FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme / specialisation: Computer Science / Reliable and Secure Systems	Spring semester, 2023 Open
Author: Fredrik Netteland	Signature: Fredrik Netteland
Supervisor at UiS: Ali Gohar Co-supervisor: Gianfranco Nencioni	
Thesis title: Security-aware Resource Allocation for Space-Air-Ground Integrated Network Using Deep Reinforcement Learning	
Credits (ECTS): 30	
Keywords: • Network Function Virtualization • SAGIN-aware • Security-aware • Deep Reinforcement Learning	Pages: 116 + appendix: 15 Stavanger, *(June 13, 2023)*

University of Stavanger

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

Security-aware Resource Allocation for Space-Air-Ground Integrated Network Using Deep Reinforcement Learning

Master's Thesis in Computer Science
by

Fredrik Netteland

Internal Supervisors

Ali Gohar

Gianfranco Nencioni

June 13, 2023

Acknowledgements

I would like to sincerely thank my supervisors for the advice provided while writing this thesis. I would also like to personally thank Ali Gohar for his outstanding guidance and commitment to answering all questions while writing this thesis.

Abstract

A Space-Air-Ground Integrated Network (SAGIN) has been proposed to extend communication network service coverage to consumer-oriented and industrial sectors where communication network coverage is either limited or unavailable. To effectively use the space, air, and ground hardware resources, Network Function Virtualization (NFV) is introduced into SAGIN. NFV enables the deployment and management of services that are represented as Virtual Networks (VN) composed of Virtual Network Functions (VNF) onto the SAGIN hardware through hardware virtualization. This enables SAGIN to support services with distinct demands from both consumer-oriented and industrial sectors.

However, by introducing NFV into SAGIN, new security vulnerabilities arise. For instance, if a malicious entity gains access to the virtualized hardware, all services utilizing the hardware are exposed to attack.

When deploying a VN onto the SAGIN hardware, also known as the Substrate Network (SN), it must be decided which SN Node (SNN) should host each VN Node (VNN) and which SN Links (SNL) should host each VN Link (VNL), also known as the Virtual Network Embedding (VNE) problem. This thesis proposes a solution to VNE in SAGIN using Deep Reinforcement Learning (DRL) while accounting for the security concerns related to NFV. To our knowledge, this has yet to be explored by other works.

We compare our solution with the well-known Global Resource Capacity (GRC) solution strategy using the acceptance rate, revenue, cost, and revenue-to-cost metrics. Our DRL-based solution strategy shows competitive performance in all metrics.

Contents

0.1 Commonly used Acronyms

Acronym	**Definition**
IoT	Internet of Things
ITS	Intelligent Transportation Systems
BS	Base Station
SAGIN	Space-Air-Ground Integrated Network
NF	Network Function
NFV	Network Function Virtualization
VNF	Virtual Network Function
VM	Virtual Machine
VNF-FG	VNF-Forwarding Graph
NFVI	NFV Infrastructure
VNF-FGE	VNF-FG Embedding
VNFP	VNF Placement
TR	Traffic Routing
InP	Infrastructure Provider
SP	Service Provider
VNE	Virtual Network Embedding
SN	Substrate Network
SNN	Substrate Network Node
SNL	Substrate Network Link
IDL	Inted-Domain Link
VNR	Virtual Network Request
VN	Virtual Network
VNN	Virtual Network Node
VNL	Virtual Network Link
DRL	Deep Reinforcement Learning
MDP	Markov Decision Process
PPO	Proximal Policy Optimization
CSPF	Constrained Shortest Path First
LTR	Long Term Average Revenue
LTC	Long Term Average Cost
LRC	Long Term Revenue Cost Ratio
ACR	Acceptance Rate
GRC	Global Resource Capacity
DSP	Dijkstra Shortest Path
SB3	Stable-Baselines3
NN	Neural Network
DSTPP	Distance from Previous Placement

Table 1: Commonly used acronyms.

Chapter 1

Introduction

The need for secure and ubiquitous communication services is witnessing a significant surge in demand. This heightened demand can be explained by the rapid growth and proliferation of various applications, encompassing both consumer-oriented and industrial sectors. Specifically, this trend is driven by technologies such as the Internet of Things (IoT) [1].

IoT interconnectivity extends beyond traditional computing devices, encompassing a wide range of objects, appliances, and systems, such as Intelligent Transportation Systems (ITS) and home appliances [1][2]. With IoT applications becoming increasingly prevalent in homes, industries, and cities, there is a growing demand for ubiquitous and secure communication services to support these interconnected ecosystems.

However, in many scenarios, the ground-based wired and wireless infrastructure is falling behind in providing consumer and industrial-grade service coverage in challenging environments, such as rural cities, ocean-based industries, and air transport [1][3]. For instance, ocean-based industries such as oil platforms require expensive subsea cables to provide network service coverage. Additionally, fixed cable communication is entirely infeasible for mobile ocean-based applications such as shipping. As an alternative to fixed cable communication, wireless Base Stations (BS) communication can be used. For instance, in a densely populated area, wireless BSs can provide service coverage to a large customer base making the BSs economically feasible. However, in a rural area, the customer base might be small and dispersed over a wide area, thus making BS coverage financially unviable. Therefore, we need a financially viable method of providing service coverage in these challenging areas. For this purpose, a Space-Air-Ground Integrated Network (SAGIN) has been proposed as a solution [1][2][4].

1.1 SAGIN

SAGIN aims to solve the mentioned issues by merging space, air, and ground-based networks into a single unified network, capable of far greater coverage than ground-based networks [5]. By leveraging this expanded coverage, SAGIN can support services for remote cities, shipping, air transport, and much more, where traditional ground-based networks are lacking.

To visualize SAGIN, we use figure 1.1. The figure shows the networks that are part of SAGIN, usually referred to as the space, air, and ground domains, and the capabilities of hardware components in each domain [2]. Firstly, the SAGIN ground domain consists of standard fixed network hardware nodes connected by wired or wireless BSs links. As we know, the ground domain alone is insufficient to support the modern service requirements of both consumer-oriented and industrial sectors. As such, SAGIN also utilizes the air domain. In the air domain, aerial vehicles include high-speed networking equipment, allowing air domain hardware nodes to forward and process communication network traffic for consumer-oriented and industrial sectors [5]. However, providing service coverage to all regions is infeasible with the air domain alone because many aerial vehicles are required and their inherent need to take off and land periodically. As such, SAGIN also considers the space domain, which adds networking infrastructure in the form of satellite hardware nodes. If placed in a stable orbit, satellites will provide long-lasting massive service coverage [5].

Satellites have very different capabilities than infrastructure in other domains, mainly regarding communication delay, range, and hardware node capabilities such as processing, memory, and storage [5]. In terms of communication delay and range, Low Earth Orbit (LEO) satellites have limited coverage[1] due to the earth's curvature but give lower communication delay than Medium Earth Orbit (MEO) satellites due to their closer proximity to Earth [5]. For example, the distance from a ground domain BS to an LEO satellite is a minimum of 160km, while MEO is at a minimum of 2000km [5]. Hence, MEO experiences increased delay. Moreover, the increased delay is even more prevalent for Geostationary Orbit (GEO) satellites placed even farther from the BS than MEO satellites at 35786km [5].

Regarding satellite hardware, satellite launches use rockets which incur huge expenses. The more hardware added to a satellite, the larger the weight, meaning a heavier lift rocket is used, incurring even more significant expenses. As such, satellites typically have limited hardware capabilities relative to the air and ground domains.

[1]Limited coverage relative to MEO and GEO.

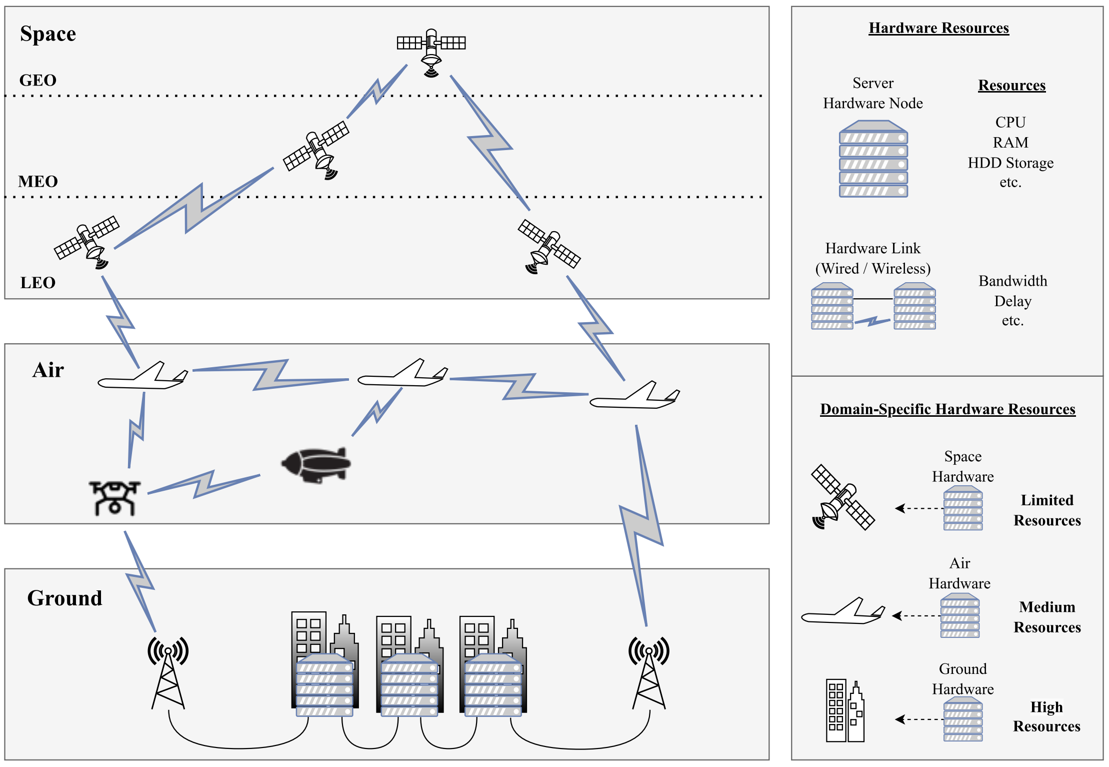

Figure 1.1: Visualization of a SAGIN network.

Considering these factors, it is crucial to recognize that satellite networks offer unique advantages and face specific limitations when compared to air and ground-based networks. Understanding the disparities in communication delay, coverage, and hardware capabilities is vital in leveraging the benefits of satellite communication and effectively integrating satellite networks within the broader communication infrastructure in SAGIN.

1.1.1 NFV

When communications network traffic is forwarded through the hardware nodes and links in SAGIN, we say each hardware node performs some "processing" on the traffic, also known as performing a Network Function (NF) [6]. As an example, consider a network user requesting a web page. In this example, two necessary NFs could be a DNS NF for domain name lookup and a routing NF for forwarding the communication network traffic[2] [7].

Furthermore, when NFs process communications network traffic, this is typically done in an ordered fashion, which is referred to as providing a service [7]. Hence, to provide a service in SAGIN, NFs can be deployed at space, air, and ground domain hardware. Moreover, depending on the order of NF processing required by a service, the service

[2]Many more NFs exist [7].

traffic can be forwarded through multiple space, air, and ground hardware nodes, where each hardware node performs an NF required by the service [2][7].

Naturally, when an NF performs some processing for a service, the NF requires sufficient processing, memory, and storage resources to perform the processing, which also varies in scale from one NF to another [6][7]. For instance, a firewall NF might require more computing resources than a DHCP NF. Hence, different services generally have heterogeneous resource requirements, and it is often necessary to scale the available SAGIN hardware resources to accommodate new services or changes to services.

However, scaling and managing NF resources effectively in a traditional network is infeasible due to a lack of management control and each NF utilizing dedicated hardware [7][8]. This results in inefficient use of NF resources. For example, an NF might not be fully utilized, thus wasting resources. Alternatively, the NF might have insufficient resources to support a new service, leading to difficulties deploying the new service.

However, the Network Function Virtualization (NFV) standard proposes a solution to these problems [9]. In NFV, NFs no longer utilize dedicated hardware. Instead, each NF is represented as a Virtual NF (VNF), which is a software representation of the NF. Moreover, by incorporating a virtualization layer in general-purpose hardware, as shown in figure 1.2, VNFs can be hosted on any hardware node [6][8]. This opens the possibility of dynamically scaling VNF resources when required and allows placing VNFs when and whenever required [7].

NFV also provides some obvious advantages for SAGIN. For example, consider the space domain of SAGIN. Before NFV, placing multiple NFs on a satellite would be infeasible due to the weight of dedicated hardware. However, NFV allows any VNFs to be deployed to general computing hardware on the satellite such that any NF can be performed.

NFV Operation

We now know that services such as web browsing require an ordering of multiple VNFs. As such, a common method of representing a service is to use a Directed Acyclic Graph (DAG), more often known as a VNF-Forwarding Graph (VNF-FG) in NFV terminology [6][8][3].

We visualize the VNF-FG of a service in figure 1.3. The figure depicts VNFs as circles with hardware requirements such as processing, memory, and storage. Furthermore, the ordering of VNFs is reflected through the chain of VNFs, where a virtual directed link connects each consecutive VNF. The directions of the arrows indicate that traffic must

[3]Also known as a Service Function Chain (SFC)[6].

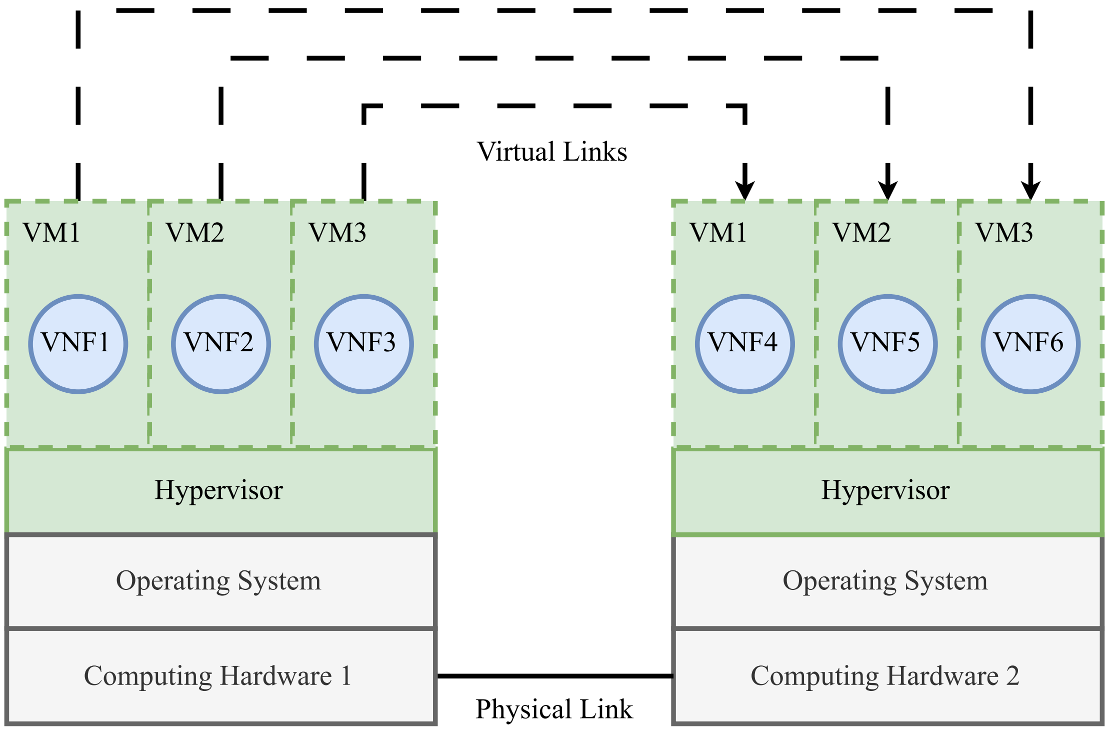

Figure 1.2: Server hardware in figure 1.1 now runs multiple VNFs on one or more VMs.

