Automation and Multi-Objective Optimization of Virtual Network Embedding

Pedro Martinez-Julia, Ved P. Kafle, and Hitoshi Asaeda
Network Science and Convergence Device Technology Laboratory
National Institute of Information and Communications Technology
4-2-1, Nukui-Kitamachi, Koganei, Tokyo 184-8795, Japan
Email: {pedro,kafle,asaeda}@nict.go.jp

***Abstract*—The need for automated management is continuously increasing, especially with the advent of network virtualization and slicing technologies. However, finding the optimum configuration for a virtual network before it is embedded onto the substrate network is a problem that cannot be resolved by exact and deterministic mathematical operations. In this paper we propose a novel heuristic for building an algorithm based on genetic programming for optimizing the placement of virtual network function instances before they are deployed, so more instances can be deployed on the same substrate network without incurring in overloads and delays. Each solution given by our algorithm is based on a previous solution, following dynamic programming scheme to minimize processing and enforcement efforts. Therefore, the algorithm accomplishes with the time constraints set by current demands. We demonstrate this quality and compare our algorithm to previous solutions, also based on genetic programming and already providing quite fast responses for the embedding problem.**

I. Introduction

The wide adoption of Virtual Networks (VNs) is depending on their ability to be quickly adapted to the new situations found in the changing environments they usually operate. Network Function Virtualization (NFV) and Network Slicing (NS) facilitate the capabilities required for such adaptation, although they raise their own challenges [1]–[3]. The most outstanding challenge is to find out the best configuration for embedding a VN onto a Substrate Network (SN).

In this paper we introduce the FADE architecture, standing for Fast Analysis and Driving of virtual network Embedding operations. It has the objective of automating the management of large VNs, emphasizing the inclusion of fast and efficient embedding operations. This is achieved by choosing new configurations and enforcing them when needed. For this, FADE includes a new algorithm based on the Genetic Programming (GP) and Dynamic Programming (DP) methodologies, together with a new heuristic function that is particularly designed to exploit deductive reasoning techniques to adapt VNs to dynamic requirements, as detailed below. This algorithm makes our solution to perform better and be more scalable than existing proposals.

The novelty behind the new solution is based on the application of intelligent reasoning to the main steps of the management process. First, performance measurements obtained from the VN are analyzed together with events notified by external detectors in order to find the most appropriate amount of Virtual Network Function (VNF) instances needed to keep the VN as much efficient as possible while avoiding the disruption of its normal operation. After this, the best configuration for the VN to be embedded onto the SN, is automatically reasoned from current configuration by considering the new amount of VNF instances and avoiding the costly operation of removing/adding instances as much as possible.

An assignation of nodes from the SN (NSNs) to a VN is formalized as the Virtual Network Embedding Problem (VNEP). A solution to this problem is a map from VNF instances to NSNs. It must ensure that all nodes and links from the VN are embedded onto the SN. These solutions must ensure that the goals set by the tenants of both the VN and the SN are respected. However, it is well known that conventional methods to resolve the VNEP cannot be both subject to find the optimum and resolved in polynomial time [4]. Therefore, in this paper we design an algorithm, A, that constructs valid configurations, viz. solutions of the VNEP, guided by a new heuristic function. Our solution is based on GP to avoid getting stuck in local optimums and the heuristic is specifically designed to exploit the benefits of DP.

The most relevant innovation of FADE resides, on the one hand, in the sequential application of H to analyze different alternatives while A builds a new configuration. On the other hand, FADE has improved scalability in comparison to State of the Art (SotA) solutions. In addition, FADE considers the impact of embedding whole VNs instead of individual Virtual Machines (VMs), so the resulting configurations have better consistency for both the VN and the SN. In summary, our algorithm is 6 times faster than SotA solutions to estimate a proper configuration for embedding a new VN or changing an existing VN to a new configuration.

The remainder of this paper is organized as follows. First, in Section II we formalize the VNEP and discuss the related work that contextualize the present work. Second, in Section III we describe our proposal to approach the VNEP. Third, in Section IV we introduce the FADE architecture that incorporates the new enforcement algorithm. Fourth, in Section V we evaluate the solution to demonstrate our claims. Fifth and finally, in Section VI we conclude the paper and discuss our future work.

978-3-903176-32-4

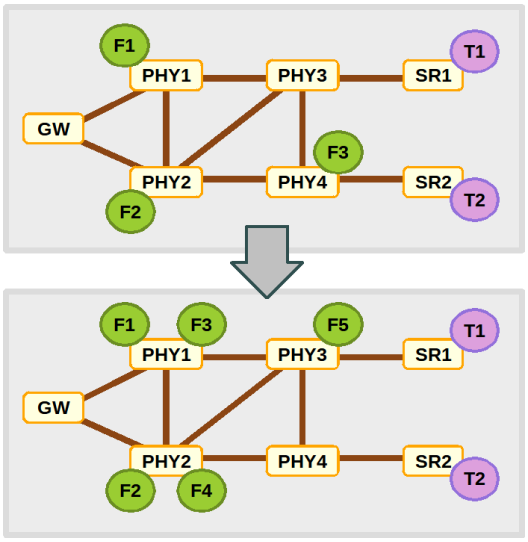

Figure 1. Illustrative example of the changes required to adapt a VN after deciding to add two new VNF instances and resolving the VNEP to get the best configuration. *GW* is the gateway, *PHY* are NSNs that instantiate functions, labeled as *F*, and *SR* nodes are high-end servers hosting resources, labeled as *T*.

