Master's Thesis

Title

An Evolvable Approach for the Virtual Network Function Placement Problem in Varying Environments

Supervisor

Professor Masayuki Murata

Author

Mari Otokura

February 12th, 2016

Department of Information Networking

Graduate School of Information Science and Technology

Osaka University

Master's Thesis

An Evolvable Approach
for the Virtual Network Function Placement Problem
in Varying Environments

Mari Otokura

Abstract

Recently, communication services are becoming more diverse and dynamic with an increasing number of users who are connecting to communication networks. Network Function Virtualization (NFV) is an effective technique to deal with this situation; it enables the operator to change configurations of network functions dynamically. In the research field of NFV, many researchers have tackled the VNF placement problem as a facility location problem which decides the placements of virtual network functions (VNF) according to a specific objective. If we consider dynamic changes of processing requests from users, we have to also solve this VNF placement problem dynamically. The goal of the dynamic VNF placement problem is to reduce the cost for dynamic placement, such as reconfigurations of virtual machines (VMs) after changes of processing requests for VNF, in addition to optimizing each placement at a specific time according to the current objective.

To tackle this problem, we utilize knowledge from biological evolution. When organisms continuously evolve towards varying goals, they genetically adapt from the old goals to the new goals and finally find a common structure which only requires small changes for adaptation. Modularly Varying Goals (MVG), which is based on genetic algorithms (GAs), introduces the concept of biological evolutions in varying environments. MVG can generate individuals which can adapt to varying goals with only a few changes. Evolutionary Varying Goals (EVG) is a generalization of MVG which is not restricted to a modular structure. In this research, we propose 2 methods named Evolvable VNF Placement (EvoVNFP) and Evolvable VNF Placement-2 (EvoVNFP-2). We apply EVG and MVG to the VNF placement problem to reduce the number of reconfigurations after the changes of processing requests from users while keeping delay on traffic routes of the users short.

Evaluation results show that our method can reduce the number of reconfigurations of VMs when processing requests from users dynamically change in comparison to conventional GAs and Integer Linear Problem by generating placements which use a sufficient number of CPU cores to adapt to varying processing requests. Furthermore, it is shown that averages of the delay on traffic routes of users in our method are only a little increased compared to optimal values.

Keywords

Network Function Virtualization
Service Function Chaining
Dynamic VNF Placement Problem
Biological Evolution
Genetic Algorithms

Contents

List of Figures

List of Tables

1 Introduction

In recent years, the number of users who are connecting to communication networks is increasing in addition to an increase of their required bandwidths. Furthermore, communication services required from users become more diverse and dynamic. Considering the realization of Internet of Things (IoT) in the future, these trends will become even stronger. Previously, communication service providers utilize network functions, such as firewalls and intrusion detection systems, implemented by hardware to provide their services. However, it is difficult to deal with the increasing number of users and the diversification of the communication services by utilizing hardware solutions because it requires large capital expenditure (CAPEX) and operating expenditure (OPEX) to add and maintain new communication services.

Network Function Virtualization (NFV) is a suitable way to deal with this situation and it is currently being discussed in many organizations such as the European Telecommunications Standards Institute NFV Industry Specification Group (ETSI NFV ISG) [2–7]. The basic idea of NFV is to separate the network functions from physical computational resources such as CPU, memory, and storage [8]; the network functions are implemented in software and they are run on commodity physical computational resources. The network functions implemented in software are called *Virtual Network Functions* (VNFs) and are running in *virtual machines* (VMs) or containers. NFV enables us to change the performance of network functions dynamically because it manages and orchestrates the VNFs and physical computational resources, and can add and reduce the physical computational resources for VNFs. Furthermore, cooperation between NFV and *Software Defined Networking* (SDN), a virtualization technique which realizes dynamic change of traffic routes by separating control plane and data plane, enables us to provide several related network functions as a chain. This is called *Service Function Chaining* (SFC) [9].

In the research field of NFV, many researchers have tackled the VNF placement problem, deciding placements of VNFs according to a specific objective [10–16]. The VNF placement problem is an instance of the *facility location problem* [17] which is known to be an NP-hard problem although it can be formulated as an optimization problem. Furthermore, it is suggested in the ETSI NFV ISG technical specifications [18] that VNF can be broken into smaller parts leading to a higher complexity of the problem. Moreover, we have to solve the VNF placement problem dynamically in the SFC to deal with dynamic changes of processing requests from users [19].

Although there is much related work on the VNF placement problem, almost all of them focus on a static situation. In the dynamic VNF placement problem, we have to reduce the cost for dynamic placement, such as reconfigurations of VMs after changes of processing requests, in addition to optimizing each placement at a specific time according to the current objective. However, we have to add conditions to reduce the cost for dynamic placement to the conventional formulation if we formulate the dynamic VNF placement problem as an optimization problem, which will complicate the formulation even more.

To tackle this problem, we utilize knowledge from biological evolution; when organisms evolve towards varying goals, they genetically adapt to the new goals and finally find a common structure which only requires small changes when adapting between the goals. Biological systems have an inherent adaptability to varying environments, which derives from the modular organization of biological networks [20]. One of the factors of the modularity in biological networks is the evolution in varying environments [21]. Reference [22] proposes *Modularly Varying Goals* (MVG) based on a *genetic algorithm* (GA) to introduce the concept of biological evolution in varying environments. We also proposed *Evolutionary Varying Goals* (EVG) as a generalization of MVG [23], where we relax the modular structural view and relate the modularity to a temporal core-periphery structure. MVG and EVG generate individuals which can adapt to varying goals with only a few changes although they only use current states and do not use past and future states to calculate the fitness of the genomes. By applying MVG to the dynamic VNF placement problem, we are to generate placements which can adapt to dynamic changes of processing requests from users without making the problem overly complex, which facilitates the formulation as an optimization problem.

In this research, we propose 2 methods for the dynamic VNF placement problem for SFC named *Evolvable VNF Placement* (EvoVNFP) and *Evolvable VNF Placement-2* (EvoVNFP-2), which extends EvoVNFP by another time scale for adaptation. We can reduce the number of reconfigurations while keeping delays on traffic paths of users and the number of cores in the placement small. Here, the delays on the traffic paths correspond to the sum of propagation delays on physical links and queuing delays which occur in equipments.

The rest of the paper is organized as follows. In Section 2, we explain NFV and biological evolution, which form the background of this work. We then show the modeling and formulation of NFV systems for SFC in Section 3. Section 4 introduces the proposed methods EvoVNFP and

EvoVNFP-2 for the dynamic VNF placement problem. In Section 5, we show the results of the evaluation by simulations and discuss them. We conclude this paper in Section 6.

2 Background

2.1 Network Function Virtualization

In recent years, many researchers have shown significant interests in network virtualization as cloud computing, software defined networking (SDN), and network function virtualization (NFV) have been developed. Virtualization itself is not a new concept in domains of information technology; since the 1970s, we have observed trends of virtualization in many domains, such as memory virtualization or storage virtualization. For the domain of computer networking, we also have observed such trends, such as Virtual Local Area Networks (VLANs) and Virtual Private Networks (VPNs) [4]. These trends have lead to the appearance of cloud computing, SDN, and NFV.

NFV is the virtualization of network functions. Network functions perform some operation on incoming packets and are placed in the middle of the network. For example, firewalls intercept packets which have certain characteristics in their 5-tuple (source IP address, source port number, destination IP address, destination port number, protocol). Intrusion detection systems (IDS) monitor activities in networks, detect suspicious traffic, and report it to the network administrators. These network functions are conventionally implemented by hardware. In NFV, however, they are implemented by software, which are called virtual network functions (VNFs). Figure 1 shows a general architecture of NFV. The VNFs run in NFV infrastructures (NFVI) as they run in virtual environments, such as virtual machine (VMs) and containers, and these virtual environments run in physical environments. The VNFs and NFVI are managed by NFV management and orchestration (MANO). MANO manages which infrastructure each VNF runs and can change the configuration of the VNFs, such as changing the amounts of physical resources which are allocated to the VNFs. Furthermore, a VNF can be broken into multiple VNF components (VNFCs) [18]. A benefit of breaking a VNF into VNFCs is that we can reuse components which multiple VNFs have in common; for example, components of deep packet inspection (DPI) are used in many other security applications [5].

Benefits of NFV are summarized as follows:

- Cutbacks of capital expenditure (CAPEX): Since the VNFs are implemented as software the service providers no longer have to purchase expensive dedicated hardware devices.
- Cutbacks of operating expense (OPEX): NFV automates and simplifies construction and

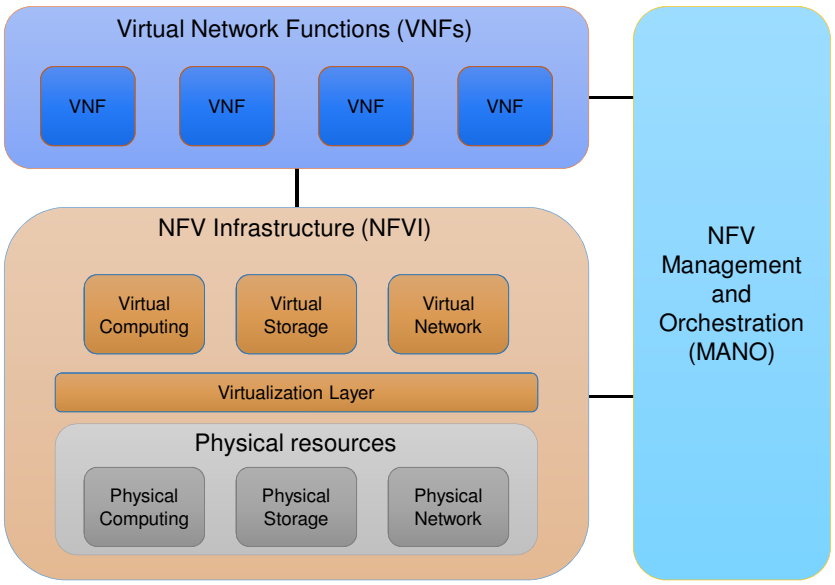


Figure 1: A general architecture of NFV [1]

maintenance of networks, which reduces the staffing costs.

- Cutbacks of energy consumption: Multiple VNFs can be integrated in a single physical machine.

- Enabling flexible and dynamic control of network functions: NFV can control the amount of the physical resources allocated to the VNFs dynamically and flexibly.

2.1.1 Service Function Chaining

Network functions are usually offered in sequence [11] even before the appearance of NFV; for example, traffic may need to be processed in the order of IDS, proxy, and firewall. Service Function Chaining (SFC) offers this sequence for each user as a sequence of VNFs: a *chain* in SFC terminology. This is realized by both SDN and NFV; SDN enables us to change the paths of chains dynamically and NFV enables us to change the scale of VNFs in the chains. Figure 2 shows a schematic view of SFC. There are 2 users, a network of a service provider, and a controller of the network. Service providers offer the chains according to requests of users in their network by using the controller. The network of the service provider exists at the edge of the Internet and directs its traffic to the core of the Internet. User 1 requests a chain of "VNF1 $\rightarrow$ VNF2" and the user 2

requests a chain of "VNF2 → VNF3." Then, the controller places VNF1, VNF2, and VNF3 in the network by using NFV technology and routes the traffic of each user according to their request by using SDN technology. Note that both of the users request VNF2 and therefore it is shared in the network among both chains.

