Efficient Service Graph Embedding: A Practical Approach

Balázs Németh, Balázs Sonkoly
Budapest University of Technology and Economics,
Budapest, Hungary
e-mail: {balazs.nemeth,balazs.sonkoly}@tmit.bme.hu

Matthias Rost
Technische Universität Berlin,
Berlin, Germany
e-mail: mrost@inet.tu-berlin.de

Stefan Schmid
Aalborg University,
Aalborg, Denmark
e-mail: schmiste@cs.aau.dk

***Abstract*—Future network services and applications, such as coordinated remote driving or remote surgery, pose serious challenges on the underlying networks. In order to fulfill the extremely low latency requirement in combination with ultra-high availability and reliability, we need novel approaches, for example to dynamically move network "capabilities" close to the users. This requires more flexibility, automation and adaptability to be added to the networks at different levels and operation planes. The key enabler of the novel features is network softwarization provided by NFV and SDN techniques. In this paper, we focus on a central component of the orchestration plane which is responsible for mapping the building blocks of services to available resources. Our main contribution is twofold. First, we propose a novel service graph embedding algorithm which is able to jointly control and optimize the usage of compute and network resources efficiently based on greedy heuristics. Besides, the algorithm can be configured extensively to obtain different optimization goals and trade-off running time with the search space. Second, we report on our implementation and integration with our proof-of-concept orchestration framework ESCAPE. Several experiments confirmed its practical applicability.**

I. Introduction

The recent paradigm shift in networking has been driven by Software Defined Networking (SDN) and Network Function Virtualization (NFV). SDN addresses the softwarization of the control plane, while the main goal of NFV is the softwarization of the data plane and of middleboxes (special purpose network elements). The combination of these techniques will provide benefits for the different stakeholders of the networking ecosystem in terms of flexibility, scalability, and costs (CAPEX and OPEX). Moreover, this softwarization approach is expected to be the enabler of future network services and e.g. 5G applications [1]. For example, the extremely low latency bound of the Tactile Internet – as small as 1ms round-trip time – will require novel further innovations in the orchestration of services as well as changes of the data planes. A typical network service consists of a series of service functions and is traditionally implemented by middleboxes, which have to be traversed in a given order by traffic flows. Due to the networking revolution driven by SDN/NFV and the evolution of cloud technologies, the concept of Service Function Chaining (SFC) has received much attention lately by both research and industry communities. SFC considers service chains (or more generally service graphs) as an abstraction to describe high level services in a generic way and to assemble processing flows for a given traffic.

According to the SFC approach, network services are virtualized on top of the underlying physical resources. Within any SFC architecture, the *Orchestrator* is the core component aiming at provisioning the shared resources of the infrastructure optimally while safeguarding the required high level service agreements [2]. The core of the resource orchestration lies in $i)$ finding suitable locations to host network functions while $ii)$ optimizing the overall resource allocations. The underlying mathematical optimization problem is closely related to the problem of Virtual Network Embedding (VNE), which is strongly $\mathcal{NP}$-hard [3]. In comparison to the VNE literature [4], we consider additional requirements which can be tricky to realize. In particular, we consider the definition of multiple delay requirements between end points for a single service chain and the sharing of virtualized network functions.

Algorithms for service orchestration can be categorized as either offline or online. Offline algorithms optimize over a large set of requests and seek optimal or near-optimal solutions, which typically comes at the expense of long runtimes. On the other hand, online algorithms process single requests arriving over time one after another providing solutions instantaneously and, hence they are typically implemented via heuristics.

In this paper, we propose a novel algorithm for service graph embedding which can jointly control and optimize cloud and networking resources focusing on modifiable objectives. Its runtime scales polynomially with the input size and we show that the obtained results are comparable to optimal solutions. It inherently supports network function sharing, making use of a simple resource model, i.e., network functions of new requests can be mapped to already instantiated ones, only slightly increasing resource consumption. As an important step from theory to practice, we have implemented the algorithm in our proof of concept orchestration framework called ESCAPE [5] and conducted several experiments.

The rest of the paper is organized as follows. Section II presents an application environment, in Section III, we introduce the problem of service graph embedding and main approaches for related problems. Section IV presents our algorithm in details. In Section V, the evaluation of the proposed mechanism is given, and Section VI concludes the paper.

