Evolutionary Computation for Latency Minimization in SDN Microservice Architectures

Evolutionary Computation for Latency Minimization in SDN Microservice Architectures

José Gómez-delaHiz*, Juan Luis Herrera†, Domenico Scotece†, Jaime Galán-Jiménez*, Javier Berrocal*, Giuseppe Di Modica†, and Luca Foschini†

* Dept. of Computer Systems and Telematics Engineering, University of Extremadura, Spain. e-mail: jagomezdh@unex.es

† Dipartamento di Informatica Scienza e Ingegneria, University of Bologna, Italy. e-mail: juanluis.herrera@unibo.it

***Abstract*—In recent years, Software-Defined Networking (SDN) research literature has proposed the integration of multiple SDN controllers into the same network, improving the scalability and reliability of the network. However, while this evolution has focused on control plane hardware, the architecture of SDN controller software is still monolithic, and its communication with the application plane through the northbound interface is done by the integration of the network-level applications' codebase with the controller software. The proposal of SDN Microservices Architectures (SDN MSAs) is aimed at transforming the application plane, from a monolithic architecture to a set of independently deployable modules named SDN microservices. However, the promising paradigm of SDN MSAs also increases the complexity of network management, as these microservices must be placed through the SDN controllers. This placement is especially complex due to its NP-hard nature. In this work, we present Genetic Algorithm for SDN MSA (GASM), an evolutionary computation-based heuristic to solve this issue in tractable times. Experimental results show that GASM represents an average speed-up of 846.33× compared to optimal solvers.**

***Index Terms*—Software-Defined Networking, Microservices, Optimization, Evolutionary Computation**

I. Introduction

The Software-Defined Networking (SDN) paradigm has revolutionized networks by enabling network programmability and enhancing flexibility.In SDN, network switches represent only the *data plane*, forwarding traffic according to the rules installed in them. These rules are installed by the *control plane*, embodied by SDN controllers, which communicate with switches through their *southbound interface*.Moreover, each SDN controller exposes a *northbound interface*, allowing the *application plane* to interact with the network.The application plane thus enables network-level applications, such as service discovery, firewalls, or traffic engineering, to define the behavior of the network. Although the original SDN definition proposes the SDN controller as a centralized entity, such an approach can lead to issues such as a single point of failure and limits the scalability of the network [1]. Therefore, the recent literature on SDN proposes deploying multiple SDN controllers, enabling a distributed control of the network [2].

In practice, the northbound interface of each of these SDN controllers is commonly offered through *SDN controller frameworks*, i.e., specialized software that is installed in SDN controllers [2]. The northbound interface is often available as a set of programming libraries that allow developers to define the behavior of the SDN controller in response to certain events (e.g., through *callbacks* that integrate into the SDN framework's runtime loop) [2]. In these cases, the northbound interface requires the integration of the network-level applications with the code of the SDN framework itself. Hence, from a software architecture perspective, SDN controller software is *monolithic*: a single program that, while modifiable, holds all the logic for the framework and all network-level applications in a single process. This approach clashes with having distributed control of the network, as the multiple SDN controllers must run full replicas of the same monolithic software.

In the field of software engineering, Microservice Architectures (MSAs) were introduced as an architecture designed for distributed systems, in contrast with the traditional monolithic architecture [3]. An MSA-based application is structured as a set of loosely-coupled modules, named *microservices*, that can be deployed independently, where each microservice is only responsible for a small subset of the complete application's functionality [3]. The microservices in an MSA can collaborate to perform complex functionalities, thus providing the same functionality as a monolithic application [3]. MSAs allow microservices to be replicated across multiple devices without requiring each device to host every microservice, enable collaboration across microservices developed with different technologies, can be deployed in a massively distributed manner, and are highly evolvable [3]. All these characteristics are also desirable for the SDN application layer, and thus, there is an ongoing research effort on the development of SDN controller frameworks based on MSAs [2]. This new architecture is an *SDN MSA*, i.e., an SDN network whose controller is based on an MSA. The data plane, which contains the network topology (i.e., switches and links) does not suffer changes w.r.t. a monolithic SDN, nor does the southbound interface or the control plane. Nonetheless, the northbound interface changes the application plane, as each network-level application is now developed as a microservice, enabling each controller to deploy a different set of microservices. Moreover, these microservices can be replicated across the controllers, and the instantiation of new or different microservices does not require a modification of the SDN controller framework, enabling the microservice deployment to be modified in runtime.