II. Background and Problem Formulation

The main objective of the solution proposed in this paper, as introduced above, is to automate the adaptation and embedding of VNs on SNs. The ultra-low-latency target imposed by the *Tactile Internet* [5] require the VNEP to be resolved within a restricted time. In this section we formalize the VNEP and discuss the most relevant existing solutions for it.

A. Virtual Network Embedding Problem

Resolving the VNEP provides configurations for the VN that consist on the assignation of a NSN to each element of the VN. The goal of the VNEP is set by administrators. It is, usually, the minimization of the resources used from the SN but, as discussed below, there can be additional goals and/or constraints, such as minimizing the number of elements from the VN deployed onto the same NSN.

We denote a solution to the VNEP as C_i. In Figure 1 we illustrate a minimal network configuration, before and after an adaptation that consists on adding two new VNF instances, *F4* and *F5*. Meanwhile, *F1* and *F2* are not changed, *F3* is moved from *PHY4* to *PHY1*, and the new functions respectively deployed in *PHY2* and *PHY3*. Each C_i is a set of element pairs, $(f_i, s_i) \in F \times S$. All elements of the VN must be assigned, so all elements from F must be in C_i. However, C_i only has a subset of S because it is not required to use all NSNs. This definition is formalized in the following equation:

$$\begin{aligned} C_i = &\{(f_1, s_1), (f_2, s_2), \ldots, (f_n, s_n)\} \\ &(f_i, s_i) \in F \times S, \quad \forall f_i \in F, \quad \exists s_i \in S \end{aligned} \tag{1}$$

With this definition for a configuration we proceed to define the estimation of cost, Q_{C_i}. As it must have a general application, it must consider not just the VMs but also the network links and/or paths. Therefore, the cost, Q_{C_i}, must be directly proportional to the number of VNF instances deployed in the network and directly proportional to the lengths of the paths that connect the gateway of the network and the NSNs used in the configuration. In addition, to reflect the effect of overloading single nodes and/or the links associated with them, Q_{C_i} must be also proportional to the number of VNF instances that are deployed in every NSN. This cost is formalized in the following equation:

$$Q_{C_i} = \sum_{s_i \in C_i} \left(L_i + Z^{N_i}\right) \tag{2}$$

It relies on the definition of configuration shown in Equation 1, so C_i is a set of associations between f_i and s_i. L_i is the length of the path from the gateway and s_i, N_i is the number of VNF instances deployed in s_i, and Z, which is defined by administrators, is exponentiated to N_i to define the affinity (or repulsion) for several VNF instance to be deployed onto the same NSN.

So far we have considered not just the deployment of VNF instances onto the NSNs, which is where most solutions end, but we have also considered the effect on the network and the potential overloading of single nodes. However, as discussed above, the embedding procedure must also choose the configuration that minimizes the changes required to be effected. Therefore, if the amount of servers is being increased, the current configuration, C_0, must be a subset of the new configuration, C_i. Otherwise, if the amount of server is being decreased, the new configuration, C_i, must be a subset of the current configuration, C_0. This constraint can be formulated in the following equation:

$$\text{Max } |C_i \cap C_0| \tag{3}$$

As the VNEP targets the minimization of of such cost function, we can include the cost from Equation 2 to formulate the global objective of the problem in the following equation:

$$\text{Min } \left(\frac{1}{|C_i \cap C_0|} + \sum_{s_i \in C_i} \left(L_i + Z^{N_i}\right) \right) \tag{4}$$

This general objective targets both minimizing the cost of configuration and minimizing the changes needed to enforce it. Both are not related, and cannot be linearly dependent, so they justify the multi-objective of our solution. This way, using this multi-objective target, the solution obtained for the VNEP is as much efficient as possible in terms of both cost of the resulting configuration and the cost of making the necessary changes to obtain it.

B. Related Work

We can find in previous work several approaches to resolve the VNEP. However, since Integer Linear Programming (ILP) methods cannot provide a solution within the time constraints we target in this paper, as introduced above, we discard all approaches that use it. Instead, we consider the most outstanding approaches that make use of Artificial Intelligence (AI)

methods, focusing on those that use a Genetic Algorithm (GA) or build their solutions based on GP. We will compare our soulution to the most relevant solutions using GA.

First, the architecture proposed by [6], [7] allows optimizing the capacity of any VN, and modifying the amount of resources assigned to its constituent VNF instances in response or anticipation to different stimuli. It decides to increase or decrease the resources the VN it manages by using Machine Learning (ML). It also makes use of Case Based Reasoning (CBR) for obtaining explanations to such decisions, relying on both telemetry information taken directly from the VNF instances and information reported by external event detectors of different types. This makes the solution able to consider, for instance, the occurrence of an earthquake or a heavy rainfall to adapt VN [8]. Although this solution can provide fast and efficient decisions for the allocation of VNF instances, it lacks consideration of efficient embedding of the decided instances, which is very important.

The work discussed in [9] presents a GA derived from the definition of a problem for ILP, demonstrating that this is not suitable for online scaling of VNFs in response to traffic changes, measuring hours to finish. In contrast, by applying GA to the problem, the authors are able to achieve a solution that can decide server and network allocations for hundreds of policies in milliseconds. However, the performance of GA highly depends on the number of nodes. Moreover, it is necessary to rely on *local* solutions because *global* solutions require much more time to be found and, what is worse, they require to re-arrange the network configuration to match the optimum assignation of resources.

