

# Final Report

## Minimising Latency in Distributed Software Defined Networks utilising Network Slicing

James Borgars

Submitted in accordance with the requirements for the degree of  
MEng, BSc Computer Science

2023/24


COMP3931 Individual Project

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Final Report	PDF file	Uploaded to Minerva (01/05/24)
Link to online code repository	URL	Sent to supervisor and assessor (30/04/24)

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) 

## Summary

This report provides a solution for the placement of virtualised network components to form function chains, within a network utilising a weighted-sum heuristic-based algorithm.

The problem space is protocol-agnostic but could be applied to mobile telecommunications networks such as those utilising the 5G standard. The solution aims to reduce latency by using context it knows about the network, to develop heuristics used for network traversal, and a weighted-sum objective function to estimate the proposed solution in regards to the latency KPI.

The solution is focused on being latency-aware, and the use of the weighted-sum approach allows for specialised deployments that can be tailored to the wants and limitations of a specific network.

This report details a formulation of the problem and the proposed solution, whilst also implementing these, as well as other solutions as benchmarks, achieving a simulation environment.

Three benchmarks have been implemented: a randomised generation algorithm, a more generic latency-aware solution, and a cost-aware heuristic algorithm found in literature. The implementation of the proposed solution consistently outperforms all benchmarks, by differing amounts.

Testing has been developed showing the effect of changing variables in the problem space, such as variance in number of nodes, and presence of pre-existing VNFs in the network prior to executing the algorithms.

The test results are then evaluated, drawing conclusions about the strengths and weaknesses of the proposed solution, as well as the benchmarks that have also been implemented.

It is concluded that the proposed algorithm is a viable solution, with also scope for several different directions for further development such as Deep Learning algorithms and the idea of hybrid solutions. This is detailed further in Chapter 4.

The appendices contains a detailed self-evaluation reflecting on the dissertation process and reflecting on what went well and areas for improvement, as well as figures, tables, and code listings.

### **Acknowledgements**

Many thanks to my supervisor Arash Bozorgchenani, for his consistent advice and support throughout the entire dissertation process. Thanks also to my assessor Tom Ranner, who gave useful insights in the assessor meeting.

# Contents

<b>1</b>	<b>Introduction and Research</b>	<b>1</b>
1.1	Introduction . . . . .	1
1.1.1	Context . . . . .	1
1.1.2	Report Contributions and Objectives . . . . .	2
1.1.3	Report Organisation . . . . .	2
1.2	Background Research . . . . .	3
1.2.1	Software-Defined Networking . . . . .	3
1.2.2	Network Function Virtualisation . . . . .	3
1.2.3	Network Slicing . . . . .	4
1.3	Literature Review . . . . .	4
<b>2</b>	<b>Methods</b>	<b>8</b>
2.1	Problem Formulation . . . . .	8
2.1.1	Network Modelling . . . . .	8
2.1.2	Resource Modelling . . . . .	9
2.1.3	Assumptions . . . . .	9
2.1.4	Constraints . . . . .	11
2.2	Solution Formulation . . . . .	11
2.2.1	Algorithm Initialisation . . . . .	11
2.2.2	Network Slice Generation . . . . .	12
2.2.3	Objective Function . . . . .	14
2.2.4	Key Performance Indicators . . . . .	16
2.3	Tooling . . . . .	16
2.3.1	Version Control . . . . .	16
2.3.2	Implementation Choices . . . . .	16
2.4	Project Management Methodology . . . . .	17
<b>3</b>	<b>Results</b>	<b>18</b>
3.1	Implementation . . . . .	18
3.1.1	Simulator Implementation . . . . .	18
3.1.2	Solution Implementation . . . . .	19
3.1.3	Benchmark Implementation . . . . .	21
3.1.4	Implementation Evaluation . . . . .	22
3.2	Testing . . . . .	23
<b>4</b>	<b>Discussion</b>	<b>27</b>
4.1	Results Evaluation . . . . .	27
4.2	Ideas for Future Work . . . . .	29

<b>References</b>	<b>31</b>
<b>Appendices</b>	<b>34</b>
<b>A Self-appraisal</b>	<b>34</b>
A.1 Critical Self-Evaluation . . . . .	34
A.2 Personal Reflection and Lessons Learned . . . . .	34
A.3 Legal, Social, Ethical and Professional Issues . . . . .	35
A.3.1 Legal Issues . . . . .	35
A.3.2 Social Issues . . . . .	35
A.3.3 Ethical Issues . . . . .	36
A.3.4 Professional Issues . . . . .	36
<b>B External Material</b>	<b>37</b>
<b>C Figures, Tables and Listings</b>	<b>38</b>
C.1 Figures . . . . .	38
C.2 Tables . . . . .	39
C.3 Listings . . . . .	40
C.3.1 Code Snippets . . . . .	40
C.3.2 Network Definitions . . . . .	45
C.3.3 Solution Implementation Definitions . . . . .	50

# Chapter 1

## Introduction and Research

### 1.1 Introduction

#### 1.1.1 Context

As the capabilities of portable technology increases, the variety of services that need to be facilitated by wireless networks has become apparent, such as *Internet of Things* (IoT) based services (Corno et al., 2018), and autonomous driving (Golkarifard et al., 2021). These different services can benefit from service-specific traffic processing in order to maximise efficacy.

However, historically, service deployment within telecommunications networks has led to issues regarding a lack of dynamism and flexibility due to the usage of specialised, function-specific hardware (Li et al., 2017). This would render the idea of service-specific processing of traffic difficult to realise, as it would be necessary for traffic of different services to be routed through different hardware.

To overcome this inflexibility, *Software Defined Networking* (SDN) is an architectural approach that allows for software-based communication between network infrastructure, utilising methods such as *Application Programming Interfaces* (APIs).

Built on top of SDN, *Network Function Virtualisation* (NFV) is an architectural approach in which what would be traditionally function-specific hardware is replaced with software instances that are functionally equivalent. This allows for general-purpose hardware to replace commercial, specialist hardware within a network, resulting in lower capital expenditure and easier equipment acquisition. Instances of these software-based functions are called *Virtual Network Functions* (VNFs).

*Service Function Chaining* (SFC) is the process of combining VNFs into a logical path in which network traffic can be routed through. As these chains are composed of logical connections, it is possible to create, modify, and deconstruct chains programmatically. These capabilities of NFV and SFC promote dynamism and flexibility within a network, rendering service-specific traffic handling and network topology modification possible, based on demand.

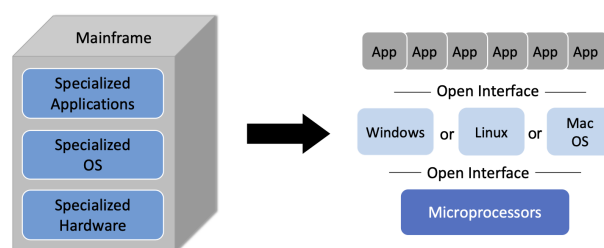


Figure 1.1: Comparison of Mainframe topology to SDN (Systems Approach LLC, 2022)

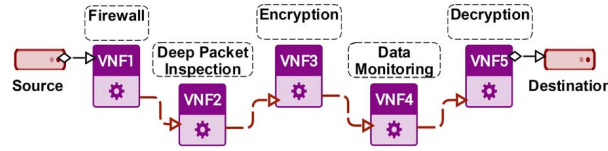


Figure 1.2: Example of a Service Function Chain (Gil Herrera and Botero, 2016)

There are setbacks as to using these technologies, however. NFV makes predicting performance prior to service embedding difficult due to factors such as multiple functions usually residing on the same node (Savi et al., 2021). The usage of NFV and SFC also gives rise to the question of node placement (i.e. the location of physical machines within a network) and virtual machine (VM) placement (i.e. on which node should a given virtual machine be present).

The placement of VNFs within a network, and connecting them, known as the *VNF Forwarding Graph Embedding* (VNF-FGE) problem, is, according to Gil Herrera and Botero, a generalisation of the Virtual Network Embedding problem, a problem which is  $\mathcal{NP}$ -hard, therefore it can also be deduced that the VNF-FGE problem is also  $\mathcal{NP}$ -hard (Gil Herrera and Botero, 2016). This report aims to propose a solution to this problem, focusing on minimising latency, but also factoring in other variables present within a network, such as bandwidth.

### 1.1.2 Report Contributions and Objectives

This report provides a weighted-sum heuristic-based solution that assumes Service Function Chain/Network Slice placement should optimally start at the network edge (i.e. where radio equipment is likely to be present), and suggests replacement chains with respect to existing chains based on a traffic sample, in order to minimise latency.

To assess the outcomes of the report, Report Objectives have been set to clearly evaluate the results of the work produced. The objectives and target outcomes are as follows:

- **RO1:** Propose, and implement in simulation, a viable network graph traversal heuristic that attempts to find a viable node concerning latency, available resources, and bandwidth.
- **RO2:** Propose, and implement in simulation, a heuristic solution to the VNF-FGE problem, which aims to propose a Service Function Chain with (relative) low latency, whilst also avoiding bad network planning decisions.
- **RO3:** Implement benchmarks within simulation to allow for testing of the algorithm to measure its performance and provide sufficient results for analysis.

### 1.1.3 Report Organisation

The remainder of this chapter is used to demonstrate background research and provide a literary review of related work, explaining their relevance and utility to this report, but also why they are not entirely suitable to solve the issues raised in this report.

Chapter 2 describes the formulation of the problem and its solution, with justification of modelling decisions provided throughout.



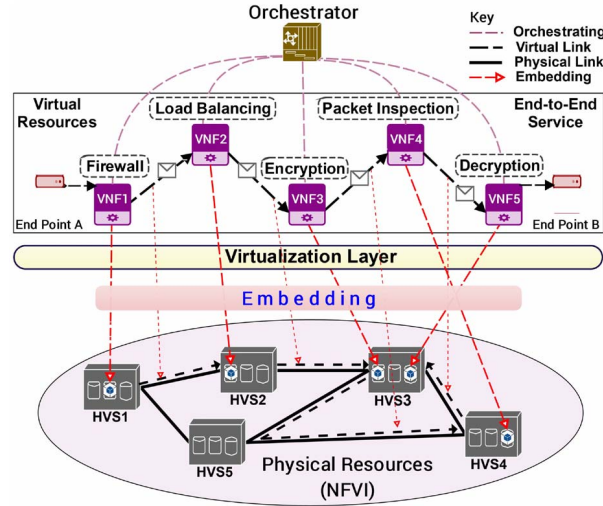


Figure 1.3: VNF embedding onto a network (Gil Herrera and Botero, 2016)

Chapter 3 explains the implementation of the formulated solution into a simulation environment, as well as the implementation of other solutions found in literature to act as benchmarks in our results analysis which will also be present in this chapter.

Chapter 4 describes the conclusions that can be drawn from our results and possible explanations for results, as well as how research could be furthered into this area in future.

The appendices contain a self-evaluation and reflection, as well as a record of external materials and pertinent figures, tables, and listings.

## 1.2 Background Research

### 1.2.1 Software-Defined Networking

SDN, as mentioned in the introductory context, is an architectural approach which differs from previous approaches, by taking advantage of the portability of software and general purpose hardware, as opposed to solely specialised hardware.

This approach allows for increased control, customisable infrastructure, and improved security, also allowing for better flexibility, even in ways such as locating equipment off-premises (VMware, no year).

### 1.2.2 Network Function Virtualisation

As the majority of major applications are delivered by *Cloud Service Providers* (CSPs), whilst using the infrastructure of ISPs, this, according to Hmaity et al., has led to negative impact on the revenues of ISPs as they have to handle the increase in traffic as data is now stored off-premises for businesses and other use cases (Hmaity et al., 2020).

As mentioned in the Literature Review, the use of NFV and VNFs can be used to handle different services across a network, but can also be used for load balancing by replicating services and forming multiple chains (Carpio et al., 2017).

As mentioned in Schardong et al. (2021), to implement NFV fully, three problems must be handled: the *VNF Chain Composition* (VNF-CC) problem which is concerning the idea of chaining VNFs together to form a chain that can process traffic effectively; then there is the aforementioned VNF-FGE problem, which is the combination of the placement of VNFs onto the network graph (the *placement* problem), and connecting those VNFs together using the physical links present within the network (the *chaining* problem); the final problem is the *VNF Scheduling* (VNF-SCH) problem, which focuses on aiming to re-use pre-existing VNFs in new chains if appropriate, aiming to take advantage of VNF idle time in order to reduce hardware resource use and avoidance of spin-up penalties.

This report focuses on solving the VNF-FGE problem with a view to reduce latency.

### 1.2.3 Network Slicing

The solution proposed in this report is going to be through the lens of a Network Slicing topology. Network Slicing places VNFs within 'slices' that are dedicated to a specific service, creating isolation between services, therefore reducing the chance of a VNF being inundated traffic due to multiple services utilising the same VNF.

Zhang (2019) describes and explains the process of implementing the Network Slicing architecture for 5G traffic. It is broken down into four phases: Design, Orchestration & Activation, Run-time Operation, and Decommissioning. This report focuses on Design (creation of a Network Slice based on Network Function Components, with placement being decided based upon maximisation of KPIs) and Orchestration & Activation (Instantiating the Network Slice and steering traffic to the newly created Slice)

Zhang shows that in the implementation of 5G into a Network Slicing architecture, traffic from *user equipment* (UE) must, after registering with the mobile network, provide *Network Slice Selection Assistance Information* (NSSAI) to the base station. The NSSAI contains information which the network uses to identify which *Access and Mobility Management Function* (AMF) the UE should use if it is not a new device to the network. Otherwise, the AMF used by the UE is determined by the *Network Slice Selection Function* (NSSF) which determines the NSSAI/which AMF the UE should use.