However, while the properties of MSAs are beneficial to SDN, they also bring an increase in the complexity of the

management of SDN controller software. Rather than having a monolithic software that must be deployed and replicated in all controllers at the same time, it is necessary to decide on the number of replicas for each microservice, as well as on which SDN controller to deploy each replica, as these decisions affect the Quality of Service (QoS) of the network [4]. The problem of deciding on the replication of microservices and where to place them to optimize QoS in the software domain is known to be NP-hard [4], and hence, its SDN analogous is also NP-hard. Although it is possible to use techniques such as mathematical programming to solve this problem, its scalability is limited, as the computational resources required to solve the problem and the time taken to find a solution grow factorially with the size of the network and the number of SDN microservices [4]. Therefore, it is necessary to develop heuristics, able to obtain near-optimal solutions in tractable times and with a smaller resource footprint.

In this paper, we present Genetic Algorithm for SDN MSA (GASM), a heuristic based on evolutionary computation to replicate and place the microservices of SDN MSAs in short times and with a small resource footprint. GASM optimizes the latency experienced by the flows of the network, and its short execution time allows network operators to quickly adapt the microservice placement and replication to changes in the distribution of traffic. The main contributions of this paper are: i) the description of the SDN microservice placement and replication problem, ii) the proposal of GASM, a heuristic based on evolutionary computing for the solution of the problem, iii) the evaluation of GASM's performance in a realistic network scenario, and iv) the comparison of GASM's performance with alternative solutions.

The remainder of this paper is structured as follows. Section II introduces related works on SDN MSAs. Section III describes the model of an SDN MSA as used by the heuristic. The design of the heuristic is detailed in Section IV, and its implementation is evaluated in Section V. Finally, Section VI concludes the paper.

II. Related works

Traditionally, the research community has focused on the study and development of SDN networks with centralized control in a single SDN controller, for example, implementing a proactive rule installation mechanism to decrease the delay of rule requests between network devices and the controller [5]; performing tasks such as improving IoT data analysis by using Machine Learning to unify SDN and virtual network functions [6]; and even combining these last two paradigms with multi-access edge computing, thereby reducing latency and increasing capacity at the network edge, thus meeting the requirements of the IoT ecosystems [7].

However, the decentralization of SDN control by having multiple SDN controllers in the same network rather than a single controller is currently advised as a good practice in the literature [1], [8]. Nonetheless, the use of multiple SDN controllers also brings a higher complexity to their management. In this regard, one of the most important problems is to decide where to place the different SDN controllers in the network topology [8]. This problem is known in the literature as the SDN Controller Placement Problem (CPP), and its solution is currently an open research topic [8].

Multiple authors have proposed the use of different techniques to solve the CPP. In [9], Zhang *et al.* present a heuristic solution for the CPP that considers three objectives: latency minimization, reliability maximization, and load balancing. Load balancing is especially interesting: it states that, as the computing capacity of SDN controllers is limited, it is necessary to split the processing of flows across controllers. SDN MSAs provide a more effective manner to split the processing by deploying different microservices. Another approach to the CPP is presented in [10], which proposes the term of *capacitated CPP*, extending the CPP to consider the limit in the computational capacity of SDN controllers as a crucial feature. Overall, the CPP is a related, but different problem, tackled at the network planning phase, and that must be solved before an SDN MSA is deployed. Nonetheless, the considerations in the capacitated CPP and the interest in load balancing highlight the relevance of SDN MSAs.

On the other hand, it is also important to highlight the similarities and differences between SDN MSAs and Service Function Chaining (SFC) [11]. SFC is an architecture that leverages SDN to perform multiple operations over traffic flows as they are routed over the network. SFC considers the existence of different *service functions*, which are similar to SDN microservices, as they perform complex functionalities over traffic flows. These service functions can be requested by flows to be executed as part of a *chain*, that is, in a given order. To do so, service functions are installed on switches, and traffic flows must be routed through a certain path (*Service Function Path*), that must traverse switches with the corresponding service functions installed, and in the corresponding order.