978-1-5090-0933-6/16/$31.00 ©2016 IEEE

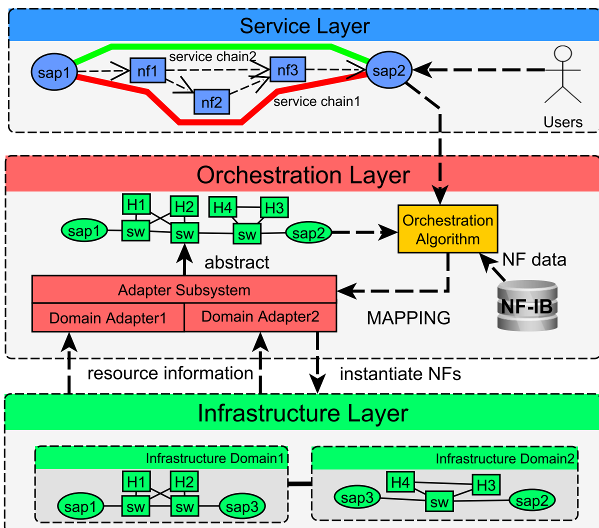


Figure 1: High-level view of the UNIFY architecture

Table I: Mathematical notations used in this paper.

Notation	Description
$\mathrm{V}(G), \mathrm{E}(G)$	Vertices and edges of G
$\mathrm{P}(G)$	All simple paths of G
$\mathrm{P}_S(G)$	Simple paths of G starting and ending in SAPs
T	Set of possible NF types
$s : \mathrm{V}(RG) \mapsto \mathbb{P}(\mathrm{T})$	Supported NF types
$t : \mathrm{V}(SG) \mapsto \mathrm{T}$	Function type of an NF
$\mathrm{R}(RG)$	Set of node resource types
$req_r : \mathrm{V}(SG) \times \mathrm{V}(RG) \mapsto \mathbb{R}_0^+$	Required and available resources of type $r \in \mathrm{R}(RG)$
$cap_r : \mathrm{V}(RG) \mapsto \mathbb{R}_0^+$	
$l_b, l_d : \mathrm{E}(SG) \mapsto \mathbb{R}_0^+$	Bandwidth and delay requirement of links
$a_b, a_d : \mathrm{E}(RG) \mapsto \mathbb{R}_0^+$	Available bandwidth and delay of links
$d : \mathrm{V}(RG) \times \mathrm{V}(RG) \mapsto \mathbb{R}_0^+$	Delay measured between hosts
$\Psi(SG) = \{c_p \mid p \in \mathrm{P}_S(SG)\}$	Set of E2E chains
$c_p = (B_p, D_p, p)$, $B_p, D_p \in \mathbb{R}_0^+$	E2E bandwidth and delay requirement on path p

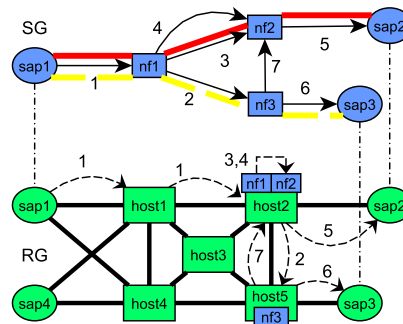


Figure 2: Illustration of Service Graph Embedding

II. Our background: UNIFY

In UNIFY[1], an EU-funded FP7 project, we proposed a novel SFC control plane architecture with a joint virtualization and resource programming interface unifying cloud and network resources [9]. Our SGE algorithm internally builds upon the UNIFY hierarchical resource abstraction, namely the “Big Switch with Big Software” (BiS-BiS) nodes, which generalizes the common Big Switch model used for topology abstraction by additionally indicating available resources. Hence, a BiS-BiS node may either represent a single physical node or a whole data center, hence providing a unified resource and capability model. Generally, a BiS-BiS may run any supported NFs and allows for arbitrary forwarding between infrastructure ports or NF ports and is the central component in our substrate representation. This approach allows recursive orchestration, i.e. a BiS-BiS node may have an internal orchestrator which handles its hidden resources to improve scalability.

The multi-layer hierarchy of the architecture is shown by Figure 1. The lowest level Infrastructure Layer encompasses the resources (both compute and network resources) from different technological domains. An abstract BiS-BiS based resource view is exposed by each infrastructure domain. These information is gathered by the Orchestration Layer and a multi-domain abstract resource view is constructed and periodically updated. The core component of this level is the orchestration algorithm including the task of service graph embedding. The service requests with predefined requirements come from the users via a dedicated Service Layer. The main goal of the Orchestration Layer is to map these requests to currently available (virtual) resources in an optimal way, and to initiate the deployment of the service.

Besides the architecture proposal, we have implemented a proof of concept prototype called ESCAPE [5]. It is a multi-domain orchestrator realizing the relevant parts of the UNIFY control plane architecture. The key component of the resource orchestration process is our proposed service graph embedding algorithm. Moreover, ESCAPE is capable of handling real-world domains abstracting physical resources, such as OpenStack and Docker or experimental infrastructure, such as Mininet. The inter-domain network resources are virtualized using VXLAN technology. ESCAPE was used in several demonstrations showcasing real use-cases with on-demand service creation and resource orchestration [10].

