Intelligent Network Service Embedding using Genetic Algorithms

Panteleimon Rodis and Panagiotis Papadimitriou
University of Macedonia, Greece
{rodis, papadimitriou}@uom.edu.gr

***Abstract*—Network Function Virtualization (NFV) opens us great opportunities for network processing with higher resource efficiency and flexibility. Nevertheless, intelligent orchestration mechanisms are required, such that NFV can exploit its potential and fill up to its promise. In this respect, we investigate the potential gains of embracing Artificial Intelligence (AI) for the virtual network function (VNF) placement problem. To this end, we design and evaluate a genetic algorithm, which seeks efficient embeddings with runtimes on par with heuristic methods. Our proposed embedding method exhibits innovations in terms of network representation and algorithm design, thereby, deviating from typical genetic algorithms. Compared to a heuristic, the proposed genetic algorithm yields higher request acceptance rates, stemming from more efficient resource utilization. We further study a range of factors and parameters that affect the efficiency of the genetic algorithm.**

I. Introduction

Network Function Virtulization (NFV) decouples network functions from the underlying specialized devices, known as middleboxes, which are commonly deployed in enterprise and telco networks [1], [2], [3]. As such, network functions, such as firewalls, proxies, network address translation (NAT), intrusion detection, and redundancy elimination, can be deployed using software that runs on virtualized commodity servers [4]. Such virtualized network functions (VNFs) can be either executed on the network operator's premises or can be leased from cloud providers, in the form of Network Function-as-a-Service [5], [6], [2], [7], [8]. In this respect, NFV enhances flexibility and resource efficiency compared to middleboxes, whereas it also spurs innovation by lowering the barrier for introducing new functionality into the network.

A key requirement for the deployment of VNFs, either on private datacenters or on public cloud infrastructures (usually termed as NFV infrastructure – NFVI), is VNF-graph embedding, *i.e.,* the assignment of VNFs and the corresponding VNF-graph edges onto the respective NFVI counterparts (*i.e.,* servers and network paths). The general case of this problem (*i.e.,* topology embedding) is known to be NP-hard [9] and not efficiently solvable, even when polynomial boundaries [10] or restrictions [11] are applied on the problem parameters. The problem complexity is retained even for special types of substrate network topologies [12].

VNF-graph mapping optimization has been mainly tackled using heuristics [13], [6], [8] and exact methods [14], [5], [15], [16], [7], [17]. Heuristics and greedy algorithms generate embeddings with typically low solver run-times, but usually at the expense of a considerable optimality gap. On the other hand, exact methods may achieve near-optimal solutions; however, their associated computational complexity introduces significant scalability limitations. As such, exact methods constitute a feasible approach only to small-scale embeddings.

Artificial intelligence (AI) comprises a promising alternative to these VNF-graph embedding methods, since it can generate efficient solutions under an acceptable solver run-time [18], [19]. In this respect, we study genetic algorithms as an alternative approach to the VNF-graph embedding problem. Genetic algorithms have been very rarely employed for topology embeddings; hence, their applicability and efficiency is not well understood. As such, we aim at shedding light into the efficacy of genetic algorithms on network service embedding by examining a variety of factors and their impact on embedding efficiency using simulations. Our proposed genetic algorithm yields adaptability in different types of network topologies, based on evaluations conducted on structured (*i.e.,* fat trees) network topologies.

The remainder of the paper is organized as follows. Section II provides background information on genetic algorithms. In Section III, we present the network and request models utilized in the design of the proposed genetic algorithm. We further introduce an innovative representation of edge vectors with the aim of reducing memory consumption. In Section IV, we discuss in detail the VNF-graph embedding problem and our proposed solution. In Section V, we discuss the efficiency of the proposed genetic algorithm based on simulation results. Finally, Section VI highlights our conclusions.

II. Genetic Algorithms

Genetic algorithms are optimization techniques inspired by Darwin's theories on the evolution of species. They provide efficient solutions in computationally hard problems, such as combinatory NP-hard problems [20]. Initially, a genetic algorithm generates a population of possible solutions. The population is usually generated randomly, but may also be the product of a heuristic procedure [21]. Our proposed solution employs both methods. The members of the population are called *chromosomes*; this name implies that the functionality of the algorithm simulates biological procedures. Every chromosome is a string that encodes a possible solution and every symbol of the string is called *gene*. A crucial factor for the

978-1-6654-2744-9/21/$31.00 ©2021 IEEE

operation of the algorithm is the fitness function, which defines the criterion for the margin between the solution encoded in a chromosome and the desired solution. The algorithm iteratively executes the following procedures.

