conclusion

Initially, I have demonstrated the newest technologies, which provide the focus of today's networking related research fields. Software Defined Networking and Network Function Virtualization gave the novel optimization and design possibilities in networking. Breeding from these new technologies, Service Chaining enables researchers to develop new principles and architectures, which can make network description more abstract for human to understand better and for machines to optimize better.

There are European research projects that tries to exploit the possibilities provided by the previously mentioned technologies, and to develop an architecture to support full network virtualization and abstract description with service chaining.

According to up-to-date requirements, I have defined the problem of VNF mapping, and I have explored the state-of-the-art solutions of any useful related problem, involving Virtual Network Embedding, Virtual Data Center Allocation and graph pattern matching. I have decided to design my own algorithm utilizing the experience I have gathered from the recent research results.

Then I have presented my design choices and general decisions about the algorithm. I have also made the input and output specific enough for proper design and implementation.

I have presented and explained the approximation algorithm of my own design in details by analyzing the pseudo-code of both of its main parts: preprocessor and core. The algorithm searches for a subgraph of the substrate network where an entire service chain should be mapped. Then the algorithm divides the request graph into disjoint subchains to support the mapping process. Further placement criteria is given to VNF instances that are used by multiple service chains. The subgraphs are used the core mapping process as a heuristic to make the latency requirement delivering easier. Hosts are selected to the VNFs by a composite objective function consisting of preference value of the node and the cost of the path leading there. The mapping process can be directed and fine-tuned by many parameters.

The focus of my work was on research result exploration, algorithm design and implementation, but I have also presented some basic test cases to evaluate my algorithm. As one of the tools for this I have proposed a generalized objective function for measuring the preference of a mapping. I used this objective function to define and calculate the optimal

mapping on a sample input, and I compared the output of my approximation algorithm to the optimal value. Which result I have found satisfying so far.

My plans for future work are replacing the simplification of dynamic latency to hop-count handling; extending the algorithm with backtracking support (so it could try more possible mappings, if it fails, but keeping it in polynomial time); fine-tuning the parameters of the objective function by replacing them to heuristics depending on the networking environment; and finally running thorough testing on carrier scale networks and real topologies and requests.

7 References

[1] Dan Pitt, Rick Bauer, Cassandra Blair, Beth Most, Daisuke Saso, "Open Networking Foundation website," Open Networking Foundation, [Online]. Available: https://www.opennetworking.org. [Accessed 14. 10. 2014.].

[2] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, Jonathan Turner, "OpenFlow: Enabling Innovation in Campus Networks," 2008.

[3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, David Walker, "NetKAT: Semantic Foundations for Networks," in *POPL*, San Diego, CA, USA, 2014.

[4] Roy Chua, Matt Palmer, Craig Matsumoto, "SDN Central – The independent community & #1 resource for SDN and NFV," [Online]. Available: http://www.sdncentral.com/. [Accessed 17. 10. 2014.].

[5] Margaret Chiosi, Steve Wright, Don Clarke, Peter Willis et al, "ETSI NFV," 15-17. 10. 2013.. [Online]. Available: http://portal.etsi.org/NFV/NFV_White_Paper2.pdf. [Accessed 17. 10. 2014.].

[6] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, Felipe Huici, "ClickOS and the Art of Network," in *NSDI '14*, Seattle, WA, USA, 2014.

[7] Jim Guichard, Thomas Narten, Alia Atlas, "datatracker.ietf.org," [Online]. Available: https://datatracker.ietf.org/wg/sfc/charter/ . [Accessed 17. 10. 2014.].

[8] J. Halpern, C. Pignataro, "IETF," 20. 09. 2014.. [Online]. Available: http://tools.ietf.org/pdf/draft-ietf-sfc-architecture-02.pdf. [Accessed 17. 10. 2014.].

[9] András Császár, Róbert Szabó, "UNIFY - Unifying Cloud and Carrier Networks," [Online]. Available: https://www.fp7-unify.eu/. [Accessed 17. 10. 2014.].

[10] Amit Kumar, Rajeev Rastogi, Avi Silberschatz, Bulent Yener, "Algorithms for Provisioning Virtual Private Networks in the Hose Model," in *SIGCOMM'01*, San Diego, California, USA, 2001.

[11] Matthias Rost, Stefan Schmid, Anja Feldmann, "It's About Time: On Optimal Virtual Network Embeddings under Temporal Flexibilities," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, Phoenix, AZ, USA.

[12] Robert Soule, Shrutarshi Basu, Robert Kleinberg, Emin Gün Sirer, Nate Foster, "Managing the Network with Merlin," in *HotNets '13,*, 2013.

[13] Robert Soule, Shrutarshi Basu, Robert Kleinberg, Emin Gün Sirer, Nate Foster, "Merlin: Programming the Big Switch," 2013.

[14] Gabriel Valiente, Conrado Martinez, An algorithm for graph pattern-matching, Carleton University Press, 1997.

[15] Jiefeng Cheng, Jeffrey Xu Yu, Bolin Ding, Philip S. Yu, HaixunWang, "Fast Graph Pattern Matching," in *ICDE '08 Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, Washington, DC, USA, 2008.

[16] N. M. Mosharaf Kabir Chowdhury, Muntasir Raihan Rahman, Raouf Boutaba, "Virtual Network Embedding with Coordinated Node and Link Mapping," in *IEEE INFOCOM 2009*, 2009.

[17] Carlo Fuerst, Stefan Schmid, Anja Feldmann, "Virtual network embedding with collocation: Benefits and limitations of pre-clustering," in *Cloud Networking (CloudNet), IEEE 2nd International Conference*, San Francisco, CA, USA, 2013.

[18] B. L. Chamberlain, "Graph Partitioning Algorithms for Distributing Workloads of Parallel Computations," 1998.

[19] Chuanxiong Guo, Guohan Lu, Helen J. Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, Yongguang Zhang, "Secondnet: a data center network virtualization architecture with bandwidth guarantees," in *ACM CoNEXT*, 2010.

[20] Wenfei Fan, Xin Wang, Yinghui Wu, "Incremental Graph Pattern Matching," *ACM Transactions on Database Systems*, 2013.

[21] E. W. Weisstein, "Floyd-Warshall Algorithm," in *MathWorld*, 2009.

[22] Aric A. Hagberg, Daniel A. Schult, Pieter J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference (SciPy2008)*, Pasadena, CA USA, 2008.

[23] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Computing In Science & Engineering*, vol. 9, pp. 90-95, 2007.

[24] L. Rizzo, "Netmap: a Novel Framework for Fast Packet I/O," in *Proc. of USENIX ATC*, June 2012.