III. Problem definition

In this section we formally define the resource allocation problem – called the Service Graph Embedding (SGE) problem – considered in this paper. Concretely, similar to approaches in the VNE literature [4], services are modeled as graphs, where the network functions are modeled as nodes and communication between network functions are modeled using links. The overall goal is to find a mapping of the (virtual) nodes and edges of service graphs onto the shared physical substrate network, such that the cumulative resource allocations on any physical node or edge does obey capacity (and other) requirements. In the following, we formally define the model attended to in this paper.

An illustrative example for the SGE problem is given in Figure 2. A service request is given as a Service Graph (SG),

[1]http://www.fp7-unify.eu

consisting of arbitrary logical connections between service elements, called Network Functions (NF). A request can optionally include Service Chains (SC), which define QoS requirements, such as maximal allowed latency, on specific end-to-end paths of the Service Graphs. In Figure 2, SCs are denoted by red continuous lines and yellow dashed lines. The computing and networking resources are described by another graph, namely the Resource Graph (RG). In Figure 2, a mapping of the SG to the RG is shown, e.g. $nf1$ and $nf2$ are collocated on $host2$. Users or other domains are connected by Service Access Points (SAPs). The notations used in the subsequent sections are summarized in Table I. A mapping of an SG to the RG solving the SGE is a tuple of node and link mappings, denoted by μ and λ respectively, such that the requirements (1)-(7) are satisfied.

The mapped path of an SG link must start at the host of its source NF and end in the host of its destination NF (2), and all NF types requested in the SG have to be supported by the host node of the infrastructure (3). The capacity constraints on node resources (4) and on link resources (6) must be satisfied by the mapping, respectively. Link-wise latency requirements must be respected along the mapped path, as (5) indicates[2]. Service Chains (their collection is denoted by $\Psi(SG)$) are defined on paths between the endpoints (SAPs) of a SG, and their end-to-end (E2E) bandwidth requirements are handled in addition to link-wise bandwidth requirements, as shown by (6). The goal of the SGE is to minimize network resource utilization and maximize the number of embedded SGs.

$$\mu : \mathrm{V}(SG) \mapsto \mathrm{V}(RG) \quad \text{and} \quad \lambda : \mathrm{E}(SG) \mapsto \mathrm{P}(RG) \tag{1}$$

$$\forall e = (n_1, n_2) \in \mathrm{E}(SG) : \tag{2}$$
$$\mu(n_1) = \lambda(e).first \quad \text{and} \quad \mu(n_2) = \lambda(e).last$$

$$\forall n \in \mathrm{V}(SG) : t(n) \in s(\mu(n)) \tag{3}$$

$$\forall h \in \mathrm{V}(RG), \forall r \in \mathrm{R}(RG) : \tag{4}$$
$$\sum_{\{n | \mu(n) = h | n \in \mathrm{V}(SG)\}} req_r(n, h) \leq cap_r(h)$$

$$\forall e \in \mathrm{E}(SG) : \sum_{l_i \in \lambda(e)} a_d(l_i) \leq l_d(e) \tag{5}$$

$$\forall l \in \mathrm{E}(RG), M(l) := \{e_i | l \in \lambda(e_i) \wedge e_i \in \mathrm{V}(SG)\} : \tag{6}$$
$$\sum_{e_i \in M(l)} \left(l_b(e_i) + \sum_{\{B_p | c_p \in \Psi(SG) \wedge e_i \in p\}} B_p \right) \leq a_b(l)$$

$$\forall c_p \in \Psi(SG) : \sum_{e_j \in p} \sum_{l_i \in \lambda(e_j)} a_d(l_i) \leq D_p \tag{7}$$

Some existing VNE graph algorithms can be used to partially solve an SGE problem [6] ((1), (2) and (4) are also required by VNE), but generally, there are some aspects of SGE which are not supported by any VNE algorithm. Contrary to VNE, SGE considers the functional types of logical nodes and maps them according to substrate network capabilities (3). More importantly, the maximal allowed latency requirement on individual SG links or on E2E paths are considered ((7) in SGE definition), which is getting more attention from current research of service orchestration [1], [2], but is not yet well studied by most VNE algorithm so far [7].

[2]We note that the delay contribution of nodes on the mapped path is also considered in the actual implementation but is not considered in this paper to ease notation.

As a generalization of VNE, SGE is also $\mathcal{NP}$-hard [3], [7].

IV. Proposed Algorithm

To solve the problem of SGE, we have designed and implemented an algorithm making use of a greedy backtracking approach with coordinated node-link mapping. In other words, the mapping of nodes and links of service graph happens at the same time. Our algorithm offers many orchestration parameters and metrics to provide scalability and may hence be customized or tuned for other applications as well.