What has been clearly stated by previous work is that the algorithms used to find out an optimal (or optimum) configuration for the network to satisfy new requirements must consider the current configuration in order to minimize the disturbances to the existing workloads, viz. computation processes and traffic flows. This encourages us to work in advancing the SotA with the application of GP with a better heuristic and the inclusion of DP. This advancement is important because, although it is specialized on *local* optimums, with a proper heuristic, it can find the *global* optimum in less time than ILP. We will elaborate this case in future work to find out the best alternative.

III. Enforcement Algorithm

In this section we discuss the algorithm that resolves the VNEP for Spade, the overall management solution proposed in this paper, as described in Section IV. The major quality of this algorithm is that it is able to meet the time constraints stated above while also being able to embed large VNs. To do so, the algorithm is based on an heuristic function that guides the selection and evolution of possible configurations for embedding a provided VN onto a provided SN.

By using an heuristically driven search approach, with most heuristics, an algorithm can be stuck into a *local optimum*. Although most of the time such kind of optimums are good enough for most VN embeddings, they can provide some disparate configurations. To prevent such situation, our algorithm follows the GP methodology. GP allows to consider some degree of randomness from one iteration to the next, so disparate solutions are also considered, breaking any local optimum barrier. Our solution makes use of such randomness by including the *creation* method of generating new solutions, as detailed below. This extends the search space beyond the path guided by the heuristic function and allows the algorithm to find better optimal configurations, closer to the *global optimum*. Moreover, this algorithm also follows the Dynamic Programming (DP) methodology to build the configurations incrementally, so the most relevant partial configurations are *cached* for allowing the find procedure to avoid re-calculating all of them when exploring different paths.

The configuration returned by our solution will be closer to the optimum when the algorithm does more iterations, so it makes more generations of configurations, some of which will be very *far* from the best configuration of each iteration. To find the global optimum, the algorithm should, theoretically, run forever. However, with just as few as 10 generations, as demonstrated in the evaluation below, our solution obtains configurations with heuristic values that are deviated less than 5% from the distance between the optimum and the worst case.

In particular, our solution forces the algorithm to extend its generations until it finds a configuration whose heuristic value is separated more than 99.99% from the first *local optimum*, so it is quite stubborn to find the optimum without incurring in too much time expenses. To design this algorithm we first define the heuristic function in the following equation:

$$h(c) = \sum_{s_i \in c} g(s_i) + \sum_{s_j \in S} Z^{\sum_{s_k \in c} \begin{cases} 0 & \text{if } s_k \neq s_j \\ 1 & \text{if } s_k = s_j \end{cases}} \tag{5}$$

This function sums the lengths of all paths from the gateway of the SN to all the NSNs that form part of the configuration, s_i. The paths are computed previously by the controller of the SN, usually an SDN controller, and stored in a database that is queried by a function denoted as $g(x)$. The function proceeds by summing for each NSN, $s_j \in S$, the result of elevating a constant defined by a parameter, Z, to the number of NSNs, s_j, that are present in c. As the algorithm we design will build incremental configurations, as stated by DP, we need to refine the heuristic function in the following equation:

$$\begin{aligned} h(C_j) = h(C_i) - Z^{\sum_{s_k \in C_i} \begin{cases} 0 & \text{if } s_k \neq s_j \\ 1 & \text{if } s_k = s_j \end{cases}} \\ + g(s_j) + Z^{\sum_{s_k \in C_j} \begin{cases} 0 & \text{if } s_k \neq s_j \\ 1 & \text{if } s_k = s_j \end{cases}} \end{aligned} \tag{6}$$

As seen in the equation above, the heuristic function is split in two parts. First, it retrieves the heuristic value of the configuration that is built immediately before it, C_i, and adjusts it to obtain the heuristic value for the current

Algorithm 1 Building a valid configuration using the heuristic function defined in Equation 6.

```
1: procedure MAKECONF(C_i, F, S)
2:     C_j ← C_i
3:     C ← Empty
4:     for all f_i ∈ F do
5:         for all s_i ∈ S do
6:             C ← C ∪ [C_j ∪ (f_i, s_i)]
7:         end for
8:         SORTBYH(C)
9:         C_j ← FIRST(C)
10:    end for
11:    return C_j
12: end procedure
```

configuration, C_j, by subtracting the exponent of Z for C_i and adding both the length of the path of the newly selected NSN as well as the exponent of Z for C_j. It is therefore making use of the relation described in the following equation:

$$C_j = C_i \cup [f_j, s_j] \tag{7}$$

This defines the tail recursion, through the right tail, for the configurations. For this relation to be consistent, the elements included in the representation of a configuration must be sorted. Moreover, to get a configuration C_j from a configuration C_i, the VNF instance, f_j, that is added to C_i will be the first of the VNF instances not added to C_i. They will be, therefore, also sorted. This ensures the right tail recursion is consistent for all values of $h(C_j)$.

In Equation 6, the exponent derived from Z is the penalty of instantiating more than one VNF instance onto the same NSN. Therefore, the parameter Z is used by administrators to control how hard the management solution will work to avoid overloading nodes in contrast to avoiding spreading VNF instances among all available NSNs.