Due to the complications involved by having a *Common Core Network Function* which would need to be used by all traffic in order to direct it to a slice, this is not included in the modelling done in this report, it is assumed all traffic is of a known service and has already had a slice selected for it to be routed through. The report aims to solve the issue of finding if the slice can be restructured to better handle the packets flowing through the pre-existing slice with respect to minimising latency.

## 1.3 Literature Review

### Hmaity et al. (2020)

Hmaity et al. (2020) demonstrates VNF placement within a Metro network, and applies NFV into the area of *Fixed and Mobile Convergence* (FMC). FMC is when fixed traffic (e.g. home

broadband), and mobile traffic (e.g. 5G data) are transported across the same network. Hmaity et al. believe that VNF placement within the Core Network or Data Centres could encounter issues regarding latency due to the physical distance between nodes (i.e. servers) in these networks, even though it is thought to be more cost-effective.

Hmaity et al. (2020) presents a new heuristic algorithm, inspired from Savi et al. (2021), taking latency and capacity into account when attempting to place VNFs. The algorithm in this Individual Project report simplifies the Access-Metro-Core model demonstrated in ISP networks due to the increase in complexity this would cause in generating a suitable problem-space. However, the solution does make the assumption that, in order to minimise latency, Service Function Chains should start at an Edge node, in order to reduce distance between User Equipment and the Service Function Chain entry-point.

### **Savi et al. (2021)**

Savi et al. (2021) looks into the idea of VNF consolidation. It is evident that financial cost of deploying a networking utilising NFV can be minimised if multiple VNFs can be placed in one node, however Savi et al. explain how there are penalties in doing this, primarily in CPU context switching, and parallelisation costs when a VNF is ran across multiple cores.

Savi et al. present a heuristic cost-aware algorithm (HCA) which assumes that each physical link within the network will always have sufficient bandwidth, due to the idea that there has been over-provisioning in the design of the Core network, where this algorithm would be implemented. The solution presented in this report *is* aware of bandwidth remaining in physical links, and will not oversubscribe a physical link, and rewards proposed solutions which take links that utilise links with vast amounts of spare bandwidth.

The algorithm implements a greedy solution, attempting to re-use VNFs and hardware nodes wherever possible, and if the latency SLA is not met, a second phase is utilised then places the chain at the shortest path concerning latency. This paper proposes similar network modelling to this report, but provides a solution focusing on a different objective (cost-aware). For this reason, it is used as a benchmark algorithm when testing the efficacy of the proposed solution from this report.

### **Li et al. (2018)**

Li et al. (2018) address issues present in the *Network Function Virtualisation Resource Allocation* (NFV-RA) problem, namely problems occurring from the *Virtual Network Function Chain Composition* problem (i.e. deciding the order of VNFs in a Service Function Chain), and the VNF-FGE problem.

Li et al. present a heuristic solution to solve the VNF-CC and VNF-FGE problems, with the aim of minimising the cost of implementing the services and to increase the number of services feasible on a network through lowering these costs.

**Zhang et al. (2019)**

In Zhang et al. (2019), the paradigm of Network Slice is proposed to allow for service-specific traffic routing over 5G networks. As described in Zhang (2019) and mentioned in Background Research, each network slice has its own set of VNFs designed handle traffic of a specific service.

Zhang et al. propose a methodology on how a slice should be designed, with VNFs being managed by an *Adaptive Interference-Aware* (AIA) heuristic algorithm, which has the aim of maximising network throughput utilising an 'Edge Cloud' and a 'Core Cloud' topology.

Zhang et al. mention the idea that *VNF consolidation*, the idea of placing multiple VNFs degrades network throughput, which can be an issue in latency-sensitive and throughput-sensitive scenarios.

Overall, Zhang et al. (2019) is focused on the specifics of 5G NFV, considering the necessities of 5G services, such as VNFs in both the Core and the Edge, the algorithm proposed in this report aims to be more service-agnostic and technology-agnostic, whilst still being latency-aware.

**Chetty et al. (2021)**

Chetty et al. (2021) proposes the use of *Deep Reinforcement Learning* (DRL) techniques to solve the VNF-FGE problem, as opposed to the majority of solutions found in literature that are heuristic or meta-heuristic.

Chetty et al. provide a solution using *Deep Q-Learning*, a form of DRL, where the reward is received when a successful chain is deployed and the latency cap of 30ms is not broken. Using DRL techniques is a very intriguing approach to this problem space, however this report is going to draw from the vast majority of literature, which focuses on heuristic-based solutions.

**Golkarifard et al. (2021)**

In Golkarifard et al. (2021), there is, as claimed by the authors, the first dynamic VNF placement solution. Golkarifard et al. discuss the aforementioned dynamic placement, resource allocation, and traffic routing in respect to 5G services. They propose a solution which bases its decisions on knowledge garnered from requests over a given period, this report uses packet samples as a similar method to gain 'context' about the network.

The network definition is similar to what is defined in this report, but the solution by Golkarifard et al., an algorithm called *MaxSR*, aims to 'maximise revenue whilst minimising cost', as opposed to minimising latency.

**Thiruvassagam et al. (2021)**

Thiruvassagam et al. (2021) presents polynomial-time heuristic algorithms for VNF placement as part of a Service Function Chain, and also for *Virtual Monitoring Functions*, which, according to Thiruvassagam et al., identify and mitigate both service degradation and security issues.

Thiruvassagam et al. focus on a latency-aware but also resilient solution in case of failures of VNFs within the network. The modelling of the network is by having a Physical Network as an

undirected graph, the Virtual Network as sub-graph of the Physical Network, with Service Function Chains being directed sub-graphs of the Virtual Network. This is similar to the problem definitions of other relevant literature found in this review, and the problem formulation in this report is based off of the definitions provided by Thiruvassagam et al..

#### **Khebbache et al. (2017)**

Khebbache et al. (2017) focus on the *VNF Chain Placement Problem* (VNF-CPP), the paper provides a service-agnostic optimisation algorithm, and solely focuses on optimising VNF placement based on available CPUs and link speeds utilising a matrix-based solution.

Khebbache et al. mention that the exact solution does not scale and provide an exact algorithm based on Service Function Chains consisting on 3 VNFs. This report proposes a more broad and scalable solution at the cost of optimality.

#### **Chiaraviglio et al. (2019)**

Chiaraviglio et al. (2019) discuss algorithms for routing 5G traffic based on entities that are similar to VNFs called *Reusable Function Blocks* (RFBs), which are more specialised to handling 5G traffic as opposed to the generalised VNF.

Two heuristics algorithms are proposed and focus on the reduction of installation costs and maximising the capable number of users respectively.

The work in Chiaraviglio et al. (2019) is highly-tailored to 5G functionality, and does not focus on the same KPI as in this report, which also aims to provide a more generalised solution, not focusing on service-specific variables.

#### **Mahboob et al. (2020)**

Mahboob et al. (2020) address dynamic VNF placement to address the issues that occur when users are changing physical location, a common occurrence of users of services such as 5G, by relocating VNFs when it is viable based on available resources within the network. This has relevance as minimising inefficiencies is an important aspect into reducing latency.

However, Mahboob et al. model a *Software Defined Wireless Network* (SDWN), thereby abstracting away the modelling of physical links, a constraint that this report takes into account,

Mahboob et al. conclude that their propositions reduce blocking rates within the Service Function Chain, and leads to an increase in network throughput.

#### **Chai et al. (2023)**

Chai et al. (2023) target the issues arising in the VNF-FGE problem and its scaling, the paper considers the dynamism of traffic, resource capacity, as well as the different costs such as VNF maintenance, and turn-on costs for VNFs.

Similarly to the solution proposed in this report, it focuses on the Network Slicing topology. However, the solution focuses on being capacity-aware and scaling.

# Chapter 2

## Methods

### 2.1 Problem Formulation

To define the problem, it is necessary to form mathematical expressions of the physical network, the virtual network, and the network slices which are formed on top of the virtual network.

The problem space needs to be formulated such that it is possible to extract required information to solve the problem, and that it is feasible to apply our solution to the model. The modelling of the network is based on Thiruvassagam et al. (2021).

#### 2.1.1 Network Modelling

Let  $G_P = (N, L)$  be an undirected graph representing the physical network infrastructure, where  $N$  is the set of physical nodes (i.e. servers) and let  $L$  be the set of physical links between nodes in  $N$ , therefore:

$$\forall l \in L : l = (a, b) \mid \exists a, b \in N \wedge a \neq b \quad (2.1)$$

By taking advantage of routing, it is possible to design a virtual network on top of the physical network, allowing for an abstraction of traffic to move between two VNFs whose host nodes are not physically connected.

Let  $G_V = (V, E)$  be an undirected graph representing the virtual network, where  $V$  is the set of VNF instances in the network, and  $E$  is the set of virtual edges between connected VNFs.

It will be necessary for our solution to be able to deconstruct edges into its physical links. Let  $\mathcal{P}(e)$  be a function that returns the set of physical links that make up an edge  $e$ .

The problem space will be modelled as a network using the network slicing topology, which creates chains of VNFs (SFCs) that are specific to a service over the 5G network.

This allows for service isolation and compartmentalisation, at the cost of power and financial efficiency. However, since this report is looking from the perspective of a core network, these disadvantages are not of maximum importance, as core networks already use expensive, specialised hardware which suffers from the same issues whilst also not being easily flexible to changes in traffic demand.

Let  $S_x^\alpha = (V_x, E_x)$  be a directed graph representing a network slice  $x$ , for a given service  $\alpha$ , where  $V_x \subseteq V$  is the set of VNFs present within the slice, and  $E_x$  is the set of virtual links between the VNFs in  $V_x$ .

### 2.1.2 Resource Modelling

In order to be aware of the necessity to find a solution, as well finding a solution in of itself, it is imperative to have information about the congestion of both the nodes and physical links of our network, which are the vertices and edges of our graph models.

To measure the usage and availability of nodes, a computational resource heuristic will be present within the model, the exact representation in application will not be in the scope of this report, however, every representation would likely involve RAM and processor usage, for example.

Let  $C(x) \mid x \in N \cup V$  be the function that returns the computational resources being used by a node or VNF  $x$ , depending on what it is. Let  $C_{\max}(x) \mid x \in N \cup V$  be the function that will return the maximum computational resources available by a node  $x$ , or maximum computational resources that can be used by a VNF  $x$ .

It is important to manage the amount of traffic being transferred over a physical link at a given time (bandwidth), as this will cause an increase in latency, even if all nodes involved have sufficient resources to handle the traffic.

Let  $B(l)$  be the function that returns the current amount of bandwidth used by a physical link  $l$  and let  $B_{\max}(l)$  be the maximum bandwidth that can be managed by a link  $l$ .

As minimising latency is the primary objective of the graphs generated by the solution, it is necessary to be able to measure the time taken by network slices, therefore our model requires the ability to measure the time taken:

- to traverse a physical link
- to traverse a virtual edge
- for a VNF to process a packet
- for a VNF to initialise (spin up)
- for a packet to be processed through a network slice

Assumptions are made to simplify the modelling of latency, this is explained further in the Assumptions section.

Let  $T(l) \mid l \in L$  return the time taken for a packet to traverse a physical link. Let  $T(e) \mid e \in E$  return the time take for a packet to traverse a virtual edge.

Let  $T(v) \mid v \in V$  return the average amount of time taken to process a packet by a VNF instance  $v$ . Let  $T_{\text{init}}(v)$  represents the time penalty to spin up a VNF instance of  $v$ .

$T(S_x^\alpha)$  returns the average amount of time taken for a packet to be through a network slice  $x$ , which processes traffic for a service  $\alpha$ .

### 2.1.3 Assumptions

Due to factors such as computational limitations, and time constraints in producing this report, assumptions will be made in the problem space to simplify the problem and to make it

approachable.

It will be assumed that any VNF will be running at its maximum allocated computational resource capacity at all times. This models a worst-case scenario, and allows for the solution to not consider variations in VNF resource usage, which can vary due to changes in traffic demand, and therefore render the generation of consistent solutions difficult if it was to be taken into account.

The baseline computational resource used at a node (such as Operating System resource usage) have been abstracted away from this value, and can be assumed to be a constant that has been subtracted away from the maximum computational resource value for a node.

This means, given that  $H(v) \mid v \in V$  is a function that returns some  $n \in N$  that is the node where  $v$  is hosted, that the amount of resources currently in use by a node can be simplified to:

$$C(n) = \sum_{v \in V} C_{\max}(v) \mid H(v) = n \quad (2.2)$$

Another assumption is that the time taken for a packet to traverse through a node solely for the purpose of routing is negligible such that it is ignored, this allows for the following:

$$T(e) = \sum_{l \in \mathcal{P}(e)} t(l) \quad (2.3)$$

The above equation then allows for the further construction for measuring the time taken for a packet to be processed through a slice:

$$T(S_x^\alpha) = \sum_{e \in E_x} T(e) + \sum_{v \in V_x} T(v) \quad (2.4)$$

To simplify the problem space (and therefore the solution), assumptions are made regarding latency. These assumptions are the following:

- Each piece of hardware (nodes and user equipment) has a fixed 2D co-ordinate pair representing physical location.
- Each edge node (node of degree 1) has the capability to receive radio traffic (i.e. packets from user equipment).
- The latency from a piece of user equipment to an edge node, is proportional to distance between the user equipment and the edge node.
- The processing time at a VNF is considered a fixed cost for each packet, issues such as VNF consolidation have been abstracted away from this model.
- Routing latency is assumed to be 0.