SDN MSAs have two similarities to SFC: they propose the use of functional modules that perform operations over traffic flows (service functions and SDN microservices) and require traffic to be steered according to the deployment of these modules. Nonetheless, SDN MSAs and SFC have key differences that make them different architectures and paradigms. The main difference between SDN MSAs and SFC is their network plane: SDN MSAs are an architecture for the application plane, and SDN microservices are deployed to SDN controller hardware [2], while SFC is a data plane architecture whose service functions are deployed to SDN switches [11]. Moreover, SDN microservices are more open in their functionality, as their behavior can change dynamically, while service functions are, by nature, static (e.g., a firewall implemented as an SDN microservice can block different ports at different points of time by querying an external database in execution time, while SFC would require different firewall service functions). Finally, SDN microservices are commutative, while service functions must be executed in a certain order. Therefore, while works like [12] propose heuristics for latency optimization in SFC, unlike GASM, they cannot be applied to SDN MSAs due to their differences.

III. System model

To understand the design of GASM, including the decisions it must take, the information it should have, and its objective, it is crucial to first understand its model of an SDN MSA. Concretely, the system model should describe the SDN microservice model, the traffic flow model, and the SDN control model. This section details each of these elements.

Before managing the microservices in an SDN MSA, it is necessary to know where these microservices can be deployed, i.e., what is the placement of the SDN controllers in the network topology. As the decision of SDN controller placement is made in an early stage, namely in the network planning phase [8], the SDN MSA model assumes that the placement of controller hardware has already been decided. Moreover, we assume that this placement complies with the classic SDN controller placement model: all SDN controllers have the same hardware, are co-located with an SDN switch, and each SDN switch communicates with exactly one controller [8].

The SDN MSA will therefore execute on these controllers. To do so, the SDN MSA defines the set of available microservices or *microservice types* that can be instantiated in the topology. To better illustrate the model, we will use a running example for the SDN MSA, fully depicted in Fig. 1. In this running example, there are two SDN controllers (shown as boxes), five switches (each assigned to the controller of the same color), and three microservice types: topology management (denoted *T*), service discovery (*S*), and firewall (*F*). Each of these types also declares the amount of computing resources an instance consumes in an idle state, as well as the resources it consumes to handle a request. The types will then be instantiated in the SDN controllers as decided by GASM, always considering that each controller can have up to one instance per type (e.g., a controller may or may not host the firewall, but it should not host two or more firewalls, as a single instance will process all the necessary requests). In terms of resources, the capacity of SDN controllers should not be exceeded: the sum of the resources consumed by microservices, including both those consumed by idle instances of microservices and those consumed by handled requests, must not be higher than the total resources of the controller. Moreover, as the hardware of all SDN controllers is equal, the execution time of a microservice is constant across controllers.

Next, it is important to consider that the microservices are requested by the traffic flows in the network, which can be referred to collectively as the *traffic matrix*, where each *demand* of the traffic matrix is a traffic flow. Each of these flows has an ingress node, which serves as the source of the flow in the network, and an egress node, which can be seen as its destination. These are denoted as I and E, respectively, in Fig. 1. The flow must go from the ingress node to the egress node, through a certain route. GASM must, thus, decide the routes of all flows (traffic routing). To do so, it must consider that each link in the network topology has a certain capacity that cannot be exceeded, while each traffic flow has a certain size. Moreover, each flow can request one or more

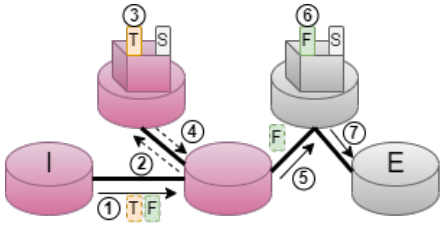

Figure 1: Example of a flow's routing and microservice execution in SDN MSA.