This heuristic function is used in every iteration of the algorithm that resolves the VNEP. The first point in which it is used is shown in Algorithm 1. It receives a base configuration, C_i, together with a set of VNF instances, F, and a set of NSNs, S. The algorithm returns a new configuration, C_j, which is the result of associating all instances to NSNs and attach the associations to C_i. On an iteration, $f_i \in F$ is assigned to all possible NSNs, and the association is attached to the current C_j to build several new configurations. From them, the configuration with lowest value of h is chosen as the new C_j. The algorithm proceeds in this manner until all $f_i \in F$ are assigned.

It is worth to note that not all possible combinations are built, just for each iteration, so the complexity of the algorithm is kept very reduced. The quality of the final solution depends on the quality of the function used for estimating the heuristic value. With this algorithm we can now proceed to define the overall algorithm to obtain a configuration that resolves the VNEP. We do not compare the configuration quality among the different algorithms evaluated because it is mostly out of the scope of this paper and it would add obscurity to the results presented here.

Algorithm 2 Getting a solution for the VNEP.

```
1: procedure GETSOLUTION
2:     C_0 ← MAKECONF(Empty)
3:     G_i ← [C_0]
4:     while N > 0 do
5:         G_{j_m} ← MUTATE(G_i, L)
6:         G_{j_b} ← BREED(G_i, L)
7:         G_{j_c} ← CREATE(G_i, L)
8:         G_j ← (G_{j_m} ∪ G_{j_b} ∪ G_{j_c}) − G_i
9:         SORTBYH(G_j)
10:        PICKHOMO(G_j, L)
11:        G_i ← G_i ∪ G_j
12:        N ← N − 1
13:    end while
14:    SORTBYH(G_i)
15:    return FIRST(G_i)
16: end procedure
```

By following the GP methodology, as shown in Algorithm 2, we first obtain an initial configuration using the algorithm described above, which is a somewhat good configuration to begin with, and iterate N times in order to explore disparate configurations from the search space. Each iteration uses three different methods to generate new configurations: mutation, breeding, and creation. Mutation obtains new configurations by changing some arbitrary assignations, breeding merges two different configurations to obtain a new one, and creation obtains a random assignation for all $f_i \in F$ over all $s_i \in S$.

All new configurations are then joined and sorted by their heuristic value. From this set, L configurations are picked homogeneously, ensuring the first (best configuration) is included. Apart from the amount of configurations picked, L is used to control the amount of new configurations generated by every method. Then, these configurations are joined to the configurations of the previous generation and a new iteration is performed. After N iterations, the best configuration from the current generation is returned as solution.

IV. Network Management Automation

In this section we discuss the design of FADE. Its major role to achieve the target of this paper is to enclose the algorithm presented above to drive its qualities to automate VN management. This requires an architecture that incorporates the required processes to retrieve telemetry data, analyze it together with information from external events, which will provide strong assertions of knowledge for the reasoning engine to perform as best as possible to determine the amount of resources required by the VN and, finally, enforce it by using the algorithm presented in this paper.

The inclusion of the algorithm presented in this paper makes FADE to advance the SotA in the domain of architectures for VN management, such as [10], [11], especially in terms of

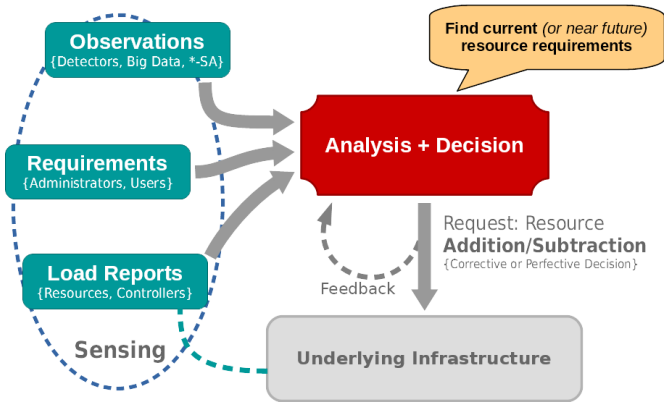

Figure 2. Abstract overview of FADE workflow.

adaptability. It is key for FADE to analyze external events, so it incorporates a CBR, inspired by the solution presented in [7]. This enables the management solution to provide explanations to its decisions, which can be rooted in some external event, so it will actually correlate the amount of VNF instances required by the VN and the information provided by the external event detectors, such as the measurements taken by a seismometer during an earthquake or the detection of a heavy rainfall [8].

In Figure 2 we show the components and overall workflow of FADE. It includes the *collector*, the *analyzer*, the *decider*, and the *enforcer*. These are the four main processes of the MAPE-K closed control loop defined by the Autonomic Computing (AC) paradigm [12], as well as the MANA and ETSI-AFI reference models [13], [14]. The *collector* is responsible of gathering and formatting the heterogeneous observations that will be used in the control cycle. Then, the *analyzer* finds the current load of the resources allocated to the managed system (i.e. the MVN), the rate of packet drops, and the occurrence of an event that can affect its normal operation. The outputs from the analyzer are therefore the *summarized* load, drops, and event. The *decider* determines the necessary actions to adjust the resources to the present or near future load of the managed system and, finally, the *enforcer* is in charge of requesting the underlying and overlying infrastructure to make the necessary changes to enforce decided actions.