### 2.1.4 Constraints

For a service  $\alpha$ , let there be an agreed average traffic latency SLA,  $\tau_\alpha$ , this gives the following constraint:

$$\forall S : T(S_x^\alpha) < \tau_\alpha \quad (2.5)$$

For all physical links, the amount of bandwidth in use must be less than the maximum bandwidth of the link (no over-subscription):

$$\forall l \in L : B(l) < B_{\max}(l) \quad (2.6)$$

The use of secondary memory (swapping) due to too many resources being allocated to a node should not occur, meaning that:

$$\forall n \in N : C(n) < C_{\max}(n) \quad (2.7)$$

## 2.2 Solution Formulation

The algorithm proposed in this report to solve the VNF-FGE problem is a greedy weighted-sum heuristic solution, factoring computational resource availability, link capacity and latency, as well as VNF spin-up costs. The algorithm proposed in this report will also be referred to as **LAGHA: Latency Aware Greedy Heuristic Algorithm**.

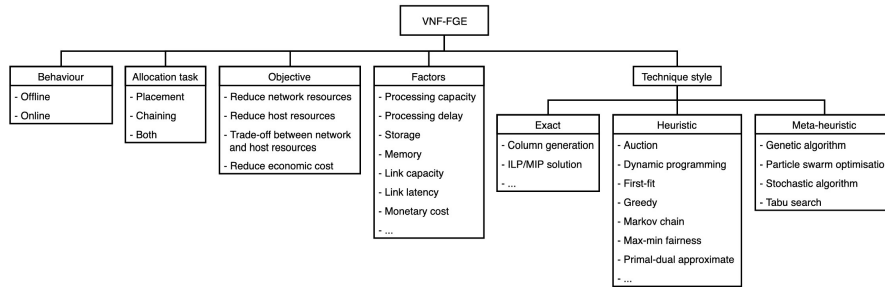


Figure 2.1: Classifications of VNF-FGE problem solutions (Schardong et al., 2021)

### 2.2.1 Algorithm Initialisation

It is necessary to consider the conditions in which an implemented solution will attempt to generate a replacement network slice. It is therefore required for a heuristic to exist that indicates and/or measures the inefficiency of a network slice.

In this case, a sample of traffic statistics is collected at time intervals to be analysed. Let  $\Omega_x$  be the set representing the sampled traffic for a network slice  $x$ , let  $S_x$  be a slice carrying traffic for a service  $\alpha$ .

To measure inefficiency, it is important to know the amount of unnecessary nodes traffic passes through on its route from the user equipment it has originated from, to the first node in the appropriate network slice for that piece of traffic. Let  $\text{avg}(\Omega_x)$  represent the average number of unnecessary hops traffic takes in the set  $\Omega_x$ .

If  $\text{avg}(\Omega_x)$  exceeds some predefined threshold  $\lambda_\alpha$ , then this would be sufficient cause to begin generation proposed network slices.

### 2.2.2 Network Slice Generation

Once conditions have been met to start generating a proposed alternative network slice, it is necessary for our solution to calculate the starting node of its proposed slice:

---

#### Algorithm 1 Candidate Starting Nodes Algorithm

---

**Require:**  $\text{avg}(\Omega_x) \geq \lambda_\alpha$  ▷ Require threshold to be hit  
 $S \leftarrow \{\}$  ▷ Let  $S$  be the set of candidate starting nodes  
 $e \leftarrow H(V_x(0))$  ▷ Let  $e$  be the node in which the first VNF is currently present  
**for**  $\omega \in \Omega_x$  **do** ▷ Iterate over every packet in the sample  
    **for**  $l = (a, b) \in \mathcal{P}((\omega_0, e))$  **do** ▷ Add every physical node involved to  $S$   
        **if**  $a \notin S$  **then**  
             $S \leftarrow S + \{a\}$   
        **end if**  
        **if**  $b \notin S$  **then**  
             $S \leftarrow S + \{b\}$   
        **end if**  
    **end for**  
**end for**  
**return**  $S$

---

The above algorithm will give us a set of candidate nodes to inspect in relation to the viability of them being the starting node in a proposed network slice.

To select a starting node from the candidates, the algorithm simulates what the average number of redirects would be if the traffic sample  $\Omega_x$  was to occur and the candidate node was the location of the first VNF of the network slice. This is represented as  $\Omega_x^s$  where  $s$  is the candidate node.

**Algorithm 2** Starting Node Selection Algorithm

---

**Require:**  $S \neq \emptyset$  ▷ Candidate nodes must be provided  
 $\Omega_x^{\min} \leftarrow \text{avg}(\Omega_x)$  ▷ Minimum simulated value of  $\text{avg}(\Omega_x)$   
 $p \leftarrow S_x^\alpha(0)$   
**for**  $s \in S$  **do**  
    **if**  $\Omega_x^s < \text{avg}(\Omega_x)$  **then** ▷  $\Omega_x^s$  represents value of  $\text{avg}(\Omega_x)$  if starting node was  $s$   
         $\Omega_x^{\min} \leftarrow \Omega_x^s$   
         $p \leftarrow s$   
    **end if**  
**end for**  
**if**  $p = s$  **then**  
    **return** NULL  
**else**  
    **return**  $(p, \Omega_x^{\min})$   
**end if**

---

As part of the iterative development and implementation process, the above starting node selection algorithms have been replaced due to its efficacy being outperformed by an algorithm calculating the Euclidean distance from the packet's origin to the closest edge node and finding the best node for the sample set as a whole.

However, the above algorithms may still be of utility if there is a large packet sample, as finding the best node through Euclidean distances has a complexity of  $\mathcal{O}(np)$  where  $n$  is the number of edge nodes and  $p$  is the number of packets, whereas the complexity of the above algorithms would be  $\mathcal{O}(p)$  due to each packet solely being inspected through how many hops it takes.

It is now necessary to define a heuristic function in order to determine which node to traverse to next, Let  $U(f, t, V^\alpha)$  be the function in which the heuristic cost to traverse from  $f$  to  $t$  is calculated.

$U_{\text{NV}}(t, V^\alpha)$  is a function that is part of the calculations made in  $U$ ,  $t$  is the node being inspected and  $V^\alpha$  is an ordered set of VNFs used in processing traffic for a service  $\alpha$ , minus the VNFs that have already been allocated.

$$U_{\text{NV}}(t, V^\alpha) = \beta \mid \beta < C_{\max}(y) \wedge \beta = \lim_{\beta \rightarrow (C_{\max}(y) - C(y))} \sum_{i=1} V^\alpha(i) \quad (2.8)$$

$U_{\text{BW}}(f, t, \Omega_x)$  calculates a heuristic cost regarding bandwidth. To summarise, it calculates how much bandwidth would be remaining in the physical link from  $f$  to  $t$ , if the link was to be used in the proposed network slice,  $B(\Omega_x)$  represents the peak amount of bandwidth used at a link in the sample  $\Omega_x$ :

$$U_{\text{BW}}(f, t, \Omega_x) = \begin{cases} 0 & B((f, t)) + B(\Omega_x) > B_{\max}((f, t)) \\ B_{\max}((f, t)) - B(\Omega_x) & \text{otherwise} \end{cases} \quad (2.9)$$

The completed heuristic algorithm,  $U$ , is as follows:

$$U(f, t, V^\alpha, \Omega_x) = (a \cdot U_{\text{NV}}(t, V^\alpha)) \cdot (b \cdot U_{\text{BW}}(f, t, \Omega_x)) \cdot (c \cdot T((f, t))) \quad (2.10)$$

$a$ ,  $b$ , and  $c$  are all arbitrary weightings that can be tweaked to suit the specific implementation of the solution.

---

**Algorithm 3** Slice Generation Algorithm
 

---

**Require:**  $p, V^\alpha$  ▷ Requires candidate node and VNF list

$V_y^\alpha \leftarrow V^\alpha$

$V_z^\alpha \leftarrow \emptyset$

$c \leftarrow p$  ▷ Start with chosen start node

**while**  $V_y^\alpha \neq \emptyset$  **do** ▷ Loop until all VNFs placed

$v \leftarrow V_y(0)$  ▷ Get first/next VNF

**if**  $C_{\max}(v) < C_{\max}(c) - C(c)$  **then** ▷ If  $c$  can facilitate  $v$

$H(v) \leftarrow c$  ▷ Place  $v$  on  $c$

$C(c) \leftarrow C_{\max}(c) + C_{\max}(v)$  ▷ Update resources used at  $c$

$V_z \leftarrow V_z + \{v\}$

$V_y \leftarrow V_y - \{v\}$

**else** ▷ Else find nearby node to facilitate  $v$

$U_{\min} \leftarrow \infty$

$n_{\min} \leftarrow \text{NULL}$

**for**  $n \in \{v(c, n) \in E \mid |\mathcal{P}(c, n)| < \text{SEARCH\_LIMIT}\}$  **do** ▷ Find nodes in certain depth limit

**if**  $U(n) < U_{\min}$  **then** ▷ Find lowest heuristic cost node to travel to

$U_{\min} \leftarrow U(n)$

$n_{\min} \leftarrow n$

**end if**

**end for**

$c \leftarrow n_{\min}$

**end if**

**end while**

**return**  $V_z^\alpha$

---

The below algorithm is then ran until all VNFs are placed, therefore providing a candidate network slice to be compared against the existing slice using the objective function.

### 2.2.3 Objective Function

Let  $O(S_x^\alpha, S_y^\alpha)$  be the objective function used to decide whether a proposed network slice,  $S_y^\alpha$ , should be implemented to replace an existing network slice,  $S_x^\alpha$ .

The function  $O$  is composed of several functions, which inspect individual aspects of a slice, with coefficients applied.

$O_{\text{PN}}$  inspects the number of physical nodes used in each network slice, it is simply the difference between the two.

$$O_{\text{PN}}(S_x^\alpha, S_y^\alpha) = |N(S_y^\alpha)| - |N(S_x^\alpha)| \quad (2.11)$$

$O_{\text{PLU}}$  analyses the number of physical links used by each network slice, it makes up part of  $O_{\text{PL}}$ .

$$O_{\text{PLU}}(S_x^\alpha, S_y^\alpha) = |\{\forall e \in E_y : \mathcal{P}(e)\}| - |\{\forall e \in E_x : \mathcal{P}(e)\}| \quad (2.12)$$

$O_{\text{SPL}}$  finds the difference between the slowest (i.e. lowest maximum bandwidth) physical link in the proposed network slice and the current network slice, it also makes up part of  $O_{\text{PL}}$ .

$$O_{\text{SPL}}(S_x^\alpha, S_y^\alpha) = \min\{\forall e \in E_y : \forall l \in \mathcal{P}(e) : B_{\max}(l)\} - \min\{\forall e \in E_x : \forall l \in \mathcal{P}(e) : B_{\max}(l)\} \quad (2.13)$$

$O_{\text{PL}}$  is the overall objective function with regards to physical links, it factors in  $O_{\text{PLU}}$  and  $O_{\text{SPL}}$ , with  $\epsilon$  and  $\zeta$  acting as weighting coefficients.

$$O_{\text{PL}}(S_x^\alpha, S_y^\alpha) = (\epsilon \cdot O_{\text{PLU}}(S_x^\alpha, S_y^\alpha)) + (\zeta \cdot O_{\text{SPL}}(S_x^\alpha, S_y^\alpha)) \quad (2.14)$$

$O_{\text{SU}}$  counts the number of spin ups that would need to be performed if  $S_y^\alpha$  was to be implemented in place of  $S_x^\alpha$ , this can be reduced if the the proposed slice was to use some pre-existing VNFs from the current slice.

$$O_{\text{SU}}(S_x^\alpha, S_y^\alpha) = |\{\forall v \in V_y : (v, H(v))\}| - |\{\forall v \in V_x : (v, H(v))\}| \quad (2.15)$$

$O_{\text{MT}}$  calculates the *minimum time* for a packet to traverse a slice (i.e. the time taken from the user equipment to the first VNF is assumed to be 0), and the time between the proposed network slice and original network slice are compared.

$$O_{\text{MT}}(S_x^\alpha, S_y^\alpha) = T(S_y^\alpha) - T(S_x^\alpha) \quad (2.16)$$

The overall objective function  $O$  is as follows:

$$O(S_x^\alpha, S_y^\alpha) = \Gamma_{\text{PN}} \cdot O_{\text{PN}}(S_x^\alpha, S_y^\alpha) + \Gamma_{\text{PL}} \cdot O_{\text{PL}}(S_x^\alpha, S_y^\alpha) + \Gamma_{\text{SU}} \cdot O_{\text{SU}}(S_x^\alpha, S_y^\alpha) + \Gamma_{\text{MT}} \cdot O_{\text{MT}}(S_y^\alpha, S_y^\alpha) \quad (2.17)$$

If  $\Gamma_{\text{PN}}, \Gamma_{\text{PL}}, \Gamma_{\text{SU}}, \Gamma_{\text{MT}} \in \mathbb{R}^+$ , then a positive result would be indicating that  $S_y^\alpha$  is a preferable network slice configuration to  $S_x^\alpha$ . This is not to say that  $O(S_x^\alpha, S_y^\alpha) > 0$  should dictate the decision, as the values of coefficients may not be tuned for this to be effective, as well as the objective function not taking into account all costs involved with modifying and replacing a network slice, however the larger the value returned, the stronger the candidate network slice is in comparison to the existing network slice.