SDN microservices. In our running example, we consider a flow that requests the topology management microservice and the service discovery microservice. These microservices must be executed for the flow *by the time it reaches the egress node*, although it is not necessary to execute them in any strict order. When the flow is routed through a switch connected to an SDN controller that hosts the microservice, the SDN switch will communicate with the SDN controller to execute the microservice. This communication is denoted as the *control flow* between the switch and the controller, and its size is also considered towards link capacity consumption. If the route of the flow includes a switch co-located with an SDN controller that hosts the microservice, it is executed on that switch without the need for control flow. In Fig. 1, the flow is first routed to an SDN switch connected to the pink controller (1). This switch establishes a control flow communication with the controller (2), while the controller executes the topology management microservice (3) and returns the result to the switch (4). Next, the flow, which now only requests the firewall is sent to the switch co-located with the grey controller (5). Hence, the firewall microservice is executed locally (6). Finally, the flow arrives at its destination (7).

The objective of GASM is, therefore, to decide where to instantiate microservice replicas, how to route traffic, and how to route control flows, to minimize the total latency experienced by the system. This latency is the sum of all the latencies of the links traversed by the flow (1, 5, and 7 in Fig. 1), as well as the latencies of control flows (2 and 4 in Fig. 1), and the execution time of microservices (3 and 6 in Fig. 1). This model is the blueprint used to build GASM, and thus, it is compliant with the presented system model.

IV. Heuristic description

The use of heuristic algorithms allows for obtaining approximately optimal solutions to optimization problems, trading off a small part of the solution's optimality by a significant decrease in execution time and resource footprint. The proposed framework, GASM, is based on a *genetic algorithm*, an evolutionary computation heuristic technique inspired by the reproduction of living beings that imitates biological evolution as a strategy to solve problems [13]. Genetic algorithms have been successfully used in computer networks research for problems such as energy efficiency optimization [14]. This technique is based on the generation of several possible solutions or a *population* of *chromosomes*, that are composed of a set of *genes* that will change their value throughout the

Algorithm 1 Pseudo-code of GASM.

Require: Network graph: $\mathcal{G} = (\mathcal{N}, \mathcal{L})$, data of the controllers installed: ϵ, available microservices: μ, traffic matrix: $\mathcal{T}$, number of shortest paths calculated per pair of nodes: n_p, population size: κ, max. generations: θ, max. stagnation (generations): ϖ, number of parents to the tournament: n_t, crossover rate: r_c, highest mutation rate: r_m^h, lowest mutation rate: r_m^l

1: create $gen \leftarrow 0$, $P_{\text{gen}} \leftarrow \emptyset$, $sol_{\text{best}} \leftarrow \emptyset$, $fval_{\text{best}} \leftarrow 0, children^* \leftarrow \emptyset, stag \leftarrow 0, gen \leftarrow 0$
2: *paths, latencies* $\leftarrow$ *shortestsPaths*$(\mathcal{G}, n_p)$
3: $P_{\text{gen}} \leftarrow$ *createPopulation*$(dim(\epsilon), dim(\mu), \kappa)$
4: **for all** chromosome c in P_{gen} **do**
5: **if** fitness$(c, \mathcal{G}, \mathcal{T}, \epsilon, \mu, path, latencies) > fval_{\text{best}}$ **then**
6: $sol_{\text{best}} \leftarrow c$
7: $fval_{\text{best}} \leftarrow$ fitness$(c, \mathcal{G}, \mathcal{T}, \epsilon, \mu, path, latencies)$
8: **end if**
9: **end for**
10: **while** $gen \leq \theta \vee stag \leq \varpi$ **do**
11: *parents* $\leftarrow$ *selectParents*$(P_{\text{gen}}, \texttt{tournament}, n_t)$
12: $elitist \leftarrow sol_{\text{best}}$
13: **while** $len(children^*) < \kappa - 1$ **do**
14: *children* $\leftarrow$ *runCrossover*$($*parents*$, r_c, \texttt{scattered})$
15: *children* $\leftarrow$ *runMutation*$($*children*$, r_m^l, r_m^h, \texttt{adaptive})$
16: *children** $\leftarrow$ *children** $\cup$ *children*
17: **end while**
18: **for all** chromosome c in *children** **do**
19: **if** fitness$(c, \mathcal{G}, \mathcal{T}, \epsilon, \mu, path, latencies) > fval_{\text{best}}$ **then**
20: $sol_{\text{best}} \leftarrow c$
21: $fval_{\text{best}} \leftarrow$ fitness$(c, \mathcal{G}, \mathcal{T}, \epsilon, \mu, path, latencies)$
22: $stag \leftarrow -1$
23: **end if**
24: **end for**
25: $stag \leftarrow stag + 1$
26: $P_{\text{gen}} \leftarrow elitist \cup$ *children**
27: $gen \leftarrow gen + 1$
28: **end while**
29: **return** $sol_{\text{best}}, fval_{\text{best}}$