The decision algorithm used by FADE to react and anticipate to changes in the environment is inspired on ARCA [6], [7]. Therefore, it makes use of ML for the analysis of telemetry and external events, more specifically, it uses a mix of Support Vector Regression (SVR), the regression application of a Support Vector Machine (SVM) [15], and a reasoner that follows the CBR methodology. The analysis procedure has self-assessment and self-correction capabilities by incorporating a mechanism based on threshold rules (THR) [16]. Combining THR with SVR provides quite effective results and retains a high degree of simplicity. However, this method itself cannot provide explanations to the decisions, which would be essential for improving the overall reasoning process. Such explanations are thus provided by the CBR. Both decisions and explanations are provided to the embedding process, which inputs them to the algorithm presented in this paper to obtain the embedding configuration and communicate it to the controller of the underlying infrastructure.

V. Evaluation and Results

In this section we evaluate Spade to contextualize its performance and general qualities among the SotA. For it we first build a proof-of-concept (PoC) implementation of our solution and then we execute it for VNs of different sizes.

A. Implementation

A key quality of the proposed solution is that, for each iteration of the algorithm, it *reasons* which would be the best configurations, from those obtained by adding an association between VNF instance and NSN, on the one hand, and which would be the best configurations to choose for every generation of the algorithm to evolve towards the best configuration for embedding the VN. This means that the algorithm heavily relies on operations and primitives from the AI domain.

Taking such quality into account we choose Prolog as the implementation language. It has, by default, strong semantics for dealing with AI problems, especially those related to reasoning and recursivity for finding solutions in a determined or undetermined search space. All statements in Prolog are, on the one hand, logical terms that define a truth, or, on the other hand, predicates that define the rules for determining the required truth for a goal to be accomplished. The Prolog runtime finds the terms (atoms) and predicates that are required to accomplish the *global goal* by reasoning on all available knowledge. The runtime implements recursion natively so it, therefore, is able to find complex solutions and is accompanied with a set of primitive and standard operations, which facilitate the coding of our algorithm.

We now proceed to implement the PoC by writing the required predicates and atoms. Below we show the most important predicates, viz. the predicate that defines the heuristic function, the predicate that defines the needed goals to build a new valid configuration, and the top-most predicate that is used to obtain the target solution.

Figure 3 shows the code for implementing the heuristic function defined in Equation 6. As seen, Prolog expressiveness allows the developer to keep the code very simple. The code shows three predicates: h, $hiter$, and $hinc$. The last predicate defines how to obtain the heuristic value for a configuration C_j after adding a new association of $F_j - S_j$ to C_i. Here resides the incremental semantics for obtaining the benefits of dynamic programming mentioned above.

The second predicate, $hiter$, defines how to obtain the heuristic value for a configuration, C, without relying on the heuristic of a previous configuration. Both $hiter$ and $hinc$ are used by the first predicate, h, to define how to globally obtain the heuristic value for any configuration. It first checks if the heuristic value has been previously recorded using hdb and, if so, it binds the value from the database to the result in H_j. If not, it gets the heuristic value for the *previous* configuration

```
h([],0).
h(Cj,Hj) :-
    hdb(Cj,Hk) -> Hj is Hk;
    (
        append(Ci,[Fj-Sj],Cj),
        (
            hdb(Ci,Hl) -> Hi is Hl;
            (
                hiter(Ci,Hi),
                rkey(Ci,Ki),
                recorda(Ki,Hi)
            )
        ),
        hinc(Ci,Fj-Sj,Hi,Hj)
    ).

hiter([],0).
hiter(C,H) :-
    csubstrate(C,S),
    sum(Hi,(
        member(Si,S),
        gpathlen(Si,Li),
        count(member(_-Si,C),Wi),
        Hi is Li + Z ** Wi
    ),H).

hinc(Ci,Fj-Sj,Hi,Hj) :-
    gpathlen(Sj,Lj),
    count(member(_-Sj,Ci),Wi),
    Hj is Hi - Z ** Wi +
        Lj + Z ** (Wi + 1),
    rkey([Fj-Sj|Ci],Kj),
    recorda(Kj,Hj).
```

Figure 3. Declarative statements for the heuristic function.

by removing the latest association $F_j - S_j$, and checking C_i in the database.

If the heuristic value for C_i is not found in the database, the code uses $hiter$ to obtain it by iterating through the elements of C_i. Finally, either being obtained from the database or from $hiter$, the heuristic value of C_i is used by $hinc$ to obtain the heuristic value of C_j and here finishes this definition.

Following the specification from the associated equation, the inner of this code indicates that the heuristic value is obtained by summing the lengths of all paths from the gateway to *Si*, which is every NSN included in the provided configuration, *C*. This is added to the sum of all *Z**x*, which is the coding of Z^x, where *x* is the count of NSNs included in the configuration *C*. If a NSN appears multiple times, the count increases per each time, otherwise it is kept to *1*.

Figure 4 shows how to build a new configuration using the heuristic function defined above. It is the translation of Algorithm 1 to a declarative form. In general, this code states that a new configuration is obtained by adding a new association $F_i - S_i$ to a previous configuration C_0, which can initially be empty.