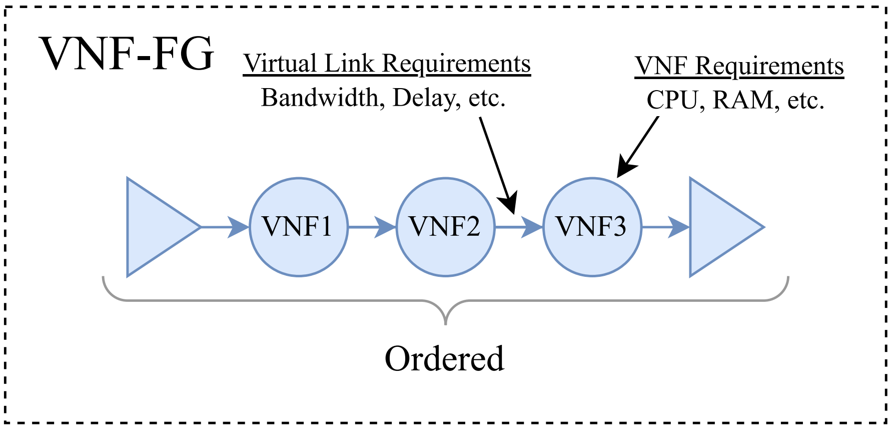

Figure 1.3: A visual representation of a VNF-FG.

flow from the left *ingress*, represented as a triangle, towards the right and out the *egress* [9]. Once the traffic exits through the egress, all processing required by the service is completed.

A VNF-FG can be considered a *blueprint* for a service because it defines every service requirement, such as the number of VNFs, types of VNFs, and resource demands. However, given that a VNF-FG is only a blueprint, we also need a method for realizing the VNF-FG on physical hardware, also known as the NFV Infrastructure (NFVI). This process is called the VNF-FG Embedding (VNF-FGE) problem [6].

VNF-FGE

The first challenge of VNF-FGE is related to the VNF Placement (VNFP). The goal of VNFP is to find NFVI hardware nodes that have sufficient available resources for hosting VNFs [6]. The available resources of NFVI hardware nodes are jointly decided based on the capabilities of the hardware and already hosted VNFs. For example, consider an NFVI hardware node with high processing, memory, and storage resources. If the node is already hosting multiple VNFs, then the node likely has low remaining resources. Hosting additional VNFs onto the same node will likely overburden the hardware, thus reducing the performance of the hosted VNFs. Hence, it is essential to account for already hosted VNFs during VNFP.

The next step of VNF-FGE is to find appropriate NFVI hardware links that can be used to represent each of the VNF-FG directed virtual links. This is referred to as the Traffic Routing (TR) problem [6][8][10].

During TR, each consecutive virtual link within the VNF-FG must be realized using one or more NFVI links with sufficient bandwidth resources and delay [10]. For example, consider a VNF-FG with three VNFs connected by two virtual links, as shown in figure 1.3. Next, consider each consecutive VNN to be placed in the space, air, and ground SAGIN domains, respectively[4]. To solve TR in this scenario, a path of ground-to-air NFVI links must be found to represent the "VNF1-VNF2" virtual link[5]. Next, a path of air-to-space NFVI links must be found to represent the "VNF2-VNF3" virtual link. If all NFVI links part of each path satisfy the demands of the virtual link, then a valid TR solution has been found [10].

Once the VNFP and TR problems are solved, the service can be hosted on the NFVI hardware using the solutions to VNFP and TR, completing VNF-FGE [6]. However, note that solving VNF-FGE is only one of several tasks which must be performed to provide a service. For example, once a VNF-FG is embedded, it must be monitored, maintained, and eventually discarded [7]. Other tasks involve the management of physical NFVI hardware, such as updating operating system software and virtualization software management [9]. The NFV architecture incorporates all these tasks into a general NFV management component known as Management and Orchestration (MANO) [9].

Due to the direct connection between the physical hardware and the MANO, we say that the owner of the NFVI hardware incorporates the MANO component, which is known as the Infrastructure Provider (InP) [9][11]. Furthermore, Service Providers (SP) want to

[4]VNFP is already solved.

[5]Although we consider the placement of ingress and egress in this thesis, we exclude them from the example for simplicity.

provide services to their tenants within the consumer-oriented and industrial sectors. As such, SPs pay the InPs to implement their services [8]. This interaction between SPs and InPs is generally represented as *service requests* sent by SPs that includes the requested VNF-FG [7][11].

1.1.2 VNE

All services should typically be well-defined to know the exact purpose of the service. For instance, a service should concretely state which VNFs are required and their exact ordering. However, as a means to represent the wide range of possible VNFs and services, this thesis instead considers a general representation of services from the viewpoint of the Virtual Network Embedding (VNE) problem [12].

In VNE, we refer to services as Virtual Networks (VN) [12]. A VN uses a general representation of the VNFs, such that each VNF can be any arbitrary VNF[6] [6]. We refer to these general VNFs and their connections as VN Nodes (VNN) and VN links (VNL) [12]. Furthermore, due to services being represented as VNs, the terminology of the *service request* changes to *VN Request* (VNR) [12].

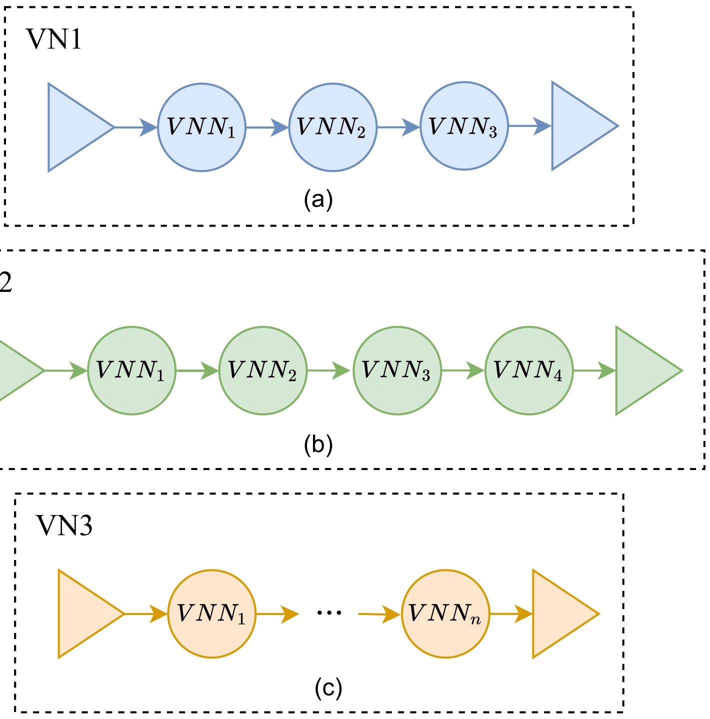

Figure 1.4: A visual representation of three VNs.

[6]i.e., each VNF provides an unknown function.

Since each VNN represents an unknown NF, we are not aware of the specific hardware requirements for each VNN. Therefore, the arrangement or ordering of the VNNs can be arbitrary.

To illustrate this, let's consider the VN1 shown in figure 1.4. In this case, the VNNs within VN1 are arranged randomly. Since we don't have prior knowledge of the specific NF associated with each VNN, there are no predetermined rules dictating their order. Therefore, the VNNs can be arranged in any order, as their NFs are unknown.

Furthermore, due to the general representation of services, the specific number of VNNs required for the service is also arbitrary. Figure 1.4 visualizes this point. Firstly, "VN1" arbitrarily has three VNNs, which represent some unknown "service A". Next, "VN2" arbitrarily includes an additional VNN, representing some other "service B". Lastly, "VN3" shows how a service could contain any number of VNNs.

Generally, works on VNE do not consider the ordering of VNNs and hence include no directed links [6][12]. However, since we define our VN as providing an arbitrary service where ordering is important, we consider VNE in terms of both the VNFP and TR problems.

Lastly, we provide Figure 1.5 as an illustration of VNE. Figure 1.5 depicts a potential embedding solution of a VNR. The VNR includes VNNs with a processing resource demand and VNLs with a bandwidth resource demand[7]. Next, Figure 1.5 includes the Substrate Network (SN), which is the alternative terminology for NFVI used in VNE [12].

[7]We limit the resources considered in the example to processing and bandwidth for simplicity.

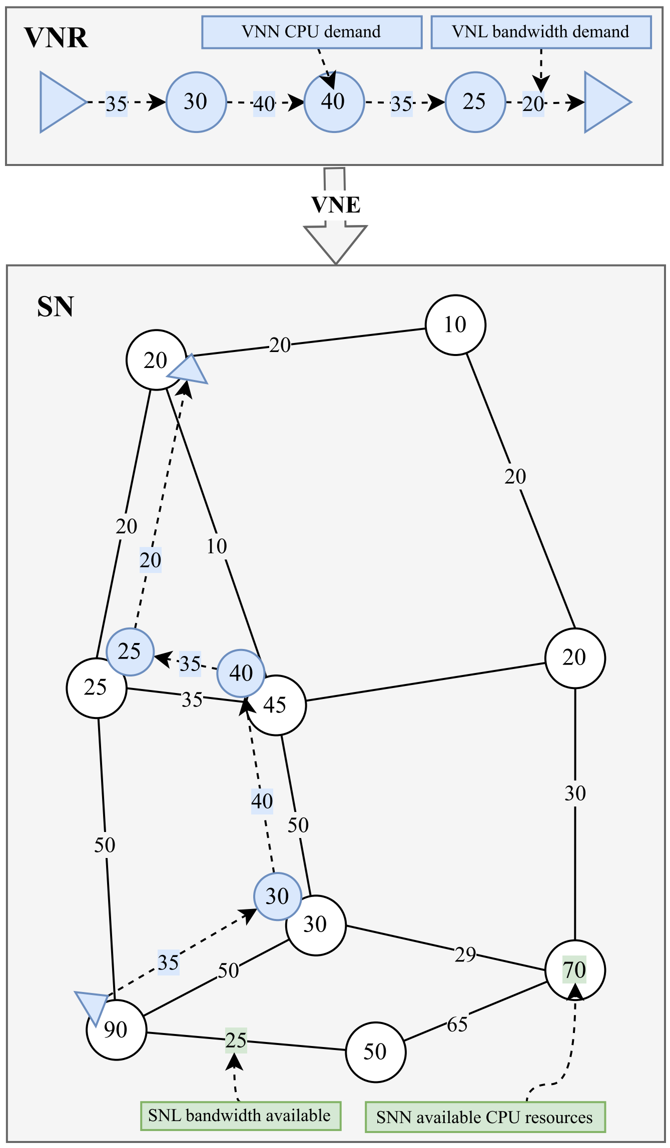

Figure 1.5: A simplified visual representation of VNE.

The SN in Figure 1.5 is represented as a graph structure, where SNNs and SNL represent physical hardware in SAGIN's space, air, or ground domains. Each VNN is placed onto an SNN with equal or greater available resources than the demands made by the VNN. Next, based on these placements, a path of SNLs is found for each VNL. Each SNL satisfies the bandwidth resource demands of the VNL.

Online and Offline VNE

Solving the VNE problem aims to find an efficient solution strategy viable for deployment at a real-world InP. For instance, the VNE solution strategy should embed as many VNRs as possible while using the least amount of InP resources, such that the InP gains maximum profits. Moreover, the time spent calculating the VNE solution should be as low as possible, which allows the InP to handle high rates of received VNRs [12].

However, even if a VNE solution strategy is able to prove high effectiveness, the value of the VNE solution strategy decreases if a real-world scenario is not considered. Hence, we highlight three important considerations for representing a real-world scenario, starting with the *online* or *offline* approaches [12].

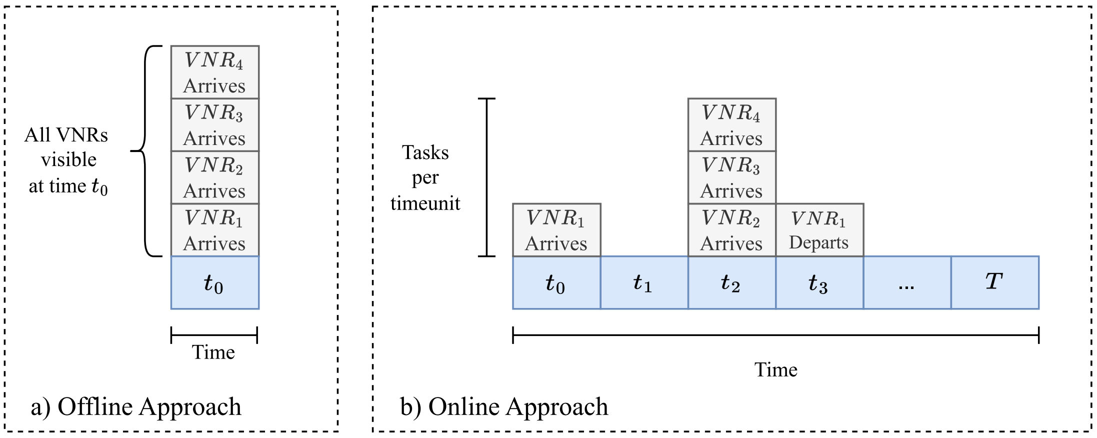

Figure 1.6: Visualization of online and offline approach.

Figure 1.6 (a) depicts the offline approach with four VNR *arrivals* and one *departure* represented as rectangles. An arrival represents a time when a VNR must be embedded into the SN, and a departure represents when an embedded VNR should be removed from the SN [12].

In the case of the offline approach shown in Figure (a), the InP has all VNRs available and may choose to embed in any order. Given that the InP knows the resource requirements of all VNRs, the InP can calculate the optimal ordering of embeddings such that as many VNRs are embedded as possible [12].

In contrast, Figure (b) shows the online approach, where the ordering is decided by a specific arrival time. Hence, the InP can no longer find an optimal embedding solution for all VNR since the InP is unaware of arrivals in subsequent time slots. This accurately depicts how a real-world SP would send VNRs to the InP at different times [12].

Dynamic and Static VNE

Secondly, we note the distinction between *dynamic* and *static* VNE approaches [12].

The *dynamic* approach simulates how computing hardware in a real network may be added, removed, or upgraded during the network's lifetime. This can be represented as dynamically adding, removing, or modifying SNNs, SNLs, and their resources between VNR embeddings. However, if these dynamic events are impossible, we say the VNE problem is static [12].

To provide an example of a dynamic SN, we use figure 1.7. Figure 1.7 visualizes a removal event, where an SNN is removed from the SN.

At a discrete time t_0, "VNR1" is already embedded into the SN. However, at t_1, the dynamic nature of the SN results in the removal of an SNN hosting VNN_3. Hence, VNN_3 no longer has a valid placement, and the InP must re-embed "VNR1".