### 2.2.4 Key Performance Indicators

In order to analyse the performance of an algorithm, the main focuses are present in the objective function shown above, to provide clarity indicators to be analysed in the performance of an algorithm include:

- Average latency from the user equipment to the end of a network slice/service function chain
- Minimum path latency (latency from first VNF to last VNF)
- Number of VNF spin-ups required to implement a network slice/service function chain
- Congestion caused on physical links (number of physical links used, amount of bandwidth remaining in physical link, etc.)

## 2.3 Tooling

### 2.3.1 Version Control

In regards to implementation, version control will be utilised to maintain consistency throughout the development process. Git will be used as it is the de facto standard, and a GitHub repository is provided by the University to store code related to the Individual Project.



Figure 2.2: Gitflow Diagram (Bitbull, 2016)

The Gitflow workflow will then be applied to handle the development process, with features being developed in separate branches, which in turn are merged into a develop branch, which is then merged into the main branch at the end of each sprint.

### 2.3.2 Implementation Choices

As part of the implementation, a simulation and benchmarking environment must be prepared, to test the algorithm's functionality, and measure its improvements regarding its objectives, and compare it to other algorithms present in literature.

This environment shall be coded in Python, and will take advantage of the `networkx` library, a package used for graph/network manipulation (Hagberg et al., 2008). This provides implementations of useful methods such as Dijkstra's Algorithm, which can be used when

searching for candidate nodes. Visualisation of data can be handled using `matplotlib` (Hunter, 2007), which `networkx` has good integration with.

Modern Python features such as built-in type hints have been used to increase code legibility and ease of code modification and expansion.

## 2.4 Project Management Methodology

For the development of the implementation, an iterative project management methodology will be used, including sprints similar to Agile methods.

The first sprint will aim to create the simulation environment, consisting of the graphs representing the physical and virtual networks, as well as the ability to simulate traffic across the networks. An implementation of the algorithm formulated in this report will also be developed by the end of this sprint.

The second sprint consists of, but not limited to, fixing bugs, implementing a benchmark algorithm from literature, extraction of benchmark results, and visualisation features if possible.

# Chapter 3

## Results

### 3.1 Implementation

#### 3.1.1 Simulator Implementation

As mentioned in Chapter 2, the implemented environment has to simply emulate the network as the graphs they have been modelled as in the Problem Formulation. All infrastructure has been coded as classes within Python, and networks consist of instances of these components, and are an instance of a class themselves.

#### Physical Network

The `Function` class (Listing C.16) acts as a descriptor of a VNF, it contains information such as the function's name, the computational cost of a VNF instance of this function, and the latency it would add.

Using the `Function` definition, we can now define a `Service` simply as a tuple of `Function` classes, a `Service` would then act as a descriptor for a Network Slice, or more generally a Service Function Chain.

In regards to this specific environment implementation, a `Node` (Listing C.17) refers to the physical hardware on which VNFs can be placed. For most real-world implementations of NFV, a 'node' would therefore be a server. An instance of `Node` has a unique identifier, as well as tracking how much of its computational resource is being used, as well as what its limit is. 2D co-ordinates are also used to model the physical location of the hardware to simulate latency.

In order to implement the Physical Network in the simulation environment, it is necessary to implement both its vertices and edges (hardware nodes and physical links respectively).

The `Link` class (Listing C.18) is the representation of the aforementioned edges. It contains information about its source node, destination node, latency, and current/maximum bandwidths. The `Link` class also has methods `allocate` and `deallocate` which monitors how much bandwidth is in use on said link (Listing C.1).

Now that the physical nodes and links have been modelled, it is now possible to model the physical network in its entirety. The `PhysicalNetwork` (Listing C.19) class has a set of nodes and links like any graph, but also contains a `Graph` structure from the `networkx` library, which represents the physical network, allowing the use of `networkx` functionality such as Dijkstra's Algorithm, filtering edges and vertices, and visualisation in conjunction with `matplotlib`.



## Virtual Network

The **Edge** class (Listing C.20) is to the Virtual Network as the **Link** class is to the Physical Network. As previously mentioned, Virtual Network edges traverse between two VNF instances, meaning that they do not need to be connected *directly* (i.e. an edge traversal may consist of multiple physical link traversals). This means that upon instantiation of a Virtual Network edge, there must be verification that the path consisting of physical links exists, the code which performs can be seen in Listing C.2.

VNFs are the vertices of the Virtual Network, the **VNF** class (Listing C.21) simply stores a **Node** instance which hosts the VNF and the **Function** that the VNF is an instance of.

The **VirtualNetwork** class (Listing C.22) stores its vertices and edges (VNFs and Edges respectively) but also the **PhysicalNetwork** instance it is a subgraph of, this is very useful when generating paths using a solution as, by being aware of the Virtual Network, the algorithms are fully informed about the Physical Network it is a part of. Methods are implemented with validation such that the integrity of the simulation is not violated (Listing C.3).

## Packets

Traffic is implemented through a **Packet** class (Listing C.23). The class holds two co-ordinates to represent a physical location of a packet's origin, and calculates the latency by adding up VNF and path latency, and finding the Euclidean distance from the origin to the first VNF and multiplying this by a set weight representing a distance-to-latency conversion.

## Network Slices & Service Function Chains

The outputs of the implemented algorithms will be proposed Network Slices or Service Function Chains. They have similar classes implemented to store this information: **NetworkSlice** (Listing C.24) and **Chain** (Listing C.25) respectively. They both have methods for adding VNFs and setting a path (tuple of Edges) to connect the aforementioned VNFs.

### 3.1.2 Solution Implementation

The initialisation function of the proposed solution (Listing C.26) works as follows:

1. Find a suitable starting node
2. Generate a slice starting at found starting node
3. Execute the objective function with the proposed and original network slices
4. Return the proposed network slice with its objective function evaluation

## Starting Node Selection

As mentioned in the Solution Formulation in Chapter 2, the initial idea for selection of a starting node was to count the number of hops taken by packets in a sample traffic set. However, after running simulation tests, it was seen that this method was not effective in

formulating solutions that effectively reduce the target KPI (latency). Code Listing C.4 is a code snippet showing how this is implemented.

In its place (Listing C.27), the hardware nodes at the edge of the physical network are iterated over, and the one that produces the lowest average Euclidean distance over the sample traffic set is chosen. This is effective as in the model, latency from the device to the network edge is proportional to its distance.

The integration of `networkx` into the simulated network definitions allows for easy retrieval of leaf nodes (See Listing C.5).

### Proposed Slice Generation

Once a starting node is chosen, the proposed slice can be created, it aims to place as many VNFs at a node as possible, similarly to Savi et al. (2021), and traverses to the node which has the lowest distance based on the heuristic defined in Chapter 2. The full function code can be found at Listing C.28.

The traversal heuristic, as shown in the Solution Formulation, consists of the weighted-sum of three factors:

1. How many VNFs can be placed at a node (Listing C.6)
2. How much bandwidth remains on the links if it was to be traversed to (Listing C.7)
3. The theoretical latency to traverse to the node

When searching for a new hardware node to place VNFs, the function looks a certain degree away from the current node, in this implementation it is three, however this can be altered based on computational power, however, there will be diminishing returns as the depth increases. A code snippet which gives a comprehensive understanding of the implementation is available at Listing C.8

### Objective Function

The objective function compares two network slices that serve the same service. In this case, it is used to compare the proposed slice generated by the above code listing, and the pre-existing slice it is trying to improve upon. Coefficients can be tuned to prioritise different factors in each network.

It is implemented in the same way it was formulated, as the weighted sum of several different heuristics:

The Physical Nodes Heuristic finds the difference in number of different hardware nodes used to implement the slice (See Listing C.9).

The Links Used Heuristic returns the difference in number of physical links used between slices (See Listing C.10).

The Slowest Link Heuristic return the bandwidth delta between the slowest link found in each slice (i.e. the bottleneck) (See Listing C.11).

The Spin Ups Heuristic returns the number of spin ups **avoided** by re-using existing VNFs (See Listing C.12).

The Minimum Time Heuristic returns the time taken to traverse the slice path, assuming routing latency and traversal to first VNF latency is zero (See Listing C.13).

### Coefficient Tuning

Since the solution proposed in this report is a weighted-sum heuristic algorithm, it is important to have acceptable coefficient values. For this reason, tuning code was written that correlates coefficient change to positive KPI changes (latency reduction).

The code changes a coefficient value and attempts the same increment/decrement two more times to see if this change enforces this correlation, if so, its effect will be noted, after iterating over given bounds, it will present the coefficient value which produced the most effective KPI change.

`weights.py` is the implementation of the code, and outputs logs of the results at each step of the tuning, so this can be inspected manually which may gleam more information regarding how to tune a given coefficient.

This code will not produce the most effective coefficients, as this can be network-specific, and tuning will benefit from more complex methods, this is something that is discussed in Chapter 4's Ideas for Future Work section.

### 3.1.3 Benchmark Implementation

To apply comparisons concerning efficacy of the proposed solution, it is necessary to compare it to other available solutions, this can be the most primitive case (i.e. 'brute-force' placement of VNFs) and also other proposed solutions found in literature. These are then compared against possible KPIs to find the strengths and weaknesses of each approach.

In this report, the proposed solution will be compared against:

1. Randomised placement of VNFs within a network, and then connecting them together using shortest paths (Dijkstra's Algorithm, edge weights are latency-based) between them
2. A generic placement algorithm, starting by placing a VNF at an edge node, and then placing as many VNFs at a node until out of resources, and then placing at node that is connected to the previously mentioned node.
3. The proposed heuristic cost-aware solution by Savi et al. (2021), this was chosen due to the similar problem formulation and solution family, with the solution having different focus KPIs.

### Randomised Algorithm

The randomised algorithm is the most basic implementation of Network Slice/Service Function Chain generation present in the simulator, it picks a random hardware node (with sufficient resources) to place a VNF, and then repeats the process until all VNFs are placed.

Once all VNFs are placed, it used Dijkstra’s Algorithm to find the shortest path from VNF to VNF (in chain order), the Physical Network graph implementation uses the link latencies as the edges.

### Generic Algorithm

The generic benchmark algorithm starts by selecting a random leaf node, Listing C.5 shows how to retrieve the leaf nodes from the graph

The placement of VNFs is then handled by placing as many as possible in a VNF and then traversing to a neighbouring node to perform the same action (it can backtrack if necessary), Listing C.14 shows the precise implementation details.

The second half of the VNF-FGE problem (connecting the VNFs together) is handled by the `create_edges` function (See Listing C.15).

### Savi et al. (2021) Heuristic Cost-Aware Algorithm

In order to compare the report’s proposed algorithm, an algorithm from literature has been chosen to benchmark it. In Savi et al. (2021), a cost-aware heuristic solution is shown. Cost-aware means that the algorithm is focused on a financial KPI, therefore minimising use of different nodes and reusing VNFs wherever possible.

The implementation shown by Savi et al. has two phases, for the purpose of this report, we are going to be focusing on Phase 1, as this will show the impact of different types of awareness within algorithms whilst focusing on different KPIs.

Implementing this algorithm involves sorting VNFs by distance and hardware nodes by available capacity, this was implemented using in-place bubble sort in the simulation environment.

#### 3.1.4 Implementation Evaluation

To conclude the implementation phase of the report, it is important to reflect on the issues that arose during the implementation process. The vast majority of issues were simple logical issues when generating paths for service function chains/network slices, as cycles forming around the graph when generating a path can lead to the program hanging.

The vast majority of bugs that caused issues were different edge cases being reached, and the implementation not sufficiently handling it. There are multiple different type of remedies on a case-by-case basis, and can include re-running a sub-procedure, or solving the theoretical edge case in a unique manner.

All critical bugs have been mitigated by the completion of this report, however, it is worth noting that due to time constraints, that there is room for improvement within the implementation that would allow for the retrieval of more qualitative data for analysis.

## 3.2 Testing

Since the problem space, proposed solution, and other algorithms have been implemented, it is possible to run benchmarks on the algorithms, to extract results on their efficacy in different scenarios focusing on the latency KPI. Tests include:

1. Varying the number of Core nodes in the network (random network generation)
2. Varying the number of Edge nodes in the network (random network generation)
3. Varying the maximum bandwidth on physical links (random network generation)
4. Varying the maximum computational resources available on hardware nodes (random network generation)
5. Testing on a given small-sized network
6. Testing on a given medium-sized network
7. Testing on a given large-sized network
8. Testing on a congested network (i.e. many slices already placed in the network)

To test the Savi et al. algorithm, two of each VNF of the test service are placed randomly across the network (this is done for all tested algorithms) in the Core and Edge Nodes Test as well as the Small, Medium, Large, and Congested Network tests, allowing for the main use of the algorithm (cost-aware) to be shown, although this shows it's unfavourable performance in latency-focused KPI benchmark.