The key quality of this procedure is that the association chosen for each F_i is the association that has a resulting configuration with minimum heuristic value, by using the

```
makeConf([],_,Ci,_,Cf) :-
    sort(Ci,Cf).

makeConf([Fi|FT],S,Ci,Hi,Cf) :-
    combine(5,S,SPU),
    sort(SPU,SP),
    makeConfLoop(Fi,SP,Ci,Hi,C,_),
    selectFirst(h,C,NC), !,
    member(Cj,NC),
    h(Cj,Hj),
    makeConf(FT,S,Cj,Hj,Cf).

makeConfLoop(_,[],_,_,[],[]).
makeConfLoop(Fi,[Si|ST],C0,H0,
             [Ci|CT],[Hi|HT]) :-
    append(C0,[Fi-Si],CiU),
    sort(CiU,Ci),
    hinc(C0,Fi-Si,H0,Hi),
    makeConfLoop(Fi,ST,C0,H0,CT,HT).
```

Figure 4. Declarative statements that implement the procedure of building a new configuration guided by the heuristic function.

```
getSolution(C) :-
    makeConf(C0),
    getSolutionLoop([C0],G1a),
    append([C0],G1a,G1b),
    sortBy(h,G1b,[C|_]).

getSolutionLoop(CG,NG) :-
    getSolutionLoop(CG,CG,10,NG).
getSolutionLoop(_,_,0,[]).
getSolutionLoop(CG,PG,N,NG) :-
    N > 0,
    generate(CG,NGi),
    subtract(NGi,PG,NGj),
    sortBy(h,NGj,NGk),
    pickH(NGk,25,NGl),
    append([PG,NGl],NPG),
    N1 is N-1,
    getSolutionLoop(NGl,NPG,N1,NGr),
    append([NGl,NGr],NG), !.

generate(CG,NG) :-
    mutateCollection(CG,MG),
    breedCollection(CG,BG),
    createCollection(CG,RG),
    append([MG,BG,RG],NG0),
    sort(NG0,NG).
```

Figure 5. Code for obtaining the best configuration from several generations of configurations, which are obtained by being guided by the heuristic function.

selectFirst predicate, which chooses the first elements of equal value of h. Then, an arbitrary C_j is chosen from those elements. This is done on each step until all F_i are already associated to the necessary S_i. When needed, the Prolog runtime will automatically loop through all possible C_j from those first elements in order to satisfy the required goal.

As global objective for the code we define the predicates shown in Figure 5. It states that to obtain such configuration, the engine must get a new generation of configurations,

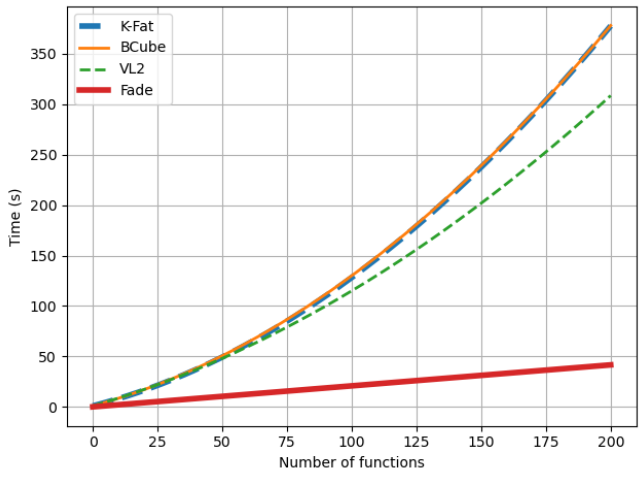

Figure 6. Evolution of the time required by each solution evaluated to obtain a valid configuration for VNs with different number of functions to be embedded onto the SN of the same size.

NG, derived from the current generation, CG, by using the $generate$ predicate. Then, those configurations are sorted by their heuristic value and 25 are homogeneously picked by *pickH*. These configurations are provided to a *loop* that finishes when all generations are done, which defaults to 10 generations. The best configuration is then returned.

The $generate$ predicate receives a generation of configurations and obtains a new generation by three different methods: mutation, breeding, and creation. The mutation consists of getting new configurations by changing the assignation of some arbitrary F_i to a different S_j. The breeding obtains new configurations by choosing two arbitrary configurations from the provided generation and choosing an assignation from each one for each F_j, so each *child* is a mixture from the *parents*. The creation method gets new assignations of F_i to arbitrary S_i, so that the generated configurations are random. All those new configurations are put together, removed duplicates, and returned as the new generation of configurations.

Using these operations we build the PoC solution that resolves the problem and obtains a valid and optimal configuration for embedding the provided VN to the provided SN. We now proceed to evaluate it to demonstrate its qualities.

B. Results

We execute the PoC solution described above for different number of functions of the VN to be embedded in a SN of the same size. For each execution, we measure the time required by our solution to find a configuration for the VN, viz. a set of associations between VNF instances and NSNs. We compare such time with the results obtained by the solutions proposed by related work, which are all based on GA, as presented in [9], viz. K-Fat, BCube, and VL2, which are named after the topologies to which they are best suited, as described in Section II.

In Figure 6 we show, for each solution evaluated in this paper, viz. K-Fat, BCube, VL2, and Spade, the plot that illustrates the relation between the number of functions of the virtual network and the time required to obtain a valid embedding configuration. We can see that they have very different rates of growth. While the algorithms presented in the related work have a quadratic growth, Spade grows linearly. This is a great improvement for allowing the virtual network to scale up without requiring a different management solution.