Selection. This step simulates the procedure of natural selection, at which the stronger members of a population survive in the next generation while the weakest members do not survive.

Crossover. During the procedure of crossover, two chromosomes exchange parts of their genetic material that are randomly chosen and generate offspring that represents different solution than their parents.

Mutation. The procedure of mutation refers to the random change of the value of a gene, similar to the biological notion of mutation. Mutation may generate solutions that are not produced by crossover, thereby, directing the search in different parts of the search space.

Each round of sequential execution of the procedures is called *generation*. Eventually, the population becomes homogeneous converging to a strong solution. This outcome stems from the fact that the chromosomes of higher fitness prevail through the procedure of selection over the weaker chromosomes. The genes that are primarily responsible for the high fitness are spread through crossover to a large part of the next generation population.

III. Network Model and Representation

A. Network and Request Model

Network Model. For the substrate network, we define the graph $G_S(V_S, E_S)$, where V_S denotes the set of compute nodes (*e.g.,* servers) at which VNFs can be hosted. The edges in E_S represent the network links. Every edge $p(u,z) \in E_S$ denotes the shortest path that connects the nodes u, z of the substrate network. In every substrate node s, we assign a weight w_S which denotes its residual capacity. For the edges of the substrate network graph, we define two properties in the form of weights, *i.e.,* the available bandwidth of the substrate links and the respective communication cost, which is proportional to the hop-count of the path between each pair of nodes. In this respect, we assign in p the weight w_p denoting the available bandwidth on the link and the weight h for its communication cost.

Request Model. Each request consists of the VNF-graph, modeled by $G_V(V_V, E_V)$. Each node $n \in V_V$ denotes a VNF, whereas its weight w_n expresses its computing demand. We further model each edge between VNFs a and b of the VNF-graph, as $e(a,b)$. The weight w_e denotes the bandwidth demand for each edge of the VNF-graph. The assignment of the VNF-graph elements is represented by an n-dimension vector H, where n is the order of the VNF-graph. In position i of H, we place the value of node $s \in V_S$, which hosts the node $i \in V_V$.

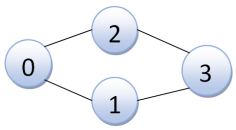

Fig. 1: Edge vector representation: $G : \langle 110011 \rangle$.

B. Edge Vector Representation

Graph representation is crucial for the efficiency of the algorithm, as the substrate and the service chain topology are modeled as graphs. More precisely, the choice of graph representation can be mandated by the memory demands of storing a graph and the computational burden of encoding and decoding the elements of the graph in the chosen representation. The most common representations are the adjacency matrix, the edge list, and the adjacency list [22]. These representations yield high memory demands and, thereby, are not suitable for our genetic algorithm. Instead, we propose a new graph representation that requires significantly lower memory, whereas encoding and decoding is of linear computational complexity. This representation is termed as *Edge Vector*. In the following, we provide definitions for representing undirected and directed graphs in this form.

Edge vector for undirected graph. Let s be a vector that represents undirected graph $G(V, E)$, consisting of v nodes. We further enumerate the elements of s in the range $[0, e)$, where $e = v(v-1)/2$ is the maximum possible number of edges in G. The symbol in position q of s is w, if nodes a and b are adjacent and 0 if they are not adjacent, where $q = a + \sum_{x=0}^{b-1} x$, for $a < b$ and $a, b \in [0, v)$. The value of w is set to 1 for non-weighted graphs, whereas for weighted graphs the respective value corresponds to the weight of the edge (a, b) (see the example in Fig. 1).

Edge vector for directed graph. Let vector d represent the weighted graph $G(V, E)$ and let us also enumerate its elements in the range [0, e). In position q of d, as q is defined above, and for non-weighted graph G we assign 0, if nodes a and b are not adjacent, whereas we assign 1 for $a \rightarrow b$, 2 for $b \leftarrow a$ and 3 for $a \longleftrightarrow b$. For weighted graphs, we place 0, $1w$, $2w$ and $3w$, respectively, where w is the weight of the edge (a, b), for $a < b$ and $a, b \in [0, v)$. In the case of graph G with weighted nodes, the node weights are stored in a *Node Weight Vector*. Therefore, in vector k:$\langle w_0, w_1, \ldots, w_v \rangle$ the value w_i represents the weight of node i.