### Core Nodes Test

A test was ran from 4 core nodes, up to an including 50 core nodes, with a step of 2 nodes each test. At each test, 100 iterations of the algorithm was executed and the average path latency taken on a randomly generated network.

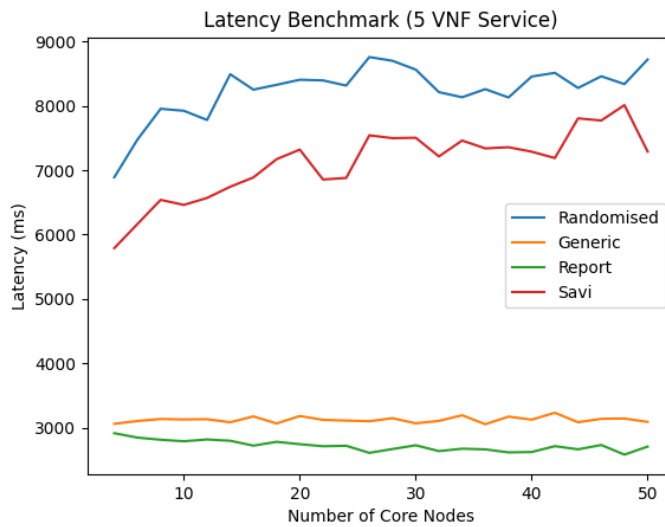


Figure 3.1: Variance of Core Nodes Amount Test Results

### Edge Nodes Test

Similar to the Core Nodes Test, testing started at 4 edge nodes, but up to 30 edge nodes, with a step of 2 per test. 100 iterations of each algorithm was performed at each step with average latency taken on a randomly generated network.

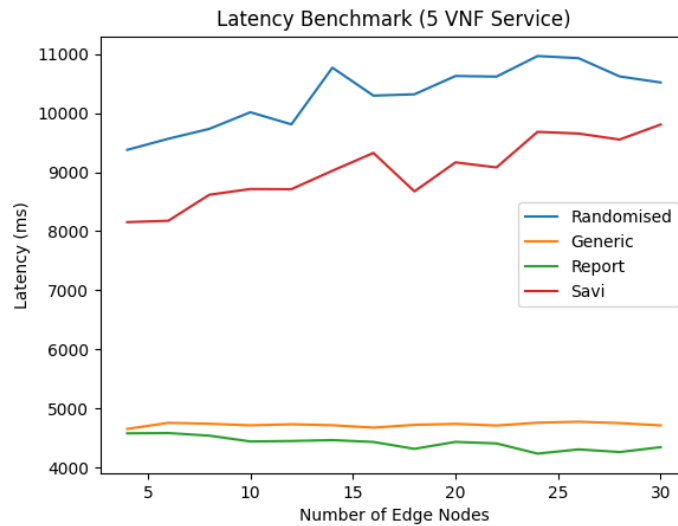


Figure 3.2: Variance of Edge Nodes Amount Test Results

### Bandwidth Test

The Bandwidth Test aims to see how algorithms react when the maximum bandwidth of physical links are set to certain values. Each algorithm was executed for 100 iterations at each value.

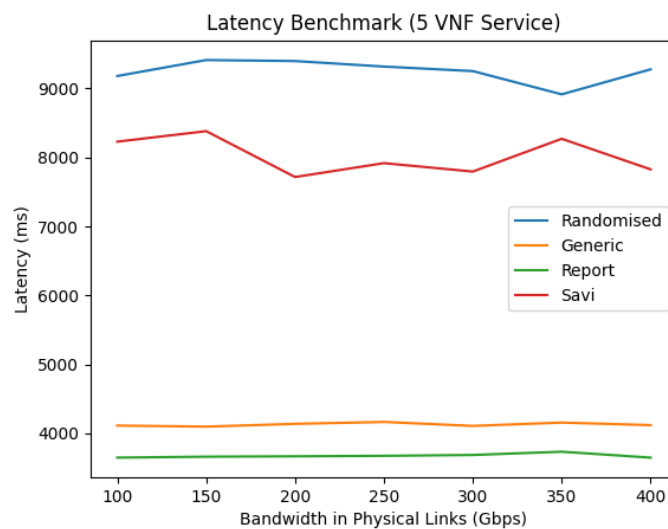


Figure 3.3: Bandwidth Variance Test Results

### Computational Resources Test

In this test, the amount of computational resources is set to the same value for each node, and is altered to a range of values, where each algorithm was tested 100 iterations at each value.

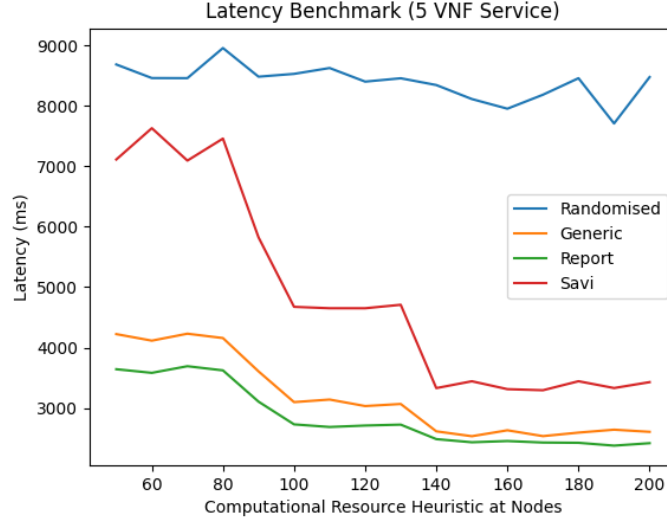


Figure 3.4: Variance of Available Computational Resource Test Results

### Small Network

A pre-defined network consisting of five core nodes, with six links between them, and three edge nodes each with one link to a core node. 100 runs of each algorithm were executed and the average of each taken.

Algorithm Type	Latency (ms)	Edge Node %
Randomised	4502	40.6
Generic	3650	43.7
Report	3635	29.8
Savi et al.	4260	28.5

Table 3.1: Small Network Test Results (100 iterations)

### Medium Network

A pre-defined network consisting of fifteen core nodes, with twenty links between them, and five edge nodes with one link to a core node each. The medium network is an extension to the small network, with same nodes and links, just with more added. Like with the Small Network test, 100 iterations of each algorithm were executed.

Algorithm Type	Latency (ms)	Edge Node %
Randomised	4224	22.6
Generic	2673	42.4
Report	2640	33.0
Savi et al.	3903	15.0

Table 3.2: Medium Network Test Results (100 iterations)

### Large Network

A pre-defined network with 25 core nodes, 40 links between them, and ten edge nodes with a link to a core node each. As with both the Small and Medium Network tests, 100 iterations for each algorithm were executed.

Algorithm Type	Latency (ms)	Edge Node %
Randomised	4624	26.6
Generic	2709	39.5
Report	2697	35.0
Savi et al.	4413	16.7

Table 3.3: Large Network Test Results (100 iterations)

### Congested Network

Uses the same pre-defined network as in the Large Network test, but adds a set number of VNFs/SFCs within the network to see how the algorithms are effected by the presence of numerous VNFs already in the network.

Algorithm Type	Latency (ms)	Spin Ups (/5)
Randomised	4543	5
Generic	2772	5
Report	2681	5
Savi et al.	3919	0

Table 3.4: Congested Network Test Results (100 iterations)



# Chapter 4

## Discussion

### 4.1 Results Evaluation

When analysing the test results focusing on the latency KPI, it is trivial to notice the following trends:

The randomised placement algorithm is the least efficient, this is evidently due to that no context is used to aid VNF placement.

The cost-aware heuristic algorithm found in Savi et al. (2021) outperforms the randomised placement algorithm in terms of path latency. In most realistic cases, the algorithm will be initialised in an environment where the VNFs present in the desired chain are already present in the network, the algorithm will prioritise those existing VNFs at the cost of latency.

The results shown in Figure 3.1 and Figure 3.2 show a big disparity between the Randomised algorithm and the implementation of Savi et al. (2021) compared to the generic latency-aware algorithm and LAGHA (the report algorithm) which perform similarly.

The Savi et al. algorithm performs similarly to the randomised algorithm due to VNFs already present in the simulation (that have been placed randomly) which the Savi et al. algorithm tries to take advantage of.

As the number of edge nodes increases, the performance gap between the generic algorithm and LAGHA also noticeably increases, this is due to LAGHA being able to select more appropriate starting nodes concerning latency, demonstrating that selection node algorithm (which was changed during the implementation phase) is effective.

Limited conclusions can be drawn from the bandwidth test results in Figure 3.3, this can be explained by a lack of persistent pre-existing VNFs/Service Function Chains/Network Slices in the network, thereby causing a lack of bandwidth exhaustion on physical links.

In Figure 3.4, the Savi et al. algorithm closes the gap between itself and the two latency-aware algorithms as there are no pre-placed VNFs in this test and can place more VNFs at the same node, much like the latency-aware algorithms. This is due to the Savi et al. algorithm being able to take advantage of placing multiple VNFs on a single node, similarly to the other latency-aware solutions.

Table 3.1 shows how the algorithms perform on a small network. The network is identical for each algorithm, and the results are consistent with tests mentioned above. It is worth mentioning the negligible difference between the latency of the Generic and Report algorithms, with the Report algorithm only showing a 0.4% improvement over the Generic algorithm in this case.

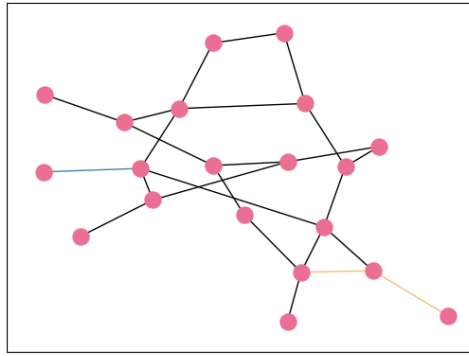


Figure 4.1: Generic Algorithm (orange) and Report Algorithm (turquoise) paths on medium test

When the network size is increased in the Medium Network Test, we see an improvement in latency of all four algorithms (see Table 3.2) although as explained later, there may be other factors contributing to this, as well as the performance gaps between algorithms being closed. The lower latencies in comparison to the small test can be credited due to the small size of the network being a performance constraint in the previous test.

The large network test (Table 3.3) reinforces the similar pattern behaviour of the previously mentioned tests, with a near identical trend of the two latency-aware algorithms performing very similarly. This indicates that the medium-sized network and large-sized network have similar constraints in their environment.

Inspecting the percentage of nodes used being Edge nodes in the Small, Medium, and Large network benchmarks, we can see a recurring pattern that the report algorithm uses Edge nodes less often than the generic latency-aware solution. This is due to the traversal heuristic proposed favouring higher bandwidth links (to avoid link congestion, a KPI), which tend to be core links as opposed to edge links, as this is likely the case in a real-world scenario.

The Savi et al. algorithm has the lowest edge node utilisation percentage due to it favouring pre-existing VNF instances, and the shortest paths are more likely to be at VNFs in core nodes, as it removes the need to have to traverse to the edge of the physical network.

When the network is more congested, the algorithm proposed in Savi et al. (2021) seems to perform slightly closer to the latency-aware algorithms, as it can save cost (i.e. spin ups) and be able to pick a more latency-efficient path (see Table 3.4).

The number of spin-ups is also shown in Table 3.4, to show how the algorithms' performance can be viewed differently when looked in the perspective of different KPI, it is shown in the Congested Network scenario, the report algorithm still prefers a route without utilising spin ups, this may mean coefficients could be tuned to take advantage of spin ups more if the network would like the algorithm to be more cost-aware.

It is worth noting that for the Small, Medium, Large, and Congested Network tests, the packets generated (which are the same for each algorithm), are randomly generated per iteration, meaning that test to test comparison efficacy is reduced due to random component of

packet locations. However, the comparison between algorithms within the test leads to accurate conclusions.

Taking into account the report objectives mentioned in Chapter 1, the RO1 (proposing and implementing a viable graph traversal heuristic), has been achieved as we can see the report algorithm consistently outperform a generic latency-aware algorithm due to its traversal and starting node algorithms. RO2 (overall solution maximises KPI considerations) has been achieved, as the constraints were successfully implemented and solutions are only provided that meet said constraints, however this is an area which can be worked upon further, alongside others that are mentioned in the next section. RO3 has also been achieved with not only a literature cost-aware heuristic algorithm being implemented for comparisons on strengths and weaknesses, but also two other benchmarks to gain further insight on algorithmic performance.

## 4.2 Ideas for Future Work

To further progress in this area of research, it seems evident that it is necessary to harness the power of machine learning models. As the power of AI models increase, utilising them in monitoring of NFV deployments could lead to a level of flexibility, granularity and dynamism that is not achievable by standard numerical methods. This area has been somewhat investigated in Chetty et al. (2021).

In closer relation to the work described in this report, further work concerning tuning the coefficients could be done, to maximise the efficacy of the algorithm, as it is heavily reliant upon the selection of appropriate coefficients for its functionality. Meta-heuristic solutions are an avenue of research which would target this area of heuristic tuning.

In regards to improve the simulation environment, simulating a consistent live network (consistent traffic flows over time, multiple network slices running at a time consistently), would allow for more qualitative data to be analysed from the tests, especially the bandwidth test. This would also allow analysis into 'triggers' for algorithm initialisation, and this can be investigated to further improve algorithmic design and implementation.