execution of the algorithm to find a better solution, evaluating the *fitness* of each chromosome to determine the best solutions in each generation. In the case of GASM, each chromosome defines the placement of microservices on each of the controllers installed on the network. These chromosomes are then generated and modified with the objective of reducing the maximum latency of all the flows in the network. To do so, each gene in a chromosome represents whether a microservice is deployed on a controller or not. The routing is not directly encoded in the *genotype* of the chromosome (i.e., its genes), and is instead algorithmically calculated and considered in the fitness function as part of its *phenotype*. In the following, the logic of GASM, whose pseudocode is specified on Algorithm 1, is detailed:

Phenotype Interpretation: One of the key design choices of GASM is that traffic routing is not in the genotype of each chromosome, instead, it is interpreted as its phenotype. This interpretation depends on some pre-computed data: the paths across nodes and their latencies. Concretely, given the value of the n_p parameter, the phenotype computes the n_p shortest paths between each pair of nodes $n, n' \in \mathcal{N}, n \neq n'$, and the latency of each of these paths is also stored (line 2). Once this information is computed, the phenotype can be described on a flow-by-flow basis. To do so, for each of the flows, the phenotype interpretation algorithm determines the *most suitable controller*: the controller with the largest subset of microservices of those requested by the flow, and with a calculated path with a higher capacity than the flow's size. If multiple controllers are considered *most suitable*, the one with the shortest path (i.e., lowest latency) to the ingress node is chosen. If all microservices requested by the flow are executed at the most suitable controller, the flow is routed to the egress node through the shortest route with enough capacity available. Otherwise, the process is repeated, finding the most suitable controller considering the current controller instead of the ingress node, and only those microservices that have not yet been executed. Once the complete route is calculated, from the ingress node to the egress node through all the suitable controllers, the phenotype is considered to be interpreted.

Fitness Function: Before describing the algorithm itself, it is also crucial to describe the fitness function that will govern it. This fitness function first checks that the solution encoded in the chromosome is valid: all microservices requests must be satisfied, each controller can run no more than one replica of each microservice, the controller capacity must not be exceeded, and the total size of the flows routed through each link must not exceed the link's capacity either. If at least one of these constraints does not hold, the fitness of the chromosome will be considered as -1 (i.e., the solution is invalid). If these constraints hold, the fitness function interprets the phenotype of the chromosome. For, the phenotype of each of the flows, the total latency of its route is calculated and stored in an array (lat_{flows}). Finally, once the phenotype is interpreted for all the flows in the traffic matrix, the fitness of the chromosome is calculated as $\frac{1}{1+max(lat_{flows})}$.

Initialization: The initial population of the algorithm is randomly generated, its dimension governed by the value of the parameter κ (line 3). Each chromosome is a binary array of a size equal to the product of the number of controllers installed on the network times the number of microservices available. Once generated, the fitness of each of the generated individuals is calculated, also determining the best chromosome in the initial population (lines 4-9).

Parent Selection: Chromosomes will be chosen to combine, mutate, and form new chromosomes ($children$) for the next generation. Among all the existing selection mechanisms [15], GASM uses the *tournament* mechanism, in which tournaments are held between n_t random chromosomes, with the winner (i.e., the chromosome with the highest fitness in the tournament) becoming a candidate for the crossover (line 11).