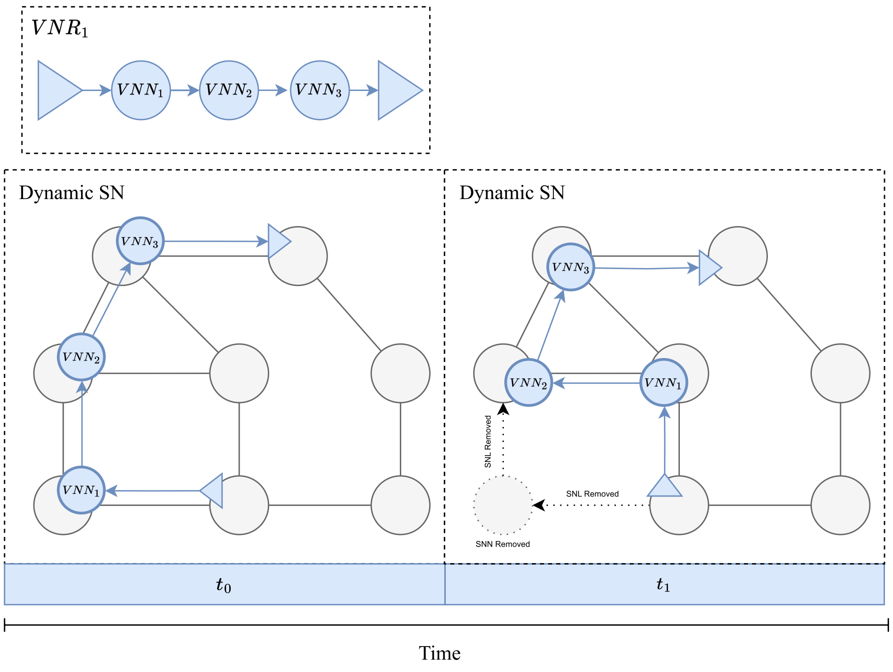

Figure 1.7: Visualization of the possible dynamic events in a VNE scenario.

Distributed and Centralized VNE

Based on our earlier description of the SN, it is owned by a single InP. However, real-world InPs typically don't have a monopoly on SN resources and the ability to provide services.

Instead, we can represent a more realistic VNE scenario by introducing the *distributed* approach, where multiple SPs send VNRs to multiple InPs, each performing VNE on their own SN [12].

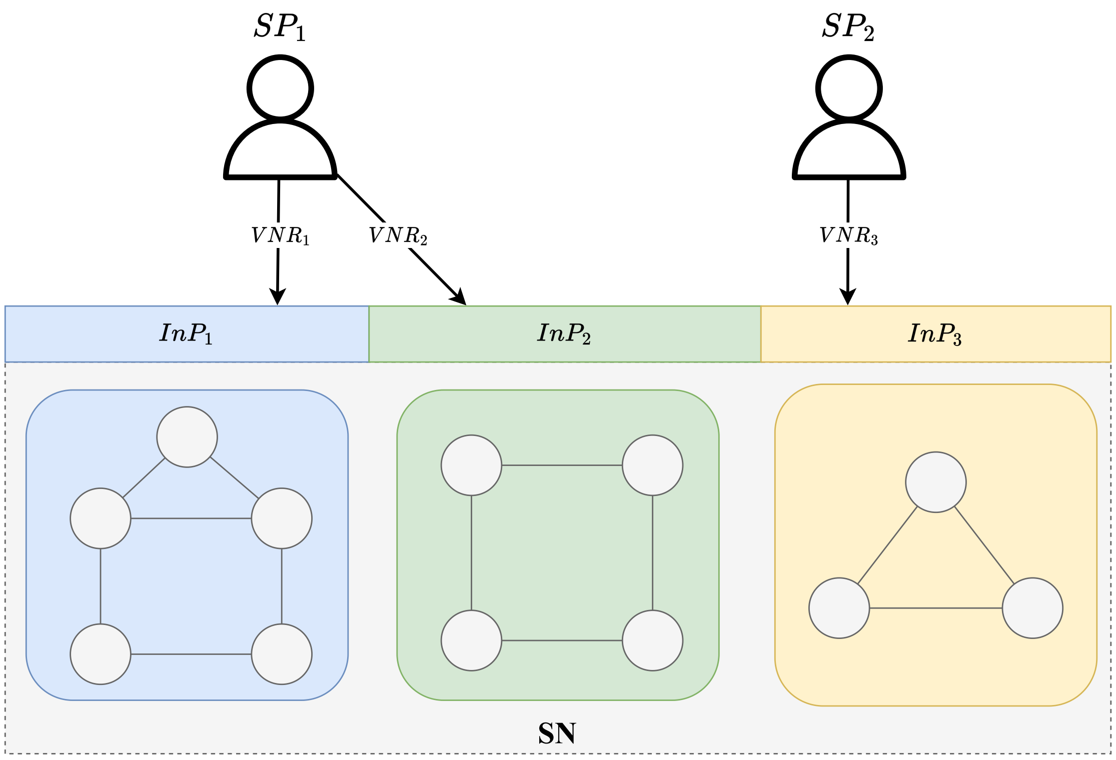

Figure 1.8: Visualization of a distributed InPs. Small SN graphs are used for simplicity.

Figure 1.8 visualizes the distributed approach and arbitrarily includes two SPs and three InPs, although any number of SPs and InPs could be included. The figure assumes each InP is entirely separate from other InPs, and each InP must solve VNE separately. However, as noted in [12], in some even more realistic scenarios, InPs could work together to decide where each VNR should be embedded in order to achieve mutual benefits [12].

Approach	**Description**
Online	VNRs arrive at discrete time steps.
Offline	VNRs have no arrival time
Dynamic	SN experiences changes over time
Static	SN experiences no changes
Distributed	SN is owned by multiple InPs
Centralized	Single InP has a monopoly on SN resources

Table 1.1: Overview over the VNE approaches.

1.2 Security

NFV brings several benefits through its virtualization of NFs but, unfortunately, also brings some security concerns [9].

Consider some customer premises equipment part of a business. Before NFV, the CPE would include non-virtualized NFs managed and purchased by the business [7]. As such, the business had physical control over the hardware components, making them responsible for the security of their network.

However, with the introduction of NFV, the NFs run on virtual machines located at the InP's premises. The InP has physical access to the SN hardware and is therefore responsible for providing security to the hardware hosting the VNNs. However, if the InP is malicious or has failed to provide security, attacks could be performed directly on the hardware [13][14].

Furthermore, InPs could also be attacked by its hosted SP tenants or other outside entities such that malicious actors gain access to SN hardware. In such a scenario, the malicious actor would have direct access to VNNs of multiple VNs [13].

Due to these new kinds of attack vectors added by NFV, SP tenants rely on safety measures provided by the InP to mitigate or prevent attacks. Depending on the kinds of attacks each SP tenant needs protection from, the SP can equip VNRs with *security demands*. Based on these demands, the InP can provide security by embedding VNNs onto SNNs that implement protective measures[8] [13][15].

1.3 Reinforcement Learning

This thesis utilizes Deep Reinforcement Learning (DRL) to make VNN placement decisions when solving VNE. The choice of DRL comes down to its ability to solve complicated decision-making problems effectively, such as optimizing moves in complex video games like Dota 2 [16]. Similarly, we also face a complex task in this thesis, namely VNE, which multiple papers state as NP-hard [2][4]. Therefore, we require a suitable solution, such as DRL, to solve SAGIN VNE effectively. Additionally, DRL has already proven its capabilities through existing works such as [2], [4], or [17], providing further merits for choosing a DRL solution.

The following sections give an introduction to DRL concepts. We list all notations in table 1.2.

[8]A more thorough introduction to security is given when reviewing the related works in section 2.2.1.

Notation	Description
t	Discrete time step. $t \geq 0$.
$\mathcal{S}$	Set of possible states in the environment.
S_t	A particular state at time step t of the environment.
s	A particular state $s \in S_t$.
$\mathcal{A}$	Set of possible actions the agent might take.
A_t	A particular action at time step t.
a	A particular action at $a \in A_t$.
$\mathcal{R}$	Reward function
R_t	Reward received at time step t, represented as a random variable.
R_{t+1}	Immediate reward when leaving state at t.
r	A particular reward given at R_t. $r \in R_t$.
$\pi(a\|s)$	The policy for choosing action a at state s.
γ	Discount factor used when finding return.
G_t	Discounted Return.
$P(s', r\|s, a)$	State transition probability from current state and action.
$V_\pi(s)$	State value. Expected reward starting from $s \in S_t$.
$Q_\pi(s, a)$	Action value. Expected reward starting from $a \in A_t$.
$\pi_\theta(a\|s)$	Policy represented as a neural network using weights and biases θ.
$J(\theta)$	Gives a score to θ in policy network $\pi_\theta(a\|s)$.
$d_\pi(s)$	On-policy stationary distribution given policy π

Table 1.2: Notations used for describing RL concepts.

1.3.1 Environment

We say the VNE problem can be viewed as a *Markov Decision Process* (MDP) [17]. Generally speaking, an MDP represents an iterative approach to solving a decision-making problem where multiple possible solutions are explored. The aim is to explore enough solutions such that the MDP can reliably choose the optimal solution in any variation of the problem [18].

Firstly, in an MDP, we consider an *environment*. The environment represents a virtual space that contains information related to a given problem [18]. For instance, in the VNE problem, the environment includes information about the SN, VNRs, and their resources [17]. Or in other words, all the information required to represent the VNE problem.

Whenever we perform a VNN embedding in the environment, we say we update the environment *state*, going from a particular state $s \in S_t$ at discrete time t to a new state $s' \in S_{t+1}$ at time $t+1$. This results in a change in the environment [18]. For example, if a VNN is embedded during VNE, the environment now includes an additional VNN that changes the state of the available SN resources. State changes such as this occur once every *time step* $t \rightarrow t+1$ until the last *terminal* state is reached in time step T [18].

Any given state s can transition to a large set of possible states, part of the state space $\mathcal{S}$ [18]. For instance, a VNN has several possible embeddings in the SN, where each possible embedding results in a unique state.

Furthermore, we know from our previous discussions on VNE that embeddings only rely on the *currently* available resources in the SN. Hence, the SN resources of *earlier* states S_{t-1} are not considered when embedding a VNN in the current state S_t. When this property holds, the *Markov Property* is satisfied, which is a requirement of MDPs [17][18].

1.3.2 Agent

We now understand the concept of an MDP environment and how it transitions through a series of states. However, if we want the series of states to end up at an optimal embedding solution, we must carefully decide on each state transition. For example, we would like to avoid a state where an embedded VNN restricts future placements or where the SN resources are not effectively utilized.

To tackle these issues, we require an intelligent system that is capable of deciding the best series of state transitions. Such a system is referred to as a DRL *agent* [18].

An agent is responsible for triggering each state transition through its *actions* $a \in A_t$, which are made based on the current environment state [18]. Based on the current state, the agent gets an overview of the remaining SN resources, part of the state, and can make an intelligent placement action to utilize the SN resources efficiently. Once the agent has made its action, the environment transitions to the new state based on a *state transition probability* dependent on the current state and action [18]:

$$P(s'|s,a) = P\{S_{t+1} = s'|S_t = s, A_t = a\}$$

Finding the best action in any state is typically challenging. For instance, there are many possible actions $a \in A_t$ for any state $s \in S_t$ that must be considered. Secondly, even if an action initially triggers a beneficial state change $s' \in S_{t+1}$, the agent might only have unbeneficial actions in subsequent states. As such, the agent must also be aware of possible future disadvantages of choosing an action. This must be learned through training the agent's *policy* [18].

The past experiences required to make beneficial actions are stored in what's referred to as a policy π [18]:

$$\pi(a|s)$$

The policy is often represented as a Neural Network (NN), which is also known as a policy network [17]. The agent inputs the current state $s \in S_t$ and receives the probability of choosing $a \in A_t$ among all other actions available in A_t [17]. Alternatively, the policy can be set to output a fixed action per state. These two policy types are called *stochastic* and *deterministic* policies, respectively [18].

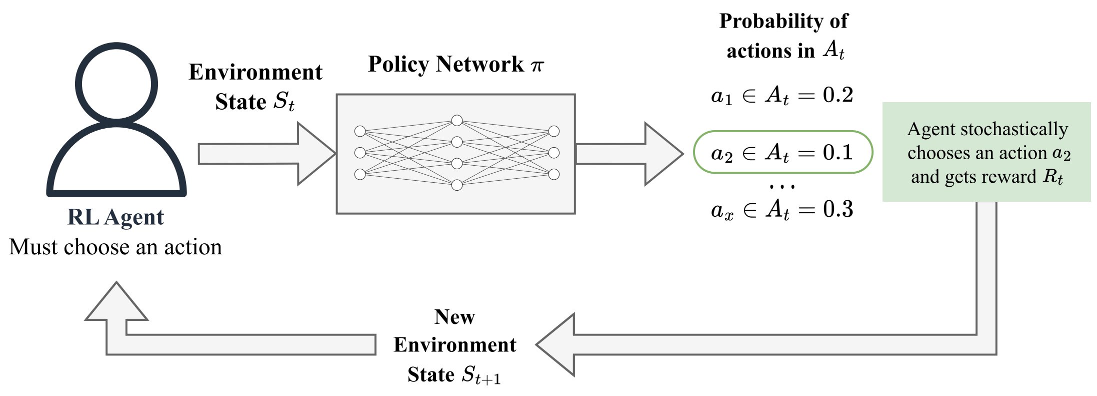

Figure 1.9: Visualization of how an agent utilizes a stochastic policy in order to make its actions.

Based on these descriptions, we see that a *stochastic policy* involves *exploring* other actions since the choice of action, determined by the policy, is stochastic. In contrast, a *deterministic policy* always *exploits* the action it views as best. Therefore, the deterministic policy will never find a better route if the agent has a false view of the optimal solution. This is the DRL *exploitation vs. exploration* problem [18][19]. We will only consider stochastic policies in the following sections compared to deterministic policies.

1.3.3 Reward

Initially, an agent does not know which actions are best due to its policy being *untrained*. Hence, we must *train* the policy network to increase the probability of good actions[9]. However, we cannot adjust the policy network without a metric for state transitions and actions, i.e., whether the agent performed a good or a bad embedding. To solve this issue, DRL includes the concept of *rewards* [18].

Whenever the agent chooses an action, a reward $r \in R_{t+1}$ is given when moving from a state $S_t \rightarrow S_{t+1}$. This is referred to as an *immediate reward*. Immediate rewards accumulate throughout all state transitions until reaching a terminal state S_T [19]. This

[9]Actions that use SN resources effectively.

is known as a *trajectory*, and the accumulated rewards during the trajectory discounted by $\gamma \in [0,1]$ are known as the *return* G_t [18][19].

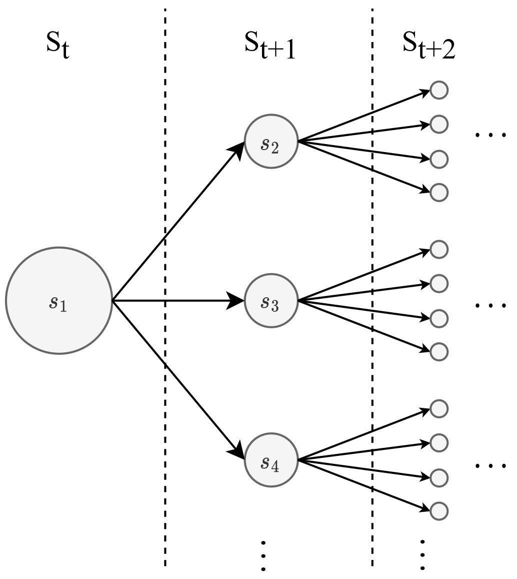

Figure 1.10: Figure showing the huge range of possible states.

Based on the concept of return, the agent can now quantify the disadvantage of making an earlier action. To illustrate this point, imagine the agent has the following trajectory[10]:

$$s_1 \in S_t \rightarrow s_2 \in S_{t+1} \rightarrow s_3 \in S_{t+2} \rightarrow s_4 \in S_{t+3} \rightarrow \dots$$

If the agent wants to know whether its action that triggered $s_2 \in S_{t+1}$ was a good choice, then the agent can find the trajectory return starting from state s_1, indicating whether the future state transitions were bad.

However, as figure 1.10 illustrates, this is only one of several possible trajectories starting from s. As such, a more accurate return value can be found by combining all these trajectory returns into an average value. This concept of calculating returns for all trajectories from state s is known as the value of the state $V_\pi(s)$, or *state-value*.

Alternatively, we can limit the possible actions A_t available in a given state $s \in S_t$ to a single action $a \in A_t$, giving us an *action-value* $Q_\pi(s,a)$ instead [19].