We examine the descriptive complexity of the aforementioned representations. The complexity is estimated by the number of symbols required in each case for the representation of graph $G(V, E)$ of n nodes, k edges and average node degree g.

- The adjacency matrix requires a binary matrix of size $n \times n$. The size of the representation is n^2 symbols.
- In the edge list, each edge (i, j) is represented by the numbering of nodes i and j. Let m be the average length of a string that represents the number of a node, the size of the representation is $k \times 2 \times m$ symbols.

- Each record of the adjacency list stores the adjacent nodes of one node. Let m be the average length of a string that represents the number of a node. The size of the representation is then $n \times g \times m$ symbols.
- In the edge vector, each graph G is represented by $n(n-1)/2$ symbols.

For dense graphs, where $k \to n(n-1)/2$ and $g \to (n-1)$, edge vector provides the shortest representation. Only in the cases of sparse graphs the edge list representation has significant advantages. Our work mainly considers full and dense graphs. The computational complexity of encoding and decoding graphs in the edge vector representation is not inferior to the complexity of other representations. As shown in the following algorithms, both procedures yield linear complexity.

The inputs in Algorithm 1, which describes the encoding procedure, are two adjacent nodes a, b in G. The output is the place of the edge in the edge vector. In Algorithm 2, the input is the place q of an edge in the vector, whereas the output is the pair of the adjacent nodes of G.

Algorithm 1 Encoding

Input: nodes a, b where $a < b$
Output: place of edge (a, b) in vector
1: $x = (b(b-1))/2$
2: **return** $x + a$

Algorithm 2 Decoding

Input: edge q in vector
Output: nodes that q connects
1: $y =$ round down$(\sqrt{2q})$
2: increment y
3: $x = q - ((y^2 - y)/2)$
4: **if** $x < 0$ **then**
5: decrement y
6: $x = q - (y^2 - y)/2)$
7: $result = \{x, y\}$
8: **return** $result$

Implementations of the algorithms in Java and JavaScript are available online in [23]. We compare our edge vector representation against a set of graphs generated for a DIMACS challenge [24]. The DIMACS graph format is based on the edge list representation. We exclude the sparse graphs and convert the 69 dense graphs of the set using Algorithm 1. The edge vector representation leads to a significant reduction in the number of symbols by 72.75% and, also in the size of the generated files by 53.87%, on average.

IV. Problem Definition and Solution

A. Problem definition

Let weighted graphs $G_S(V_S, E_S)$ and $G_V(V_V, E_V)$ model the substrate and the service chain topology respectively, as defined in Section III-A. We define the mapping m of the nodes in V_V to the nodes of subset $V'_S \subseteq V_S$, such that:

$$\forall n \exists z \to n \in V_V \wedge z \in V'_S \wedge c(n,z)$$

where $c(n,z)$ denotes that node n corresponds to z. By m we define the mapping of the edges of G_V to edges of G_S, so that

$$\forall e(a,b) \exists p(u,z) \to e(a,b) \in E_V \wedge p(u,z) \in G_S \wedge c(a,u) \wedge c(b,z)$$

where $e(a,b)$ is the edge connecting nodes a, b of G_V, while $p(u,z)$ is the path connecting nodes u and z of G_S. This represents the shortest path of the substrate network that connects the two nodes.

Let w_z, w_n be the weights of nodes z and n, respectively, and w_p, w_e the weights of edges $p(u,z)$ and $e(a,b)$. By h we also define the communication cost of using p. The efficiency of the mapping is therefore defined as:

$$C_m = \sum (w_z - w_n) + \sum [(w_p \times h) - w_e]$$

Equation C_m corresponds to the fitness function of the genetic algorithm and expresses the efficiency in terms of resource utilization and cost. The value of the difference $w_z - w_n$ represents the efficiency in using the available resources of node z of the substrate network. Similarly, the difference $w_p - w_e$ expresses the efficiency in utilizing the available bandwidth of link p. The value of $w_p \times h$ corresponds to the cost of using path p, where h is the hop-count of the path. The value of C_m is minimized when all VNFs are co-located on the same substrate node. The problem at hand is then defined as: *Find* m that $min(C_m)$.

B. Genetic algorithm

During the development of genetic algorithms, various critical issues are raised that are inherent in the nature of genetic algorithms. In order to address them, it is often necessary to apply variations on the typical design of the genetic algorithms [21], [25].

Maintenance of efficient solutions. In the common approach, the procedures of crossover and mutation modify the genotype of the population without preserving the initial chromosomes. This design may fail to preserve an efficient solution on the population that may take part to mutated or crossover procedures. This issue can be resolved by including in the same population both the parents and their offspring.