Analyzing these results we get, on the one hand, that our measurements are fit to the polynomial described in the following equation:

$$y = 208x + 30 \tag{8}$$

On the other hand, the results obtained with the solution found in related work for K-Fat, BCube, and VL2 are respectively fit to the polynomials described in the following equations:

$$y = 6.107x^2 + 662.5x + 948.8 \tag{9}$$

$$y = 5.948x^2 + 695.8x + 992.6 \tag{10}$$

$$y = 3.972x^2 + 744.0x + 925.2 \tag{11}$$

Comparing Equation 8 with Equations 9, 10, and 11, we can see that our solution has improved previous work by a polynomial degree, moving from a complexity of $O(n^2)$ of the related work to a complexity of $O(n)$ of our solution, for the time required to provide an embedding solution for a VN to be deployed in the SN in relation to the number of instances of the VN. We can achieve such reduction by using the dynamic programming scheme described above. This way, each iteration reuses as much operations as possible from previous iterations. In effect, a new configuration in our solution mostly relies on existing configurations, so minimizing the changes required to embed a new configuration and allowing the reduction of such degree of complexity. Putting this result in terms of typical sizes of VNs, such as around 200 nodes, we find that our solution requires less than the sixth part of the time, so it is six times faster. Moreover, as the complexity of our solution is linear we can determine that this is able to manage networks twice as large by requiring just twice the time, while the related work is not able to manage such network sizes.

Once we have shown the performance of the solution in relation to the amount of VNF instances, we proceed to evaluate the configurations that are actually enforced in the network. For this we explore the distribution of the VNF instances among all available NSNs when using different values for the parameter Z, which determines the behavior of the solution in such terms, using a small VN (10 VNF instances) in a small SN (20 nodes). As shown in Figure 7, from $Z = 0$ to $Z = 1$, the number of NSNs used remains 1, so all VNF instances are deployed in the same node.

As Z gets a value slightly greater than 1, the number of NSNs increases quite suddenly. Then, when all VNF instances

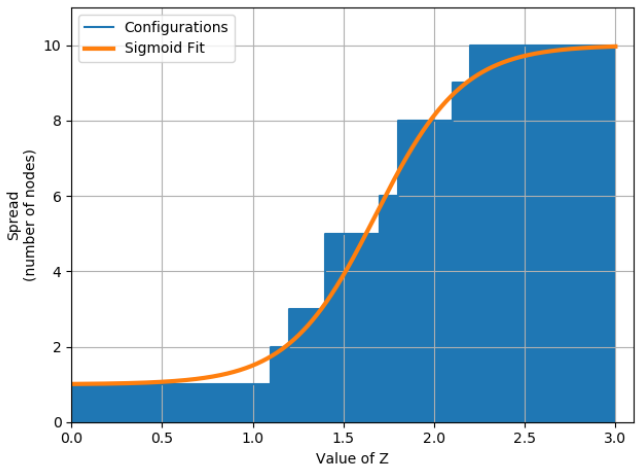

Figure 7. Relation of the value of Z and the number of NSNs used when deploying the resulting configuration.

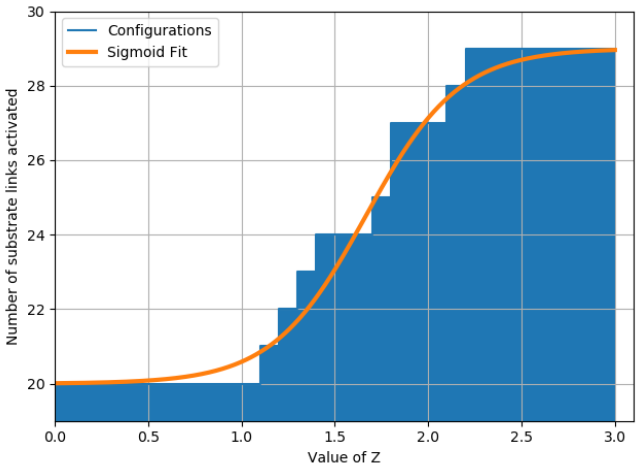

Figure 8. Relation of the value of Z and the number of links of the SN that are activated for the resulting configuration.

are assigned, the growth of the spread stops. This occurs when Z gets a value close to $\frac{5}{2}$. After calculating and representing the best fit to the obtained values for the spread, we can easily see that the evolution of this behavior has a sigmoid shape, which in this case is $\frac{l}{1+e^{-k(x-x_0)}} + b$, with $l = 9$, $x_0 = 1.68$, $k = 4.17$, and $b = 1$. It means that there is a tight the response of the spread to Z, which is important for the administrators of the network to determine the overall behavior of the embedding operation.

In addition to the number of NSNs used, to determine the effect on the underlying network of the configurations provided by our solution, we now measure the number of links of the SN that are directly or indirectly activated for a particular configuration, which also changes depending on Z. This is important because deploying a VNF instance on a node that is far from the gateway will activate more links that when deploying it in a node closer to the gateway. Therefore, the performance of the whole network will be affected by this metric. As shown in Figure 8, the VN begins with 20 links activated, as it represents the mimum number of links needed for all the VNFs and services deployed in the experimental network. Then, being parallel to the evolution of the number of NSNs used, the number of links also increases quickly between $Z = 1$ and $Z = \frac{5}{2}$.

As a key point for this result we can confirm that the number of links does not increase faster or to a greater extent than the number of NSNs, being 29 the maximum number of links used, which correspond to the additional 9 NSNs involved in the configuration after spreading the VNF instances. This remarks the scalability properties of our solution. Moreover, after obtaining the fit for this measurements we see that the shape of its evolution also follows a sigmoid function $\frac{l}{1+e^{-k(x-x_0)}} + b$, with $l = 9$, $x_0 = 1.67$, $k = 4$, and $b = 20$. This shape has a close relation with the shape of Figure 7. It remarks the importance of Z, which will definitely be a determinant factor (and parameter) for network administrators to define the behavior of the network.