The orchestration algorithm greedily maps a Service Graph node and an adjacent link together in one step and we refer to these type of pairs as *leg*. Concretely, a leg is denoted by (e_n, n), where $e_n \in \mathrm{E}(SG)$ terminates in $n \in \mathrm{V}(SG)$ and the respective link is always mapped together with the node. Mapping a leg is considered as a single greedy step of the orchestration procedure. If a greedy step does not find a suitable embedding, then our algorithm backtracks to explore other possible mappings. The pseudocode of the main procedure is shown in Algorithm 1. The order in which legs are mapped is given by the Function DivideIntoSubchains() depicted in Algorithm 2.

Concretely, the mapping process iterates over the legs (e_n, n) and tries to map these based on customizable preference parameters using heuristics (detailed in MapOneNF() shown in Algorithm 3) as indicated in Line 7 of Algorithm 1. If the embedding step fails, PreviousLeg() is used to retrieve the previous SG link - SG node pair, both denoted by primed variables in Line 9. The existing mapping and resource reservation for (e'_n, n') is undone. In case of a successful mapping step, the resources are reserved for (e_n, n) and the mapping proceeds until all elements of SG are embedded to the Resource Graph and the result is returned.

In addition, the backtracking process can be customized by its depth limit and branching factor, meaning how many alternative *leg*s should be saved for further exploration. Hence, our algorithm allows for trade-offs between run time and the explored search space.

A. Basic Preprocessing Steps

Initially, the Preprocess() of Algorithm 1 prepares the input structures for the embedding process, and a helper structure is created. The end-to-end bandwidth requirement of Service Chains are added to link-wise bandwidth requirement of SG, so from now on bandwidth parameter B_p of all SCs $c_p \in \Psi(SG)$ are incorporated by l_b functions of SG links on path $p \in \mathrm{P}_S(SG)$.

Secondly, the mapping of SAPs from the SG can be done unambiguously to the SAPs of RG, so this is calculated in

Algorithm 1 Core SGE Algorithm

1: **procedure** MAP($SG, RG, \Psi(SG)) \hookrightarrow \mu, \lambda$
2: $SG, RG, \mathcal{G}^{RG}_{\Psi(SG)} \leftarrow$ PREPROCESS($SG, RG, \Psi(SG)$)
3: $\Psi_{div}(SG) \leftarrow$DIVIDEINTOSUBCHAINS($SG, \Psi(SG)$)
4: **for all** $q \in \Psi_{div}(SG)$ **do**
5: **for all** $(e_n, n) \in q$ **do**
6: **while** $\lambda(e_n) = \emptyset$ **or** $\mu(n) = \emptyset$ **do**
7: $success \leftarrow$MAPONENF(e_n, n, q)
8: **if** $\neg success$ **then**
9: $(e'_n, n') \leftarrow$PREVIOUSLEG()
10: Undo mapping of (e'_n, n')
11: $(e_n, n) \leftarrow (e'_n, n')$
12: **end if**
13: **end while**
14: **end for**
15: **end for**
16: **return** λ, μ
17: **end procedure**

Algorithm 2 SG Division Algorithm

1: **procedure** DIVIDEINTOSUBCHAINS($SG, \Psi(SG)$)
2: Sort $\Psi(SG)$ ascending by D_p
3: $SG' \leftarrow$COPY(SG)
4: $used(n) \leftarrow \begin{cases} T & \text{, if } n \in SAPs \\ F & \text{, else} \end{cases}$ **for all** $n \in \mathrm{V}(SG)$
5: **while** $\mathrm{E}(SG') \neq \emptyset$ **do**
6: **for all** $c_p \in \Psi(SG) \cup \{\emptyset\}$ **do**
7: **for all** $u \in$ ORDERED($\mathrm{V}(SG')$) **do**
8: **if** $c_p \in$ GETSCS(u) **then**
9: **choose** $e = (u, v) \in \mathrm{E}(SG')$, s.t. $c_p \in$ GETSCS(e)
10: **select** $q \in \mathrm{P}(SG')$, s.t. **for all**$(i, j) \in q$ holds
11: $q.first = u$, $used(q.last) = T$, GETSCS$(u, v) =$ GETSCS(i, j)
12: **for all** $w \in q$ **do**
13: **if** $used(w) = T$ **then**
14: SETHOSTRESTRICTIONS(w, GETSCS(u, v))
15: **end if**
16: $used(w) \leftarrow T$
17: **end for**
18: Remove all edges of q from $\mathrm{E}(SG')$
19: Add q to $\Psi_{div}(SG)$
20: **end if**
21: **end for**
22: **end for**
23: **end while**
24: **return** ORDERED($\Psi_{div}(SG)$)
25: **end procedure**

the PREPROCESS() method, stored in the μ structure and kept fixed during the whole orchestration.