Premature convergence to local optima. A significant issue concerning genetic algorithms in general is the convergence to an undesired solution. This stems from the restriction of the search in a part of the state space that contains only local optima. In this case, the population becomes homogeneous, before converging to the optimal solution.

The technique developed for the avoidance of premature convergence, in our work, is based on the concept of competition that takes place in stages and in sets of competitors. The competitors that prevail in each set form the groups of competitors that will compete in the next stage of the competition. The winners of the group stage are nominated in the last stage of the competition, which designates the stronger competitor as the winner of the competition.

The genetic algorithm is executed in n groups of n sets of chromosomes. In each set, there are p chromosomes that

constitute the population for the algorithm that is executed for g generations. The best solution generated in each set is promoted to form the population of the group that it belongs. Subsequently, in each group, the algorithm is executed over the population generated in the sets. The final output is computed by executing the algorithm on the population generated on the groups. In this method, the algorithm combines in the group stage the local optima discovered in the sets, avoiding the confinement within a single local optimum. Simulation results demonstrating the advantages of this technique are presented in Section V-C. A benefit of this approach is that computations in sets and groups can be executed in parallel, thereby, reducing solver run-time in multi-core servers.

The operation of the genetic algorithm consists of the generation of an initial population of chromosomes and the iterative execution of the genetic procedures, as shown in Algorithm 3. The population is generated randomly and also the result of a heuristic is handled as input in order to include the exploration of parts of the search space that are expected to lead to an efficient solution. Our simulation results show that the genetic algorithm could efficiently adapt in random substrate network topologies. Instead, in structured topologies (*i.e.*, fat trees) the algorithm is not so efficient. This limitation can be overcome by using a heuristic for the generation of some members of the initial population. This heuristic is described in detail in Section V-B.

The execution of the genetic procedures in each chromosome depends on its fitness, computed by fitness function C_m. For the formation of sets and groups and their computation, we use two parameters called *generations* and *supergenerations*. These two parameters determine the iterative structures of the algorithm. The thresholds of mutation and crossover probabilities along with a random number generator determine whether the two procedures are executed in every chromosome. In the following, we elaborate on the three genetic procedures.

Selection. The selection procedure, described in Algorithm 4, is executed only if the population is not homogeneous; otherwise, its execution is pointless. Whereas the procedures of crossover and mutation increase the population in the current generation, selection maintains a constant size for the population of the next generation. We maintain a population of constant size in order to control the complexity of the algorithm. Homogeneity is evaluated by computing the deviation of the population fitness. If the population is deemed to be heterogeneous, the fitness of each chromosome is computed and if its value is computed less than c, the chromosome is added in the future population. This occurs for c=m+$(d \times p)$, where m is the value of the strongest chromosome of the current population, d the deviation of the fitness values of the population, and p is a randomly generated value.

Crossover. For every pair of sequentially chosen chromosomes (*e.g., i, j*), we generate a random value p. If p is smaller than the crossover probability, the two chromosomes produce two offsprings. Given a random number h, chromosome i copies its first h genes to the first offspring and the rest to the second one. Likewise, chromosome j copies its first h genes to the second offspring and the rest to the first one (see Algorithm 5).

Algorithm 3 Genetic Algorithm

Input: graphs G_S, G_V, *generations*, *supergenerations*
Output: best chromosome

```
1: run init1()
2:
3: Procedure init1()
4: for t = 0 to supergenerations do
5:    for s = 0 to supergenerations do
6:       generate population1
7:       run init2(population1)
8:       store best chromosome in population2
9:    run init2(population2)
10:   store best chromosome in population3
11: run init2(population3)
12: return best chromosome
13:
14: Procedure init2(population)
15: compute deviation in population
16: if deviation > 0 then
17:    for i = 0 to generations do
18:       run crossover(pop)
19:       run mutation(pop)
20:       run selection(pop)
21: return best chromosome
```

Algorithm 4 Selection

Input: population *pop*, *newpop_maxsize*
Output: new population *newpop*

```
1: compute deviation in population
2: if deviation > 0 then
3:    while newpop.size < newpop_maxsize do
4:       generate random value p
5:       c =(minimum pop fitness)+deviation × p
6:       for i = 0 to pop.size do
7:          if pop[i].fitness< c then
8:             add pop[i] to newpop
9: return newpop
```

Mutation. This procedure is sequentially executed in every chromosome. If a randomly generated value p is smaller than the mutation probability, a copy of the chromosome is generated and mutated. A gene of the copied chromosome is randomly chosen and a random value is assigned to it. The mutated chromosome is added to the population, as described in Algorithm 6.