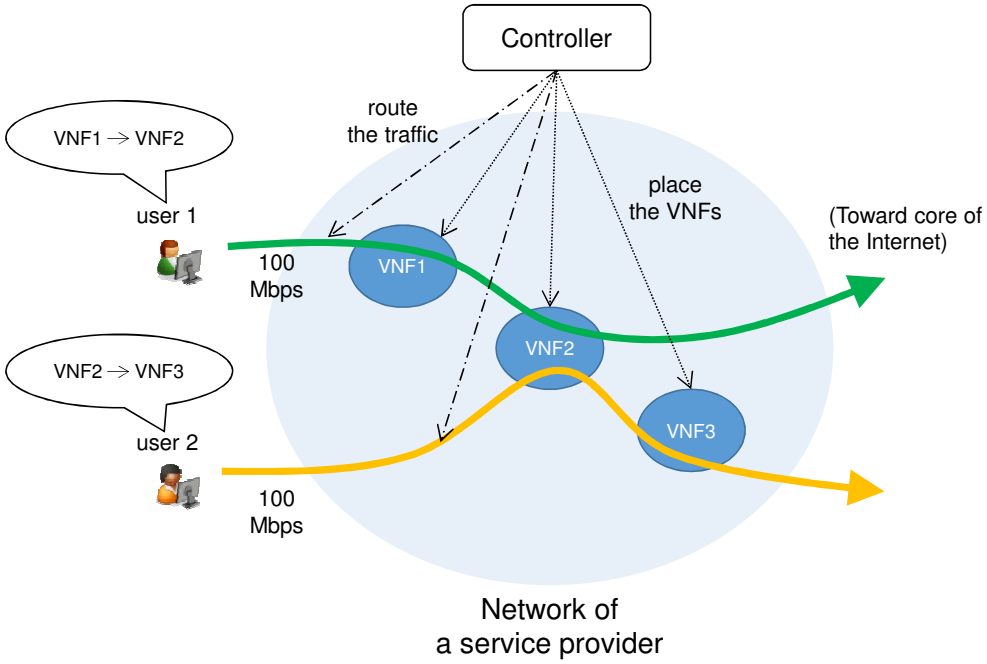


Figure 2: A schematic figure of two users requesting SFCs in the network of a service provider

2.1.2 Dynamic VNF Placement Problem

The control of NFV is modeled as a VNF placement problem which decides the placements of VNFs according to certain objectives, such as minimizing utilization of the physical network and minimizing delay arising on traffic paths. Such facility location problems have been known as NP-hard problems [12].

Furthermore, the requests of the users may change dynamically in the SFC. For example, user 1 changes his request in Fig. 3 from Fig. 2; the second element of the chain of user 1 is changed from VNF2 to VNF3. In this case, the controller reroutes the traffic of user 1 so that it goes through VNF1 and VNF3 and does not pass VNF2 anymore. Other examples are shown in Figs. 4 and 5, where user 1 and user 2 change their transmission rates from 100 in Fig. 2 to 300, which makes it

unable for VNF2 to process all the ingress traffic. There are 2 solutions to deal with this situation. One is shown in Fig. 4: the controller enlarges VNF2, i.e., it allocates more resources. The other is shown in Fig. 5, where the controller creates a new VNF2 and changes the route of user 1. The former one seems to be better in terms of the number of the reconfigurations. However, it depends on the future situation which is really better; for example, if the transmission rate of user 1 and user 2 continues to increase, the VNF2 in Fig. 4 may become unable to process all the ingress traffic and the controller finally must generate the new VNF2.

Therefore, in order to control SFCs, we have to solve VNF placement problems dynamically, which is called dynamic VNF placement problem. To provide a stable service to the users, we especially have to reduce the cost for dynamic placement, i.e., the number of reconfigurations of the placements in addition to optimizing each placement at a specific time in terms of utilization, delay, etc. A simple method of solving the dynamic VNF placement problem is solving the optimization problem which includes the number of reconfigurations in the optimization function; however, it will take much time to solve because it is an extension of the static VNF placement problem, which is NP-hard.

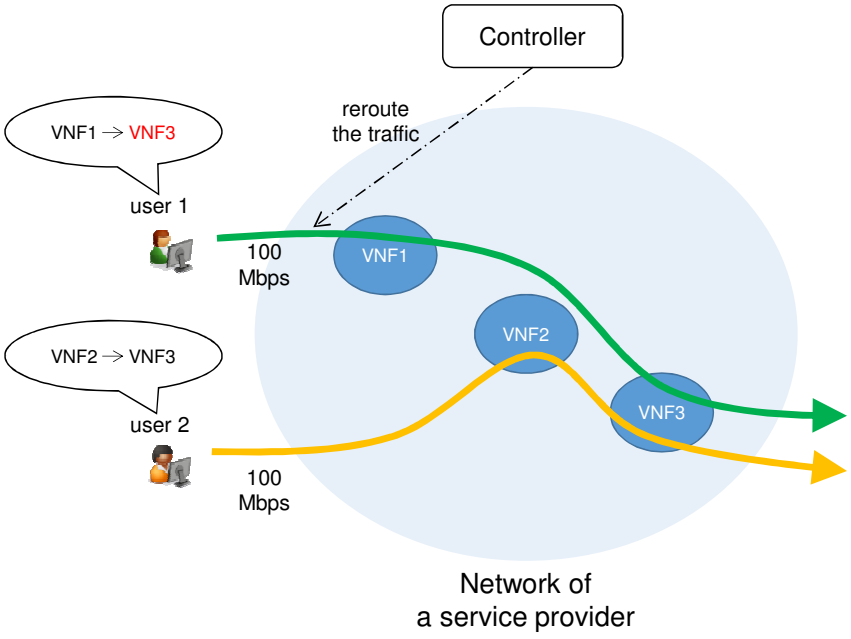


Figure 3: An example of changing the SFC for user 1 from VNF2 to VNF3 and rerouting the traffic

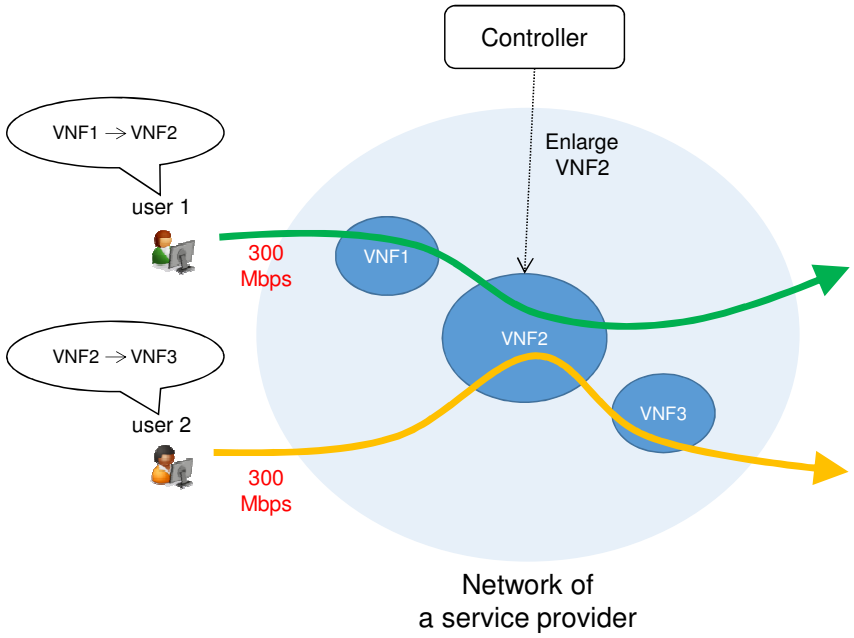


Figure 4: An example of enlarging VNF2 due to increased transmission rates by both users

2.2 Biological Evolution in Varying Environments

To tackle the dynamic VNF placement problems, we utilize knowledge from biological evolution. Biological systems have high adaptability to varying environments; they can easily adapt to new environments with only small changes of their structure. If we see their structure as a biological network, we can see a high degree of modularity, which is a property that the network can easily be decomposed into some modules which correspond to necessary functions. When the environments change, they only have to change the links between modules and do not have to change the internal structures of the modules. Although the reason why modularity emerged is not completely clear, a hypothesis is proposed in [22]: evolution in varying environments is the cause of modularity. Reference [22] validates this hypothesis by using a special genetic algorithm called Modularly Varying Goals (MVG).

We can see 2 analogies between the dynamic VNF placement problem and the biological evolution in varying environments. One is the varying environment in biology is similar to the varying requests of the users. Biological systems have to adapt to the varying environment and the controllers of the NFV systems have to decide suitable placements according to the varying requests

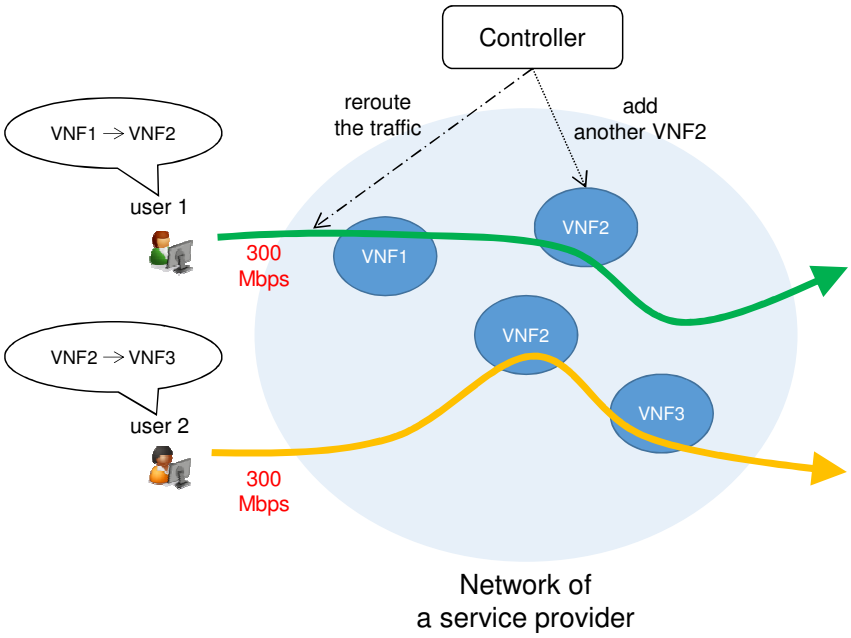


Figure 5: An example of adding another instance of VNF2 due to the increase in transmission rate

of the users. The other is the high adaptability of the biological systems to the varying environments and the requirement of the dynamic VNF placement problem. Biological systems can adapt to the new environments with small changes and we have to keep the number of reconfigurations small in the dynamic VNF placement problem.

In the following subsection, we will explain the concepts of conventional genetic algorithms, MVG, and Evolutionary Varying Goals (EVG), which is a generalization of MVG.

2.2.1 Genetic Algorithms

Before we explain MVG and EVG, we explain their basis: genetic algorithms (GAs) [24]. GAs are computational heuristics which imitate biological evolution, where biological systems evolve so as to adapt to an environment by repeating crossover, mutation, and selection.

Figure 6 shows a general algorithm of GAs. There is a main loop in the algorithm, which imitates lifetimes of the biological systems in the biological evolution. Each execution of the loop is therefore called *generation*.

Initialization of population In the beginning, we prepare a *population*, which is composed of

multiple *genomes*. The genomes are numerical strings and represent the solutions. The initial population is usually generated randomly.

Calculation of fitness We calculate *fitness* of how well a genome has adapted to the goal for each genome in the population by using the *fitness function*.

Selection We select the population for the next generation according to the fitness of each genome. There are many ways of selection, such as roulette wheel selection, ranking selection, and tournament selection. Additionally we can use elite selection, where the genomes which have the best fitness in the current population are directly selected for the next population.

Mutation/Crossover We change the genomes in the population by using *mutation* and *crossover* at a *mutation rate* and *crossover rate*, respectively. In mutation, we change the genome slightly, e.g., invert a bit of a genome represented as a binary string. In crossover, we combine 2 genomes, e.g., cut 2 parent genomes represented as a binary strings called G_A and G_B at the same point and create 2 new children genomes called $G_{A'}$, which is composed of the former part of G_A and the latter part of G_B, and $G_{B'}$, which is composed of the former part of G_B and the latter part of G_A.

Termination criteria are met? We judge whether the termination criteria are met, and if so, we stop the algorithm. The termination criteria are decided in advance. They could be defined for instance by reaching a predefined maximum number of generations or by a genome reaching a threshold value of best fitness.

2.2.2 Modularly Varying Goals

Reference [22] proposes MVG and introduces the concept of biological evolutions in varying environments to conventional GA. This is realized in MVG by providing multiple goals and switching between these goals during evolution. Note that MVG does not re-initialize its population when the goals switch. Figure 7 shows an outline of the algorithm of MVG. It is similar to Fig. 6 except for the blocks for switching the goals in Fig.7.