The definition of the helper structure, calculated in the preprocessing stage, is denoted by $\mathcal{G}^{RG}_{\Psi(SG)}$ and formally introduced in (8). It contains an induced subgraph of the RG for each Service Chain, that excludes substrate nodes if they cannot potentially satisfy latency requirements with respect to the already mapped nodes.

$$\mathcal{G}^{RG}_{\Psi(SG)} = \{G_{c_p} | G_{c_p} \subseteq RG, \forall c_p = (B_p, D_p, p) \in \Psi(SG), \\ \forall h \in \mathrm{V}(G_p) : d(\mu(p.first), h) + d(h, \mu(p.last)) \leq D_p\} \quad (8)$$

The matrix d, denoting the shortest path distances, is calculated by Floyd-Warshall algorithm using delay a_d as the weight function for the edges of RG. This computation needs to be done only once for a substrate network and delay matrix d is cached for further orchestrations.

B. Division of the Service Graph into Subchains

The goal of DIVIDEINTOSUBCHAINS() procedure is to find a partition of $\mathrm{E}(SG)$ into subchains, and determine an ordering between the subchains, which will in turn determine the order in which the legs are mapped. Its pseudo-code is shown in Algorithm 2.

The iteration on the set of end-to-end SCs lasts until there are no more SG links in SG'. At least one SG link is removed in every iteration of the *while* loop, so the algorithm always terminates.

The GETSCS() function, used in Line 8, retrieves the set of end-to-end Service Chains whose path includes the given SG node or link, or $\{\emptyset\}$.

After a starting SG node u contained by Service Chain c_p of path p is found, an adjacent SG link (u, v) can be found, which is also contained in c_p. The SC set of link (u, v) is obtained by GETSCS(u, v), which is used for finding a subchain, starting from $u \in \mathrm{V}(SG')$ and ending in an already used SG node. So the path q is a subchain whose every link has the same set of contained Service Chains as its first SG link (u, v), as shown in Line 11.

If any SG node of q is already used by a previous iteration (already included by an earlier subchain), a placement criterion is set. In other words, Line 14 narrows the set of potential hosts of the current NF to the intersection of all RG subgraphs calculated for the involved end-to-end SCs. This way, the current node u will be mapped in the core process to a host which satisfies the end-to-end latency requirement of all involved SCs.

The next few lines maintain the helper structures $used$, SC' and the output subchain set $\Psi_{div}(SG)$.

If there are elements of SG, which are not in any end-to-end chain (i.e. GETSCS() returns $\{\emptyset\}$), they are also included in $\Psi_{div}(SG)$, but have low priority during the mapping procedure.

The output structure is sorted by a composite key consisting of the predecessor criterion and secondly by the end-to-end delay requirement. Concretely, the predecessor criterion between two subchains $c_q, c_p \in \Psi_{div}(SG)$ decides the order if, and only if $(q.first \in p)$ XOR $(p.first \in q)$ evaluates to true, i.e. that the start node of one of the service chains lies *within* the other service chain, but not vice-versa. In this case, the service chain containing the other's start node is prioritized. On the other hand, if this logical expression is false, then the ordering between the two subchains is decided by their end-to-end delay requirement. The chain with the lower end-to-end delay requirement[3] is stricter, so this subchain comes earlier.

[3]If multiple end-to-end chains contain a given subchain, then the lowest end-to-end delay requirement is taken into account.

Most importantly, the mapping order of NF and adjacent SG link pairs defined by Algorithm 2 maps the parts having the strictest requirements first to provide a less loaded RG for the greedy mapping. Thanks to the predecessor ordering, the first and last NF of a subchain will always be mapped (at least temporarily, which can be undone by backtracking) at the time of its embedding in Algorithm 1.

C. Metrics for Mapping Decisions

The pseudo-code of a single greedy mapping step is shown by procedure MAPONENF() in Algorithm 3. Its task is to map the given SG link - SG node pair to the best hosting RG path and node, in addition to saving some other good candidates for backtracking.

The algorithm uses the intersection of those subgraphs of RG, which correspond to the end-to-end chains involved by this subchain, and iterates on this $G_{sub} \subseteq RG$ to find potential hosts for n and path for e_n. In other words, G_{sub} defines the possible places where the current leg (e_n, n) can be hosted. This subgraph calculation is shown in Line 2.

The set of possible hosts is narrowed heuristically, the neglected RG nodes are considered to be too distant in terms of latency ($\mathcal{G}^{RG}_{\Psi(SG)}$ was calculated based on end-to-end latency requirements), and all the possible hosts of all other NFs of the current subchain are still in G_{sub}. So possible solutions are not lost, but complexity can be reduced dramatically by this easy step.