C. Parameter Adjustment

The functionality of a genetic algorithm is determined by various parameters, such as the population size, the crossover and mutation probabilities, the number of generations, and, in our case, the supergeneration parameter. Identifying appropri-

Algorithm 5 Crossover

Input: population *pop*, crossover probability c
Output: new population *newpop*
1: l =chromosome length
2: **for** $i = 0$ to *pop*.size-1 with step 2 **do**
3: add *pop*[i], *pop*[i+1] to *newpop*
4: generate random value $p \in [0, 1]$
5: **if** $p \leq c$ **then**
6: generate empty chromosomes *temp*1, *temp*2
7: generate random value $h \in [0, l)$
8: add genes 0 to h of *pop*[i] in *temp*1
9: add genes h+1 to l of *pop*[i+1] in *temp*1
10: add genes 0 to h of *pop*[i+1] in *temp*2
11: add genes h+1 to l of *pop*[i] in *temp*2
12: add *temp*1, *temp*2 to *newpop*
13: **return** *newpop*

Algorithm 6 Mutation

Input: population *pop*, mutation probability m
Output: new population *newpop*
1: l =chromosome length
2: **for** $i = 0$ to *pop*.size **do**
3: add *pop*[i] to *newpop*
4: generate random value $p \in [0, 1]$
5: **if** $p \leq m$ **then**
6: generate random values $h \in [0, l)$, n
7: place n in gene h of *pop*[i]
8: add *pop*[i] to *newpop*
9: **return** *newpop*

ate values for these parameters entails a significant challenge and is also subject to the problem that the genetic algorithm is applied on. We approach this parameter adjustment as an optimization problem and, thereby, have implemented a genetic algorithm in order to determine near-optimal values. This algorithm is described in Section IV-B. At first, we investigate optimal values for the crossover and mutation probabilities and, subsequently, we employ these values to properly adjust the parameters for the population size, generations, and supergenerations. In each phase, the chromosomes are set to the values of the parameters under study. The fitness function is computed by running instances of the VNF-graph embedding algorithm with the values of the parameters stored in every chromosome. The objective of this tuning procedure is to identify the parameters of the embedding algorithm that would lead to the minimum embedding cost.

The crossover and mutation probabilities generated by this tuning procedure are 0.59 and 0.78, respectively. We compared these values against typical values (*i.e.,* 0.85 for crossover probability and 0.05 or less for mutation), as well as with a setup where both probabilities are set to 1. We find that our optimized setup (0.59, 0.78) yields embedding efficiency on par with both values set to 1, due to a more thorough exploration of the search space. In contrast, employing the typical values (0.85, 0.05) yields lower efficiency and instability (*i.e.,* results vary significantly across different runs), due to more frequent premature convergence to local optima. The same tuning algorithm is also used on population size, number of generations, and supergenerations (see Table I for the respective parameter values).

TABLE I: Optimal parameter setup.

nodes of VNF-graph	chromosomes	generations	supergenerations
5	148	40	6
6	176	156	4
7 - 8	132	42	6
9 - 10	244	86	6

V. Evaluation

A. Evaluation Environment

The algorithm and the evaluation environment are implemented in Java 8 and executed on a computer with dual-core CPU at 2.60 GHz and 4 GB RAM. The simulated topology is a 3-layer fat-tree datacenter network topology, which comprises a common NFVI. The simulated substrate network consists of 250 servers, each one with 10 GHz computing capacity. The capacity of links connecting the servers with the Top-of-the-Rack (ToR) switches is 1 Gbps, whereas the links at the upper layers of the topology have capacity 10 Gbps.

Each VNF-graph request consists of a diverse number of nodes, picked randomly within the range of 5 to 10. The computing demand for each VNF in the request varies between 2 and 6 GHz. Likewise, the bandwidth demands in the VNF-graph vary between 20 and 100 Mbps. The VNF-graph requests are expiring. In particular, after 90 requests have been embedded into the substrate network, an embedded VNF-graph is randomly selected and removed from the network. This occurs for every incoming request, after the first 90 requests.