The state and action values are typically not calculated directly, they are instead estimated using several trajectories. Based on this, the state value can be defined as [19]:

$$V_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] \tag{1.1}$$

[10]For simplicity, actions, and rewards are not shown.

And the action value as [19]:

$$Q_\pi\left(s,a\right) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{1.2}$$

Given a state $s \in S_t$, we can now find two types of values for that state. The *state-value*, which is based on the estimated return from multiple trajectories from a given starting state, and the *action-value*, which starts its trajectories with a particular action available in the state. As such, the *state-value* is a more general value than *action-value*.

Based on this, the *state-value* will capture both good and bad trajectories from all actions. We can use this value to give an estimate of whether a particular action a generally outperforms any other action in state s. This is known as the *advantage* and is defined as follows [18]:

$$\hat{A}_\pi(s,a) = Q_\pi(s,a) - V_\pi(s) \tag{1.3}$$

1.3.4 Bellman Equations

Above, we mentioned how the V_π and Q_π values must be estimated based on several possible trajectories. Since most problems will include a huge number of possible trajectories, this estimation is a slow process.

However, one important property provides a solution: If we are calculating V_π or Q_π for a state S_t, then we can re-use previously calculated V_π or Q_π values to find the value of S_t. The value of S_t can be calculated directly from these existing values and the immediate rewards. This property is captured through the Bellman equations [18][19]:

$$\begin{aligned}
&\text{Bellman State-Value Equations}\\
V_\pi(s) &= E_\pi[G_t|S_t = s]\\
&= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots|S_t = s]\\
&= \mathbb{E}_\pi[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \ldots)|S_t = s]\\
&= \mathbb{E}_\pi[R_{t+1} + \gamma(G_{t+1})|S_t = s]\\
&= \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1})|S_t = s]
\end{aligned} \tag{1.4}$$

$$\begin{aligned}
&\text{Bellman Action-Value Equations} \\
Q_\pi(s,a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\
&\dots \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma(G_{t+1})|S_t = s, S_t = a] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1})|S_t = s, A_t = a]
\end{aligned} \tag{1.5}$$

Using the Bellman equations, the agent is now able to rely on previously calculated values when performing value estimation, significantly improving estimation time in equation 1.4.

Note that the state value $V_\pi(s)$ is based on trajectories starting from any action a with policy π. Since the action value $Q_\pi(s,a)$ defines the value of each action, we can rewrite $V_\pi(s)$ as the sum of all action values multiplied by its policy probability [19]:

$$V_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s)\mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1})|S_t = s, A_t = a] \tag{1.6}$$

$$V_\pi(s) = \sum_{a\in A} \pi(a|s)Q_\pi(s,a) \tag{1.7}$$

1.3.5 Training

When we train the DRL agent, we run several *epochs*. During an epoch, multiple trajectories are started from random states until reaching a final state. The time step of the final state is referred to as the *horizon* of the trajectory and is specified as a parameter in the epoch. After each epoch finishes, the returns are compared and used to adjust the policy [20]. The connection between epoch, trajectories, and the horizon is seen in Figure 1.11.

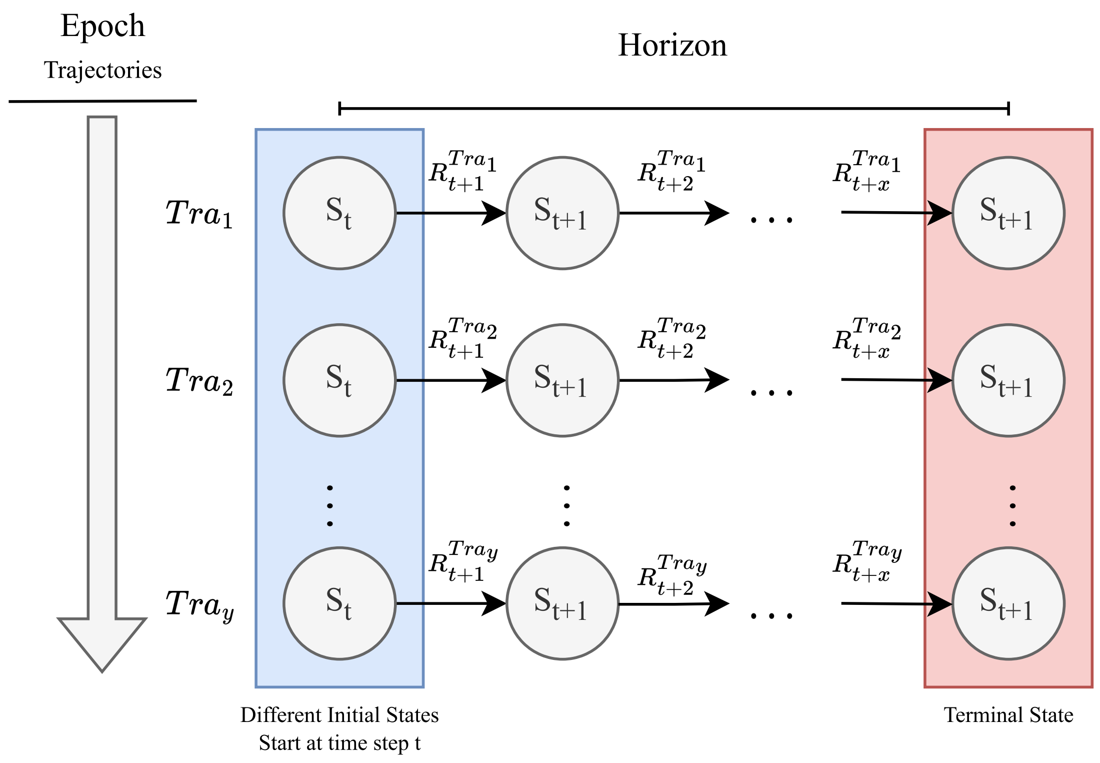

Figure 1.11: Figure showing the connection between epoch, trajectory, and horizon.

This thesis considers the Proximal Policy Optimization (PPO) method for adjusting the policy network used to make VNE decisions[11].

PPO is referred to as a *policy gradient* method. In policy gradient methods, the trajectories contained in each epoch are used to update the weights and biases θ of the policy network. This is achieved using the gradient of a reward function $J(\theta)$. Based on the direction of the gradient, the weights and biases θ of the policy network can be adjusted using back propagation such that the agent is more likely to choose the beneficial actions and less likely to choose bad actions following policy π_θ [21].

Since a policy is represented as a NN, the parameters θ of the NN must be updated to improve the policy π_θ. To give a score to a specific configuration of the parameters θ, the formula 1.8 can be used. Note also how the formula can be expanded using formula 1.7 [21]:

$$J(\theta) = \sum_{s \in \mathcal{S}} d_\pi(s) V_\pi(s) = \sum_{s \in \mathcal{S}} d_\pi(s) \sum_{a \in \mathcal{A}} \pi_\theta(a|s) Q_\pi(s,a) \tag{1.8}$$

[11]See section 5 for specific details on PPO.

Moreover, note that equation 1.8, sums over all states $s \in \mathcal{S}$ when calculating the score of the policy. This is required since the overall policy score must be based on the values of all states. However, the sum of state value $V_\pi(s)$ alone is insufficient to represent a policy score. Hence, $V_\pi(s)$ is multiplied by the stationary distribution $d_\pi(s)$. This ensures that state values are scaled appropriately such that unlikely states $d_\pi(s') \approx 0$ with high values $V_\pi(s')$ don't contribute much to the overall policy score [21].

Furthermore, the gradient of equation 1.8 is given by equation 1.9 [18][21]. Note that the proof for equation 1.9 can be found in [21]:

$$\Delta J(\theta) = \mathbb{E}_\pi[\nabla_\theta \ln \pi_\theta(a|s) Q_\pi(s,a)] \tag{1.9}$$

Lastly, given the gradient, we can improve the policy [19]:

$$\theta = \theta_{old} + \alpha \Delta J(\theta) \tag{1.10}$$

1.4 Objectives

The main goal of this thesis is to solve the VNE problem in SAGIN while accounting for the issues related to NFV security. To achieve this goal, we will cover the following points:

- We review the related works to gain insight into the VNE problem, the security aspect, and SAGIN.
- Describe and define our VNE problem including the evaluation metrics and our objectives.
- Implement a simulator that is capable of simulating our version of the VNE problem.
- Evaluate the performance of our solution strategy compared to other solutions.

The thesis is divided into the following sections: Firstly, we discuss the related works in section 2. Secondly, we describe and define our problem in the approach section 3. Thirdly, we describe the implementation in section 4. Next, section 5 contains an evaluation of our performance compared to the existing solutions. Lastly, we conclude our work in section 6.

Chapter 2

Related Works

This section provides a review of the related works. This section aims to provide insight into how the VNE, security, and SAGIN are considered by the related work on which our work is based.

The related works section is divided into the following sub-sections:

- Fundamental considerations for VNE: These works cover fundamental considerations related to the VNE problem.
- Security-aware VNE: These works focus on enhancing the security of VNE solutions.
- SAGIN-aware VNE: Works that provide solutions to VNE in the context of SAGIN.

2.1 Fundamental Considerations for VNE

In the earlier works on VNE research, most works did not focus on security or SAGIN. Nevertheless, early works researched essential aspects of the problem that should still be considered today. For example, early works provided the foundations for future works by proposing SNN and SNL parameters[1] that adequately describe real networks and objectives that reflect the goals of VNE.

For instance, the work done by Nogueira et al. [22] proposes a solution to VNE with extensive processing parameters, such as CPU frequency, CPU cores, and CPU load, for their SNNs. Although this provides a good description of SNN processing capabilities, most network applications don't have requirements related specifically to CPU frequency or core count. As such, most works consider these processing parameters to be surplus

[1]Such as processing, memory, and storage.

to requirements and only consider the more generalized processing (CPU) parameter instead [15][23].

Furthermore, Nogueira et al. [22] proposes SNN memory (RAM) and SNL bandwidth parameters. RAM is important since SNNs need to perform the processing of data. For instance, a VNN performing DNS may require RAM for processing. As such, VNE works should include CPU and RAM parameters to avoid congestion of SNNs. The same holds for the bandwidth parameter since SNLs are likewise limited in capacity and should not be utilized if congested. However, [22] does not mention whether their bandwidth resource is based on wired or wireless SNLs. Moreover, an identical bandwidth is used for all SNLs, which does not accurately depict the available SNL bandwidth in a real network.

A general description of the VNE solution proposed by Nogueira et al. [22] is as follows. Firstly, [22] implements a custom algorithm that scans the SN for remaining SN resources and makes VNN placements based on VNNs with the most available resources. However, when placing the VNLs, [22] uses a Constrained Shortest Path First (CSPF) algorithm to select SNLs with sufficient remaining resources. This approach, which separates VNN and VNL placement, is commonly known as a "two-stage" approach in VNE terminology [12].

We highlight another work that is proposed by Zhang et al. [24], which improves the aspects of Quality-Of-Service (QoS) and resiliency, which is not covered by [22]. Firstly, Zhang et al. [24] does not include either processing, memory, or bandwidth parameters since their focus is solely on resiliency and QoS. Instead, [24] proposes to include the SNL delay parameter. The delay parameter is essential for QoS since it represents the time spent transmitting data from the ingress VNN to the egress VNN. This is important for some delay-sensitive applications, such as video streaming, where response times are affected by the delay. By introducing delay into SNLs, the VNE solution provided by [24] is able to ensure high QoS for its embedded VNRs.

Secondly, the work by Zhang et al. [24] addresses the issue of resiliency, which was not covered in the previous work by Nogueira et al. [22]. Resiliency is a critical aspect to consider when dealing with failure-sensitive applications. In the event of failures occurring on SNNs hosting VNs, the VNs may become dysfunctional, making resiliency an essential factor to consider.

Lastly, the simulation environment of both the [24] and [22] works has room for improvement. Firstly, neither [22] nor [24] provides an online environment, thus considerably reducing the value of their simulation in terms of applying their solution to a real network. Furthermore, neither [22] nor [24] mentions that a dynamic or distributed approach is used, so one can assume that both works use a static and centralized approach. As such, both solutions may prove incapable of maintaining VNs in a more challenging real-world environment where the dynamism and scale of the environment would likely result in significantly lower embedding performance.

Paper	Approach		SAGIN	Security	Objective	Parameters		Environment		
	Algorithm	Stages				SNN	SNL	Online/ Offline	Dynamic/ Static	Distributed/ Centralized
[22]	Heuristic	Two-stage CSPF	×	×	Even distribution of SN resources	CPU Load CPU Freq. CPU Cores RAM Storage VMs	BW	Offline	Static	Centralized
[24]	Exact & Heuristic	Two-stage BFS	×	×	QoS and resiliency	Degree	Delay	Offline	Static	Centralized

Table 2.1: Overview over reviewed early works.

The table 2.1 provides an overview of the main considerations made by [22] and [24].

2.2 Security-Aware VNE

We have now gained an overview of some parameters and considerations in earlier works. However, none of these works consider security, which is a vital consideration for any service. The subsequent sections will review how security is considered by related works that cover security.

2.2.1 Security Levels

Section 1.2 gave a high-level description of the vulnerabilities faced by NFV. These vulnerabilities have prompted the creation of several works proposing VNF-FGE and VNE solution algorithms capable of ensuring security. Such works are said to provide *security-aware* solutions to the VNE problem [23]. This section primarily focuses on how these works consider security.

Gong et al. [13] provides a security-aware solution to VNE and provides a high-level definition of attacks and some defenses in VNE. In terms of attacks, [13] highlights attacks between the VNE actors, namely the tenants, SPs, and InPs. The management roles of SPs and InPs make them candidates for malicious activity. InPs can access the physical hardware of the SN and can therefore perform any malicious actions directly on the hardware. In addition, the SP can perform similar attacks due to its control over the leased VNs [13].

Gong et al. [13] also notes that InPs are susceptible to attacks by SP tenants who operate at InP virtualized hardware. Since SP tenants operate in a VM, they should not have any control or overview over their host OS. However, if the protection provided by the VM is insufficient, the SP tenant could perform attacks directly on the host OS. Some works refer to this general type of attack as a *VM escape* [14].

Regarding mitigation strategies, [13] gives a limited description. For example, [13] states a list of security NFs, such as Intrusion Prevention Systems (IDS) and firewalls, which could detect and protect against attacks. However, this is not a comprehensive list of NFV security mechanisms [14].

To reflect these general security mechanisms, [13] combines them into a *security level* parameter. However, [13] does not directly map these security mechanisms to the security levels.

SNN Security Level

In order to incorporate the security levels into their VNE simulation, Gong et al. [13] firstly incorporates the security level parameter in their SNNs. Secondly, a *security demand* parameter is included in the VNNs, which limits the possible embeddings to SNNs of equal or higher *security level* [13]. This combination of security and demand provides an accurate representation of how SPs request higher security in VNRs when demanded by SP tenants.

We highlight why it is logical to include security levels in SNNs based on the work by Lal et al. [14], who provides a comprehensive review of NFV attacks and mitigations. Most mitigation techniques noted by [14] involve direct implementations on SNN hardware[2], highlighting why security levels on SNNs accurately represent NFV security.

For example, [14] notes the possibility of *hypervisor introspection*. This gives the InP additional monitoring capabilities over VMs running VNFs. Suppose a VNF behaves maliciously, such as attempting a *VM escape*. In that case, the InP will be able to detect the activity and react before significant damage can be done to the InP or other hosted VMs [14]. Figure 2.1 visualizes the implementation of hypervisor introspection on the OS, as described by Pattaranantakul et al. [25].

We highlight a second security technique presented by [14], which could also be applied to SNNs. This security technique protects against the problem of InP employees tampering with VMs. For instance, if an InP employee attempts to access data from a VM, then the data should not be readable. In [14], it is therefore stated that the InP should encrypt

[2]Not inside VMs.