The shortest path from the host of the previous NF $\mu(n_1)$ to the RG node h under examination, is calculated using the reciprocal of available bandwidth as weight function on the edges[4].

Firstly, bad hosting paths and nodes are filtered, according to the SGE definition, in Line 8. Furthermore, the placement criterion of n is also tested here.

The REMAININGDELAY() function determines how much delay is left from the delay budget determined by the strictest involved end-to-end or link-wise delay requirement. Initially, the mapping budget is the end-to-end requirement itself which is decremented during the greedy mapping of the SC. The latency forward checking in Line 8 sorts out some possible hosts from G_{sub} to avoid some predictable backtracking step.

The objective function value of a mapping of (e_n, n) onto the path p and the host h respectively, is determined by the following three components:

- ϱ_{bw} – average link and node bandwidth utilization on path p;
- ϱ_{lat} – the mean of (1) delay used by path p divided by the remaining delay budget, (2) sum of delay distance from last used host ($d(\mu(n_1), h)$) and delay distance until the chain end ($d(h, \mu(q.last))$), scaled between $d(\mu(n_1), \mu(q.last))$ (i. e. the delay of the shortest path), and REMAININGDELAY(e_n, n, q);
- ϱ_{res} – weighted sum of preference values of resource utilization of type r on host h.

[4]The available bandwidth of nodes on the shortest path is also considered in the actual implementation.

Algorithm 3 Mapping of one SG Link and Node

```
1: procedure MAPONENF(e_n, n, q)
2:   G_sub ← ⋂_{sc_i ∈ GETSCS(e_n)} G_i where G_i ∈ 𝒢^{RG}_{Ψ(SG)}
3:   n_1, n_2 ← e_n; best_hosts ← ∅                    // n_2 is always n.
4:   if μ(n) = ∅ then
5:     for all h ∈ V(G_sub) do
6:       p ← Shortest path from μ(n_1) to h based on 1/a_b
7:       path_delay ← Σ_{(x,y)∈p} a_d(x, y)
8:       if Each link on p has l_b(e_n) bandwidth and Each resource
          requirement is satisfiable on h and t(n) ∈ s(h) and
          h complies to HOSTRESTRICTIONS(n) and d(h, μ(q.last)) ≤
          ≤ REMAININGDELAY(e_n, n, q) − path_delay then
9:         ϱ_bw ← GETAVGLINKNODEBANDWIDTHUTIL(p)
10:        ϱ_lat ← 1/2 path_delay/REMAININGDELAY(e_n, n, q)+
                   1/2 SCALEDDISTANCEFROMSHORTESTPATH(h, q)
11:        ϱ_res ← Σ_{r∈R(RG)} w_r φ_r(NODEUTIL(h, r))
12:        Add (p, h, αϱ_bw + βϱ_lat + γϱ_res) to best_hosts.
13:      end if
14:    end for
15:    best_hosts ← ORDERED(best_hosts)
16:    λ(e_n), μ(n) ← First element from best_hosts.
17:    Save elements from best_hosts for backtracking.
18:  else
19:    λ(e_n) ← Shortest path from μ(n_1) to μ(n) based on 1/a_b
20:  end if
21: end procedure
```

Component (1) of ϱ_{lat} ensures that one path does not consume too much delay, and (2) directs the greedy mapping towards the subchain end, $q.last$.

Preference value of resource utilization can be defined by any real function $\phi_r : [0, 1] \mapsto [0, 1]$, which should quantify how much a utilization state is preferred. The bigger the preference value, the less the state is preferred. The default function for all $r \in \mathrm{V}(RG)$ resource types is shown in (9).

$$\phi_r(u) = \begin{cases} 0 & \text{, if } u \leq 0.2, \\ \frac{3}{4}u + \frac{1}{4} & \text{, if } u > 0.2 \end{cases} \tag{9}$$

The (p, h) pair with the smallest objective function value is chosen for mapping, and a few other possible pairs are saved for backtracking purposes, as shown in Line 17.

The last SG node of the subchain always already mapped onto a host, i.e. $\mu(n) \neq \emptyset$, because it is either a SAP or the predecessor ordering provided by DIVIDEINTOSUBCHAINS() ensures this SG node is already handled by the greedy mapping. So if the execution is here the shortest path between the two RG nodes is chosen based on the reciprocal available bandwidth as edge weight, shown in Line 19.