Before the arrival and processing of VNF-graph requests, traffic load is randomly inserted in the datacenter, which resembles real conditions at which a NFVI will not be entirely unutilized. In particular, CPU load equal to 50% of the computational capacity is inserted to 10% of the nodes, whereas traffic equal to 50% of the bandwidth capacity is injected to 10% of the network paths. Our simulation scenario handles the embedding of 6000 requests, which are either accepted or rejected, depending on the outcome of each executed algorithm.

B. Comparison Method

To assess the efficiency of our proposed genetic algorithm, we perform a comparison against a baseline algorithm, which strives to achieve VNF consolidation. More precisely, the baseline is a greedy algorithm that sorts the nodes of the substrate network and the VNF-graph in descending order based on their weights. Subsequently, in every substrate node of the sorted list with the largest available capacity, the algorithm maps sequentially the VNFs with the largest computing demand that can fit into the respective node, while taking into account also

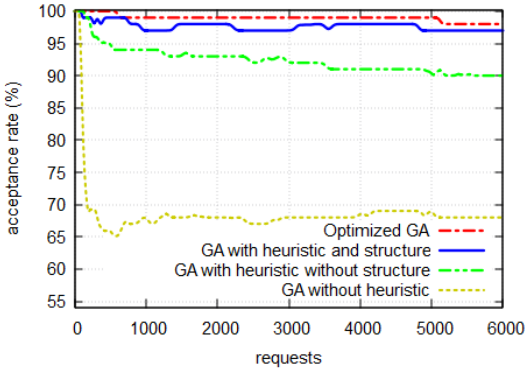

Fig. 2: Request acceptance rates of diverse GA variants.

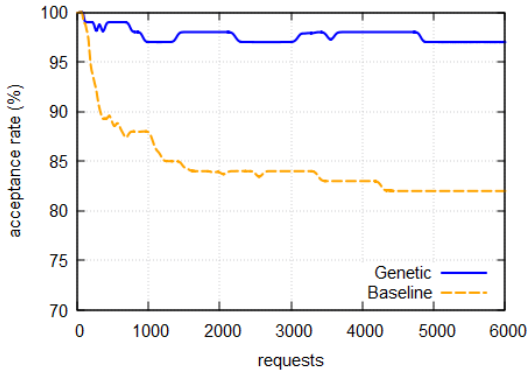

Fig. 3: Request acceptance rates of the GA and the baseline.

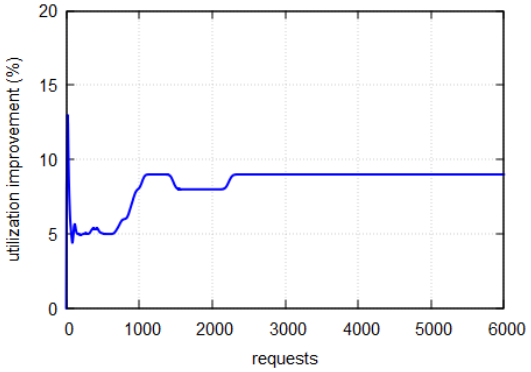

Fig. 4: Resource utilization improvement of the GA over baseline.

bandwidth constraints. The algorithm terminates as soon as all VNFs have been mapped. If the mapping is not feasible (*i.e.,* when either the computing or bandwidth demands are not met), the execution of the algorithm is terminated with the rejection of the request.

The generated mapping from the greedy algorithm (when feasible) is included in the initial population of the genetic algorithm. More specifically, it is employed as a heuristic for the generation of one member of the initial population. As shown in the following, this technique empowers the genetic algorithm to adapt to the structured topology of the network and generate more efficient solutions.

C. Evaluation Results

Our proposed algorithm uses an initial population of 250 chromosomes, whereas the generations and supergenerations parameters are set to 25 and 2, respectively. This leads to balance between efficiency and solver runtime (*i.e.,* 85 ms per request, on average). Furthermore, in each request, the output of the greedy algorithm is inserted in the initial chromosome population. The crossover and mutation parameters are set to their optimal values, as computed by the tuning procedure described in Section IV-C.

Fig. 2 illustrates the request acceptance rate for different variants of the genetic algorithm (GA). Using the GA without the heuristic for the generation of the initial population yields low acceptance rates. This stems from the fact that the search space is not directed to any specific areas, and, thereby, the algorithm fails to adapt effectively in the search space. Next, we focus on the comparison between three other GA variants, which employ the heuristic and lead to notably higher acceptance rates. *GA with heuristic and structure* and *GA with heuristic without structure* differ in the sense that the former utilizes the structure of procedures of sets and groups, exemplified in Section IV-B. This structure of procedures enables a more advanced search of the feasible solution space, generating better embeddings with higher acceptance rates, as shown in Fig. 2. Last, the *Optimized GA* is adjusted based on the the optimal setup of Table I, yielding acceptance rates approximately at 99%, but at the expense of runtime (over 1 sec per request, in our system).