Crossover and Mutation: The chromosomes selected in the previous step, with a probability equal to r_c, will be crossed to generate a child that will be mutated with a probability between r_m^l and r_m^h. As far as the crossover operator is concerned, a *scattered* combination is performed: the genes of the two parents are considered and, with a probability of r_c, randomly exchange genes with each other to generate new chromosomes (line 14). On the other hand, an *adaptive* mutation is applied, i.e. each generated offspring has a different mutation probability, depending on its fitness (line 15).

Next Generation: The above process is repeated until $\kappa - 1$

chromosomes are obtained (lines 13-17), which joins with the best solution of the previous generation (line 12) to give rise to the new generation (line 26).

Stop Condition: At this point, it is checked if some of the following stopping conditions are met (line 10): exceed the maximum number of generations (θ), or the number of successive generations that have not improved the solution is greater than ϖ. If so, the algorithm is terminated by a stagnation of the algorithm to save time.

V. Performance evaluation

This section evaluates GASM under a realistic network environment. The setup used to evaluate the heuristic is presented in Section V-A, and the obtained results are then described in Section V-B.

A. Evaluation setup

The evaluation has been performed on an SDN MSA with a total of five microservice types: topology management, firewall, encryption, service discovery, and IoT data aggregation. In the scenarios used for evaluation, all flows are considered to request the topology manager microservice, as well as the security microservices (encryption and firewall). Some of the flows are considered calls to application services, and thus, also request for service discovery. Moreover, flows coming from IoT devices also request data aggregation. Hence, each traffic flow requests between 3 and 5 microservices. The MSA is considered to be implemented under the Ryu runtime, which can be divided into microservices [2]. In terms of hardware, the evaluation scenarios consider between 3 and 5 SDN controllers to be placed, with a computing capacity extracted from [16]. The evaluation has been performed on data extracted from real network topology [17]: Abilene. The traffic matrices for these scenarios were also extracted from [17], considering four traffic matrices, each of them representing a quartile of the total traffic volume across all the traffic matrices available. It is important to note that, as the traffic matrices are obtained from real data, the microservice requests and traffic distribution may vary from one to another.

The evaluation has three objectives: the assessment of the optimality gap in terms of latency between an optimal solution and the heuristic, the impact of the use of the heuristic over link loads w.r.t. the use of the optimal solution, and the comparison of execution times between the heuristic and the optimal solution. To achieve such objectives, GASM has been implemented using PyGAD[1], while the optimal solution has been implemented with Gurobi[2]. In relation to GASM, the parameters of the heuristic have been set as follows: $n_p = dim(\epsilon), \kappa = 50, \theta = 50, \varpi = 5, n_t = 3, r_c = 0.9, r_m^h = 0.9, r_m^l = 0.1$. Moreover, due to the memory consumption of this type of solver, the implementation of the optimal solution may finish by either finding the global optimum or by consuming all the available memory, yielding the best solution it was able to find, if any. Both GASM and the optimal solution have been executed in the same machine, with an Intel Xeon Gold 6238R CPU and 16 GB of RAM, under Ubuntu 22.04, to ensure the fairness of the comparison.

[1]https://pygad.readthedocs.io/en/latest/

[2]https://www.gurobi.com/

B. Evaluation results

The first analysis is aimed at comparing the latencies that the network flows experience with GASM and those of the optimal solutions, as shown in Fig. 2. It is important to note that in some cases (e.g., 4 controllers, traffic matrix Q1), the optimal solver cannot find a valid solution before filling up the machine's memory completely, and hence. It is important to note that there are no cases where GASM does not find a solution but the optimal solver does. In these results, the latencies obtained by the flows are only slightly impacted by the number of controllers deployed, with a reduction of 1 ms in average per additional controller in the optimal case, and a 2 ms average difference in GASM. The traffic matrix has also negligible effects on the experienced latencies which fluctuate more on different distributions of traffic than on different volumes. Overall, the standard deviation of the latencies is small, approximately 1.68 ms for GASM, and 3.6 ms for the optimal solution. The optimality gap between GASM and the optimal solution is slightly higher, at 7.6 ms on average, but still very low considering the average link latency of 3.11 ms. Thus, the solutions that GASM yields are considered acceptably similar to the optimal solutions.