V. EVALUATION

In addition to the presented algorithm, we have implemented a Mixed-Integer Linear Programming (MILP) based solution for the SGE problem, which maximizes the number of embedded SGs, while minimizing link bandwidth utilization. MILP provides a reasonable comparison basis for our evaluation, because it implements all the constraints (1)-(7) of SGE. Both orchestration algorithms have been evaluated on a real world topology taken from SNDlib[5] [8], which has 42 nodes and

[5]The dfn-gwin topology was used with additional network parts (6 nodes have been attached to an access network) and computing resources (another 6 nodes have been attached to a data center).

Table II: Performance comparison on SC sequences.

	Average result	15th %ile	85th %ile
MILP-based algorithm	207.37	200	216
Heuristic algorithm	131.57	119	163

157 edges, representing access, aggregation and core network parts, equipped with computation resources. All computation nodes had the same available resources (i.e. 400 units of CPU each).

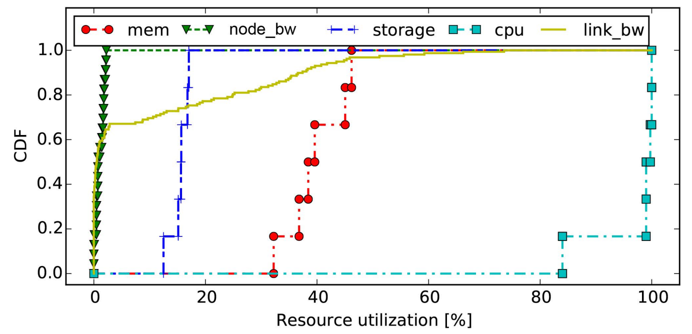
Simple Service Chains with 1 to 8 NFs connected between two SAPs have been generated with a uniform distribution. Similarly to the other requested resource types, the CPU values for each NF have been generated independently with uniform distribution between 1 and 4 units. The SCs had infinite lifespans and the Service Graphs can be disconnected.

To evaluate the performance of the proposed heuristic, we compare its embeddings to the optimal solutions of the (non-polynomial time) MILP. Concretely, our proposed heuristic tries to embed batches containing 4 service graphs as long as all requests can be embedded incrementally onto the shared RG. Hence, the embedding process aborts once a single request cannot be embedded anymore. Using the MILP as a baseline we compute the maximal number of successful consecutive embeddings possible in an offline fashion, i.e. the MILP may re-embed all previously given requests. The number of successfully mapped SCs can be used to compare the performance of the presented algorithm to the MILP-based solution for SGE.

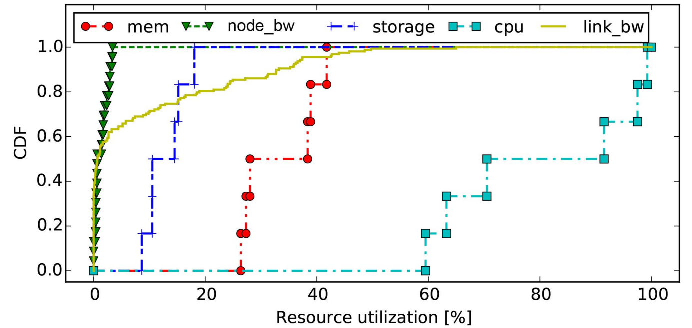
Firstly, we have executed every evaluation scenario with 100 different Service Chain sequences to provide a reasonable sample. Having the test cases ordered by the successfully mapped SCs, the first 15% is below 200 and the top 15% is above 216 in the case of the MILP-based solution. Similarly, the results for the heuristic algorithm are shown in Table II. In conclusion, our online heuristic can embed around the $\frac{2}{3}$ (varying between 59-75%) of the optimal, offline calculated number of requests.

Secondly, the Cumulative Distribution Functions (CDF) of resource utilizations of the network elements can be used to compare the quality of the two orchestration approaches. The goal of the mappings is to maximize the number of provisioned Service Graphs, and to balance the load on the network elements. The resource utilization CDFs of fully loaded states of the two approaches are shown in Figure 3. Figure 3a shows that the MILP-based approach balances the load well throughout the network, but as Figure 3b shows the heuristic algorithm does not fall much behind, especially in the case of less scarce resource types.

Finally, to evaluate the running time of the heuristic algorithm, we have simulated a scenario, when it had to map the SC sequence batched together into one big request, just like the MILP-based algorithm did in the previous examples. This also demonstrates the presented algorithm's capability to operate as an offline or semi-online *Orchestrator*. The simulation was conducted on computers with Intel Core i5 processors and 8 GB RAM. The results are shown in Figure 4, where the error bars represent the minimal and maximal running times



(a) Network state after mapping 212 SCs in one batch by the MILP-based algorithm.



(b) Network state after napping 180 SCs in batches of four by the heuristic algorithm.

Figure 3: CDF of resource utilizations after mapping as many SCs as the orchestration algorithms could.