Furthermore, the goals prepared for MVG must be modular, i.e., they must have common parts. For example, in an example presented in [22], the authors used MVG to create logic circuits

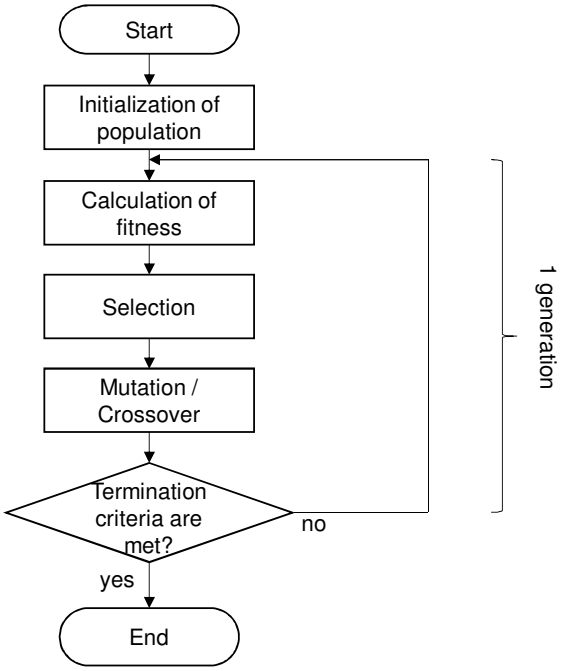


Figure 6: A flowchart of the general algorithm of GAs

according to corresponding boolean functions. In this case, the goals are the boolean functions shown below and they are switched every 20 generations:

- $G_1 = (X \oplus Y) \wedge (W \oplus Z)$
- $G_2 = (X \oplus Y) \vee (W \oplus Z)$

These goals share the common parts: "$(X \oplus Y)$" and "$(W \oplus Z)$." While evolving the logical circuits with MVG, they gradually obtain structures which have corresponding modules to the common parts; finally the logic circuit for goal G_1 can change to the one for goal G_2 by only exchanging the logical "$\wedge$" with an "$\vee$".

2.2.3 Evolutionary Varying Goals

In reference [23], we proposed Evolutionary Varying Goals (EVG) as a generalization of MVG. EVG and MVG are very similar as shown in the algorithm in Fig. 7. The only difference between EVG and MVG is that the goals prepared for EVG are not necessarily modular; they can be random ones which have only small changes.

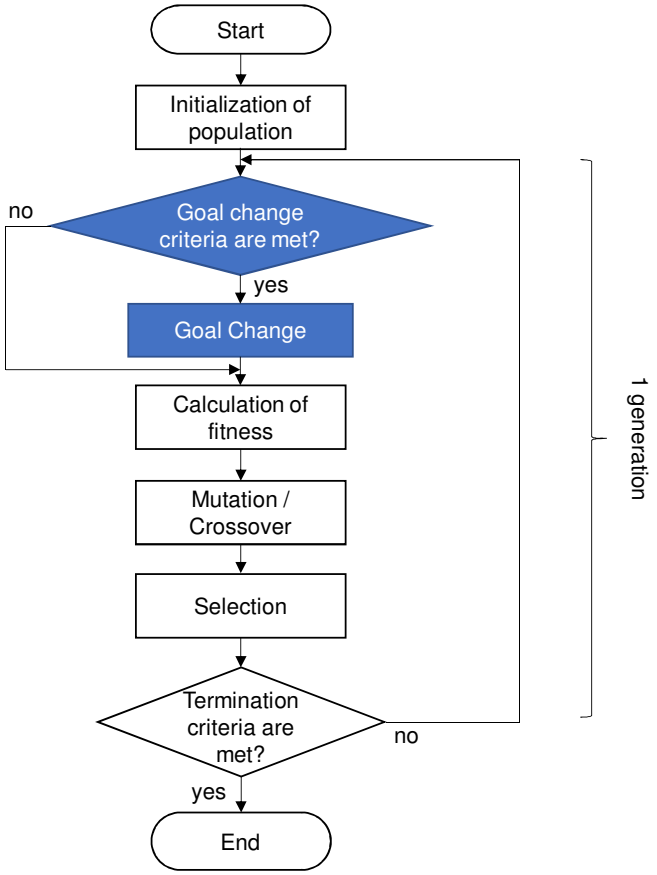


Figure 7: A flowchart of the general algorithm of MVG and EVG

We discuss in [23] that there is always a common part in the random goals if the changes between varying goals are small, which we can regard as the core of a temporal core-periphery structure [25]. While evolving the genomes with EVG, the resulting system gradually obtains a temporal core-periphery structure where the core corresponds to the common part and does not change at every switch of the goals. The smaller the changes between goals are, the larger the core is and the smaller the periphery is; it means that the genome for a goal can change to the next goal by only few genetic operations if the changes between goals are small.

3 System Model and VNF Placement Problem

3.1 Modeling of NFV Systems for SFC

In this section, we will explain the model of the NFV system for SFC which is used in this research.

3.1.1 Schematic View of the System

Figure 8 shows a schematic view of the system. In this system, users request service chains in the form of processing requests, which include information of ingress routers, egress routers, order of VNFs (chains), and transmission rates. The user in the figure requests a service chain of 100 Mbps which starts at r_1, goes though VNF1 and VNF2, and ends at r_3. When the controller of the system receives the processing requests for a VNF chain, it breaks the chain into components and then places them on the physical machines (PMs) in the physical network. In the figure, the controller breaks VNF1 and VNF2 into 2 components, respectively. It then places the decomposed chain in the physical network. The circles of the physical network r_1–r_5 represent routers and the squares PM1–PM10 represent PMs. Traffic of a user enters the physical network through its ingress routers, traverses all components in its chain in a specified order, and leaves through its egress routers. In the case of Fig. 8, the traffic of the user who sends the processing request enters through router r_1, sequentially passes through of VNF1 Comp1, VNF1 Comp2, VNF2 Comp1, and VNF2 Comp2, and leaves through router r_3. While the traffic passes through all components in the physical networks, 2 kinds of delays will occur: propagation delays and queuing delays. Propagation delay occurs when the traffic goes through links. Queuing delay occurs when the traffic is processed by network equipment, such as routers and VMs. The queuing delays can be analytically approximated because we assume that the VNFs are security applications, which continuously receive packets. In this work, we regard only CPU as the computational resources although other resources such as storage and memory are regarded as the computational resources in [8]. We assume that the components are placed on VMs and the VMs are placed on PMs in the physical network. When the VMs are placed on the PMs, the VMs occupy CPU cores of the PMs. Similarly, when the components are placed on the VMs, the components occupy CPU cores of the PMs. We can place multiple components on a VM. If there are insufficient remaining cores in the VMs or PMs, we cannot place the components or the VMs on them. A schematic view of the occupation of CPU cores is shown in the lower right of Fig. 8.

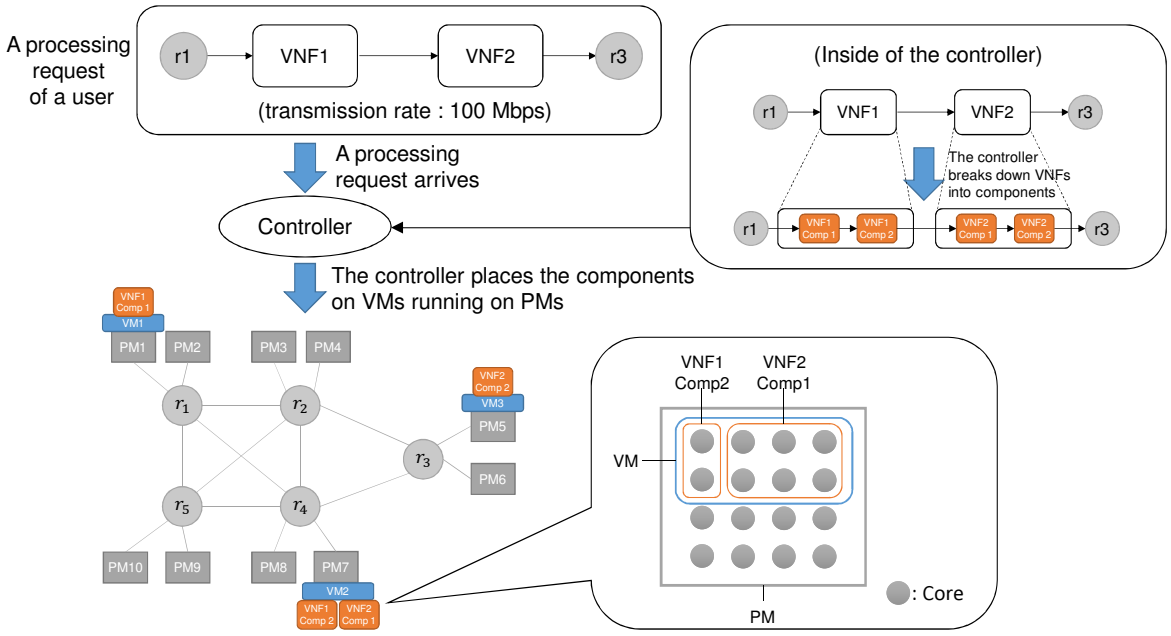


Figure 8: A schematic view of how VNFs are broken down into components that are placed on VMs at PMs

3.1.2 Definitions Related to Performance of a Component

To relate the number of cores of a component with its performance, we define the term of *processing amount*. Processing amount represents the workload of cores, such as the number of instructions which the cores can process. We furthermore define the term *processing capacity* as the maximum rate of processing amount which cores can process per second.

We assume that the performance of a PM and a VM proportional to the number of cores. Therefore, the processing capacity of a machine which has n cores is $n \cdot C$, where we define C as the processing capacity of a core.

3.1.3 Definition of Request Flows

We define *request flows* as flows of the traffic of the users who send *processing requests* to the controller. A processing request includes 4 kinds of information and is represented as $R_u = (src_u, dst_u, chain_u, b_u)$ for a user u: src_u is the ingress router, dst_u is the egress router, $chain_u$ is the chain of VNFs, and b_u (in bit/s) is the requested transmission rate.

Figure 9 shows an example of request flows. There are 2 request flows for users u_1 and u_2: $R_{u_1} = (r_1, r_3, [\text{VNF1} \rightarrow \text{VNF2}], b_{u_1})$ and $R_{u_2} = (r_5, r_3, [\text{VNF2}], b_{u_2})$. The 2 request flows come from their respective ingress routers r_1 and r_5, go through the components they need, and leave through the egress routers. Note that in this example they share the components of VNF 2; if multiple chains of processing requests include the same VNFs, their VNF components can be shared.

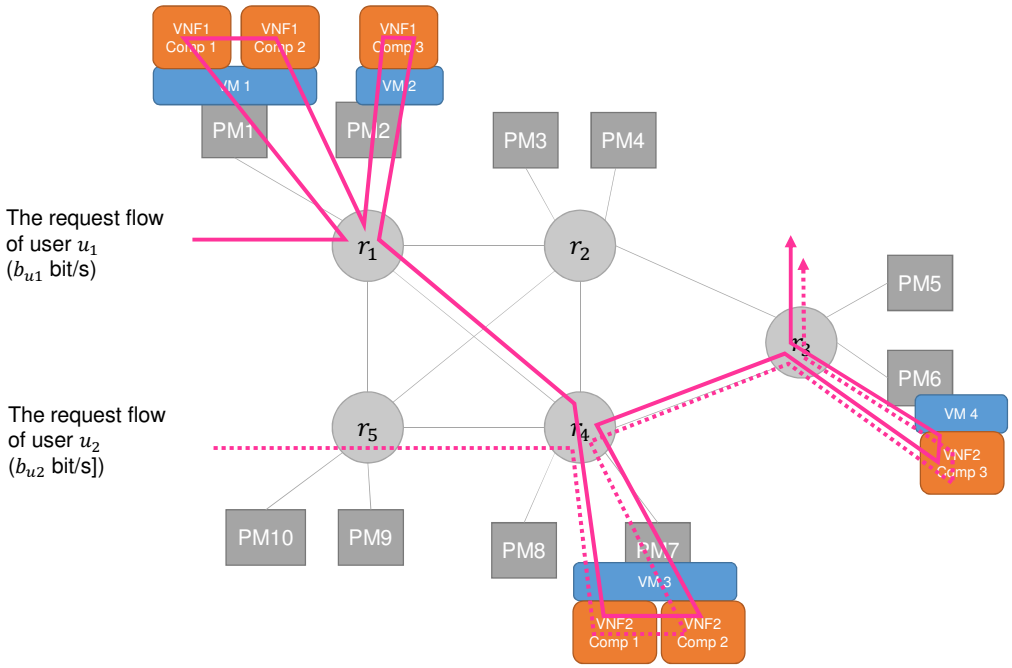


Figure 9: An example of two request flows and their allocated SFCs

3.1.4 Calculating the Delay of a Placement

As stated in Section 3.1.3, a request flow undergoes a delay which includes the queuing delay and the propagation delay. We assume that queuing delay occurs in both routers and components in this study. It is not constant because it depends on the current performance of the components and the traffic entering them. The propagation delay on the other hand is assumed as a constant delay when the request flow passes through a particular link. We define *request flow delay* t_u as the sum of all queuing delays and all propagation delays of the request flow of user u.