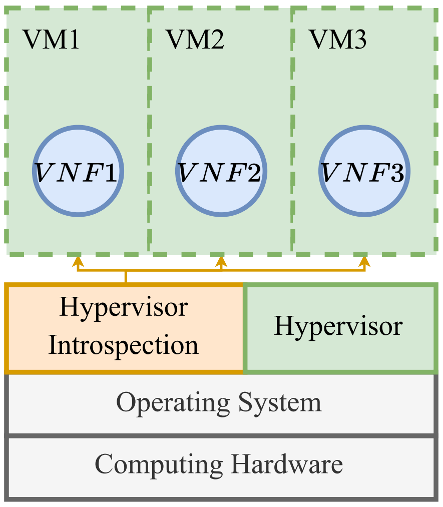

Figure 2.1: Visualization of hypervisor introspection on SN hardware based on the description by [25].

all stored data. Furthermore, [14] notes the possibility of signing VNF images, which are used to instantiate VNFs. When the images are signed, employees cannot add malicious software to the image without this being detected by InP management.

The two security techniques mentioned above are only two of several security techniques presented by [14] that could be applied to SNNs. Hence, we argue the security levels in SNNs proposed by [13] gives a good abstract description of SN security measures.

VNN Security Level

Secondly, Gong et al. [13] proposes security levels also in its VNNs that must match the security demand of the host SNN. Gong et al. [13] claims this is required for protecting the underlying SNN and other co-hosted VNNs.

However, we argue security levels in VNNs and security demand in SNNs are not required to represent SN security. We base this argument on the NFV security overview provided in [14]. In their overview, [14] presents multiple security techniques that could be applied directly to the SNNs, not VNNs. Hence, we argue security levels in VNNs do not add any additional benefits when representing the security issues introduced in NFV.

Additionally, the purpose of the VNN security level is not properly defined. For example, consider a VNN. If we say the VNN arbitrarily represents a firewall VNF, which we know is located inside a VM, then how can the VNN provide any protection to the SNN when isolated by the VM? This is not mentioned in [13].

As such, we highlight the work by Zhao et al. [26], who proposes an alternative utilization of security levels. In [26], security levels are not associated with the VNNs, and SNNs have no demanded security from their hosted VNNs. We argue this is a more accurate representation of NFV security. In addition, [26] includes security levels in SNLs, which is not covered in [13].

2.3 SAGIN-aware VNE

Until now, we have seen works with some differences in parameters, solution algorithms, and simulation considerations. However, most of these works describe their SN as a single domain, which does not accurately represent SAGIN [2]. Therefore, this section reviews works that consider VNE in the context of SAGIN, which we call SAGIN-aware works. Our review highlights how these works represent SAGIN and which VNE considerations that are essential for representing SAGIN.

2.3.1 SAGIN SN Representation

The first SAGIN work we consider is proposed by Zhang et al. [2]. Their simulation description highlights how special considerations must be taken for SAGIN compared to non-SAGIN works. Firstly, earlier works such as [22] sampled their SNN and SNL parameters using fixed-range probability distributions for a single domain only. As a result, every SNN and SNL parameter value would be within the same distribution range as all other SNNs and SNLs.

However, as we know from the description of hardware resources in SAGIN from section 1, this parameter configuration would be inadequate for SAGIN due to the hardware differences between the domains [2]. As such, Zhang et al. [2] proposes an alternative representation of the SN by defining it as three main graph components representing the domains. Moreover, [2] gives each domain unique parameter ranges to each domain, providing the unique accommodations required for each SAGIN domain [2].

However, since SAGIN aims to create an integrated network, each SN domain cannot be disconnected from the other. Hence, [2] defines the concept of *inter-domain links* (IDL) as a way of connecting SN domains into a single *integrated* SN. Figure 2.2 visualizes IDLs as red connections.

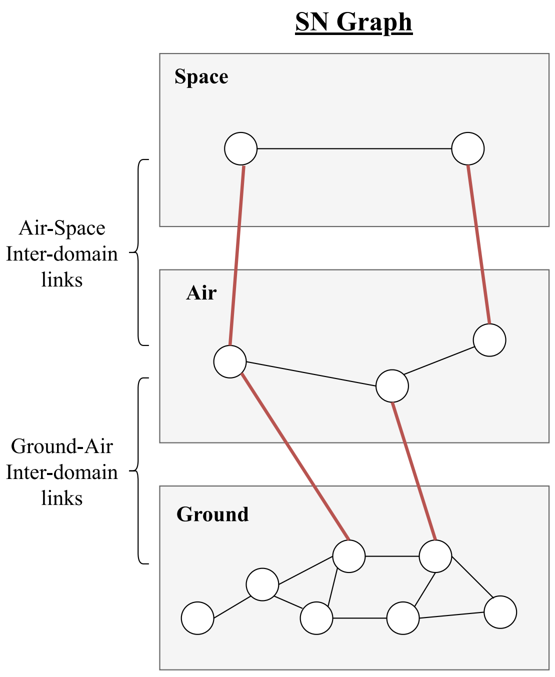

Figure 2.2: Visualization IDLs.

Regarding parameters, [2] applies the SNN CPU parameter to all SNNs, with unique sampling ranges based on the domain of each SNN. However, [2] does not consider storage or memory parameters. Next, [2] notes that SAGIN inherits QoS challenges related to the physical nature of the space domain, where satellites must communicate through vast distances, thus incurring transmission delays [2]. Due to this issue, [2] also considers the delay parameter in all SNLs.

We also consider a second work on SAGIN proposed by Wang et al. [17], who considers a similar problem as [2] but with some differences regarding parameters and VNE environment considerations. One of the differences in the parameters is the *candidate domain* proposed by [17].

Candidate domains are based on how SAGIN SNNs and SNLs are inherently given a domain label based on which domain they are located in. In [17], this domain label is used to give further capabilities to VNRs by including a set of domains in each VNN, indicating which domains each VNN can be embedded in, referred to as their *candidate domains* [17]. The concept of candidate domains is also interesting for the security aspect since it could potentially be used to provide physical security. For example, air or space-based equipment is inherently difficult to access physically. It could therefore be considered a potential security measure to request VNNs in the space or air domains[3].

[3]We will not consider candidate domains as a security measure in the problem of this thesis.

Similarly to the parameters in [2], [17] includes most of the basic parameters besides the candidate domain parameter. However, the RAM parameter is replaced by a storage parameter not considered by [2].

2.3.2 VNE Considerations for Representing SAGIN

We have now seen how two works that accommodate SAGIN by modifying the structure of the SN and by including the appropriate parameters for SAGIN. This section instead highlights the differences in the VNE environment considerations, listed in table 1.1, of both works.

Firstly, the non-SAGIN works we've seen in sections 2.1 and 2.2 all consider a static environment instead of a dynamic one. Based on the assumption that these works utilize a single ground-domain SN, then using a static representation is suitable since ground-based hardware can be assumed to be mostly physically static.

However, for SAGIN, this conclusion no longer holds due to the inclusion of the air and space domains, which are inherently dynamic [17]. For example, when a satellite or air vehicle leaves its original position after some time, it may break the SNL connected to it, thus also breaking VNs embedded in the SNL. As such, SAGIN works should ideally consider a dynamic environment to simulate SAGIN effectively [17].

Although both the [2] and [17] works state the importance of using a dynamic environment, neither actually considers the dynamic environment and instead considers a simplified static environment. The argument for this approach provided by both works is that air vehicles and satellites will not move much during a smaller time frame [2][17]. Therefore, both [2] and [17] essentially use a static environment.

As we know, SAGIN includes additional VNNs and VNLs compared to a single-domain network. Hence, the SAGIN network is much larger than a single-domain network. Furthermore, since satellites can be anywhere around Earth, a SAGIN network could also span the entire Earth. Due to the size of SAGIN, a logical assumption would be that multiple InPs would perform embeddings. Hence, Wang et al. [17] proposes a distributed SAGIN solution. Their solution includes splitting their SN even further into local areas of SNLs and SNNs. Furthermore, each area is incorporated into their DRL solution by using multiple DRL agents, where each agent is responsible for their area of the network. Using this setup, [17] can closely resemble how the embeddings would be handled in a real-world deployment of VNE in SAGIN.

Paper	**Approach**		**SAGIN**	**Security**	**Objective**	**Parameters**		**Environment**		
	Algorithm	**Stages**				**SNN**	**SNL**	**Online/ Offline**	**Dynamic/ Static**	**Distributed/ Centralized**
[2]	DRL	Two-stage BFS	✓	×	Maximize profits	CPU Avg. distance	BW Delay	Online	Static for time t	Centralized
[17]	DRL	Two-stage BFS	✓	×	Maximize profits	Candidate domains CPU Storage Avg. distance	BW	Online	Static for time t	Distributed

Table 2.2: Overview over review related works on SAGIN.

The table 2.2 provides an overview of the main considerations made by the reviewed works on SAGIN.

2.4 Thesis Contributions

In the review of the related works, we have seen both security-aware works and SAGIN-aware works. However, none of these works or other works that we are aware of consider both security and SAGIN. Hence, the main contribution of this thesis is to effectively solve VNE when considering both SAGIN and security using a DRL solution approach.

Chapter 3

Problem Description and Formulation

Based on the background sections and our review of related works, we now have the required background knowledge to define and implement a simulator for our variation of the VNE problem. This chapter is divided into three main sections:

1. **Problem Description**: General problem description.
2. **Problem Formulation**: Concrete mathematical definitions of our problem.
3. **Implementation**: Description of essential simulation details, design, and challenges encountered during implementation.

3.1 Problem Description

The core of our problem is to provide a security-aware solution to VNE in SAGIN using DRL.

The following sections provide the details of our VNE problem.

3.1.1 Bossiness Model

Our business model can be described with the help of Figure 3.1. The figure shows an SP that needs a new service for one of its tenants. As such, the SP generates a new VNR satisfying the service and all its requirements.

The SP sends the VNR to an InP, who deploys the VN to the SN using a DRL solution to the VNE problem. If the InP successfully embeds the VN, it gains revenue based on the number of resources used, which is paid by the SP. Moreover, the InP incurs operational costs[1] related to the maintenance and operation of the leased hardware. However, if the InP fails to embed the VN, no revenue is gained and the InP incurs no maintenance costs related to the VNR.

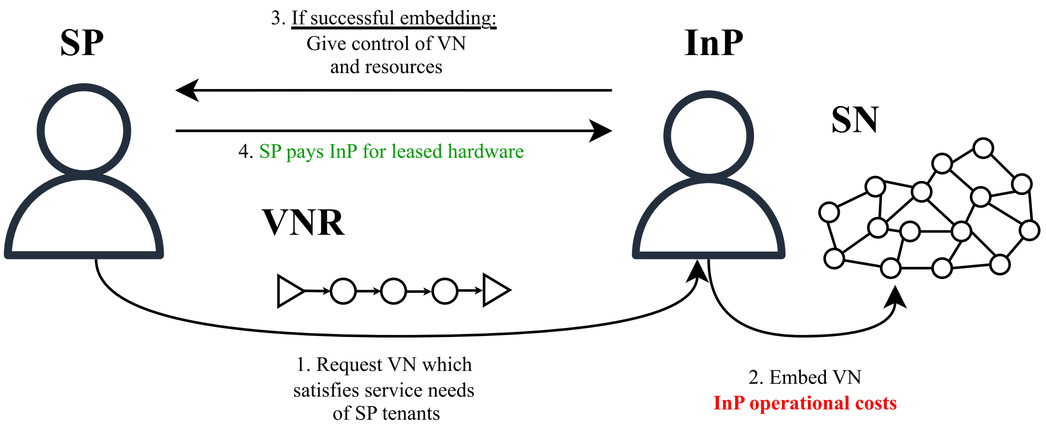

Figure 3.1: Visualization of a typical interaction between tenants, SP, and InP. InP receives revenue and costs from embedding.

The main objective of this thesis is focused on the InP. Specifically, we can state our objective as *minimize InP cost while maximizing InP revenue*. In other words, optimal usage of SN resources.

The following problem description sections will highlight our VNE considerations.

[1]Cost serves as an indicator of how effectively the InP uses its resources.

3.1.2 SN Description

The SN represents the resources that can be leased by the SP. This section describes the assumptions we make regarding the SN.

SAGIN Description

We extend the traditional SN representation with SAGIN domains, a recent development for extending communication network service coverage.

We consider our SN to have three separate domains, namely the *ground*, *air*, and *space* domains. Note that the GEO, MEO, and LEO satellites discussed in section 1.1 are considered part of a single more general space domain for simplicity.

We number the ground, air, and space domains as 1, 2, and 3, respectively.

Each domain is represented as a network of SNNs and SNLs, as described in section 1.1.2. However, the domains have the following unique characteristics:

- The air and space domains have fewer hardware resources to represent the costs and difficulty of placing hardware in these domains.
- The air domain has fewer SNNs and SNLs than the ground domain.
- The space domain has fewer SNNs and SNLs than the air domain.

Next, we consider all domains to be connected using IDLs to create an integrated SN, which is one of the goals of SAGIN. Figure 2.2 shows a visualization of IDLs. The ground domain connects to the air domain, and the air domain connects to the space domain.

Security

As discussed in sections 1.2 and 2.2.1, NFV includes various security threats. As such, we consider the abstract *security levels* measure to indicate how much protection is given in SNNs:

- Level 3 $\rightarrow$ High Security
- Level 2 $\rightarrow$ Medium Security
- Level 1 $\rightarrow$ Low Security

Furthermore, we want tenants to be able to specify their level of security. Hence, we assume tenants provide information to SPs regarding desired security. This is reflected by a *security demand* in the VNNs.

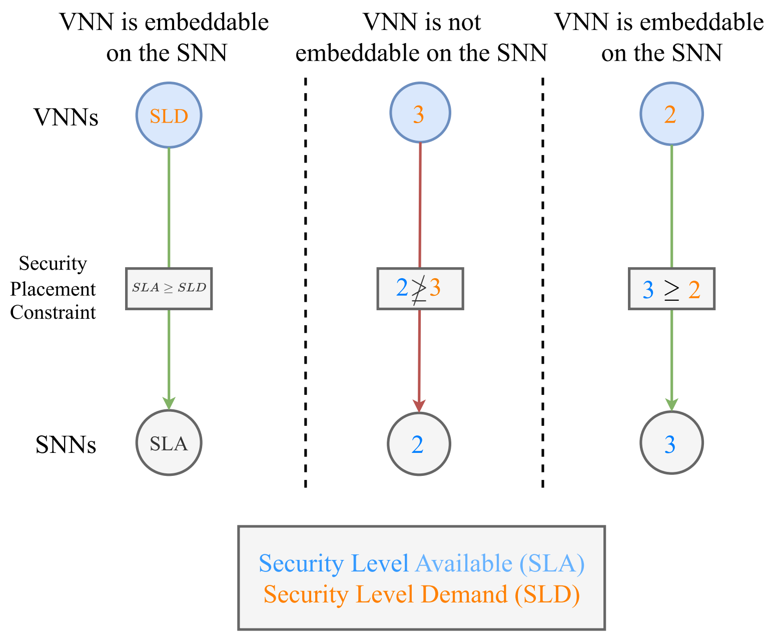
VNN is embeddable on the SNN
VNN is not embeddable on the SNN
VNN is embeddable on the SNN
VNNs
SLD
3
2
Security Placement Constraint
$SLA \geq SLD$
$2 \not\geq 3$
$3 \geq 2$
SNNs
SLA
2
3
Security Level Available (SLA)
Security Level Demand (SLD)

Figure 3.2: Visualization of how security levels and security demand constraints VNN placements.

We illustrate the interaction between the security levels and the security demand in figure 3.2. If the security demand is not matched with an adequate security level, embedding is impossible.

3.1.3 VNR Description

A VNR represents a VN and its requirements requested by an SP. This section describes the assumptions we make regarding VNRs.