In the following, we compare the *GA with heuristic and structure* with the baseline in terms of acceptance rate. Fig. 3 indicates that both methods converge to a steady state, as the outcome of expiring requests. Nevertheless, the GA achieves a notably higher acceptance rate, due to its ability to adapt to the current condition of its environment. This observation is also substantiated by the fact that when there is no random traffic in the network and low utilization from the embedded requests, neither of the algorithms leads to rejections.

To gain more insights, we also compare the two methods in terms of resource utilization. To this end, we specifically compare the embeddings generated by the baseline and fed into the population of the GA with the final embedding computed by the GA. For the comparison of the generated mappings, we use their fitness as a metric for resource utilization. According to Fig. 4, the GA yields a significant improvement in terms of resource utilization. This stems from the fact that the GA examines a larger part of the search space compared to the baseline. This essentially empowers the genetic procedures to improve the embeddings generated by the baseline or identify better alternatives.

To further explain the efficiency gains achieved by the GA, we measure the server and link cost (*i.e.,* in terms of CPU cycles and bits per second, respectively). The respective gains compared to the baseline are illustrated in Fig. 5. Additional micro-benchmarks from our simulation environment indicate that the edges of the VNF-graph are mapped onto shorter

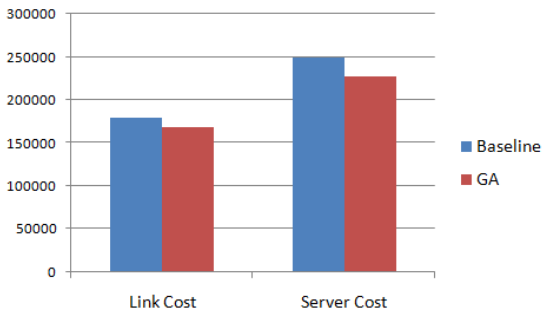

Fig. 5: Server cost measured in CPU cycles (GHz) and link cost in Gbps.

paths by the GA, which also achieves an increased VNF consolidation level. Eventually, the shorter hop-count and the better consolidation level lead to the link and server cost savings, empowering the NFVI provider to better monetize his infrastructure.

VI. Conclusions

VNF-graph mapping comprises a crucial orchestration aspect for NFV infrastructures. In contrast to various heuristics and exact methods employed to tackle this problem, we investigated the efficiency of AI-assisted embedding, leveraging on genetic algorithms. Our simulation results indicate that our proposed genetic algorithm confronts the computational complexity of VNF-graph embedding and generates efficient solutions. More precisely, the genetic algorithm yields significantly higher request acceptance rates, which stem from improved VNF consolidation and lower bandwidth consumption, compared to a baseline greedy algorithm with a similar optimization objective (*i.e.,* VNF consolidation).

In future work, we plan to explore the suitability of other methods for AI-assisted NFV orchestration, such as reinforcement learning, and perform comparisons against existing heuristic and exact methods. Furthermore, the applicability of such AI methods will be also studied in the context of other NFV orchestration aspects, such as VNF elasticity.

Acknowledgments

This work is supported by the MESON (Optimized Edge Slice Orchestration) project, co-financed by the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH - CREATE - INNOVATE (project code: T1EDK-02947).

References

[1] "ETSI Network Function Virtualization," http://www.etsi.org/technologies-clusters/technologies/nfv.

[2] M.-A. Kourtis *et al.*, "T-nova: An open-source mano stack for nfv infrastructures," *IEEE Transactions on Network and Service Management*, vol. 14, no. 3, pp. 586–602, 2017.

[3] G. Papathanail, A. Pentelas, I. Fotoglou, P. Papadimitriou, K. V.Katsaros, V. Theodorou, S. Soursos, D. Spatharakis, I. Dimolitsas, M. Avgeris, D. Dechouniotis, and S. Papavassiliou, "Meson: Optimized cross-slice communication for edge computing," *IEEE Communications Magazine*, vol. 58, no. 10, 2020.

[4] J. Sherry *et al.*, "Making middleboxes someone else's problem: network processing as a cloud service," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 13–24, 2012.