In order to consider the queuing delays analytically, we have to define service rates and arrival

rates of routers and components. We assume them as follows:

- Service rate of router r: The service rate of router r is a constant value M_r packet/s.
- Service rate of component j: The service rate of component j is $\mu_{k,j,a} = \frac{n_{k,j,a} \cdot C}{T_a}$ packet/s, where $n_{k,j,a}$ is the number of cores which the component j of VNF a occupies on VM k.
- Arrival rate at router r: The arrival rate at router r is $\lambda_r = \sum_{r'}(p^u_{(r,r')} \cdot b_u)$, where λ_r bit/s is the sum of the transmission rates of all request flows arriving at the router r and $p^u_{(r,r')}$ is a binary variable which is 1 when the request flow of a user u passes through a link between routers r and r' and otherwise 0.
- Arrival rate at component j: The arrival rate at component j is the sum of the transmission rates of all the request flows which request VNF a and it is represented as v_a bit/s.

For the sake of simplicity, we assume that each component behaves like an M/M/1 queuing system, i.e., packets in the request flow arrive as Poisson process and the service time is exponential.

The delay of a placement is calculated after the paths of the request flows have been decided. The delays of each traffic flow are calculated first and then they are averaged with weights corresponding to the transmission rates. We explain the calculation by using the example of the placement and the paths of request flows in Fig. 10. In the Fig. 10, $v_1 = b_{u_1}$ and $v_2 = b_{u_1} + b_{u_2}$. In an M/M/1 queuing system, the delay of a system with a single server and a single queue includes the waiting time in the queue and the processing time at the server. The average delay is $\frac{1}{\mu - \lambda}$, where μ is the average service rate of the system and λ is the average arrival rate at that system. We thus can calculate the queuing delay of each equipment by assigning the service rate and arrival rate of it. Then, for the example shown in Fig. 10, the request flow delay t_{u_1} of u_1 is

$$\begin{aligned} t_{u_1} = {} & d_{r_1} + d_{V1\_C1} + d_{V1\_C2} + d_{r1} + d_{V1\_C3} + d_{r1} + D_{(r_1,r_4)} + d_{r_4} \\ & + d_{V2\_C1} + d_{V2\_C2} + d_{r_4} + D_{(r_3,r_4)} + d_{r_3} + d_{V2\_C3} + d_{r_3} \end{aligned}$$

where d_x is the queuing delay of equipment x and D_{r_1,r_2} is the propagation delay of the link between routers r_1 and r_2. Finally, the delay of a placement $\hat{d}$ is calculated as the average of t_u weighted with the transmission rates of each user as:

$$\hat{d} = \sum_u (\frac{b_u}{\sum_u b_u} \cdot t_u).$$

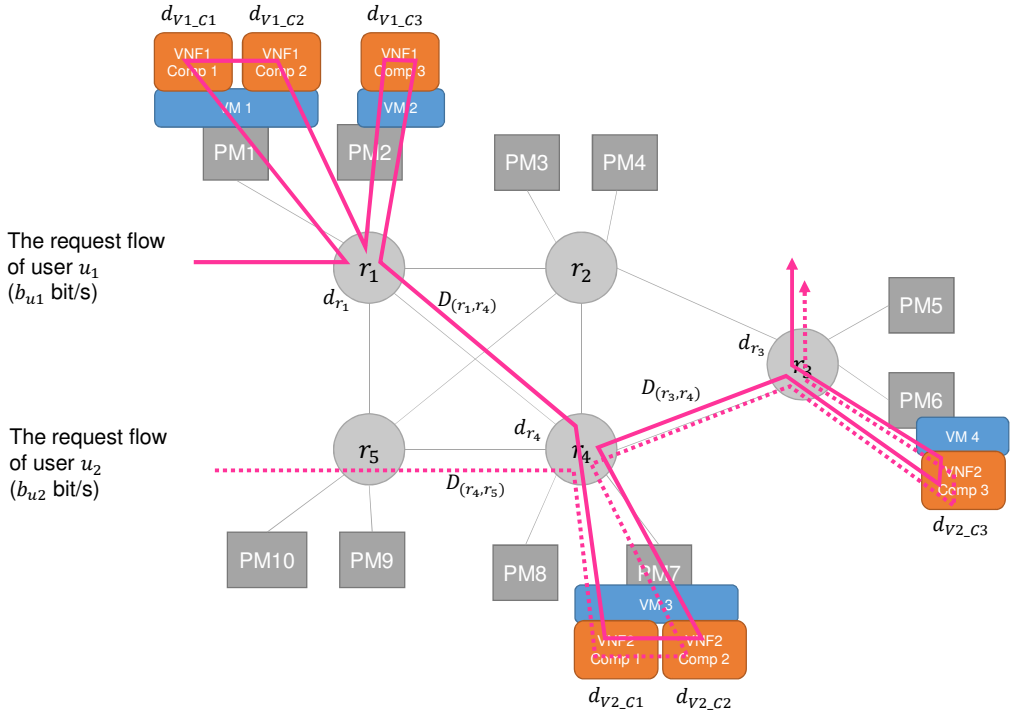


Figure 10: An example of the delays occurring in a placement

3.2 Formulation of VNF Placement Problem

In this section, we explain a simple formulation which does not consider the sharing of the VMs with multiple components of the VNF Placement Problem based on the model in Section 3.1. It is formulated as follows:

minimize:

$$\hat{d} + W \cdot \left(\sum_{i,k} m_{i,k} \right) \tag{1}$$

subject to:

$$\forall k, j, a \quad n_{k,j,a} \geq 1 \rightarrow T_a \cdot \frac{v_a}{S} < n_{k,j,a} \cdot C \tag{2}$$

$$\forall i \quad \sum_{k} m_{i,k} \leq N_i \tag{3}$$

$$\forall k, i \quad \sum_{j,a} n_{k,j,a} \leq m_{i,k} \tag{4}$$

decision variables: $m_{i,k}$, $n_{k,j,a}$, $p^u_{(r,s)}$

The term $m_{i,k}$ is the number of cores which the VM k occupies on PM i, W is a weighting coefficient which decides the importance of sum of cores, N_i is the number of cores of PM i, and T_a is the processing amount needed by VMs which have components of VNF a to process a packet: $T_a = \frac{C}{P_a/S}$. Here, P_a is the performance of the components of VNF a in a test environment where each component of VNF a has one core.

Figure 11 explains the constraints. The constraint (2) means that the controller must decide the number of cores for each components such that it has enough performance to process all the request flows which go through it. The constraint (3) means that the sum of the number of cores for VMs on a PM must not exceeds the number of cores of the PM. The constraint (4) means that the sum of the number of cores for components on a VM must not exceed the number of cores of the VM.

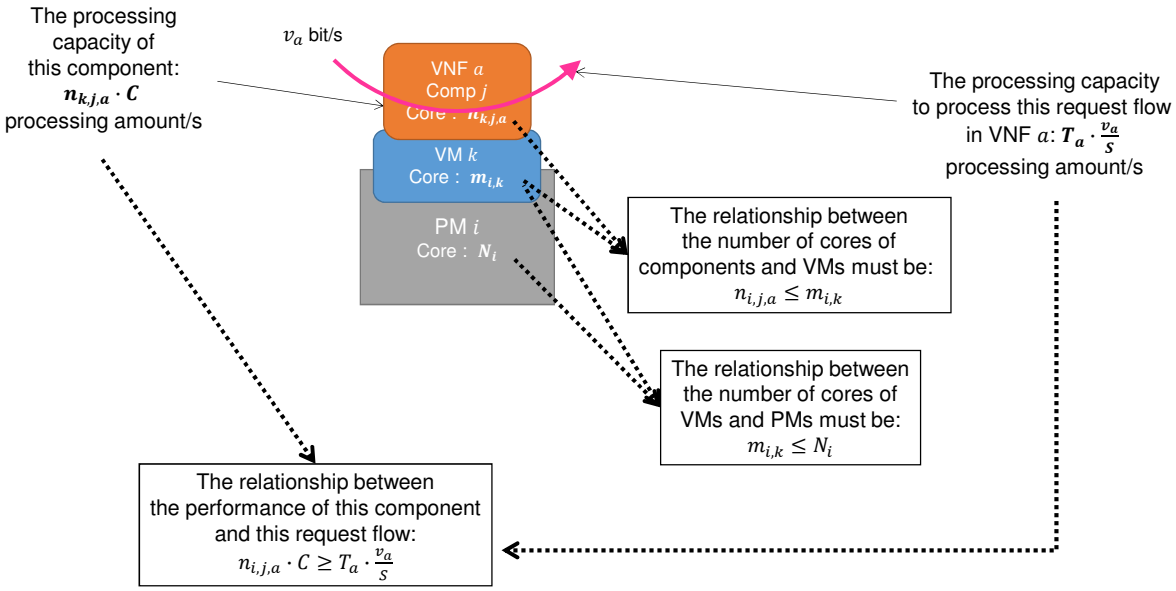


Figure 11: Graphical explanations of relationships related to the constraints in the formulation

3.3 Summary of Notation in the Model

In this section, we summarize the notation in the model in 3 tables.

Table 1: The variables used in the model and the problem formulation

Symbol	**Unit**	**Description**
b_u	bit/s	transmission rate of user u
v_a	bit/s	sum of the transmission rates of all request flows requesting VNF a
t_u	s	request flow delay of user u
$n_{k,j,a}$		number of cores occupied by component j of VNF a on VM k
$\mu_{k,j,a}$	packet/s	service rate of component j of VNF a on VM k
d_x	s	queuing delay in equipment (router or component) x
λ_r	bit/s	arrival rate at router r
$\hat{d}$	s	delay of placement
$p^u_{(r,r')}$		a binary variable which is 1 when the request flow of user u passes through a link between router r and r' and otherwise 0
$m_{i,k}$	-	number of cores which VM k occupies on PM i

Table 2: The parameters used in the model and the problem formulation

Symbol	**Unit**	**Description**
C	processing amount/s	processing amount which cores can process per second
P_a	bit/s	performance of VNF a in a test environment
S	bit	size of a packet
T_a	processing amount/packet	processing amount needed by VMs for components of VNF a to process a packet
M_r	packet/s	service rate of router r
$D_{(r_1,r_2)}$	s	propagation delay of link between router r_1 and r_2
W		weighting coefficient which decides importance of sum of cores in the minimization function
N_i		number of cores of PM i

Table 3: Other symbols used in the model and the problem formulation

Symbol	**Unit**	**Description**
R_u		processing request from user u
src_u		ingress router for user u
dst_u		egress router for user u
$chain_u$		chain of VNFs for user u

4 Methods for Dynamic VNF Placement Problem Based on EVG and MVG Principles

In this section, we will propose 2 methods named Evolvable VNF Placement (EvoVNFP) and its variant EvoVNFP-2, which uses adaptations on two time-scales, for the dynamic VNF placement problem based on EVG and MVG. We assume that users send new processing requests when they want to change the current processing requests in the dynamic VNF placement problem.

4.1 Overview of Proposed Methods

As we explained in Section 3, the controller receives the processing requests and decides the placement. We assume that the programs of the proposed methods run in the controller to decide the placement.

4.1.1 Evolvable VNF Placement (EvoVNFP)

Figure 12 is a schematic view of EvoVNFP. There are 3 processing requests R1, R2, R3 from users in the figure. When the control starts, the individuals of the genetic algorithm are initialized and the program starts with an objective which aims to meet all of the 3 processing requests. We call the number of generations from the change of objectives until the solution is obtained as an *epoch*. In this work, we assume that the length of epochs is shorter than the time intervals of the changes of the processing requests and obtain the solutions for current objectives at the end of each epoch. The solutions are then realized for the placement in the network. After obtaining the solutions for current objectives, the program waits for the next change of the processing requests. When the processing requests from users change, the objective changes without re-initialization of the individuals and the program continues its execution again.