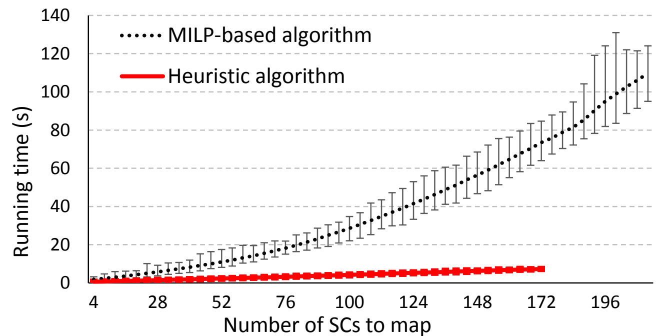


Figure 4: Comparison of the running times of MILP-based and heuristic algorithms.

among the 100 independent SC sequences. As we expected, the heuristic algorithm exhibits the polynomial scaling with the number of input service chains, while the MILP-based approach imposes impractical runtimes and exhibits worse scaling behavior.

VI. Conclusion

In this paper, we proposed a fast and flexible algorithm to solve the newly defined Service Graph Embedding problem. It supports different operation modes from online to offline operation with polynomial runtime and provides acceptable results compared to an MILP-based algorithm. Both the

"mapping quality" and performance results are promising based on our extensive Service Chain benchmark tests. The heuristic algorithm is tunable with many parameters to make the solution customizable in many networking environments. The algorithm has been integrated with our multi-domain NFV orchestration framework (ESCAPE) and it is ready to be deployed above real infrastructure, such as private data centers and ISP-sized networks.

As a future work, we plan to closely combine this algorithm with the implemented offline MILP module to realize a two-phase, hybrid operation. The incoming requests will be embedded on the fly by the online mapper, while the MILP solver running in the background will be responsible for reoptimizing the existing embeddings from time to time.

Acknowledgement

This research was supported by FP7 UNIFY, a research project partially funded by the European Commission under the Seventh Framework Program (grant agreement no. 619609), by H2020-ICT-2014 project 5GEx (grant agreement no. 671636), which is partially funded by the European Commission, by the German BMBF Software Campus grant 01IS1205, and by Aalborg University's talent project PreLytics.

References

[1] A. Vahdat. (2014). Enter the Andromeda zone, [Online]. Available: https://cloudplatform.googleblog.com/2014/04/enter-andromeda-zone-google-cloud-platforms-latest-networking-stack.html (visited on 07/12/2016).

[2] (2016). 5GEx White Paper, [Online]. Available: http://www.5gex.eu/wp/wp-content/uploads/2016/03/5GEx-White-Paper-v1.pdf (visited on 07/12/2016).

[3] E. Amaldi, S. Coniglio, A. M. Koster, and M. Tieves, "On the computational complexity of the virtual network embedding problem", *Electronic Notes in Discrete Mathematics*, vol. 52, pp. 213–220, 2016, INOC 2015 – 7th International Network Optimization Conference, ISSN: 1571-0653.

[4] A. Fischer, M. B. J. Botero, H. D. Meer, and X. Hesselbach, "Virtual network embedding: A survey", *IEEE Communications Surveys Tutorials*, vol. 15, no. 4, 2013.

[5] B. Sonkoly, J. Czentye, R. Szabó, D. Jocha, J. Elek, S. Sahhaf, W. Tavernier, and F. Risso, "Multi-domain service orchestration over networks and clouds: A unified approach", *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 377–378, 2015.

[6] C. Fuerst, S. Schmid, and A. Feldmann, "Virtual network embedding with collocation: Benefits and limitations of pre-clustering", in *IEEE 2nd International Conference on Cloud Networking, CloudNet 2013, San Francisco, CA, USA, November 11-13, 2013*.

[7] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba, "On orchestrating virtual network functions", in *11th International Conference on Network and Service Management, CNSM 2015, Barcelona, Spain, November 9-13, 2015*, IEEE Computer Society, 2015, pp. 50–56.

[8] S. Orlowski, M. Pióro, A. Tomaszewski, and R. Wessäly, "SNDlib 1.0–Survivable Network Design Library", in *Proceedings of the 3rd International Network Optimization Conference (INOC 2007), Spa, Belgium*, 2007.

[9] B. Sonkoly, R. Szabó, D. Jocha, J. Czentye, M. Kind, and F.-J. Westphal, "Unifying cloud and carrier network resources: An architectural view", in *Proc. IEEE Global Telecommunications Conference (GLOBECOM)*, 2015.

[10] R. Szabó, "Multi-domain/technology service function chain orchestration based on sdn and nfv", in *ETSI From Research to Standardization Workshop*, 2016.