Although comparative analysis has been performed in papers such as Golkarifard et al. (2021), in-depth analysis of the cases in which several algorithms perform may provide useful insights and hybrid algorithms could be implemented to handle different scenarios and allow for a better outcome for a given network.

Concerning the problem formulation, using directed graphs in place of undirected graphs can better reflect, the duplex standard in ISP networks (a pair of fibres are used for separate transmission and receive when connecting two devices together).

In future, greater focus being placed on 5G and beyond mobile networks, one of the most common beneficiaries of NFV advancements, could be further implemented into the research from this report. This could involve looking into the Common Core Network Function slice, and the intricacies involving its placement for optimal latency of all traffic involved.

In addition, focusing on the components (VNFs) that make up specific real-world services, as

opposed to utilise a generic Service Function Chain for simulation/testing, to allow for more context as to which services benefit more and less from respective algorithms, and also the different KPIs to prioritise for different services.

A final idea in regards to possible future developments, is to advance the complexity of the modelling, this could involve having a more traditional network hierarchy as is present in ISP networks, such as Metro networks mentioned in Hmaity et al. (2020), also factoring in penalties such as VNF consolidation as mentioned in Zhang et al. (2019), as although assumptions allow us to simplify a problem and reach conclusions easier, a more accurate simulation/model is always a positive improvement.

In conclusion, this report has explored the research area in detail, and uniquely attempts to design and implement a heuristic latency-aware solution within a network slicing topology in a protocol-agnostic manner. Further work may entail different solution avenues, improving tuning of coefficients and increased simulation detail such as size, physical network topologies, and different services to be tested.

# References

- Bitbull (2016). Gitflow Workflow. [Online]. [Accessed 7th March 2024]. Available from:  
**URL:** <https://www.bitbull.it/en/blog/how-git-flow-works/>
- Brodkin, J. (2024). ISPs can charge extra for fast gaming under FCC’s Internet rules, critics say. [Online]. [Accessed 18th April 2024]. Available from:  
**URL:** <https://arstechnica.com/tech-policy/2024/04/isps-can-charge-extra-for-fast-gaming-under-fccs-internet-rules-critics-say/>
- Carpio, F., Dhahri, S. and Jukan, A. (2017). VNF placement with replication for Loac balancing in NFV networks *2017 IEEE International Conference on Communications (ICC)* pp. 1–6.
- Chai, X., Wang, Y., Zhang, M. and Cong, L. (2023). Efficient VNF-FG Scaling Algorithm for 5G Network Slices *2023 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)* pp. 1–5.
- Chetty, S. B., Ahmadi, H., Sharma, S. and Nag, A. (2021). Virtual Network Function Embedding under Nodal Outage Using Deep Q-Learning *Future Internet* **13**(3).  
**URL:** <https://www.mdpi.com/1999-5903/13/3/82>
- Chiaraviglio, L., D’Andreagiovanni, F., Rossetti, S., Sidoretti, G., Blefari-Melazzi, N., Salsano, S., Chiasserini, C.-F. and Malandrino, F. (2019). Algorithms for the design of 5G networks with VNF-based Reusable Function Blocks *Annals of Telecommunication* **74**, 559–574.
- Corno, F., De Russis, L. and Pablo Sáenz, J. (2018). On The Advanced Services That 5G May Provide To IoT Applications *2018 IEEE 5G World Forum (5GWF)* pp. 528–531.
- Gil Herrera, J. and Botero, J. F. (2016). Resource Allocation in NFV: A Comprehensive Survey *IEEE Transactions on Network and Service Management* **13**(3), 518–532.
- Golkarifard, M., Chiasserini, C. F., Malandrino, F. and Movaghar, A. (2021). Dynamic VNF placement, resource allocation and traffic routing in 5G *Computer Networks* **188**, 107830.  
**URL:** <https://www.sciencedirect.com/science/article/pii/S1389128621000177>
- Hagberg, A. A., Schult, D. A. and Swart, P. J. (2008). Exploring Network Structure, Dynamics, and Function using NetworkX in G. Varoquaux, T. Vaught and J. Millman (eds) *Proceedings of the 7th Python in Science Conference* Pasadena, CA USA pp. 11 – 15.
- Hmaity, A., Savi, M., Askari, L., Musumeci, F., Tornatore, M. and Pattavina, A. (2020). Latency- and capacity-aware placement of chained Virtual Network Functions in FMC metro networks *Optical Switching and Networking* **35**, 100536.  
**URL:** <https://www.sciencedirect.com/science/article/pii/S1573427719300268>
- Hunter, J. D. (2007). Matplotlib: A 2d graphics environment *Computing in Science & Engineering* **9**(3), 90–95.

- Khebbache, S., Hadji, M. and Zeghlache, D. (2017). Virtualized network functions chaining and routing algorithms *Computer Networks* **114**, 95–110.  
**URL:** <https://www.sciencedirect.com/science/article/pii/S1389128617300087>
- Li, J., Shi, W., Ye, Q., Zhuang, W., Shen, X. and Li, X. (2018). Online Joint VNF Chain Composition and Embedding for 5G Networks *2018 IEEE Global Communications Conference (GLOBECOM)* pp. 1–6.
- Li, T., Zhou, H. and Luo, H. (2017). A new method for providing network services: Service function chain *Optical Switching and Networking* **26**, 60–68. Advances on Path Computation Element.  
**URL:** <https://www.sciencedirect.com/science/article/pii/S1573427715000764>
- Mahboob, T., Jung, Y. R. and Chung, M. Y. (2020). Dynamic VNF Placement to Manager User Traffic Flow in Software-Defined Wireless Networks *Journal of Network and Systems Management* **28**, 436–456.
- Oxford University (no year). Responsibilities under GDPR. [Online]. [Accessed 21st April 2024]. Available from:.  
**URL:** <https://researchsupport.admin.ox.ac.uk/policy/data/responsibilities>
- Savi, M., Tornatore, M. and Verticale, G. (2021). Impact of Processing-Resource Sharing on the Placement of Chained Virtual Network Functions *IEEE Transactions on Cloud Computing* **9**(4), 1479–1492.
- Schardong, F., Nunes, I. and Schaeffer-Filho, A. (2021). NFV Resource Allocation: a Systematic Review and Taxonomy of VNF Forwarding Graph Embedding *Computer Networks* **185**, 107726.  
**URL:** <https://www.sciencedirect.com/science/article/pii/S1389128620313189>
- Systems Approach LLC (2022). Software-Defined Networks: A Systems Approach - Chapter 1: Introduction. [Online]. [Accessed 16th April 2024]. Available from:.  
**URL:** <https://sdn.systemsapproach.org/intro.html>
- Thiruvassagam, P. K., Chakraborty, A., Mathew, A. and Murthy, C. S. R. (2021). Reliable Placement of Service Function Chains and Virtual Monitoring Functions With Minimal Cost in Softwarized 5G Networks *IEEE Transactions on Network and Service Management* **18**(2), 1491–1507.
- van Schewick, B. (2024). Harmful 5G Fast Lanes Are Coming. The FCC Needs To Stop Them. [Online]. [Accessed 18th April 2024]. Available from:.  
**URL:** <https://cyberlaw.stanford.edu/blog/2024/04/harmful-5g-fast-lanes-are-coming-fcc-needs-stop-them>
- VMware (no year). What is Software-Defined Networking. [Online]. [Accessed 21st April 2024]. Available from:.  
**URL:** <https://www.vmware.com/topics/glossary/content/software-defined-networking.html>
- Zhang, Q., Liu, F. and Zeng, C. (2019). Adaptive Interference-Aware VNF Placement for

- Service-Customized 5G Network Slices *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications* pp. 2449–2457.
- Zhang, S. (2019). An Overview of Network Slicing for 5G *IEEE Wireless Communications* **26**(3), 111–117.

# Appendix A

## Self-appraisal

### A.1 Critical Self-Evaluation

In regards to critically self-evaluating, I believe many of the areas for improvement tie in well with the Ideas for Future Work section, as they both cover some of the holes and avenues for strengthening this report.

The networks simulated as part of the implementation only scratch the surface of the complexity that can be present in ISP networks for example, with varying server deployments at the Metro and Core layers. This leads to different constraints and strengths of VNF placement in these layers. Combined with this, I believe modelling the network graphs as directed, with physical links being two separate directed edges, would better represent the duplex links used in network in hindsight.

As mentioned in the Literature Review, Deep Learning is an area of active research in the field of NFV, and an area I believe could have been further scrutinised in the report, however with my unfamiliarity in this area, combined with the amount of research having to be taken on to grasp the research area and provide a solution, time was unfortunately the limiting factor.

From a positive point of view, this report proves that a weighted-sum heuristic latency-aware algorithm exists for the VNF-FGE problem in a Network Slicing topology, also with proof that the added heuristics help outperform a generic latency-aware solution, with both greatly outperforming randomised generation.

I believe that every aspect of the report was achieved somewhat, which I consider a great success due to the breadth of considerations taken into account in the report. However, time management did become difficult, with one bug in particular, regarding bandwidth allocation, only getting fixed as the report was being finalised. This is reflected upon more in the next section.

Finally, in regards to implementation, I believe adding more seamless integration with the very well-designed `networkx` library would have been a good implementation decision, due to how well it helped with the development process. This could have been done by making the network definitions inherit from `networkx` classes.

### A.2 Personal Reflection and Lessons Learned

In the perspective of personal reflection, I am proud of the report I have managed to produce, especially considering the difficulty of the problem at hand, with the VNF-FGE problem being at the cutting edge of research, as evidenced by the recency of the works present in the Literature Review.



Personally, I do not find the deeply mathematically-rooted areas of Computer Science to be my strengths, especially areas such as Optimisation. However, I committed to a research project in an area that is interesting, as well as pertinent to my placement from the year prior, as opposed to tackling a dissertation that I knew was in my capabilities and could be completed with ease but for very little personal achievement.

To have released a working solution that does have a positive effect on reducing latency in a NFV environment (in a sample of 1000 iterations, the report algorithm finds improvements over the generic latency-aware benchmark around 70% of the time in a 30 node simulation), does make me proud of my efforts. As mentioned in Chapter 4 of the report and in the Critical Self-Evaluation, I do believe that there are improvements to be made from this report, but nevertheless I am pleased that my efforts have led to a tangible result.

A lesson learned in the implementation process was to start the process of result extraction, benchmarking, and testing from as early as possible, this allows for issues and bugs to arise when an exhaustive testing procedure is being developed. The issue of path traversal of algorithms utilising Dijkstra's shortest path failing (cycling) in certain cases due to bandwidth allocation issues, was picked up very late and had to be tackled when the project was not in a flexible state.

Furthermore in regards to lessons learned, I have learned vast amounts about the academic writing process in regards to structure and literature reviews, but also the difficulty of time management with balancing a dissertation of this size alongside the Level 3 taught modules. I am thankful to my supervisor, Arash Bozorgchenani, for the consistent fortnightly meetings that helped keep my project progress on track.

## **A.3 Legal, Social, Ethical and Professional Issues**

### **A.3.1 Legal Issues**

Legal issues were not of any particular relevance in the creation of this report, as this report focuses on the production of algorithm to solve the VNF-FGE problem, and simulates the network, purely by representing it as an undirected graph. No real traffic is used and generic 'packets' are used to create abstract traffic on the network.

If the work on this report was to be advanced in the future, a good form of empirical testing could be to test the algorithm on real-samples of traffic, which would have accurate per-service workload distributions, but may run into legal issues such as data not being collected for its explicit purpose, a necessary requirement under GDPR (Oxford University, no year).

### **A.3.2 Social Issues**

There are no evident social issues present within this report due to it being a network traffic formulation algorithm, and with little social impact.

The overarching topic of NFV may lead to social discourse if costs of using the network became service specific, tying in with the idea of Net Neutrality, which is mentioned below in Ethical

Issues.

### A.3.3 Ethical Issues

There are no direct ethical issues present within this report as it is the formulation of an algorithm that has been implemented in a simulated environment focusing on NFV. However, it is worthwhile to reflect on the ethical issues the wider area of NFV can present as a whole.

As NFV can provide service-specific traffic management, it is clear that this can cause issues with *Net Neutrality*, the idea that the ISP should treat all traffic across its network, such that it cannot prioritise certain user groups or services.

As SDN and NFV can distinguish traffic between services, and utilises this ability to improve a network, it is evident that it paves the way to breach Net Neutrality.

The use of Network Slicing in particular, has, as of writing, recently been thrust into the spotlight in technical journalism in particular (Brodkin, 2024).

Barbara van Schewick, Professor of Law at Stanford Law School, summarises that due to the proposed rules by the FCC, Network Slicing will "make it possible for mobile ISPs to start picking applications and putting them in a fast lane - where they'll perform better generally and much better if the network gets congested" (van Schewick, 2024).

### A.3.4 Professional Issues

There are no specific professional issues regarding this report due to the same reason there are a lack of report-specific issues in the other aforementioned areas. As long as the issues mentioned above are planned for and mitigated, then professional standards will have been upheld.

# Appendix B

## External Material

`networkx` (Hagberg et al., 2008) is an open-source Python library for graph environments. This report takes advantage of its Dijkstra’s Algorithm implementation, node/edge filtering, and integration with `matplotlib`, a Python graph plotting library (Hunter, 2007).