The next analysis, depicted in Fig. 3, addresses the average link load of the network in the same cases, i.e., scenarios where either GASM or the optimal solver are able to find a solution. In the case of the link load, it is greatly affected by the traffic matrix, especially in scenarios with a very high load (i.e., Q4). The number of controllers slightly affects the load as well, with a higher number of controllers being correlated with a higher link load, especially on the optimal solution. This phenomenon appears because a higher number of controllers makes it more efficient in terms of latency to route traffic through switches with SDN controllers, and hence, to increase the load in such links. Nonetheless, the link load is rather low in general terms (1.34% in the optimal case, and 2.07% with GASM), with a significant increase in Q4 traffic matrices, but still very low, with a maximum load of 4.1% in the optimal case and 6.87% with GASM. On average, GASM loads the links 0.73% more than the optimal solution, which is a negligible increase. Hence, the heuristic approach of routing traffic directly through SDN switches with controllers at all times does not reflect a great impact on the link load.

Finally, the last analysis from Fig. 4 shows the execution time of GASM and the optimal solver, labeled *optimization time* to avoid confusion with the execution time of SDN microservices. Starting with the optimization times of the optimal solver, they are all very similar, ranging between 3570 and 3660 seconds (3639.23 seconds on average). These similarities are due to the memory of the machine, which the optimal solver tends to fill completely after approximately one hour of execution time.. GASM, on the other hand, requires between 2 and 5.1 seconds to find a solution (4.3 on average),

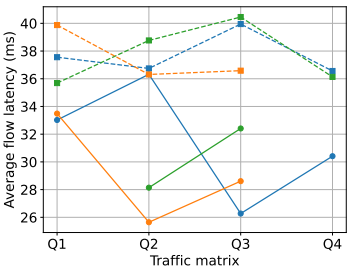

Figure 2: Comparison of the latencies achieved by GASM and the optimal solution.

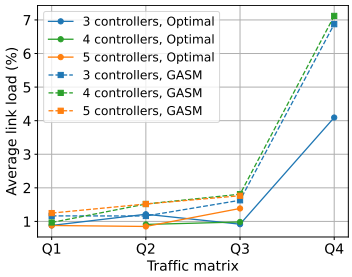

Figure 3: Comparison of the link load in GASM's solutions and in optimal solutions.

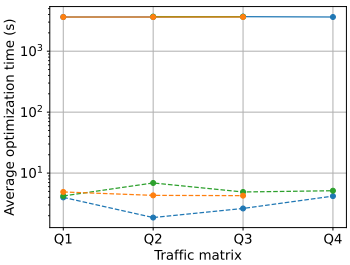

Figure 4: Comparison of the execution time of optimizations in GASM and the optimal solution.

generally taking more time on cases with more loaded traffic matrices, or with a higher problem size (e.g., more controllers). In terms of speed-up, GASM achieves an average speed-up of 846.33× w.r.t. the optimal solution. Hence, GASM is notably faster and more efficient than an optimal solver.

VI. Conclusions and Future Work

Due to the rise in interest of decentralizing the SDN control plane through the deployment of multiple, distributed SDN controllers in networks, proposals for the decentralization of the application plane have appeared in the literature. One of the most promising proposals in this regard, SDN MSAs, are promising, bringing the characteristics of MSA-based applications to the network. However, leveraging SDN MSAs also comes with the increasing complexity in management that characterizes software MSAs, including the problem of placing SDN microservices across a network topology. In this paper, we presented GASM as an evolutionary computation-based heuristic to address this NP-hard problem. The evaluation shows that GASM achieves near-optimal latencies and a 846.33× average speed-up w.r.t. optimal solvers. In the future, we expect to compare GASM with other meta-heuristic approaches to the problem. Furthermore, we also expect to perform experiments in real or emulated network testbeds.

Acknowledgements

This work was partially funded by the project PID2021-124054OB-C31 and the grant CAS21/00057 (MCI/AEI/FEDER, UE), by the grant PDC2022-133465-I00 and the project TED2021-130913B-I00 funded by MCIN/AEI/10.13039/50100011033 and by the "European Union NextGenerationEU/PRTR", by the Department of Economy, Science and Digital Agenda of the Government of Extremadura (GR21133), and by the European Regional Development Fund. This work was partially supported by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART"). CUP: J33C22 002880001.