We consider the *online* approach for VNRs. As such, we extend our VNRs with an *arrival time* and *lifetime*:

- VNR Arrival: When a tenant requests a VNR in the real world, the request may occur at any time. The VNR arrival depicts this time. To assign each VNR an arrival time slot, we use the *Poisson probability distribution*, commonly used to represent arrival events [3][23][27]
- VNR Lifetime: Besides the arrival time, we consider each VNR to have an associated lifetime. The lifetime is essential for releasing resources over time. For example, if

no VNRs are removed over time, the SN will become completely congested. We use the *exponential* distribution for sampling lifetimes, as it is commonly used to find time between the *Poisson* arrivals [4][27]. Furthermore, we can scale the intensity of the distribution to simulate more or less embedded VNR congestion in the SN.

Candidate Domain

We include the candidate domain constraint suggested by [17].

Each VNN includes a single candidate domain, signifying the VNN's only embeddable domain. Hence, the candidate domain is measured using SAGIN domain levels:

1. Embeddable in the ground domain.
2. Embeddable in the air domain.
3. Embeddable in the space domain.

By including the candidate domain, we add an additional novelty to the thesis and increase the difficulty of our problem. Furthermore, adding the candidate domain gives further capabilities to the SP who creates VNRs. In a scenario where the SP would like its VNNs in specific domains, we can accommodate such a VNR.

Ingress and Egress

Section 1.1.1 introduced the concept of *ingress* and *egress*. We include an ingress VNN to represent the ingress point of network traffic into the VN and an egress VNN to represent the outgoing traffic from the VN.

By including ingress and egress VNNs, we ensure embeddings are placed in relation to the network traffic source into the VN and the traffic destination out of the VN, which is required for the operation of the service. This is visualized in Figure 3.3 where the ingress is placed at the source, and the egress is placed at the destination in the SN.

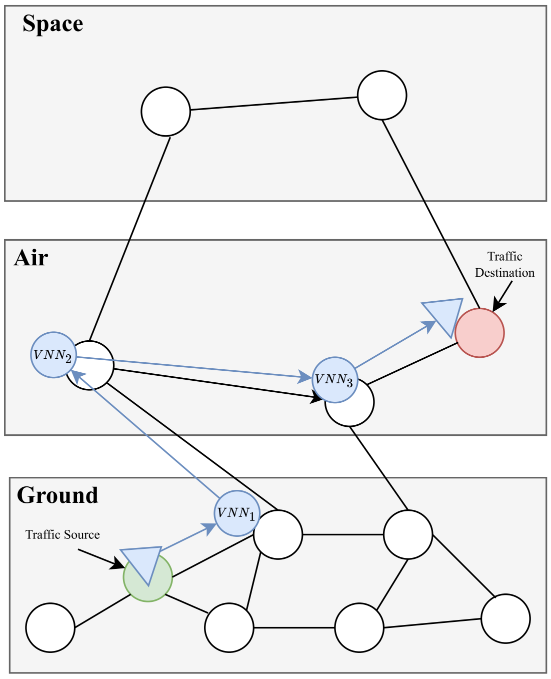

Figure 3.3: Visualization of how the ingress is embedded onto the SN traffic source and the egress onto the SN traffic destination.

We consider the ingress and egress to have no resource demands. Moreover, since most SP tenants are located in the ground domain, we consider SN source SNNs only in the ground domain, while SN destination SNNs are in any domain.

3.1.4 Parameter Description

We choose our parameters using the following points:

- The parameters should give a realistic representation of real-world resources.
- The parameters should support the research goals of this thesis.

SN Parameters

We describe our SNN parameters as follows:

- **CPU**: All computations performed by SNNs use the CPU. Hence, when hosting a new VNN onto an SNN, evaluating whether CPU resources are available is essential.
- **Security Level**: Represents the protection level.
- **Domain**: Represents the SAGIN domain in which the SNN is located.

We describe our SNL parameters as follows:

- **Bandwidth**: Max available bandwidth in an SNL.
- **Delay**: We consider delay as the travel time over SNLs, which is especially relevant for distant satellites communicating over significant distances.

The notations for the above parameters are listed in table 3.1. Note that table 3.1 categorizes each parameter as either a resource or a constraint. This signifies whether the parameter is a spendable resource or an embedding constraint.

SNN				
Parameter	Type	Measure	Data type	Description
CPU	Resource	Utilization	integer	CPU resources
SLA	Constraint	security level	integer	Security Level Available (SLA)
DOM	Constraint	domain	integer	Domain the SNN is part of

SNL				
Parameter	Type	Measure	Data type	Description
BW	Resource	Mb/s	integer	Available bandwidth
DLY	Constraint	ms	integer	Travel time between nodes

Table 3.1: SN parameter notations.

VNR Parameters

We describe our VNN parameters as follows:

- **CPU**: VNNs demand CPU resources from SNNs to meet their computational requirements.
- **Security Demand**: Demanded security from the SNN used to embed the VNN.
- **Candidate Domain** Specifies the domain in which the VNN can be embedded.
- **Ingress** Whether the VNN is an ingress or not.
- **Egress** Whether the VNN is an egress or not.

And the following VNL parameters:

- **Bandwidth**: VNL bandwidth requirement.
- **Delay** Maximum VNL delay requirement.

The notations for the above parameters are listed in table 3.2.

VNN				
Parameter	Type	Measure	Data type	Description
CPU	Resource	Utilization	integer	CPU resource demand
SLD	Constraint	security level	integer	Security Level Demand (SLD)
CAN	Constraint	domain	integer	Candidate domain
IN	Constraint	1/0	bool	VNN is an ingress
EG	Constraint	1/0	bool	VNN is an egress

VNL				
Parameter	Measure	Data type	Description	
BW	Resource	Mb/s	integer	Bandwidth demand
DLY	Constraint	ms	integer	Max delay demand

Table 3.2: VN parameter notations.

3.1.5 Evaluation Metrics

Before a VNE solution algorithm can be deployed in a real network, it must be rigorously tested and evaluated based on its objectives. This is essential to avoid introducing a low-quality embedding algorithm onto a real network, potentially wasting resources and reducing profits.

Acceptance Rate

The first metric is the *acceptance rate*, which captures the problem of failed embeddings. Failed embeddings are a common occurrence due to the limited resources found in the SN and the constraints imposed on the solution algorithm [12]. For example, if a VNN requires an SLD not supported by the SLA in any SNN, then the VNR cannot be embedded due to security constraints.

A solution algorithm with a low acceptance rate indicates the embedding algorithm is not intelligently utilizing the SN resources. To illustrate this point, figure 3.4 shows how a lower acceptance rate is achieved due to a bad VNR_1 embedding causing another VNR_2 to fail[2]. An alternative solution is shown in figure 3.5, which results in a higher acceptance rate. The second solution is preferable for the InP due to higher resource utilization.

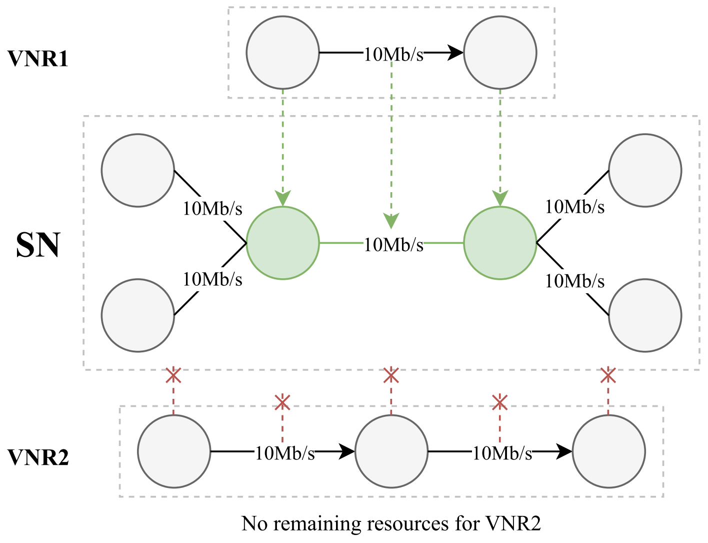

Figure 3.4: Visualization of how one embedding might cause the next to fail.

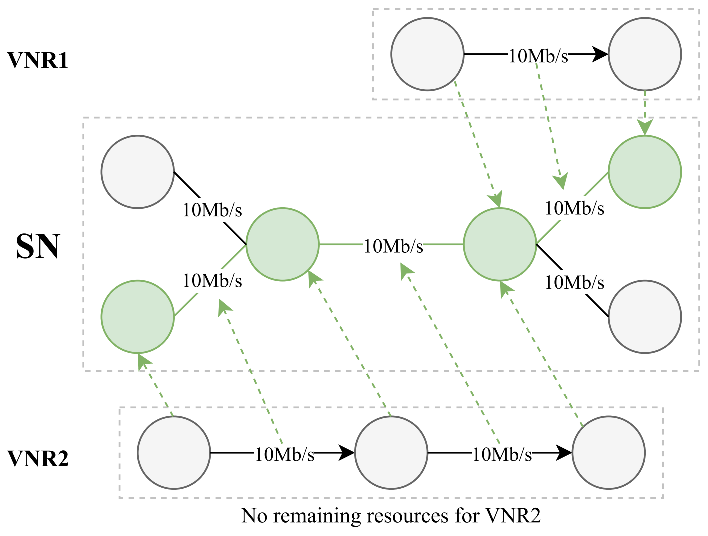

Figure 3.5: Improved embedding solution compared to figure 3.4.

Long Term Average Revenue

To evaluate our goal of maximizing revenue, we include the Long Term Average Revenue (LTR) metric. The LTR metric considers the average revenue of all embedded VNRs up

[2]Note that the ingress and egress of VNR_1 and VNR_2 are not shown for simplicity.

to a simulation time t. LTR is a good metric for comparing how revenue improves over time for different embedding scenarios [17].

Long Term Average Cost

To evaluate the overall profits, we must consider both revenue and cost. Hence, we include a similar metric to LTR, but with revenue replaced by cost, termed the Long Term Average Cost (LTC) [17].

Long Term Average Revenue To Cost Ratio

We use the Long Term Revenue Cost Ratio (LRC) to measure the ratio between revenue and cost for a given time t [17]. LRC indicates whether the InP is earning or losing money based on recordings up to time t. For instance, if the ratio between revenue and cost is below one, the InP loses money. Moreover, if the InP achieves an LRC close to or above one, then the InP uses an efficient VNE solution strategy that limits the overall costs and maximizes revenue.

3.2 Problem Formulation

The previous section 3.1 provided the informal problem description. This section provides the formulation:

1. SN Formulation.
2. VNR Formulation.
3. Parameter Formulation.
4. Metrics Formulation.

3.2.1 SN Model

Since the SN consists of SNNs and SNLs, a graph is ideal for representing the SN network using nodes and edges. This is visualized in figure 3.6.

We denote A as the SN graph containing all SNNs in a set N and all SNLs in a set E. These sets contain all SNNs and SNLs that are part of every domain, including

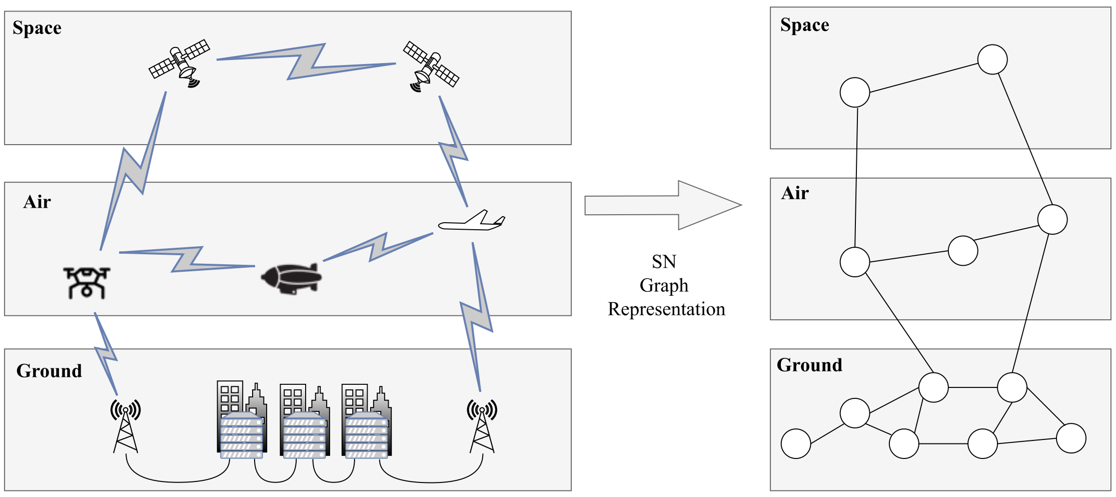

Figure 3.6: Visualization of how a real SAGIN network can be represented as a graph.

IDLs connecting the domains. Additionally, we define A as undirected since SNLs are bi-directional:

$$A = \{N^A, E^A\} \tag{3.1}$$

When referring to a specific SAGIN domain, we include a superscript $d \in \{1, 2, 3\}$, which represents the domain levels. The space, air, and ground domains are represented by domain level 3, 2, and 1, respectively. However, to make the notation more readable, we use the following letters to represent the domain levels: $s = 3$, $a = 2$, $g = 1$. Hence, $d \in \{s, a, g\}$:

$$A^d = \{N^{A^d}, E^{A^d}\} \tag{3.2}$$

Next, in order to access a specific SNN, we use the following notation where an SNN n_i is extracted from the set of all SNNs N_i^A using the set index i:

$$n_i = N_i^A \tag{3.3}$$

When extracting an SNN from a specific domain, we use the following notation:

$$n_i = N_i^{A^d} \tag{3.4}$$

To access the parameter value of an SNN, we use a superscript denoting which parameter we are accessing:

$$n_i^{CPU} \qquad n_i^{SLA} \qquad n_i^{DOM} \tag{3.5}$$

Regarding the SNLs, we use a similar notation when extracting a specific SNL:

$$e_{(n_i,n_j)} = E^A_{(n_i,n_j)} \tag{3.6}$$

In the notation above, we extract a specific SNL between the SNNs n_i and n_j.

Furthermore, accessing the parameter value of an SNL is identical to accessing the SNN parameter value as follows:

$$e^{BW}_{(n_i,n_j)} \qquad e^{DLY}_{(n_i,n_j)} \tag{3.7}$$

The notations above are listed in table 3.3.

Notation	Description
d	Specifies the domain of the SN graph
A	SN graph including all domains
N^A	Set of SNNs in all domains
E^A	Set of SNLs in all domains including IDLs
A^d	SN graph of domain d
N^{A^d}	Set of SNNs in domain d
E^{A^d}	Set of SNLs in domain d
n_i	Extracted SNN with index i
n_i^{CPU}	Available computational resources in SNN n_i
n_i^{SLA}	SLA in SNN n_i
n_i^{DOM}	Domain of SNN n_i
$e_{(n_i,n_j)}$	SNL between n_i and n_j
$e^{BW}_{(n_i,n_j)}$	Available bandwidth resources in SNL $e_{(n_i,n_j)}$
$e^{DLY}_{(n_i,n_j)}$	Delay of SNL $e_{(n_i,n_j)}$

Table 3.3: SN notations.

We also require notation for specifying the amounts of SNNs and SNLs in the SN or the number of some other element. For this purpose, we always use the η symbol. We list the SN η notations and their descriptions in table 3.4.

Notation	Description
η^s	Number of space domain nodes
η^a	Number of air domain nodes
η^g	Number of ground domain nodes
$\eta^{(d_a,d_b)}$	Number of IDLs between domains d_a and d_b

Table 3.4: SN additional notations.