# Appendix C

## Figures, Tables and Listings

### C.1 Figures

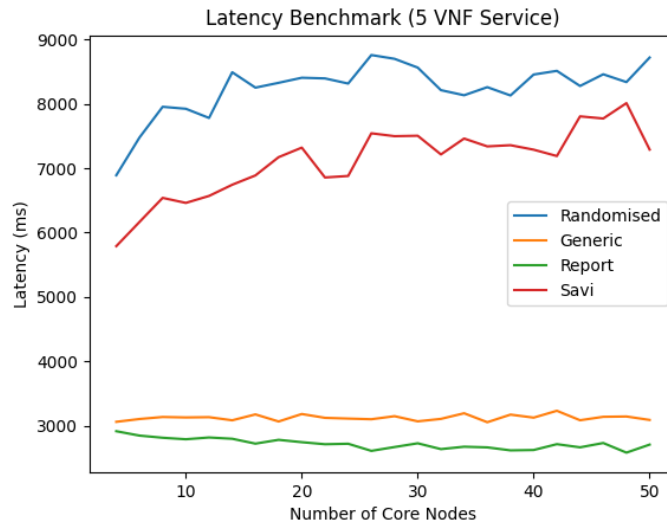


Figure C.1: Variance of Core Nodes Amount Test Results

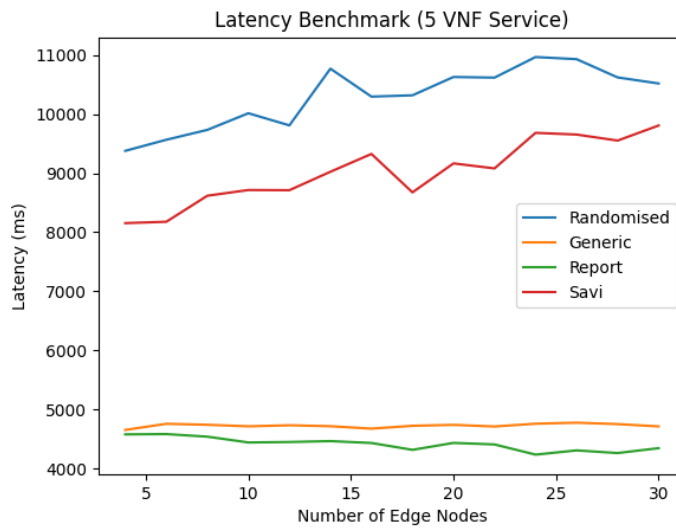


Figure C.2: Variance of Edge Nodes Amount Test Results

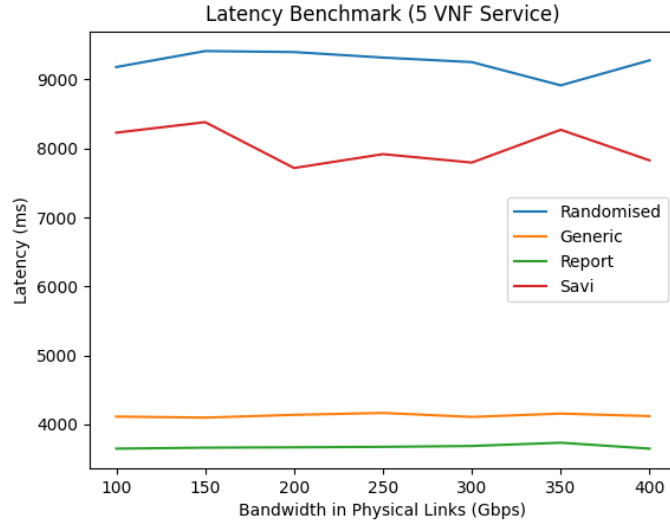


Figure C.3: Bandwidth Variance Test Results

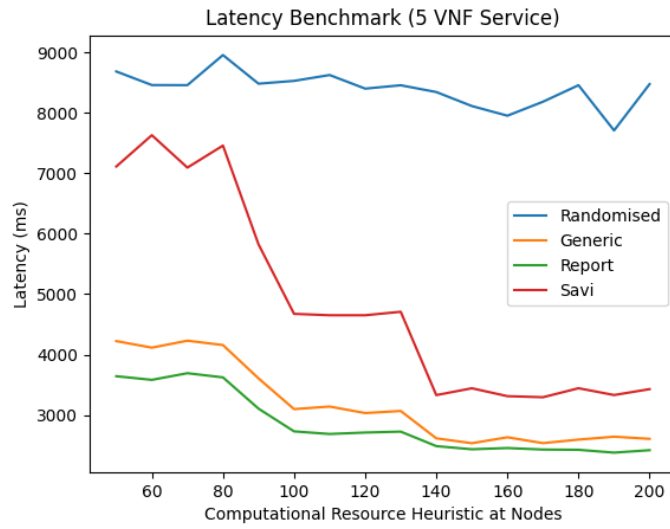


Figure C.4: Variance of Available Computational Resource Test Results

## C.2 Tables

Algorithm Type	Latency (ms)	Edge Node %
Randomised	4502	40.6
Generic	3650	43.7
Report	3635	29.8
Savi et al.	4260	28.5

Table C.1: Small Network Test Results (100 iterations)

Algorithm Type	Latency (ms)	Edge Node %
Randomised	4224	22.6
Generic	2673	42.4
Report	2640	33.0
Savi et al.	3903	15.0

Table C.2: Medium Network Test Results (100 iterations)

Algorithm Type	Latency (ms)	Edge Node %
Randomised	4624	26.6
Generic	2709	39.5
Report	2697	35.0
Savi et al.	4413	16.7

Table C.3: Large Network Test Results (100 iterations)

Algorithm Type	Latency (ms)	Spin Ups (/5)
Randomised	4543	5
Generic	2772	5
Report	2681	5
Savi et al.	3919	0

Table C.4: Congested Network Test Results (100 iterations)

## C.3 Listings

**NOTE:** The code listings present in this section are for comprehension reasons only and may not be the final version of the code, up-to-date code can be found at the provided GitHub repository.

### C.3.1 Code Snippets

#### Simulator Implementation

```

1 def allocate(self, value: int | float):
2     if self.bandwidth + value > self.bandwidth_max:
3         raise ValueError("Allocation would overload link")
4     self.bandwidth += value
5
6 def deallocate(self, value: int | float):
7     self.bandwidth -= value
8     if self.bandwidth < 0:
9         self.bandwidth = 0

```

Listing C.1: Link methods allocate and deallocate

```

1 # --CODE SNIPPED-- #
2 x = src.host

```

```

3 y = dst.host
4 latency = 0
5
6 # Verifies path goes from src to dst
7 # Also sums latency
8 for link in path:
9     if link.src == x:
10         x = link.dst
11         latency += link.latency
12     elif link.dst == x:
13         x = link.src
14         latency += link.latency
15
16 if x != y:
17     raise ValueError("Path does not link VNFs together")
18 # -- CODE SNIPPED -- #

```

Listing C.2: Edge Path Verification

```

1 # Add VNF to virtual network
2 def add_vnf(self, vnf: VNF):
3     if vnf in self.vnfs:
4         raise ValueError(f"VNF {vnf.name} already in network")
5     if vnf.function.cost > vnf.host.resource_max - vnf.host.resource:
6         raise ValueError(f"Insufficient resources at {vnf.host.name}
7             for VNF {vnf.name}")
8     self.vnfs.append(vnf)
9     self.graph.add_node(vnf)
10    vnf.host.resource += vnf.function.cost
11
12 # Add edge (VNF-VNF) to virtual network
13 def add_edge(self, edge: Edge):
14     for e in self.edges:
15         if edge.src == e.src and edge.dst == e.dst:
16             raise ValueError(f"Edge between {edge.src.name}
17                 and {edge.dst.name} already exists")
18         elif edge.src == e.dst and edge.dst == e.src:
19             raise ValueError(f"Edge between {edge.src.name}
20                 and {edge.dst.name} already exists")
21     self.edges.append(edge)
22     self.graph.add_edge(
23         edge.src,
24         edge.dst,
25         weight = edge.latency
26 )

```

Listing C.3: VirtualNetwork Methods add\_vnf and add\_edge

### Solution Implementation

```

1 total = 0
2 for packet in packets:
3     # Distance is Euclidean distance from Packet to Leaf
4     distance = sqrt(

```

```

5         (leaf.x - packet.x)**2 +
6         (leaf.y - packet.y)**2
7     )
8     total += distance * settings.LATENCY_WEIGHT
9 total /= len(packets)

```

Listing C.4: Obtaining Average Latency to given Leaf Node

```

1 leaves: list[Node] = [
2     n
3     for n in network.pn.graph.nodes
4     if network.pn.graph.degree(n) == 1
5 ]

```

Listing C.5: Obtaining leaf nodes from VirtualNetwork using networkx

```

1 # -- CODE SNIPPED -- #
2 for i in range(index, len(service)):
3     cost = service[i].cost
4     # available is the amount of computational resource available
5     # at given node
6     if available - cost > 0:
7         return num
8     else:
9         available -= cost
10        num += 1
11 # -- CODE SNIPPED -- #

```

Listing C.6: Report Algorithm Heuristic VNF Placement Snippet

```

1 # -- CODE SNIPPED -- #
2 for p in packets:
3     cost += p.bandwidth
4
5     for link in links:
6         # If cost would overload link
7         if link.bandwidth + cost > link.bandwidth_max:
8             return 0
9         elif link.bandwidth_max - cost < lowest:
10            lowest = link.bandwidth_max - cost
11    return lowest
12 # -- CODE SNIPPED -- #

```

Listing C.7: Report Algorithm Heuristic Bandwidth Snippet

```

1 # -- CODE SNIPPED -- #
2 # c is the current node
3 visited.append(c)
4 h_min = float('inf') # lowest found heuristic cost
5 n_min = None # node where h_min is found
6
7 candidates = nx.generators.ego_graph(
8     network.pn.graph,
9     c,
10    radius = 3
11 )

```



```

12
13 for n in candidates.nodes:
14     if n in visited:
15         continue
16     h = heuristic(
17         network,
18         c,
19         n,
20         packets,
21         service,
22         len(service) - len(remaining)
23     )
24     if h < h_min:
25         h_min = h
26         n_min = n
27 # -- CODE SNIPPED -- #

```

Listing C.8: Report Algorithm Node Traversal Snippet

```

1 # -- CODE SNIPPED -- #
2 for v in network_slice.vnfs:
3     if v.host not in counted_nodes:
4         counted_nodes.append(v.host)
5 # -- CODE SNIPPED -- #

```

Listing C.9: Report Algorithm Objective Node Counting Snippet

```

1 # -- CODE SNIPPED -- #
2 for e in network_slice.path:
3     for l in e.path:
4         if l not in tracked_links:
5             tracked_links.append(l)
6 # -- CODE SNIPPED -- #

```

Listing C.10: Report Algorithm Objective Links Used Snippet

```

1 # -- CODE SNIPPED -- #
2 for e in network_slice.path:
3     for l in e.path:
4         if l.bandwidth_max < slowest:
5             slowest = l.bandwidth_max
6 # -- CODE SNIPPED -- #

```

Listing C.11: Report Algorithm Objective Slowest Link Snippet

```

1 # -- CODE SNIPPED -- #
2 for i, v in enumerate(proposed_slice.vnfs):
3     if v.host == existing_slice.vnfs[i].host:
4         num += 1
5 # -- CODE SNIPPED -- #

```

Listing C.12: Report Algorithm Objective Spin Ups Snippet

```

1 # -- CODE SNIPPED -- #
2 for v in network_slice.vnfs:
3     path_time += v.function.latency

```

```

4     for e in network_slice.path:
5         path_time += e.latency
6 # -- CODE SNIPPED -- #

```

Listing C.13: Report Algorithm Objective Minimum Time Snippet

### Benchmark Implementation

```

1 # -- CODE SNIPPED -- #
2 for function in service:
3     placed = False
4     while not placed:
5         available = host.resource_max - host.resource
6         if available >= function.cost:
7             v = VNF(host, function, uuid.uuid4())
8             network.add_vnf(v)
9             s.add_vnf(v)
10            placed = True
11        else:
12            if host not in visited:
13                visited.append(host)
14            for edge in list(network.pn.graph.edges(host)):
15                if edge[0] in visited and edge[1] in visited:
16                    continue
17                elif edge[0] == host:
18                    host = edge[1]
19                    break
20                elif edge[1] == host:
21                    host = edge[0]
22                    break
23 create_edges(network, s) # s is the name of slice being generated
24 return s

```

Listing C.14: Generic Algorithm VNF Placement Snippet

```

1 slice_path = list[Edge] = []
2 for i in range(len(network_slice.vnfs) - 1):
3     # Uses Dijkstra's Algorithm to find shortest path between VNFs
4     sp = nx.shortest_path(
5         network.pn.graph,
6         network_slice.vnfs[i].host,
7         network_slice.vnfs[i + 1].host
8     )
9     # len(sp) = 1 means VNFs at same node, no edges
10    if len(sp) == 1:
11        continue
12    links: list[Link] = []
13    for j in range(len(sp) - 1):
14        links.append(network.pn.get_link(sp[j], sp[j + 1]))
15    e = Edge(
16        network_slice.vnfs[i],
17        network_slice.vnfs[i + 1],
18        links
19    )