4.1.2 Evolvable VNF Placement-2 (EvoVNFP-2)

Figure 13 is a schematic view of EvoVNFP-2. EvoVNFP-2 is derived from EvoVNFP and the difference between them is that EvoVNFP-2 includes more fine-grained changes of the objectives. Similar to EvoVNFP, the change of objectives is performed per epoch. These objectives are formed by a set of requests, e.g., R1, R2, R3 in Fig. 13, that remain constant during an epoch. Additionally,

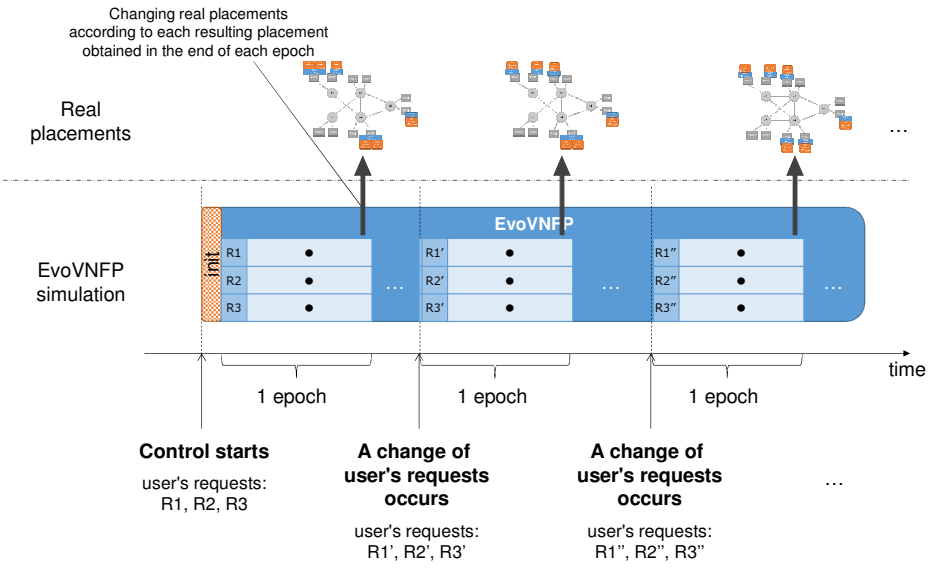


Figure 12: A schematic view of time-series processing of EvoVNFP

we now also introduce a smaller time unit called *period*. At each period, EvoVNFP-2 switches between 2 cases:

- The objective is to meet all processing requests (odd epochs).
- The objective is to meet all processing requests except for one which is selected randomly (even epochs).

This way of varying the objectives is inspired by the MVG model for RNA [26]. The authors aim to evolve the individuals toward an RNA structure with 3 hairpin loops. To do so, they switches the objective of the MVG between the 2 cases shown below:

- The objective is to make an RNA structure with 3 hairpin loops.
- The objective is to make an RNA structure with 2 hairpin loops except for the one which is selected randomly.

It is expected that the number of reconfigurations and the number of generations until obtaining a good solution will be smaller with EvoVNFP-2. The common parts of several varying goals emerge as module in the original MVG [22]. Therefore, if the objectives switch as described above, the part for all processing requests except for the one which is selected will emerge as

module. The module for each processing request then will emerge because all the requests can be selected as the excluded one. Finally, it is expected that each resulting module for the corresponding processing request will not influence each other so much.

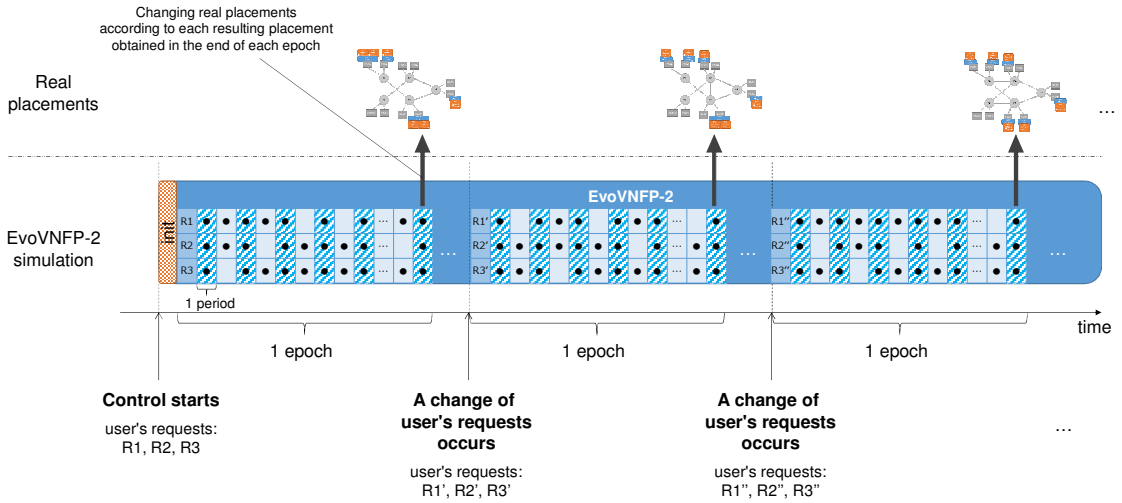


Figure 13: A schematic view of time-series processing of EvoVNFP-2

4.2 Design of Genetic Algorithm in the Proposed Methods

In this section, we will explain the genetic algorithm in the proposed methods in detail. We will discuss how an individual is encoded and how the mutation and fitness function are defined.

4.2.1 Design of an Individual

An individual in the genetic algorithm is a 3-layered network and represents a placement, i.e., components assigned to VMs, which in turn are assigned to PMs. Figure 14 is an example of an individual of the genetic algorithm representing the placement of Fig. 9. The first layer represents the PMs (PM layer), the second layer represents the VMs (VM layer), and the third layer represents the components (Component layer). Each node corresponds to each PM, VM, and component in the placement. A node in the VM layer and a node in the PM layer are connected if the corresponding VM is placed on the PM. Similarly, a node in the component layer and a node in the VM layer are connected if the corresponding component is placed on the PM. Furthermore, each node has the information about the number of cores which the corresponding PM, VM, or

component occupy. Once the placement is decided, the paths are determined as the shortest path of the all paths which go through all needed components.

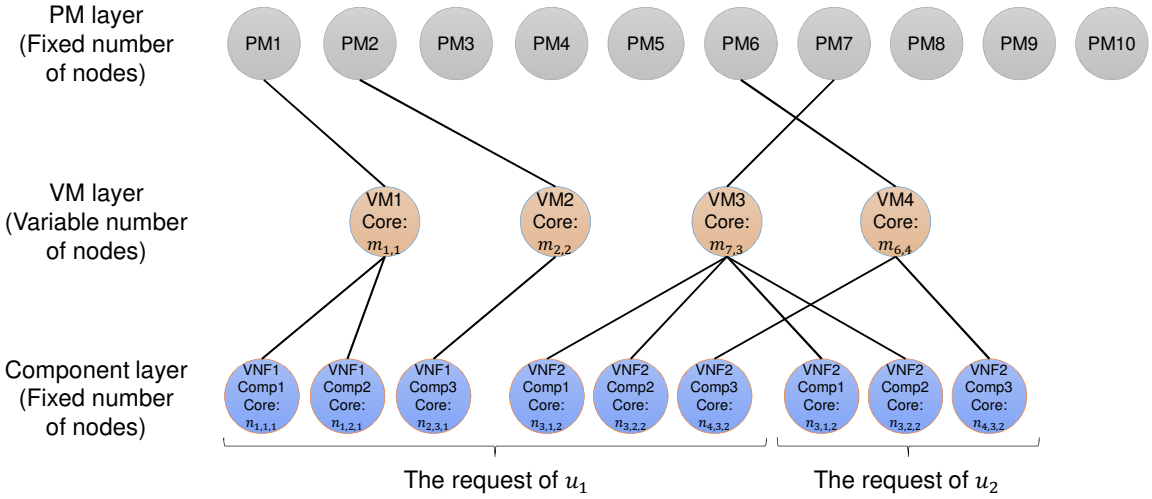


Figure 14: An example of an individual representing a placement by a 3-layered network

4.2.2 Fitness Function

The fitness function evaluates how well an individual matches the objective. In our genetic algorithm, it is a function which has the following properties:

- If the individual meets all the constraints shown in Eqs. (2)–(4), the fitness function is a positive value; the fitness F is calculated from Eq. (5), which is the reciprocal of Eq. (1).

$$F = \frac{1}{\hat{d} + W \cdot \left(\sum_{i,k} m_{i,k}\right)} \tag{5}$$

  It means the smaller the value of the minimization function is, the larger the value of the fitness function is.

- If the individual does not meet some of the constraints shown in Eqs. (2)–(4), the fitness F becomes a negative value. It is calculated from $F = (-0.1)Z$, where Z is the number of violations, so that the higher the level of violation of the constraints is, the lower the fitness is.

4.2.3 Mutation

In the mutation step of our genetic algorithm, an element of the network is changed slightly. We only use mutation to change individuals and do not use crossover. The mutation is performed for all individuals except for the elites at a predefined mutation rate. We randomly select one of the 5 kinds of operations shown below for the mutation:

- Changing a link between PM layer and VM layer: We select a link from all the links between PM layer and VM layer randomly and change the node for the PM of the link randomly.
- Changing a link between VM layer and component layer: We select a link from all the links between VM layer and component layer randomly and change the node for the component of the link randomly.
- Changing of the information about the number of cores: We select a node from VM layer and component layer and increment or decrement the number of cores by 1. This is done in such way that the number of cores does not exceed the maximum number of cores of PMs.
- Adding a node to VM layer: We select a node from VM layer and select a child of it. We then add a node to VM layer and connect the selected child to the new node. If the node selected first does not have any nodes after the process of mutation, the node is deleted.
- Deleting a node from VM layer: We select a node from VM layer, connect all its children to the other VM nodes randomly, and delete the selected node.

5 Evaluation

In this section, we will first explain the settings of the simulations and then present numerical results and their discussion.

5.1 Simulation Settings

First, we explain the parameters which we used in the simulation. The parameters of the genetic algorithms are shown below:

- Number of individuals: 1000
- Number of elites: 100
- Mutation rate: 0.8

We use the physical network shown in Fig. 15. The network is composed of 5 routers which have 2 PMs respectively. The circles r_1–r_5 represent routers and the squares PM1–PM10 represent PMs. Each PM has 16 CPU cores. The propagation delays of the physical link between 2 routers is 20 ms. In this work, we assume that the routers and their connected PMs are located in close proximity and therefore we do not consider the propagation delay on the physical link between a router and a PM.

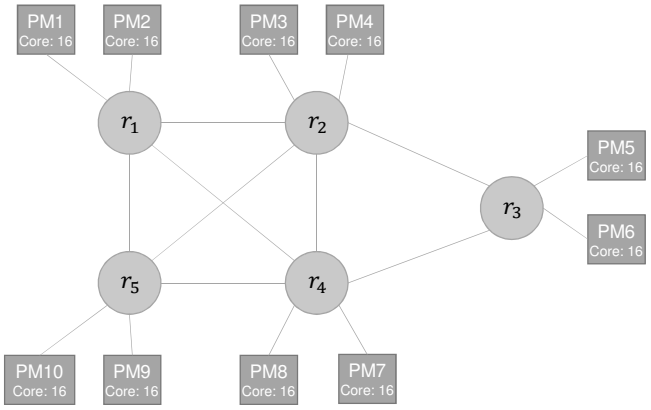


Figure 15: The physical network used in the simulation consisting of routers r_1–r_5 and PMs PM1–PM10

In the simulation, we use the 3 types of processing requests shown below:

- $R_{u_1} = (r_1, r_3, \{\text{VNF1}\rightarrow\text{VNF2}\}, b_{u_1})$
- $R_{u_2} = (r_5, r_3, \{\text{VNF2}\}, b_{u_2})$
- $R_{u_3} = (r_4, r_2, \{\text{VNF1}\rightarrow\text{VNF3}\}, b_{u_3})$

where b_{u_1}, b_{u_2}, and b_{u_3} are the transmission rates, see Section 3.1.3. In order to avoid too many changes in the requests which would make the results difficult to interpret, we limit ourselves at the moment to only changing the transmission rates b_{u_1}, b_{u_2}, and b_{u_3} at the epochs and keep the same settings of the source, destination, and VNF chain for R_{u_1},R_{u_2}, and R_{u_3}. The transition of b_u for a user u is performed as a random walk as shown in Eqs. (6) and (7):

$$b_u(0) = 500 \tag{6}$$

$$b_u(t_e + 1) = b_u(t_e) + \eta \tag{7}$$

where t_e represents the number of elapsed epochs and η represents the difference from the preceding value. η is generated according to a normal distribution with mean 0 and variance $(I/3)^2$, where I is a parameter of the strength of changes. We can obtain η from $[-I, I]$ at the probability of 99.74% by using this normal distribution.