References

[1] F. Bannour, S. Souihi, and A. Mellouk, "Distributed sdn control: Survey, taxonomy, and challenges," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 333–354, 2018.

[2] S. T. Arzo, D. Scotece, R. Bassoli, D. Barattini, F. Granelli, L. Foschini, and F. H. Fitzek, "Msn: A playground framework for design and evaluation of microservices-based sdn controller," *Journal of Network and Systems Management*, vol. 30, pp. 1–31, 2022.

[3] K. Indrasiri. Microservices in practice - key architectural concepts of an MSA. (visited on Aug. 28, 2023). [Online]. Available: https://tinyurl.com/msa-architecture

[4] J. L. Herrera, J. Galán-Jimnez, J. Garcia-Alonso, J. Berrocal, and J. M. Murillo, "Joint optimization of response time and deployment cost in next-gen iot applications," *IEEE Internet of Things Journal*, vol. 10, no. 5, pp. 3968–3981, 2022.

[5] L. Sanabria-Russo, J. Alonso-Zarate, and C. Verikoukis, "Sdn-based proactive flow installation mechanism for delay reduction in iot," in *2018 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2018, pp. 1–6.

[6] J. Serra, L. Sanabria-Russo, D. Pubill, and C. Verikoukis, "Scalable and flexible iot data analytics: When machine learning meets sdn and virtualization," in *2018 IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*. IEEE, 2018, pp. 1–6.

[7] I. Sarrigiannis, K. Ramantas, E. Kartsakli, P.-V. Mekikis, A. Antonopoulos, and C. Verikoukis, "Online vnf lifecycle management in an mec-enabled 5g iot architecture," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4183–4194, 2019.

[8] G. Deep, V. Tripathi, and A. Dumka, "A review on controller placement problem in software defined networking," in *AIP Conference Proceedings*, vol. 2521, no. 1. AIP Publishing, 2023.

[9] B. Zhang, X. Wang, and M. Huang, "Multi-objective optimization controller placement problem in internet-oriented software defined network," *Computer Communications*, vol. 123, pp. 24–35, 2018.

[10] A. K. Singh, S. Maurya, and S. Srivastava, "Varna-based optimization: a novel method for capacitated controller placement problem in SDN," *Frontiers of Computer Science*, vol. 14, no. 3, p. 143402, 2020.

[11] M. Polverini, J. Galán-Jiménez, F. G. Lavacca, A. Cianfrani, and V. Eramo, "A scalable and offloading-based traffic classification solution in nfv/sdn network architectures," *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 1445–1460, 2020.

[12] M. M. Tajiki, S. Salsano, L. Chiaraviglio, M. Shojafar, and B. Akbari, "Joint energy efficient and qos-aware path allocation and vnf placement for service function chaining," *IEEE Transactions on Network and Service Management*, vol. 16, no. 1, pp. 374–388, 2018.

[13] D. E. Goldberg, "The genetic algorithm approach: why, how, and what next?" in *Adaptive and learning systems: Theory and applications*. Springer, 1986, pp. 247–253.

[14] J. Galán-Jiménez, M. Polverini, F. G. Lavacca, J. L. Herrera, and J. Berrocal, "Joint energy efficiency and load balancing optimization in hybrid ip/sdn networks," *Annals of Telecommunications*, vol. 78, no. 1-2, pp. 13–31, 2023.

[15] L. M. Schmitt, "Theory of genetic algorithms," *Theoretical Computer Science*, vol. 259, no. 1, pp. 1–61, 2001.

[16] F. Benamrane, R. Benaini *et al.*, "Performances of openflow-based software-defined networks: an overview," *Journal of Networks*, vol. 10, no. 6, p. 329, 2015.

[17] S. Orlowski, M. Pióro, A. Tomaszewski, and R. Wessäly, "SNDlib 1.0–Survivable Network Design Library," in *Proceedings of INOC 2007*, April 2007.