```

```

20     network.add_edge(e)
21     slice_path.append(e)
22 network_slice.set_path(slice_path)

```

Listing C.15: Generic Algorithm Edge Creation Snippet

### C.3.2 Network Definitions

```

1 # Acts as a function template, instanced function is a VNF
2 class Function:
3     def __init__(self, latency: int, cost: int, name: str = None):
4         if name == None:
5             self.name = uuid.uuid4()
6         else:
7             self.name = name
8             self.latency = latency
9             self.cost = cost

```

Listing C.16: Function Definition

```

1 # Physical infrastructure that houses VNFs
2 class Node:
3     def __init__(self, resource_max: int, x: int, y: int, name: str = None):
4         if name == None:
5             self.name = uuid.uuid4()
6         else:
7             self.name = name
8             self.resource = 0
9             self.resource_max = resource_max
10
11         self.x = x
12         self.y = y
13
14         self.vnfs = list[VNF]

```

Listing C.17: Node Definition

```

1 # Connects two nodes
2 class Link:
3     def __init__(self, src: Node, dst: Node, latency: int, bandwidth_max: int):
4         self.src = src
5         self.dst = dst
6
7         self.latency = latency
8
9         self.bandwidth = 0
10        self.bandwidth_max = bandwidth_max
11
12        def allocate(self, value: int | float):
13            if self.bandwidth + value > self.bandwidth_max:
14                raise ValueError("Allocation would overload link")
15            self.bandwidth += value
16
17        def deallocate(self, value: int | float):
18            self.bandwidth -= value

```

```

19         if self.bandwidth < 0:
20             self.bandwidth = 0

```

Listing C.18: Link Definition

```

1 class PhysicalNetwork:
2     def __init__(self):
3         self.graph = nx.Graph()
4         self.nodes: list[Node] = []
5         self.links: list[Link] = []
6     class VirtualNetwork:
7     def __init__(self, pn: PhysicalNetwork):
8         self.pn = pn
9         self.graph = nx.Graph()
10        self.vnfs: list[VNF] = []
11        self.edges: list[Edge] = []
12
13    # Add VNF to virtual network
14    def add_vnf(self, vnf: VNF):
15        if vnf in self.vnfs:
16            raise ValueError(f"VNF {vnf.name} already in network")
17        if vnf.function.cost > vnf.host.resource_max - vnf.host.resource:
18            raise ValueError(f"Insufficient resources at {vnf.host.name} for VNF
19        {vnf.name}")
20        self.vnfs.append(vnf)
21        self.graph.add_node(vnf)
22        vnf.host.resource += vnf.function.cost
23
24    # Add edge (VNF-VNF) to virtual network
25    def add_edge(self, edge: Edge):
26        for e in self.edges:
27            if edge.src == e.src and edge.dst == e.dst:
28                raise ValueError(f"Edge between {edge.src.name} and {edge.dst.
29            name} already exists")
30            elif edge.src == e.dst and edge.dst == e.src:
31                raise ValueError(f"Edge between {edge.src.name} and {edge.dst.
32            name} already exists")
33            self.edges.append(edge)
34            self.graph.add_edge(
35                edge.src,
36                edge.dst,
37                weight = edge.latency
38            )
39
40    def remove_vnf(self, vnf: VNF):
41        for v in self.vnfs:
42            if v.name == vnf.name:
43                self.vnfs.remove(v)
44                break
45
46    def remove_edge(self, edge: Edge):
47        for e in self.edges:
48            if e.path == edge.path:
49                self.edges.remove(e)

```

```

47
48 # Add node to physical network
49 def add_node(self, node: Node):
50     if node in self.nodes:
51         raise ValueError(f"Node {node.name} already in network")
52     else:
53         self.nodes.append(node)
54         self.graph.add_node(node)
55
56 # Add link to physical network
57 def add_link(self, link: Link):
58     # Check existing links to see if connection already exists
59     duplicate = False
60     for l in self.links:
61         if l.src == link.src and l.dst == link.dst:
62             raise ValueError(f"Link between {l.src.name} and {l.dst.name}
already exists")
63         elif l.src == link.dst and l.dst == link.src:
64             raise ValueError(f"Link between {l.src.name} and {l.dst.name}
already exists")
65     self.links.append(link)
66     self.graph.add_edge(
67         link.src,
68         link.dst,
69         weight = link.latency,
70         bandwidth = link.bandwidth,
71         bandwidth_max = link.bandwidth_max
72     )
73
74 def get_link(self, src: Node, dst: Node) -> Link:
75     for l in self.links:
76         if l.src == src and l.dst == dst:
77             return l
78         elif l.src == dst and l.dst == src:
79             return l
80     raise ValueError(f"No link found between {src.name} and {dst.name}")

```

Listing C.19: PhysicalNetwork Definition

```

1 # Connects two VNFs
2 class Edge:
3     def __init__(self, src: VNF, dst: VNF, path: tuple[Link]):
4         x = src.host
5         y = dst.host
6
7         # Verifies path goes from src to dst
8         # Also counts total latency of traversal
9         latency = 0
10        for link in path:
11            if link.src == x:
12                x = link.dst
13                latency += link.latency
14            elif link.dst == x:
15                x = link.src

```

```

16         latency += link.latency
17
18     # If path doesn't end at destination
19     if x != y:
20         raise ValueError("Path does not link VNFs together")
21     else:
22         self.src = src
23         self.dst = dst
24         self.path = path
25         self.latency = latency

```

Listing C.20: Edge Definition

```

1 # Virtual Network Function
2 class VNF:
3     def __init__(self, host: Node, function: Function, name: str = None):
4         if name == None:
5             self.name = uuid.uuid4()
6         else:
7             self.name = name
8         self.host = host
9         self.function = function

```

Listing C.21: VNF Definition

```

1 class VirtualNetwork:
2     def __init__(self, pn: PhysicalNetwork):
3         self.pn = pn
4         self.graph = nx.Graph()
5         self.vnfs: list[VNF] = []
6         self.edges: list[Edge] = []
7
8     # Add VNF to virtual network
9     def add_vnf(self, vnf: VNF):
10         if vnf in self.vnfs:
11             raise ValueError(f"VNF {vnf.name} already in network")
12         if vnf.function.cost > vnf.host.resource_max - vnf.host.resource:
13             raise ValueError(f"Insufficient resources at {vnf.host.name} for VNF {vnf.name}")
14         self.vnfs.append(vnf)
15         self.graph.add_node(vnf)
16         vnf.host.resource += vnf.function.cost
17
18     # Add edge (VNF-VNF) to virtual network
19     def add_edge(self, edge: Edge):
20         for e in self.edges:
21             if edge.src == e.src and edge.dst == e.dst:
22                 raise ValueError(f"Edge between {edge.src.name} and {edge.dst.name} already exists")
23             elif edge.src == e.dst and edge.dst == e.src:
24                 raise ValueError(f"Edge between {edge.src.name} and {edge.dst.name} already exists")
25         self.edges.append(edge)
26         self.graph.add_edge(
27             edge.src,

```

```

28         edge.dst,
29         weight = edge.latency
30     )
31
32     def remove_vnf(self, vnf: VNF):
33         for v in self.vnfs:
34             if v.name == vnf.name:
35                 self.vnfs.remove(v)
36                 break
37
38     def remove_edge(self, edge: Edge):
39         for e in self.edges:
40             if e.path == edge.path:
41                 self.edges.remove(e)

```

Listing C.22: VirtualNetwork Definition

```

1 class Packet:
2     def __init__(self, slice: NetworkSlice, bandwidth: float, x: int, y: int):
3         self.x = x
4         self.y = y
5
6         self.bandwidth = bandwidth
7
8         init_latency: int = sqrt(
9             (slice.vnfs[0].host.x - x)**2 +
10            (slice.vnfs[0].host.y - y)**2
11        ) * settings.LATENCY_WEIGHT
12
13        self.latency = init_latency
14        for v in slice.vnfs:
15            self.latency += v.function.latency
16        for e in slice.path:
17            self.latency += e.latency

```

Listing C.23: Packet Definition

```

1 class NetworkSlice:
2     def __init__(self, service: Service, max_bandwidth: int | float, name: str =
    None):
3         if name == None:
4             self.name = uuid.uuid4()
5         else:
6             self.name = name
7         self.service = service
8         self.max_bandwidth = max_bandwidth
9         self.vnfs: list[VNF] = []
10        self.path: tuple[Edge] = ()
11
12        def add_vnf(self, vnf: VNF):
13            for v in self.vnfs:
14                if vnf.function == v.function:
15                    raise ValueError("VNF serving this function already present in
    slice")
16            if vnf.function not in self.service:

```

```

17         raise ValueError("Function not necessary as part of slice's service"
18     )
19     self.vnfs.append(vnf)
20
21     def set_path(self, path: tuple[Edge]):
22         self.path = path
23
24     def destroy(self, network: VirtualNetwork):
25         for v in self.vnfs:
26             v.host.resource -= v.function.cost
27             network.remove_vnf(v)
28         for e in self.path:
29             for l in e.path:
30                 l.deallocate(self.max_bandwidth)
31             network.remove_edge(e)
32     self.vnfs = []
33     self.path = []

```

Listing C.24: NetworkSlice Definition

```

1 class Chain:
2     def __init__(self, service: Service, name: str = None):
3         self.service = service
4         self.vnfs: list[VNF] = []
5         self.path: tuple[Edge] = ()
6
7     def add_vnf(self, vnf: VNF):
8         for v in self.vnfs:
9             if vnf.function == v.function:
10                 raise ValueError("VNF serving this function already present in
chain")
11             if vnf.function not in self.service:
12                 raise ValueError("Function not necessary as part of chain's service"
)
13         self.vnfs.append(vnf)
14
15     def set_path(self, path: tuple[Edge]):
16         self.path = path

```

Listing C.25: Chain Definition

### C.3.3 Solution Implementation Definitions

```

1 def init(network: VirtualNetwork, network_slice: NetworkSlice, packets: list[
Packet]) -> tuple[NetworkSlice, float]:
2     start = report_construction.starting(network, packets)
3     proposed = report_construction.generate_slice(network, start, network_slice.
service, network_slice.max_bandwidth, packets)
4
5     objective = report_objective.objective(network_slice, proposed)
6
7     return (proposed, objective)

```

Listing C.26: Report Algorithm Initialisation Function



```

1 # Find starting node for report algorithm
2 def starting(network: VirtualNetwork, packets: list[Packet]) -> Node:
3     # leafs: all nodes in physical network of degree 1
4     leafs: list[Node] = [ n for n in network.pn.graph.nodes if network.pn.graph.
        degree(n) == 1 ]
5
6     lowest = float('inf')
7     n = None
8     for leaf in leafs:
9         total = 0
10        for packet in packets:
11            # Distance is euclidean distance from packet to leaf
12            distance = sqrt(
13                (leaf.x - packet.x)**2 +
14                (leaf.y - packet.y)**2
15            )
16            total += distance * settings.LATENCY_WEIGHT
17        total = total / len(packets)
18        if total < lowest:
19            lowest = total
20            n = leaf
21
22    return n

```

Listing C.27: Report Algorithm Starting Node Selection Function

```

1 def generate_slice(network: VirtualNetwork, start: Node, service: Service,
    max_bandwidth: int | float, packets: list[Packet]) -> NetworkSlice:
2     c = start
3     remaining = [ s for s in service ]
4     proposed = NetworkSlice(service, max_bandwidth)
5
6     # Keep track of visited nodes to prevent infinite loops
7     visited = []
8     while len(remaining) != 0:
9         # Place VNFs in order
10        p = remaining[0]
11        available = c.resource_max - c.resource
12        if p.cost < available:
13            v = VNF(c, p)
14            network.add_vnf(v)
15            proposed.add_vnf(v)
16            remaining.pop(0)
17        else:
18            visited.append(c)
19            # h_min is lowest found heuristic cost, and n_min is the node which
20            it is found at
21            h_min = float('inf')
22            n_min = None
23
24        # Make subgraph of all nodes of 3 degree or less away from c
25        candidates = nx.generators.ego_graph(network.pn.graph, c, radius =
3)

```

```

26         for n in candidates.nodes:
27             if n in visited:
28                 continue
29             h = heuristic(network, c, n, packets, service, len(service) -
len(remaining))
30             if h < h_min:
31                 h_min = h
32                 n_min = n
33             if n_min is not None:
34                 c = n_min
35
36         # Create path of edges for proposed network slice
37         slice_path: list[Edge] = []
38         for i in range(len(proposed.vnfs) - 1):
39             if proposed.vnfs[i].host != proposed.vnfs[i + 1].host:
40                 sp = nx.shortest_path(
41                     network.pn.graph,
42                     proposed.vnfs[i].host,
43                     proposed.vnfs[i + 1].host
44                 )
45
46                 links: list[Link] = []
47                 for j in range(len(sp) - 1):
48                     links.append(network.pn.get_link(sp[j], sp[j + 1]))
49                     #network.pn.get_link(sp[j], sp[j + 1]).allocate(max_bandwidth)
50
51                 slice_path.append(Edge(
52                     proposed.vnfs[i],
53                     proposed.vnfs[i + 1],
54                     links
55                 ))
56
57         proposed.set_path(slice_path)
58         return proposed

```

Listing C.28: Report Algorithm Proposed Slice Generation Function