The other parameters are shown below. For a description of the parameters, see Tables 1 and 2:

- $C = 3.0$ Gprocessing amount/s
- $P_a = 100$ Mbit/s for VNF1, VNF2, and VNF3
- $S = 1500$ bit
- $M_r = 3.0$ Gpacket/s
- $W = 0.5$

5.2 Reference Methods

We use 2 reference methods a conventional genetic algorithm (GA) and Integer Linear Problem (ILP), which we will briefly summarize in the following.

5.2.1 Conventional Genetic Algorithm

In the conventional GA, we rerun the normal genetic algorithm every epoch. A schematic view is shown in Fig. 16. Similar to EvoVNFP in Fig. 12, the objectives change every epoch. The major difference between Figs. 16 and 12 is that there are re-initializations of the individuals at the beginning of each epoch in Fig. 16 corresponding to running an independent GA for each epoch. We use this as a reference method to clarify the differences between our proposed methods and conventional GA.

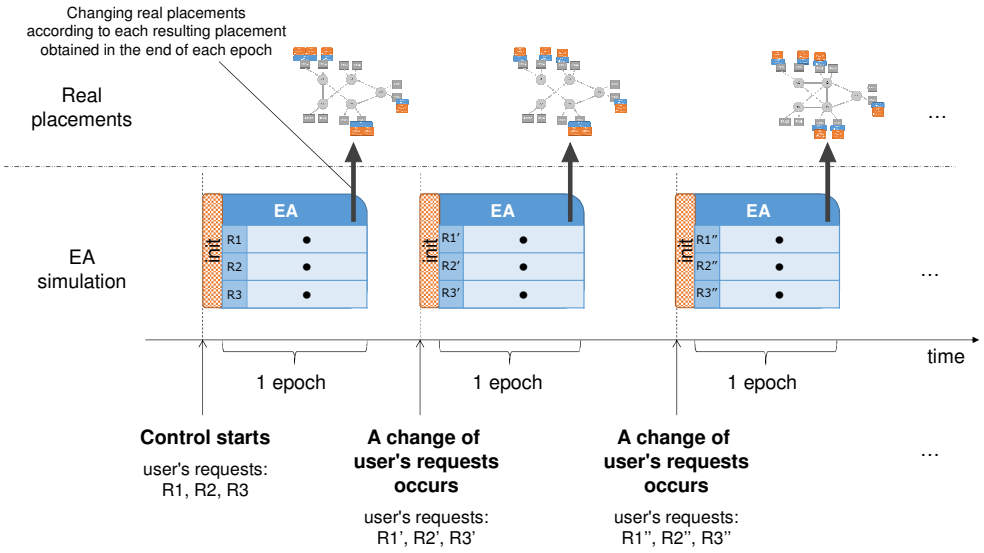


Figure 16: A schematic view of the conventional GA

5.2.2 Integer Linear Problem

ILP is a well-known optimization method which was used to represent the VNF placement problem in many references [10–16], although it is rarely directly used for solution because the problem is NP-hard. Most of the references proposed heuristics to solve the problem.

In this evaluation, we use ILP to decide the optimal placement at each epoch without considering the minimization of the number of reconfigurations. Due to the time complexity of the problem in Section 3.2, we decided to tackle the simpler problem as shown below.

- The routes of the traffic are prepared in advance and we choose the optimal routes from

them in the ILP of this subsection while we can choose arbitrary routes in the problem of Section 3.2

- The ILP of this subsection only considers the propagation delays as delays while the problem of Section 3.2 considers both of the propagation delays and queuing delays.

- The VM i can only be placed on PM i in the ILP of this subsection while all VMs can be placed on arbitrary PMs in the problem of Section 3.2

5.3 Evaluation Metrics

We use the 4 evaluation metrics shown below:

- The number of reconfigurations: we evaluate the number of reconfigurations which are needed to change placements between 2 successive processing requests. The reconfigurations include 6 operations: VM migrations, VM size changes, VM additions, VM removals, component migrations, and component size changes. VM migrations and component migrations mean that a VM or component which is placed on a PM i at epoch t is moved to another PM $j, j \neq i$ at epoch $t + 1$. VM size changes and component size changes mean that the number of cores of a VM or component is increased or decreased. VM additions mean that the number of VMs is incremented by one on a PM and removals mean that the number of VMs is decremented by one.

- Delay of a placement: we evaluate the weighted delay of each placement, described in Section 3.1.4, at each epoch.

- The number of cores in a placement: we evaluate the number of physical cores which are used by VMs in the placement at each epoch.

- The number of generations until obtaining the first feasible solution: we evaluate the number of generations until obtaining the first feasible solution in EvoVNFP, EvoVNFP-2, and GA. Here, a feasible solution refers to the individuals which have positive fitness, i.e., the individuals which can be converted to placements.

5.4 Results and Discussions

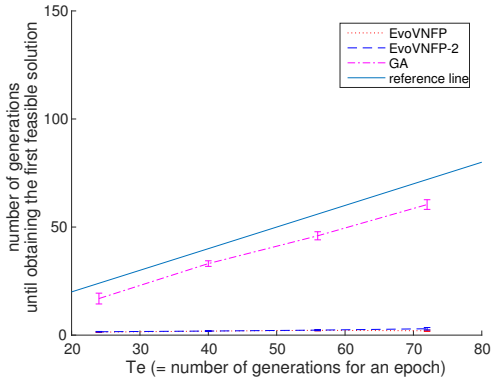
In this subsection, we show the results of the simulation and discuss them. We repeated the EvoVNFP simulation, EvoVNFP-2 simulation, GA simulation, and ILP simulation for each data point in the following figures 20 times. We used different transitions of the objectives in each simulation. An objective is composed of b_{u1}, b_{u2}, and b_{u3}, and we use 150 objectives (epochs) in each simulation. The result of a simulation is calculated from 100 placements which are obtained at the final generations of the 50th–the 150th epochs while the results from the first 49 epochs are considered as transient state and not included. If we cannot obtain any individuals which have positive fitness within an epoch, we do not consider this epoch. The results shown below are the averages of the results of the 20 simulations. The results of the EvoVNFP, EvoVNFP-2, the conventional GA, and the ILP are represented as "EvoVNFP", "EvoVNFP-2", "GA", and "ILP", respectively in the figures. Errorbars indicate the 95% confidence intervals.

5.4.1 Comparison between Different Pairs of T_e and T_p

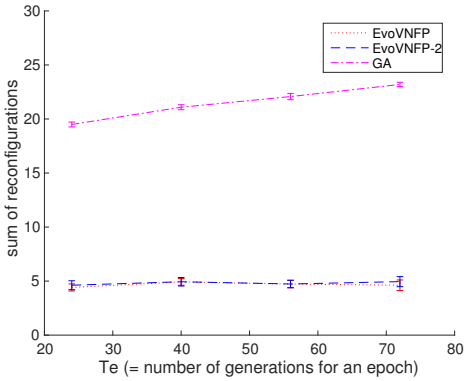
At first, we show the results of the comparison between the results obtained at different tuples (T_p, T_e), where T_p represents the number of generations for an epoch and T_e represents the number of generations for a period. Figure 17 is the results of the number of generations until obtaining the first feasible solution and the sum of the number of reconfigurations at $T_p = 8$. In order to make a fair comparison between the methods, we considered $T_e = kT_p$, with $k = 3, 5, 7, ...$ to always have a period with all three requests at the beginning and end of each epoch. Thus, in Fig.17 we computed the values for the tuples $(T_p, T_e) = (8, 24), (8, 40), (8, 56), (8, 72)$. Figure 18 illustrates the results at $T_p = 16$, and Fig. 19 is the results at $T_p = 24$, all having the same relationships between T_e and T_p. In the figures showing the number of generations until obtaining the first feasible solution, we also inserted reference lines with slope 1 and intercept 0, which correspond to the upper limits of the number of generations which can be used in an epoch.

We can see that the results of EvoVNFP and EvoVNFP-2 are very similar in all figures. By contrast, it is clearly shown that GA is the worst of the 3 methods in all figures. The results of GA are closest to the reference lines in Figs. 17(a), 18(a), and 19(a). It means GA uses almost all of the generations of the epochs to generate the feasible solutions while EvoVNFP and EvoVNFP-2 only require about 5 generations.

Figures 17(a)–19(b) also show that the larger T_e and T_p are, the larger the results of all methods become although the increases of the results of EvoVNFP and EvoVNFP-2 are very small. It means that the smaller the intervals between the changes of the objectives are, the better the performance of EvoVNFP and EvoVNFP-2 are relatively to that of GA.

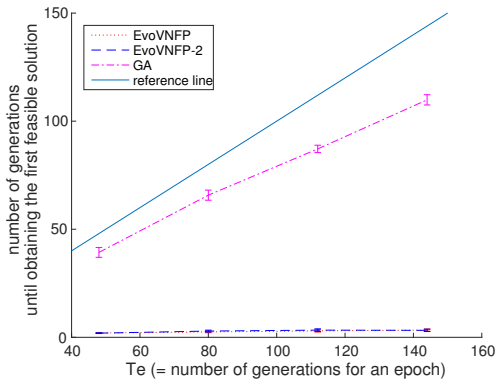


(a) The number of generations until obtaining the first feasible solution

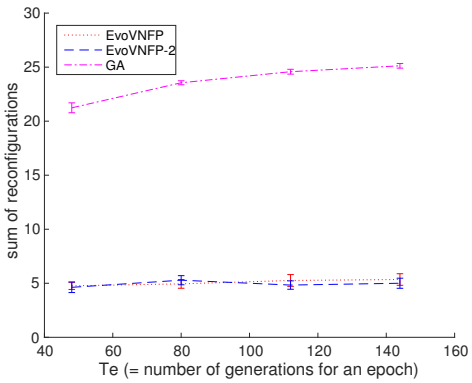


(b) The number of reconfigurations

Figure 17: Results of number of generations and reconfigurations at $T_p = 8$



(a) The number of generations until obtaining the first feasible solution

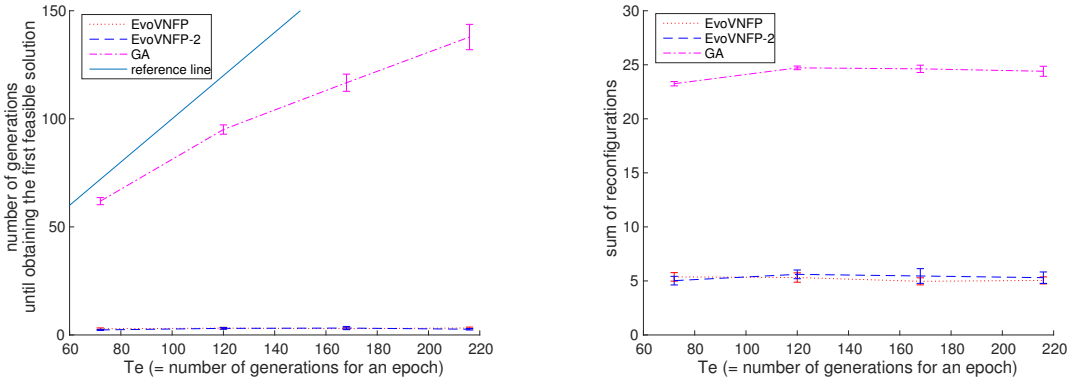


(b) The number of reconfigurations

Figure 18: Results of number of generations and reconfigurations at $T_p = 16$

5.4.2 Evaluation of the Number of Reconfigurations

In the following subsections, we will explain the detailed results by picking up the results at $(T_p, T_e) = (8, 24)$ and the results at $(T_p, T_e) = (24, 216)$, which have the lowest value and



(a) The number of generations until obtaining the first feasible solution

(b) The number of reconfigurations

Figure 19: Results of number of generations and reconfigurations at $T_p = 24$

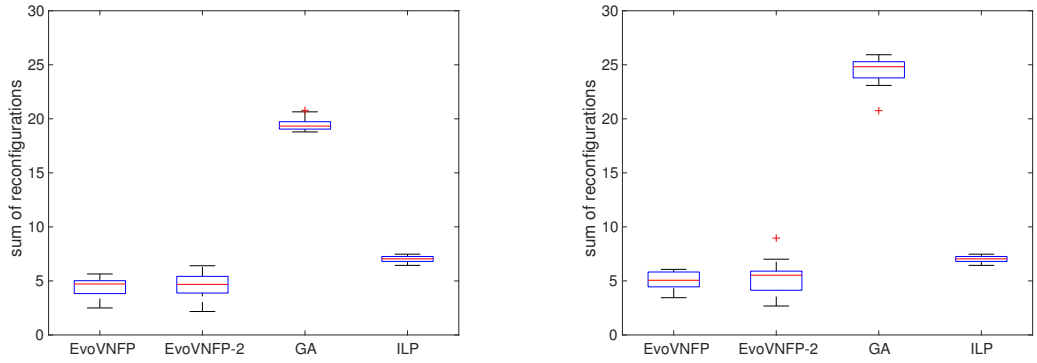
highest value of the T_e in the results in Section 5.4.1, respectively. Figure 20 shows the sum of the number of the 6 types of reconfigurations and Figs. 21–26 show the detailed results: the number of each kind of reconfiguration. The left figures are the results at $(T_p, T_e) = (8, 24)$ and the right figures are the results at $(T_p, T_e) = (24, 216)$.