3.2.2 VNR Model

The set of all VNRs is defined as R, and each specific VNR part of R is defined as r_i where i is the index of a specific VNR:

$$R = \{r_i, r_{i+1}, r_{i+2}, \ldots\} \tag{3.8}$$

Each VNR contains three main parts, the VN denoted as B, the time of arrival denoted as t^a, and the time of departure[3] denoted as t^d. We can access these parts using superscripts on r_i:

$$r_i^B, \quad r_i^{t^a}, \quad r_i^{t^d} \tag{3.9}$$

We refer to the difference between t^a and t^d as the *lifetime* of the VNR:

$$\Delta t = t^d - t^a \tag{3.10}$$

Since we consider our VN as a chain of ordered VNFs, we represent the VN using a linearly directed graph denoted as B with a set of VNNs V^B and a set of VNLs L^B as follows:

$$B = \{V^B, L^B\} \tag{3.11}$$

We access a specific VNN using index f on the set of VNNs:

$$v_f = V_f^B \tag{3.12}$$

[3]The departure is the sum of the arrival time and the lifetime.

Moreover, we can access the parameter demands of a VNN as follows:

$$v_f^{CPU} \qquad v_f^{SLD} \qquad v_f^{CAN} \qquad v_f^{IN} \qquad v_f^{EG} \tag{3.13}$$

Next, we can access a specific VNL between two VNLs v_f and v_h as follows:

$$l_{(v_f,v_h)} = L^B_{(v_f,v_h)} \tag{3.14}$$

Moreover, we can access the parameter demands of a VNL as follows:

$$l^{BW}_{(v_f,v_h)} \qquad l^{DLY}_{(v_f,v_h)} \tag{3.15}$$

The notations above are listed in table 3.5.

Notation	Description
R	Set of all VNRs
r_i	VNR with index i
r_i^{ta}	Time of arrival for a VNR
r_i^{td}	Time of departure for a VNR
$r_i^{\Delta t}$	VNR Lifetime
r^{Accept}	Whether the VNR is accepted
B	VN graph
V^B	Set of VNNs
L^B	Set of VNLs
v_f	VNN with index f
v_f^{CPU}	Computing resource demand in VNN v_f
v_f^{SLD}	SLD in VNN v_f
v_f^{CAN}	Candidate domain of VNN v_f
v_f^{IN}	Whether the VNN is an ingress
v_f^{EG}	Whether the VNN is an egress
$l_{(v_f,v_h)}$	VNL between VNNs v_f and v_h
$l^{BW}_{(v_f,v_h)}$	VNL bandwidth demand
$l^{DLY}_{(v_f,v_h)}$	VNL delay demand

Table 3.5: VN notations.

Lastly, we include table 3.6, which includes additional VNR-related notations.

Notation	Description
η_v^{max}	Maximum length of VN
η_v^{min}	Minimum length of VN
η_{train}^{R}	Number of training requests
η_{sim}^{R}	Number of requests for simulation
$\eta_{(v_i,v_j)}^{l}$	Number of SNLs used to embed VNL $l_{(v_i,v_j)}$

Table 3.6: VN additional notations.

3.2.3 Metrics

Acceptance Rate Function

We denote R_t as a set containing all VNRs that have arrived up to time t. All VNRs $r_i \in R_t$ are assumed to be either accepted[4] or rejected[5] at time t. If some VNR r_i is accepted, then $r^{Accept} = 1$. If r_i was rejected, then $r^{Accept} = 0$. This gives the following function for calculating the acceptance rate at time t:

$$\text{ACR}(R_t) = \frac{\sum_{i=0}^{|R_t|} r_i^{Accept}}{|R_t|} * 100 \tag{3.16}$$

Revenue Function

Long-term average revenue can be calculated using the revenue gained from all accepted VNRs up to time t. Therefore, we must first define the VNR revenue function. Furthermore, to define the VNR revenue function, we must also define the revenue of successfully embedding a single VNN and VNL.

We define the revenue of successfully embedded VNN as the allocated processing resources:

$$VNN\ Revenue(v_f) = s^{CPU} v_f^{CPU} \tag{3.17}$$

Note that the parameter value v_f^{CPU} in equation 3.17 is normalized based on its maximum value specified in table 5.5. Hence, parameters with a high maximum value contribute equally to the revenue as parameters with a low maximum value.

[4]The VNR was successfully embedded.
[5]The VNR could not be embedded.

Furthermore, each normalized value is scaled according to a revenue scale s, which can be used to tune the revenue gained by each parameter [28]. We base our own revenue scales shown in table 3.8 on the values proposed by related works in table 3.7[6].

Parameter	[28]		[11]		[29]	
	Revenue Scale	**Cost Scale**	**Revenue Scale**	**Cost Scale**	**Revenue Scale**	**Cost Scale**
CPU	0.5	-	1	-	1	-
BW	0.5	-	1	-	1	-

Table 3.7: Scaling factors used in related works.

Parameter	**Revenue Scale** s	**Cost Scale** c
CPU	1	1
BW	1	1

Table 3.8: Scaling factors based on values from related works in table 3.7.

In terms of revenue gained by embedding a VNL, we consider the normalized bandwidth usage scaled by s^{BW}:

$$VNL\ Revenue(l_{(v_f,v_h)}) = s^{BW} l^{BW}_{(v_f,v_h)} \tag{3.18}$$

Based on equations 3.17 and 3.18, we get the following revenue per VNR r over its lifetime Δt:

$$Revenue(r_i) = r^{\Delta t} \left(\sum_{f=0}^{|V^B|} VNN\ Revenue(v_f) + \sum_{l_{(v_f,v_h)} \in L^B} VNL\ Revenue(l_{(v_f,v_h)}) \right) \tag{3.19}$$

Lastly, we use equation 3.19 to define LTR:

$$LTR(R_t) = \frac{\sum_{i=0}^{|R_t|} Revenue(r_i)}{t} \tag{3.20}$$

Cost Function

Long-term average cost can be calculated using the costs associated with all embedded VNRs up to time t. Again, we first define the cost associated with a single VNN and a single VNL.

[6]Note that in a real-world scenario, these scales would be set by an InP.

We define the cost associated with a successfully embedded VNN v_f as the sum of the normalized processing resources. Similarly to the revenue calculations, we use a cost scaling factor c shown in table 3.8:

$$VNN\ Cost(v_f) = c^{CPU} v^{CPU} \tag{3.21}$$

Next, we define the cost of an embedded VNL. Each VNL may require more than one SNL to fully connect two VNNs, referred to as *link splitting* [17]. We denote the number of SNLs used to embed a VNL $l_{(v_f,v_h)}$ as $\eta^{l}_{(v_f,v_h)}$. We must sum over the number of SNLs used to embed the VNL to find the cost:

$$VNL\ Cost(l_{(v_f,v_h)}) = \sum_{i=0}^{\eta^{l}_{(v_f,v_h)}} l^{BW}_{(v_f,v_h)} \tag{3.22}$$

Based on equations 3.21 and 3.22, we get the following cost per VNR r_i:

$$Cost(r_i) = r^{\Delta t} \left(\sum_{i=0}^{|V^B|} VNN\ Cost(v_f) + \sum_{l_{(v_f,v_h)} \in L^B} VNL\ Cost(l_{(v_f,v_h)}) \right) \tag{3.23}$$

Lastly, we use equation 3.23 to define LTC:

$$LTC(R_t) = \frac{\sum_{i=0}^{|R_t|} Cost(r_i)}{t} \tag{3.24}$$

LRC Function

We can define LRC using equations 3.20 and 3.24 as follows:

$$LRC(R_t) = \frac{LTR(R_t)}{LTC(R_t)} \tag{3.25}$$

3.2.4 Constraints

Objective Function

Our objective is stated in section 3.1. By using our definitions of the ACR, LTR, LTC, and LRC metrics, re-state our objective as:

- Maximize ACR such that as many requests are embedded as possible
- Maximize LTR such that the InP can achieve a higher profit.

- Minimize LTC such that the InP can achieve a higher profit.
- Maximize LRC such that the InP can achieve a higher profit.

Constraints

1. **Resource Constraints**

   For a VN B, each VNN and VNL in B includes resource demands[7].

   To embed B into some SN A, the resource demand of every VNN and VNL in B must be satisfied by an equal or greater amount of resources in the host SNNs and SNLs. This constraint must hold for all VNNs and VNLs in B for the successful embedding of B.

   Resource constraint for VNNs:

   $$n_i^{CPU} \geq v_f^{CPU} \tag{3.26}$$

   Resource constraint for VNLs:

   $$e^{BW} \geq l^{BW} \tag{3.27}$$

2. **Delay Constraint**

   Every VNL contained within some VN has a limit for the maximum acceptable delay, denoted as $l_{(v_f,v_h)}^{DLY}$. Hence, for a VNL to be embeddable onto an SNL, the following constraint must hold:

   $$e_{(n_i,n_j)}^{DLY} \leq l_{(v_f,v_h)}^{DLY} \tag{3.28}$$

3. **Candidate Domain Constraints**

   Each VNN specifies a candidate domain v_i^{CAN}. This limits the possible VNN embeddings to SNNs of the same domain as the candidate domain. Hence, for a VNN v_f to be embeddable on an SNN n_i, the domain of n_i must match the candidate domain of v_f:

   $$n_i^{DOM} = v_f^{CAN} \tag{3.29}$$

4. **Security Constraint**

   Given a VNN v_f, the following security constraint must hold for the VNN to be embeddable on an SNN n_i:

   $$n_i^{SLA} \geq v_f^{SLD} \tag{3.30}$$

[7]Table 3.2 list all parameters considered as resources.

5. **Ingress Constraint**
   If a VNN is an ingress, it is denoted as $v_f^{IN} = 1$. Moreover, if a VNN is an ingress, it is only embeddable on a ground domain SNN. Hence, to embed an ingress VNN onto an SNN, the SNN must have a domain equal to the ground domain: $n_i^{DOM} = g$.

3.3 Evaluation Objective

Our main evaluation objective is to assess how effective the DRL approach is in maximizing our objectives[8] using the ACR, LTR, LTC, and LRC metrics.

Furthermore, we compare our results with the well-known heuristic Global Resource Capacity (GRC) algorithm proposed by Gong et al. [11] to evaluate how our DRL solution compares to alternative solutions. The GRC algorithm is tested on our problem, and we measure its performance against our own using the ACR, LTR, LTC, and LRC metrics. Additionally, we measure the time spent embedding both solution approaches to evaluate whether the solution could be applied to a real-world scenario.

Lastly, we evaluate the performance of both PPO and GRC in a scenario with heavy network congestion, which simulates both approaches would handle congestion in a real-world network. This is achieved by increasing the VNR arrival rate and lifetime.

Based on our findings, we state whether our DRL solution outperforms the GRC heuristic. Section 5 gives specific details on the evaluation setup, GRC, and performance evaluation.

[8]See the section 3.2.4

Chapter 4

Implementation Setup

In this section, we will delve into the specifics of the implementation and how the challenges encountered along the way were handled.

This chapter describes the following components of the implementation:

1. Generation of a multi-domain SN.
2. Generation of VNRs.
3. Embedding VNRs into a generated SN.
4. Implementation of the environment used for training and simulating the DRL agent.

4.0.1 Implementation Overview

To give a high-level overview of the implementation components, we use figure 4.1. The figure shows the components required to achieve training and testing of a DRL agent solving the VNE problem. Each component represents a set of functions used to perform the desired task of the component.

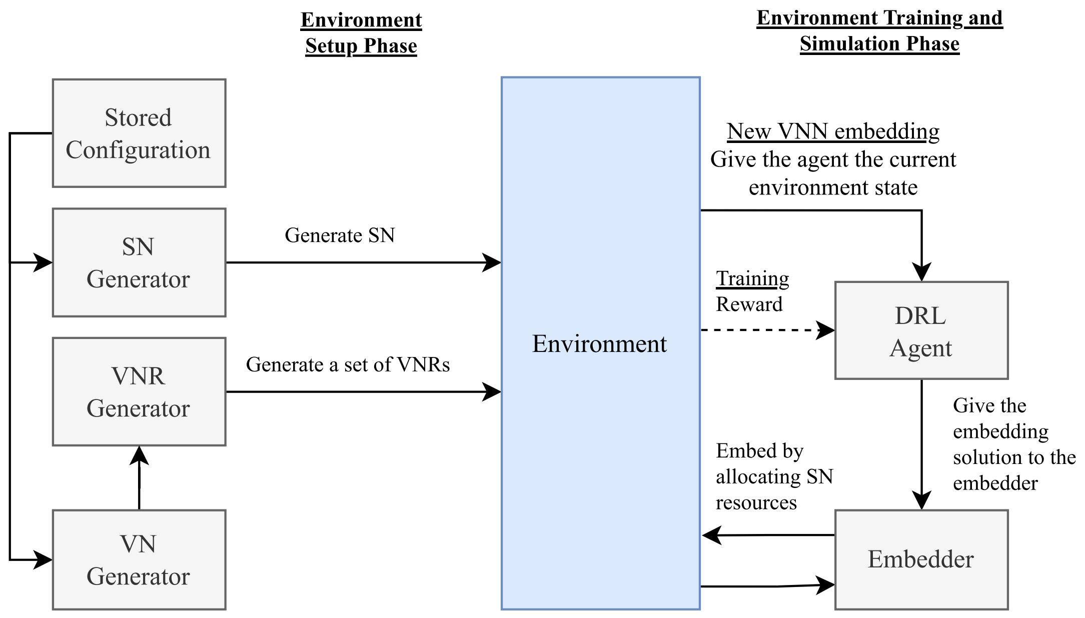

Figure 4.1: Figure showing the high-level design of the implementation components.

The main component in figure 4.1 is the environment that represents the SN, the VNRs, reward signal, and other functionality required by the agent[1]. This environment is used by the DRL agent when training and simulating VNE.

The environment relies upon several sub-components to generate its SN and VNRs. These are handled by the SN generator, which contains the required functions to generate an SN, and the VNR generator, which contains the required methods to generate VNRs.

Furthermore, each SN and VNR must be created based on the simulation configuration specified in the evaluation section 5. The configuration includes settings such as the number of domain nodes, the value ranges of parameters, and so on. These configurations are stored using the JavaScript Object Notation (JSON) format, which is represented as the *stored configuration component* in the figure.

Furthermore, once the environment has generated the SN and the VNRs, the placement tasks of each VNR are given to a solver. This solver represents the DRL agent used to find VNN placements and the shortest path algorithm used to find VNL placements. Placement solutions are given to an embedder that updates the state of the SN with the new placements such that the placements are embedded. If the agent is currently training, then the environment provides rewards to the agent used for learning.

[1]More details in section 4.0.6.

4.0.2 Programs and Libraries

The implementation uses the Python[2] programming language. This language was chosen for its ease of implementation. Furthermore, Python has a wide range of RL and network libraries which we will use extensively in our implementation. We list the main libraries we use in table 4.1.

Library	Description
networkx[3]	Used when generating SN and VNs.
stable-baselines3[4]	Provides the implementation of the training algorithm.
Gym[5]	Provides the environment interface required by the training algorithm.
matplotlib[6]	Used for various visualizations.
numpy[7]	Used for various calculations.
tensorboard[8]	Used for visualizing training and testing performance.

Table 4.1: Python libraries.

4.0.3 Generating the SN

According to our SN definition in section 3.2.1, the SN should be represented as an undirected graph with parameters added to SNNs and SNLs. To create such a graph, we use the networkx library.

Networkx is a Python library for working with graph structures and includes predefined functions for creating graphs, nodes, and edges, which we use for representing our SN and VNs [30]. Additionally, networkx enables us to assign attributes to each VNN and VNL, which is ideal for representing parameters such as CPU and bandwidth.

When creating our SN graph with networkx, we can choose from a list of graph generation algorithms. Since our SN is supposed to represent a real-world physical network, we must pick an algorithm that closely mimics a real network. For this purpose, we use the Waxman graph generator [31].

The Waxman graph generator mimics how nodes in a real network are connected by including distance in its node connection probability calculations [31]. This mimics how real network nodes typically connect to other nearby nodes, as seen in figure 4.2.