The results presented here demonstrate our claims, so our solution provides a clear improvement to the SotA while also reinforcing the demonstration that advanced AI methods, such as reasoning and GP, must be adopted by NM to break the limits and drawbacks found in ML approaches. Furthermore, we demonstrate that the configurations produced by our solution are scalable and predictable, allowing the administrators of the network to instruct the behavior of our solution by using different parameters.

VI. Conclusions and Future Work

Increasing the complexity of VNs or the SNs that support them is complicating the automation of NM tasks, not to say if they are performed by humans manually. We therefore proposed to advance the SotA with a new solution that is between 3 to 6 times faster than related work when finding the configuration for embedding a VN on a SN. The key of our proposal is the application of GP together with a powerful heuristic that allows finding highly close-to-optimum solutions.

Our next work will settle the dynamic bounding of the solution space to further improve the efficiency of our algorithm. Moreover, we will explore new methods to make new configuration generations that maximize the avoidance of local optimal solutions. Finally, we will carry our PoC implementation to a production-oriented test-bed to demonstrate its benefits in real-world environments. This will show how a whole solution works with the algorithm proposed here.

REFERENCES

[1] A. Galis, S. Clayman, L. Mamatas, J. R. Loyola, A. Manzalini, S. Kuklinski, J. Serrat, and T. Zahariadis, "Softwarization of future networks and services-programmable enabled networks as next generation software defined networks," in *Proceedings of the IEEE SDN for Future Networks and Services (SDN4FNS)*. Washington, DC, USA: IEEE, 2013, pp. 1–7.

[2] P. Martinez-Julia, A. F. Skarmeta, and A. Galis, "Towards a secure network virtualization architecture for the future internet," in *The Future Internet*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 7858, pp. 141–152.

[3] D. R. Lopez, "Network functions virtualization: Beyond carrier-grade clouds," in *Proceedings of the 2014 Optical Fiber Communications Conference and Exhibition (OFC)*. Washington, DC, USA: IEEE Computer Society, 2014, pp. 1–18.

[4] M. Rost and S. Schmid, "On the hardness and inapproximability of virtual network embeddings," *IEEE/ACM Transactions on Networking*, vol. 28, no. 2, pp. 791–803, 2020.

[5] K. Antonakoglou, X. Xu, E. Steinbach, T. Mahmoodi, and M. Dohler, "Toward haptic communications over the 5g tactile internet," *IEEE Communications Surveys*, vol. 20, no. 4, pp. 3034–3059, 2018.

[6] P. Martinez-Julia, V. P. Kafle, and H. Harai, "Exploiting external events for resource adaptation in virtual computer and network systems," *IEEE Transactions on Network and Service Management*, vol. 15, no. 2, pp. 555–566, 2018.

[7] P. Martinez-Julia, V. P. Kafle, and H. Asaeda, "Explained intelligent management decisions in virtual networks and network slices," in *Proceedings of the 23th ICIN Conference (Innovations in Clouds, Internet and Networks, ICIN 2020)*. Washington, DC, USA: IEEE, 2020, pp. 1–7.

[8] H. Saito, H. Honda, and R. Kawahara, "Disaster avoidance control against heavy rainfall," in *Proceedings of the IEEE INFOCOM 2017*. Washington, DC, USA: IEEE, 2017, pp. 1–8.

[9] W. Rankothge, F. Le, A. Russo, and J. Lobo, "Optimizing resource allocation for virtualized network functions in a cloud center using genetic algorithms," *IEEE Transactions on Network and Service Management*, vol. 14, no. 2, pp. 343–356, 2017.

[10] Y. Li, L. T. Xuan Phan, and B. T. Loo, "Network functions virtualization with soft real-time guarantees," in *Proceedings of the 35th IEEE International Conference on Computer Communications (IEEE INFOCOM 2016)*. Washington, DC, USA: IEEE, 2016, pp. 1–9.

[11] M. Ghaznavi, A. Khan, N. Shahriar, K. Alsubhi, R. Ahmed, and R. Boutaba, "Elastic virtual network function placement," in *Proceedings of the 4th IEEE International Conference on Cloud Networking (IEEE CloudNet 2015)*. Washington, DC, USA: IEEE, 2015, pp. 255–260.

[12] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[13] A. Galis *et al.*, "Position Paper on Management and Service-aware Networking Architectures (MANA) for Future Internet," http://www.future-internet.eu/fileadmin/documents/prague_documents/MANA_PositionPaper-Final.pdf, 2010.

[14] ETSI AFI GS, "Autonomic network engineering for the self-managing Future Internet (AFI); Generic Autonomic Network Architecture (An Architectural Reference Model for Autonomic Networking, Cognitive Networking and Self-Management)," 2013, ETSI GS AFI 002.

[15] U. Thissen, R. van Brakel, A. P. de Weijer, W. J. Melssen, and L. M. C. Buydens, "Using support vector machines for time series prediction," *Chemometrics and Intelligent Laboratory Systems*, vol. 69, no. 1–2, pp. 35–49, 2003.

[16] T. Lorido-Boltran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014.