The results at $(T_p, T_e) = (8, 24)$ and at $(T_p, T_e) = (24, 216)$ all have similar tendencies, but have slight differences. In Fig. 20, we can see that EvoVNFP and EvoVNFP-2 are smaller than GA and ILP on average. When we see the differences between Figs. 20(a) and 20(b), we can recognize that EvoVNFP and EvoVNFP-2 in Fig. 20(a) are a little smaller than those in Fig. 20(b). This is because the smaller length of the epochs prevent EvoVNFP and EvoVNFP-2 from overfitting to the objective at each epoch. The detailed results in Figs. 21–26 show that EvoVNFP and EvoVNFP-2 are better than ILP in terms of VM resizing, VM addition, VM removal, and component migration and they are worse than ILP in terms of component resizing. We will discuss this in Section. 5.4.5 more.

5.4.3 Evaluation of the Quality of the Best Placement at Each Epoch

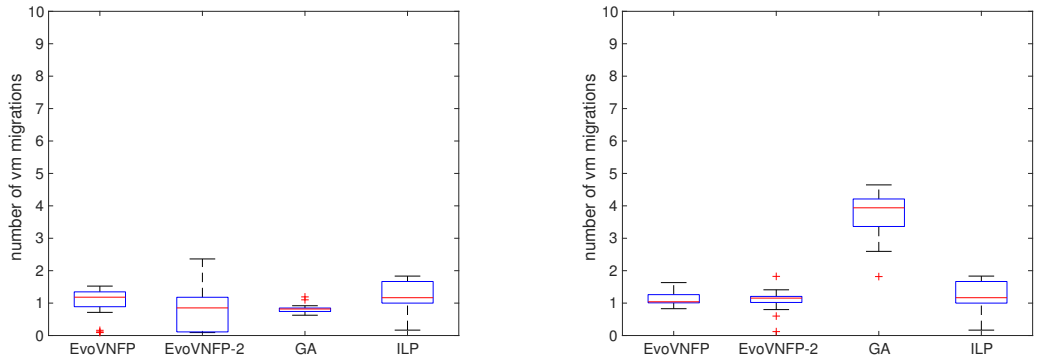
We then show the results about the quality, i.e., the number of cores and the delays, of the placement at each epoch. Figure 27 shows the results of the number of cores and Fig. 28 shows the results of the delays. The results of ILP can be regarded as the optimal values of the problem.

EvoVNFP, EvoVNFP-2, and GA require about twice as many cores as ILP in Fig. 27. On the



(a) $T_p = 8, T_e = 24$ (b) $T_p = 24, T_e = 216$

Figure 20: The number of reconfigurations



(a) $T_p = 8, T_e = 24$ (b) $T_p = 24, T_e = 216$

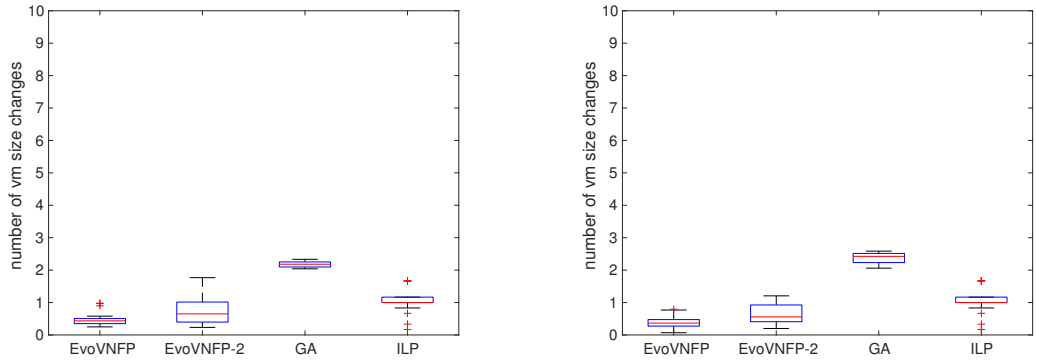
Figure 21: The number of VM migrations

contrary, EvoVNFP and EvoVNFP-2 are only a little worse than ILP in terms of the medians of the delays although GA is still worse than the other 3 methods. We will discuss these in Section. 5.4.5 more.

5.4.4 Evaluation of the Number of Generations until Obtaining the First Feasible Solution

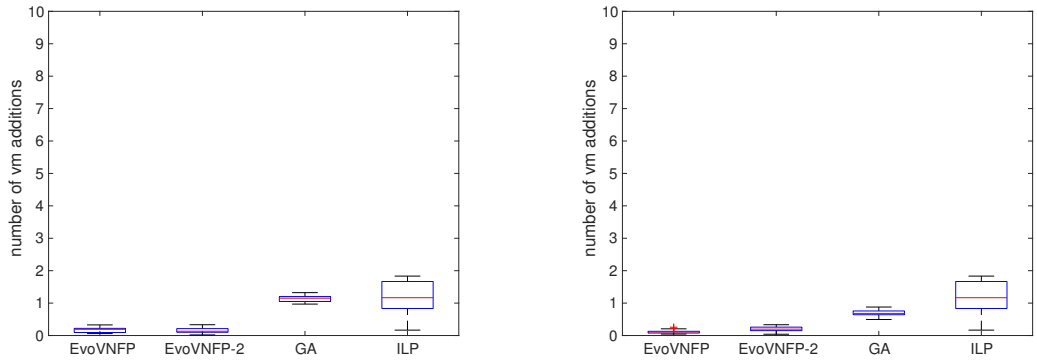
We then show the results of EvoVNFP, EvoVNFP-2, and GA about the number of generations needed until obtaining the first feasible solution. We do not consider ILP in this subsection. Figure 29(a) shows the results of $(T_p, T_e) = (8, 24)$ and Fig. 29(b) shows the results of $(T_p, T_e) = (24, 216)$.

In the Fig. 29, we can see that the number of generations of EvoVNFP and EvoVNFP-2 is



(a) $T_p = 8, T_e = 24$ (b) $T_p = 24, T_e = 216$

Figure 22: The number of VM size changes



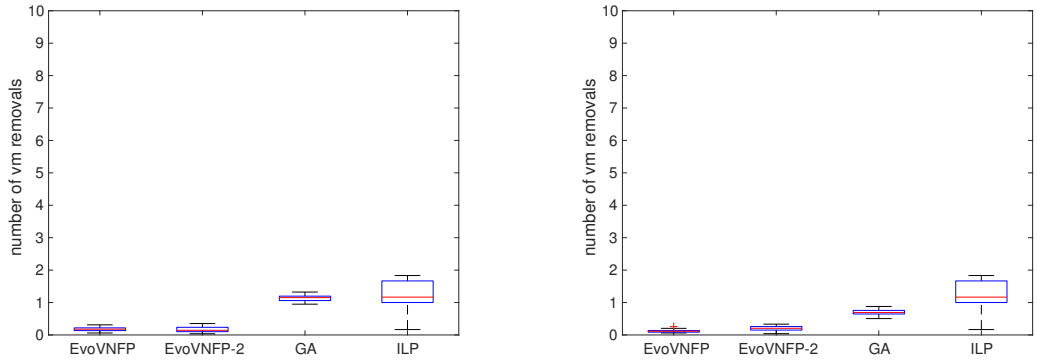
(a) $T_p = 8, T_e = 24$ (b) $T_p = 24, T_e = 216$

Figure 23: The number of VM additions

smaller than GA. It means that EvoVNFP and EvoVNFP-2 can adapt to the succeeding objectives rapidly. Furthermore, there is a big difference between the results of GA in Fig. 29. This is due to the same reason which we explained in Section. 5.4.1; GA uses almost all of the generations of the epochs to generate solutions. Note that the y-axis of Fig. 29 is in logarithmic scales.

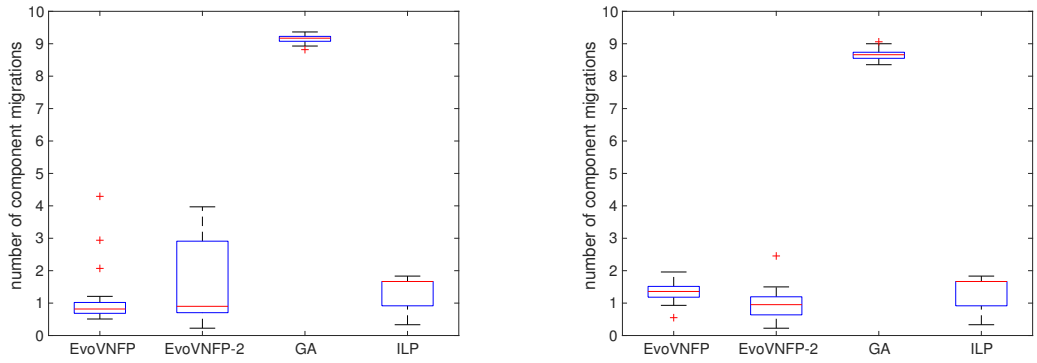
5.4.5 Discussion about Placement Strategies of EvoVNFP and EvoVNFP-2

Finally we discuss the strategy which EvoVNFP and EvoVNFP-2 select for VNF placement. In Section 5.4.2, we saw that the results of EvoVNFP and EvoVNFP-2 are smaller than those of ILP in terms of VM size changes, VM additions, VM removals, and component migrations, but larger in terms of component size changes. Furthermore, in Section 5.4.3, we saw that EvoVNFP



(a) $T_p = 8, T_e = 24$ (b) $T_p = 24, T_e = 216$

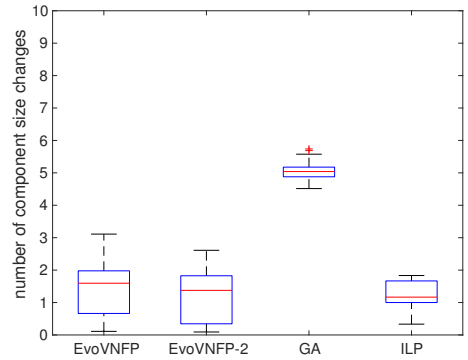
Figure 24: The number of VM removals



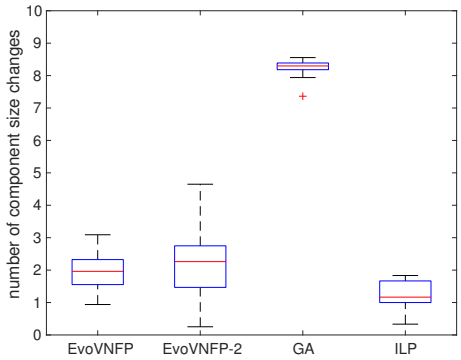
(a) $T_p = 8, T_e = 24$ (b) $T_p = 24, T_e = 216$

Figure 25: The number of component migrations

and EvoVNFP-2 are worse than ILP in terms of the number of cores. Considering these results, EvoVNFP and EvoVNFP-2 seem to spontaneously select strategies where the placements have a sufficient number of cores for a sufficient number of VMs so that they can easily adapt to the succeeding objectives by only resizing the components. The placements generated by EvoVNFP and EvoVNFP-2 tend to have extra cores which components may not need currently, but they can be used for adapting to the succeeding objectives by only resizing the components thanks to these extra cores. Therefore, the larger number of cores in EvoVNFP and EvoVNFP-2 seems inevitable to make the number of reconfigurations smaller, that is, the number of cores and the number of reconfigurations are in a trade-off relationship. Hence, we can regard the delay as the only overhead. From this standpoint, we can summarize that EvoVNFP and EvoVNFP-2 can generate