[5] D. Dietrich *et al.*, "Multi-provider service chain embedding with nestor," *IEEE Transactions on Network and Service Management*, vol. 14, no. 1, pp. 91–105, 2017.

[6] A. Abujoda and P. Papadimitriou, "Midas: Middlebox discovery and selection for on-path flow processing," in *IEEE COMSNETS*, 2015.

[7] D. Dietrich *et al.*, "Network function placement on virtualized cellular cores," in *IEEE COMSNETS*, 2017, pp. 259–266.

[8] A. Abujoda and P. Papadimitriou, "Distnse: Distributed network service embedding across multiple providers," in *IEEE COMSNETS*, 2016.

[9] A. Chakrabarti, C. Chekuri, A. Gupta, and A. Kumar, "Approximation algorithms for the unsplittable flow problem," *Algorithmica*, vol. 47, no. 1, pp. 53–78, 2007.

[10] S. Dräxler, H. Karl, and Z. Á. Mann, "Jasper: Joint optimization of scaling, placement, and routing of virtual network services," *IEEE Transactions on Network and Service Management*, vol. 15, no. 3, 2018.

[11] L. Gong, H. Jiang, Y. Wang, and Z. Zhu, "Novel location-constrained virtual network embedding lc-vne algorithms towards integrated node and link mapping," *IEEE/ACM Transactions on Networking*, vol. 24, no. 6, pp. 3648–3661, 2016.

[12] 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.

[13] A. Colorni, M. Dorigo, F. Maffioli, V. Maniezzo, G. Righini, and M. Trubian, "Heuristics from nature for hard combinatorial optimization problems," *International Transactions in Operational Research*, vol. 3, no. 1, pp. 1–21, 1996.

[14] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, "Near optimal placement of virtual network functions," in *IEEE INFOCOM*, 2015.

[15] C. Papagianni *et al.*, "Rethinking service chain embedding for cellular network slicing," in *IFIP Networking*, 2018, pp. 1–9.

[16] A. Basta, W. Kellerer, M. Hoffmann, H. J. Morper, and K. Hoffmann, "Applying nfv and sdn to lte mobile core gateways, the functions placement problem," in *Proceedings of the 4th workshop on All things cellular*, 2014, pp. 33–38.

[17] A. Pentelas, G. Papathanail, I. Fotoglou, and P. Papadimitriou, "Network service embedding across multiple resource dimensions," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, 2021.

[18] C. Renzi, F. Leali, M. Cavazzuti, and A. O. Andrisano, "A review on artificial intelligence applications to the optimal design of dedicated and reconfigurable manufacturing systems," *The International Journal of Advanced Manufacturing Technology*, vol. 72, no. 1-4, 2014.

[19] H. Guo and W. H. Hsu, "A machine learning approach to algorithm selection for *NP*-hard optimization problems: a case study on the mpe problem," *Annals of Operations Research*, vol. 156, no. 1, 2007.

[20] A. Diveev and O. Bobr, "Variational genetic algorithm for np-hard scheduling problem solution," *Procedia Computer Science*, vol. 103, 2017.

[21] P. V. Paul, N. Moganarangan, S. S. Kumar, R. Raju, T. Vengattaraman, and P. Dhavachelvan, "Performance analyses over population seeding techniques of the permutation-coded genetic algorithm: An empirical study based on traveling salesman problems," *Applied soft computing*, vol. 32, pp. 383–402, 2015.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms second edition," *The Knuth-Morris-Pratt Algorithm*, 2001.

[23] "Edge Vector libraries," https://rodispantelis.github.io/EdgeVector/.

[24] "DIMACS graph set," http://archive.dimacs.rutgers.edu/pub/challenge/graph/benchmarks/clique/.

[25] S. M. Lim, A. B. M. Sultan, M. N. Sulaiman, A. Mustapha, and K. Y. Leong, "Crossover and mutation operators of genetic algorithms," *International journal of machine learning and computing*, vol. 7, no. 1, pp. 9–12, 2017.

European Union
European Regional Development Fund
HELLENIC REPUBLIC
MINISTRY OF ECONOMY & DEVELOPMENT
SPECIAL SECRETARY FOR ERDF & CF
MANAGING AUTHORITY OF EPAnEK
EPAnEK 2014-2020
OPERATIONAL PROGRAMME
COMPETITIVENESS
ENTREPRENEURSHIP
INNOVATION
ΕΣΠΑ 2014-2020
ανάπτυξη - εργασία - αλληλεγγύη
Partnership Agreement
2014 - 2020
Co-financed by Greece and the European Union