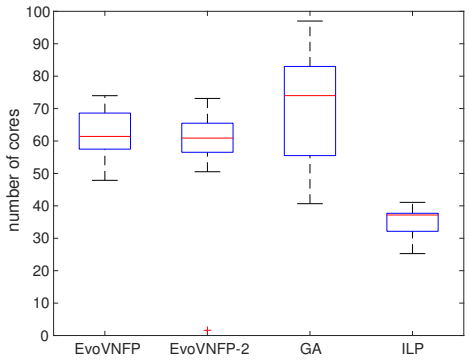


(a) $T_p = 8, T_e = 24$

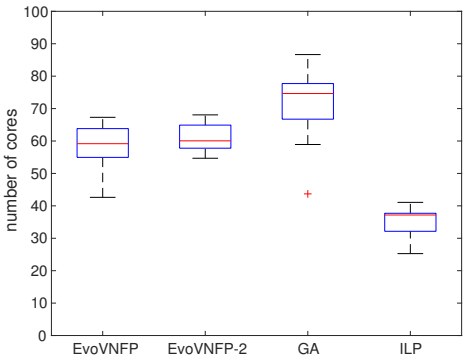


(b) $T_p = 24, T_e = 216$

Figure 26: The number of component size changes



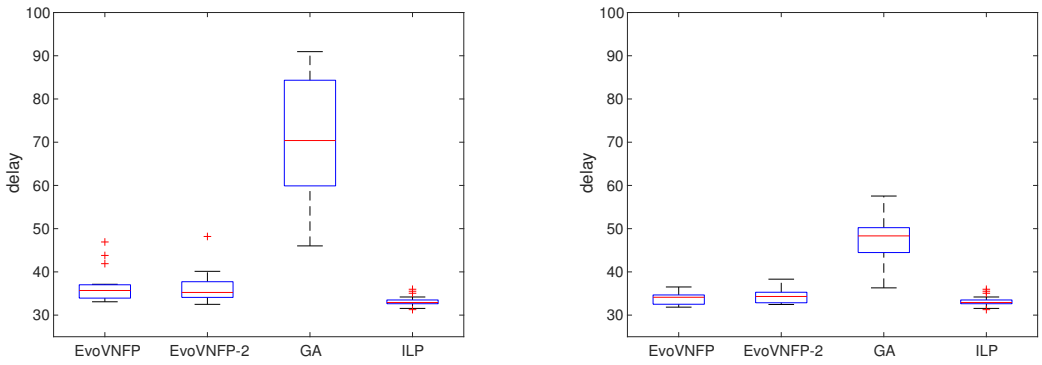
(a) $T_p = 8, T_e = 24$



(b) $T_p = 24, T_e = 216$

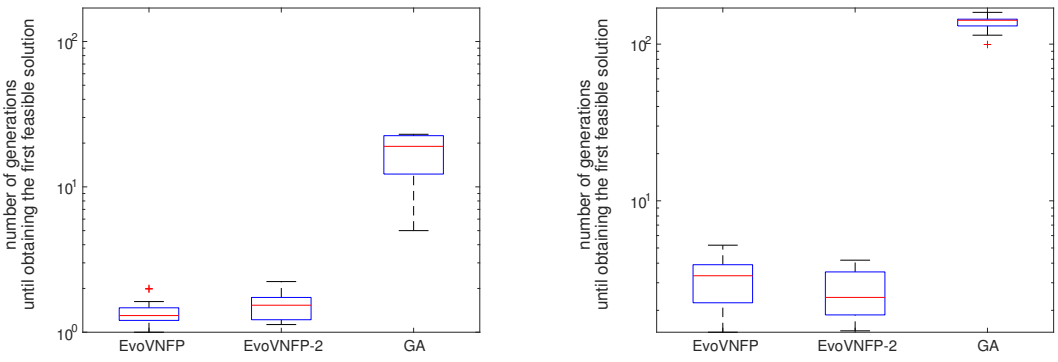
Figure 27: The number of cores

evolvable placements with small overhead.



(a) $T_p = 8, T_e = 24$ (b) $T_p = 24, T_e = 216$

Figure 28: Delay



(a) $T_p = 8, T_e = 24$ (b) $T_p = 24, T_e = 216$

Figure 29: The number of generations until obtaining the first feasible solution

6 Conclusion and Future Work

In this research, we proposed methods called EvoVNFP and EvoVNFP-2 for solving the VNF placement problem while focusing on reducing the number of reconfigurations. EvoVNFP and EvoVNFP-2 use genetic algorithms based on the MVG and EVG concepts of biological evolution under varying environments. EvoVNFP generates placements which meet the current processing requests by the genetic algorithms. When the processing requests change, the objectives of the genetic algorithms change without re-initialization of the population. This leads to placements which are robust to the varying processing requests. The resulting placements need a small number of reconfigurations to adapt to succeeding processing requests. EvoVNFP-2 includes more fine-grained changes of the objectives in addition to the mechanism of EvoVNFP. It is expected that EvoVNFP-2 will generate placements with modular structures corresponding to the processing requests, which leads to evolvability and placements that are robust to varying processing requests.

The results of the simulations show that EvoVNFP and EvoVNFP-2 can reduce the number of reconfigurations with only small overhead in terms of the delays compared to the reference methods. EvoVNFP and EvoVNFP-2 seem to generate evolvable placements which have a sufficient number of cores and can adapt to the succeeding objectives by only changing the components.

Our future work includes improving EvoVNFP-2 so that it achieves better adaptation by generalizing from past environments [27].

Acknowledgment

This thesis would not have materialized without a lot of supports of many people. First, I would like to express my sincere gratitude to my supervisor, Professor Masayuki Murata of Osaka University, for his valuable advice, his kind guidance, and providing me this precious study opportunity in his laboratory. I am also deeply grateful to Guest Associate Professor Kenji Leibnitz of Osaka University. He spent a lot of time for me and gave me great support and encouragement. I also would like to express my great appreciation to Assistant Professor Daichi Kominami of Osaka University. He gave me a lot of insightful advice on my studies and enormous help to me for making documentations.

Furthermore, I am deeply grateful to Assistant Professor Yuki Koizumi of Osaka University. He supported me a lot with considering the aspects of information networking in my studies. I also would like to express my great appreciation to Guest Associate Professor Tetsuya Shimokawa of Osaka University. He gave me a lot of valuable comments from the viewpoint of biology.

Moreover, I would like to appreciate Associate Professor Shin'ichi Arakawa and Assistant Professor Yuichi Ohsita of Osaka University for insightful comments on my studies. I am also grateful to Dr. Satoshi Imai of Fujitsu Laboratories for providing me precious information about NFV. I also would like to express my great appreciation to my senior associates, Ms. Naomi Kuze and Mr. Shinya Toyonaga, for their great help and encouragement.

Finally, I would like to thank all members of the Advanced Network Architecture Laboratory at the Graduate School of Information Science and Technology, Osaka University, for meaningful discussion about my research and kind support.

References

[1] ETSI, GS NFV 002 - V1.1.1 - Network Functions Virtualisation (NFV); Architectural Framework, Oct. 2013.

[2] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network Function Virtualization: State-of-the-Art and Research Challenges," IEEE Communications Surveys & Tutorials, vol. PP, p. 1, Sept. 2015.

[3] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network Function Virtualization: Challenges and Opportunities for Innovations," IEEE Communications Magazine, vol. 53, pp. 90–97, Feb. 2015.

[4] R. Jain and S. Paul, "Network Virtualization and Software Defined Networking for Cloud Computing: A Survey," IEEE Communications Manazine, pp. 24–31, Nov. 2013.

[5] A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral, "Deep Packet Inspection as a Service," in Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies, pp. 271–282, Dec. 2014.

[6] J. Soares, C. Goncalves, B. Parreira, P. Tavares, J. Carapinha, J. Barraca, R. Aguiar, and S. Sargento, "Toward a Telco Cloud Environment for Service Functions," IEEE Communications Magazine, vol. 53, pp. 98–106, Feb. 2015.

[7] W. Kiess, X. An, and S. Beker, "Software-as-a-Service for the Virtualization of Mobile Network Gateways," in Proceedings of IEEE Global Communications Conference 2015, Dec. 2015.

[8] ETSI, GS NFV-INF 003 - V1.1.1 - Network Functions Virtualisation (NFV); Infrastructure; Compute Domain, Dec. 2014.

[9] W. John, K. Pentikousis, G. Agapiou, E. Jacob, M. Kind, A. Manzalini, F. Risso, D. Staessens, R. Steinert, and C. Meirosu, "Research Directions in Network Service Chaining," in Proceedings of 2013 IEEE SDN for Future Networks and Services, pp. 1–7, Nov. 2013.

[10] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba, "On Orchestrating Virtual Network Functions in NFV," in Proceedings of 11th International Conference on Network and Service Management Mini-Conference, Nov. 2015.

[11] M. Ghaznavi, N. Shahriar, R. Ahmed, and R. Boutaba, "Service Function Chaining Simplified," eprint arXiv:1601.0075, Jan. 2016.

[12] R. Cohen, L. Lewin-Eytan, J. Naor, and D. Raz, "Near Optimal Placement of Virtual Network Functions," in Proceedings of 2015 IEEE Conference on Computer Communications, pp. 1346–1354, Apr. 2015.

[13] H. Moens and F. De Turck, "VNF-P: A Model for Efficient Placement of Virtualized Network Functions," in Proceedings of the 10th International Conference on Network and Service Management, pp. 418–423, Nov. 2014.

[14] M. Xia, M. Shirazipour, Y. Zhang, H. Green, and A. Takacs, "Network Function Placement for NFV Chaining in Packet/Optical Datacenters," Journal of Lightwave Technology, vol. 33, pp. 1565–1570, Apr. 2015.

[15] M. Bouet, J. Leguay, and V. Conan, "Cost-based Placement of vDPI Functions in NFV Infrastructures," in Proceedings of the 1st IEEE Conference on Network Softwarization, pp. 1–9, Apr. 2015.

[16] M. Bouet, J. Leguay, and V. Conan, "Cost-Based Placement of Virtualized Deep Packet Inspection Functions in SDN," in Proceedings of 2013 - 2013 IEEE Military Communications Conference, pp. 992–997, Nov. 2013.

[17] C. Aikens, "Facility Location Models for Distribution Planning," European Journal of Operational Research, vol. 22, pp. 263 – 279, Dec. 1985.

[18] ETSI, GS NFV-SWA 001 - V1.1.1 - Network Functions Virtualisation (NFV); Virtual Network Functions Architecture, Dec. 2014.

[19] S. Clayman, E. Maini, A. Galis, A. Manzalini, and N. Mazzocca, "The Dynamic Placement of Virtual Network Functions," in Proceedings of the IEEE International Conference on Network Operations and Management Symposium, pp. 1–9, May 2014.

[20] G. P. Wagner, “Homologues, Natural Kinds and the Evolution of Modularity,” American Zoologist, vol. 36, pp. 36–43, Feb. 1996.

[21] G. P. Wagner, M. Pavlicev, and J. M. Cheverud, “The Road to Modularity,” Nature Reviews Genetics, vol. 8, pp. 921–931, Dec. 2007.

[22] N. Kashtan and U. Alon, “Spontaneous Evolution of Modularity and Network Motifs,” Proceedings of the National Academy of Sciences, vol. 102, pp. 13773–13778, Sept. 2005.

[23] M. Otokura, K. Leibnitz, T. Shimokawa, and M. Murata, “Evolutionary Core-Periphery Structure and its Application to Network Function Virtualization,” to appear in IEICE Transactions on Nonlinear Theory and Its Applications, vol. E7-N, Apr. 2016.

[24] L. Davis, Handbook of Genetic Algorithms. Van Nostrand Reinhold, 1991.

[25] M. P. Rombach, M. A. Porter, J. H. Fowler, and P. J. Mucha, “Core-Periphery Structure in Networks,” SIAM Journal on Applied Mathematics, vol. 74, pp. 167–190, Feb. 2014.

[26] N. Kashtan, E. Noor, and U. Alon, “Varying Environments Can Speed Up Evolution,” Proceedings of the National Academy of Sciences, vol. 104, pp. 13711–13716, Aug. 2007.

[27] M. Parter, N. Kashtan, and U. Alon, “Facilitated Variation: How Evolution Learns from Past Environments to Generalize to New Environments,” PLoS Computational Biology, vol. 4, p. e1000206, Nov. 2008.