[2]https://www.python.org/
[3]https://networkx.org/
[4]https://stable-baselines3.readthedocs.io/
[5]https://www.gymlibrary.dev/
[6]https://matplotlib.org/
[7]https://numpy.org/
[8]https://www.tensorflow.org/tensorboard

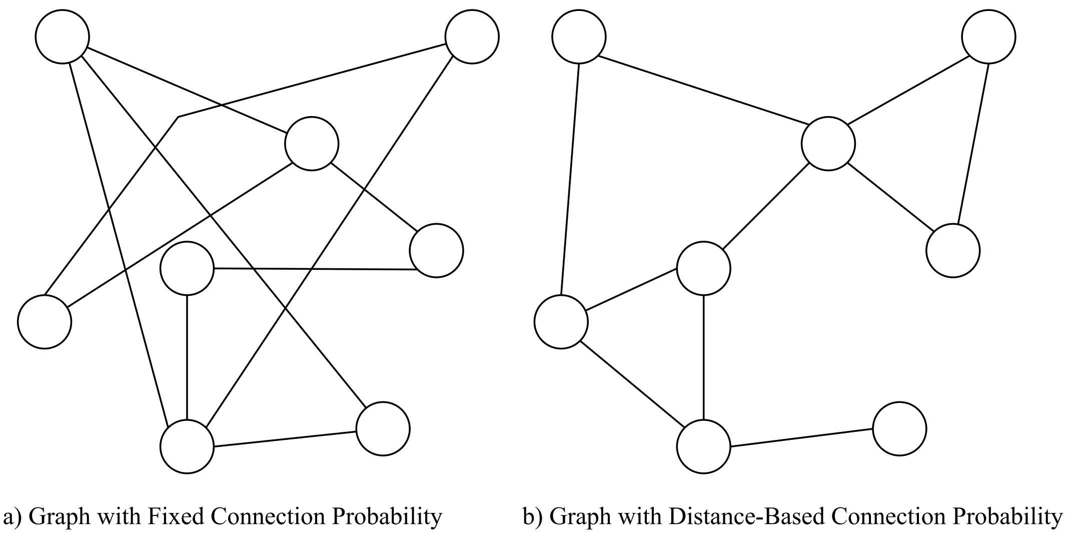
a) Graph with Fixed Connection Probability
b) Graph with Distance-Based Connection Probability

Figure 4.2: Figure showing two example graphs. Graph (b) includes distance in its connection probability and is better at mimicking a real network while graph (a) does not and includes only random connections.

We are interested in the following parameters when we generate the Waxman graph: Firstly, we adjust the $\alpha \in [0, 1]$ and $\beta \in [0, 1]$ values, which relate to the connection probability function [31]:

$$p = \beta e^{-\frac{d}{\alpha L}} \tag{4.1}$$

From this formula, β scales the entire connection probability p and can be used to limit the maximum possible probability. Secondly, α scales the distance d, making nodes less likely to connect if α is close to zero. Note there is also the parameter L, which is set automatically by networkx [31]. We provide the exact α and β values used by this thesis in the evaluation section 5.

Graph Connectivity

An issue encountered when generating Waxman graphs was that outputted graphs could contain multiple components leading to a disconnected graph. This happens because the node connection probability in equation 4.1 only gives a probabilistic guarantee of a connected graph. For example, consider a graph with three nodes as shown in figure 4.3. Each pair of nodes has a probability p of connection. If any node or group of nodes are not connected by edges, the entire graph becomes disconnected. Figure 4.3 shows one such scenario where a single node is created as a separate component.

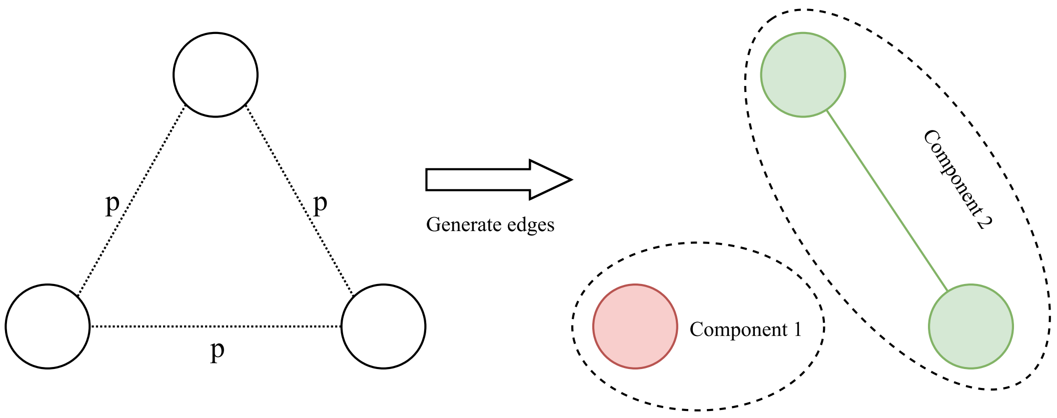

Figure 4.3: Figure showing how a Waxman graph might be generated as a disconnected graph.

If a disconnected graph is used as the SN, we encounter the following issues:

- Each component of the SN graph would represent separate networks. However, we want a single integrated network.
- A VNR can only be embedded into one component due to a lack of connections to the other components. Hence, the other components' resources are unavailable for that VNR, reducing embedding performance.
- Based on the previous point, the agent would have to learn to place entire VNRs within one component. This would increase the learning difficulty of the agent.

To avoid the possibility of using a disconnected SN graph, we re-generate the Waxman graph until the graph is connected. This will be highlighted further when we introduce algorithm 4.1.

Multi-Domain

To accurately depict SAGIN, we must ensure the SN is generated with a specific number of SNNs in the space, air, and ground domains. Furthermore, we must ensure the SNNs and SNLs in each domain are given parameter values sampled according to the distributions associated with the domain. In the subsequent sections, we refer to these parameters as the *domain-specific* parameter values. Table 5.4 in the evaluation section lists the domain-specific value ranges used by this thesis.

Intuitively, the domains of the SN could be generated by running the Waxman graph generator three times, generating each domain separately before joining the domains into a single SN graph with IDLs. Some benefits of this approach are:

- Number of SNNs in the respective domain can be provided to the Waxman graph generator. Hence, each domain graph could be generated using only a few lines of code.
- Can specify different connectivity for each domain if desired. For instance, the Waxman generator could generate each domain with different α and β values.

However, this approach considerably reduced the speed of SN generation when creating smaller graphs, like the space domain graph. This decrease in speed was due to the connectivity problem mentioned in section 4.0.3, which is more pronounced for smaller graphs due to a higher probability of a disconnected graph, resulting in several costly re-generation steps.

To illustrate this point, consider two graphs of a and b with three and four nodes, respectively, shown in figure 4.4.

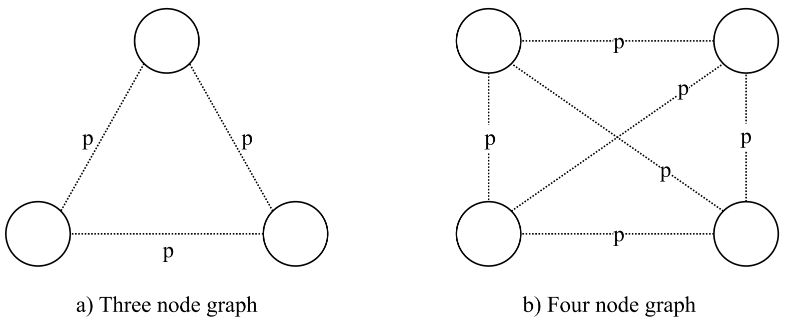

Figure 4.4: Illustration of two graphs with different probabilities of being connected.

Additionally, consider a simplified fixed connection probability between any two nodes in each graph as $p = 0.5$. The probability of a single node being disconnected from any other node is given by $(1 - p)^{n-1}$. Hence, the probability of a single node being disconnected in the graph a is given in equation 4.2 as P_a and similarly for b in equation 4.3 as P_b.

$$P_a = (1 - 0.5)^{3-1} = 0.25 \tag{4.2}$$

$$P_b = (1 - 0.5)^{4-1} = 0.125 \tag{4.3}$$

As such, it is clear that each node has a lower probability of being disconnected when considering larger graphs. Hence, when the SN was generated as three small graphs, it often resulted in either of the graphs containing multiple components. Domain re-generation was then performed multiple times, significantly reducing the generation time.

The running time was tested using sixty ground SNNs, thirty air SNNs, and ten space SNNs. Furthermore, the α and β Waxman graph parameters were set to 0.5.:

```
Generation of ground domain took 5.99 ms
Generation of air domain took 0.99 ms
Generation of space domain did not complete
```

Although the ground and air domains were generated quickly, the Waxman graph generator could not create a connected graph for the smaller space domain after running for over 15 seconds.

In order to avoid this problem, we employed a slightly different approach to generate the multi-domain SN. Instead of the abovementioned method, the SN is created as a single Waxman graph. The number of SNNs in this graph n^{waxman} is set to be equal to the sum of SNNs across all domains:

$$n^{waxman} = \eta^s + \eta^a + \eta^g = \text{Sum of SNNs in all domains} \tag{4.4}$$

Once the graph is generated, the domain graphs are extracted using a BFS node selection and the selected nodes are removed from the original graph. The selected nodes are subsequently used to form the new domain graph.

By generating the graph in this way, we ensure the Waxman graph generator can create all domains, even if some domains only include a few nodes. With the new setup, the Waxman graph generation only took $10ms$:

```
Generation of the combined graph took 9.99 ms
```

Algorithm 4.1[9] shows the steps to generate our SN. The algorithm starts by generating the entire SN graph that contains the SNNs of all domains using the Waxman graph

[9]Note that we refer to the stored configuration using the *config* keyword in our algorithms. The config can be assumed as globally accessible.

Algorithm 4.1 *GenerateSN*

```
$\alpha \leftarrow config[waxman\_alpha\_value]$
$\beta \leftarrow config[waxman\_beta\_value]$

$\eta^s \leftarrow config[number\_of\_SN\_space\_nodes]$
$\eta^a \leftarrow config[number\_of\_SN\_air\_nodes]$
$\eta^g \leftarrow config[number\_of\_SN\_ground\_nodes]$
$\omega \leftarrow \eta^s + \eta^a + \eta^g$

{Generate Waxman graph using networkx (nx)}
$A^g \leftarrow nx.generateWaxmanGraph(\omega, \alpha, \beta)$
if $isConnected(A^g)$ = false then
    return $GenerateSN()$
end if

$A^g, A^a \leftarrow splitGraph(A^g, \eta^a)$
if $isConnected(A^g)$ = false then
    return $GenerateSN()$
end if

$A^g, A^s \leftarrow splitGraph(A^g, \eta^s)$
if $isConnected(A^g)$ = false then
    return $GenerateSN()$
end if

{Set domain-specific parameter values}
$A^s \leftarrow setDomainParameters(A^s, s)$
$A^a \leftarrow setDomainParameters(A^a, a)$
$A^g \leftarrow setDomainParameters(A^g, g)$

$A \leftarrow addInterDomainLinks(A^g, A^a, A^s)$

return $A$
```

generator (line 9). We initially call this graph A^g since it will eventually represent the ground domain graph. However, at this point in the algorithm, A^g contains the nodes of all domains.

Next, if the generated Waxman graph A^g is disconnected, it is re-generated using a recursive step to avoid the previously mentioned problems of a disconnected graph (lines 10-13). Once the graph is connected, the domains are split from the graph using algorithm 4.2 (line 15, 20).

Algorithm 4.2 takes as input the graph and the number of SNNs to split from the graph. Once called, the algorithm selects an arbitrary starting SNN n_i in the graph (line 2) and then runs a BFS selection of nodes starting from n_i (line 4). The BFS algorithm stops when the required number of SNNs is picked. Using networkx, the selected nodes are

Algorithm 4.2 splitGraph

```
Input: Current ground domain graph $A^g \leftarrow \{N^g, E^g\}$, Number of nodes $\eta$ to split
 1:
 2: $n_i \leftarrow$ {Pick random node from $A^g$}
 3:
 4: $nodeSelection \leftarrow BFSNodeSelection(A^g, \eta)$
 5: $subGraph \leftarrow nx.subgraph(A^g, nodeSelection)$
 6: $A^d \leftarrow nx.Graph(subGraph)$
 7:
 8: $A^g \leftarrow nx.remove\_nodes\_from(A^g, nodeSelection)$
 9:
10: return $A^g, A^d$
```

formed into a subgraph, which includes the selected SNNs and all SNLs connecting the SNNs (line 5). The subgraph is then used to form a new graph A^d representing the new domain (line 6), and all SNNs and SNLs part of the subgraph are removed from the original graph A^g (line 8).

Algorithm 4.1 calls algorithm 4.2 twice, once for the air domain (line 15) and once for the space domain (line 20). Note that for each split, another connectivity check is required on A^g (lines 16-18,20-22). This is required since the removed nodes might split the A^g graph into two components, as shown in figure 4.5. Although this scenario is unlikely for medium to large-sized graphs, the issue should still be addressed by including the connectivity checks to avoid program exceptions during simulation.

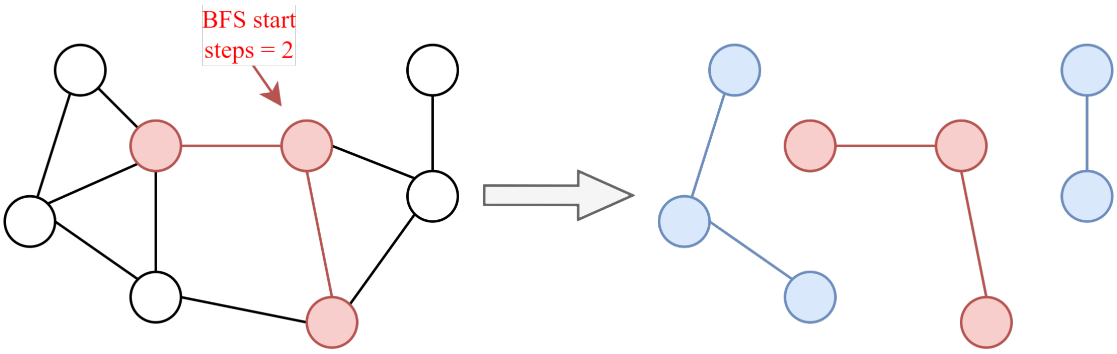

Figure 4.5: Illustration of how domain generation might create a disconnected graph.

Once all domains are generated as separate graphs, algorithm 4.3 is called (lines 26-28) to set the domain-specific parameter values of each SNN and SNL for each generated domain A^s, A^a, and A^g. This is performed by iterating over each SNN and SNL in the provided graph and then assigning them the sampled parameter values according to the configurations specified in table 5.4.

Lastly, we join the SAGIN domains into a single SN using IDLs as shown in algorithm 4.4. Ground-air IDLs are added by randomly selecting two SNNs, one from the ground

Algorithm 4.3 setDomainParameters

```
Input: SN graph $A \leftarrow \{N^A, E^A\}$, domain $d$
1:
2: for $n_i \in N^A$ do
3:     $n_i^{CPU} \leftarrow$ {Sample CPU from SN domain $d$ config}
4:     $n_i^{SLA} \leftarrow$ {Sample SLA from SN domain $d$ config}
5:     $n_i^{DOM} \leftarrow d$
6: end for
7:
8: for $e_{(v_i,v_j)} \in E^A$ do
9:     $e_{(v_i,v_j)}^{BW} \leftarrow$ {Sample BW from SN domain $d$ config}
10:    $e_{(v_i,v_j)}^{DLY} \leftarrow$ {Sample DLY from SN domain $d$ config}
11: end for
12:
13: return B
```

Algorithm 4.4 addInterDomainLinks

```
Input: Domain graphs $A^g \leftarrow \{N^g, E^g\}$, $A^a \leftarrow \{N^a, E^a\}$, and $A^s \leftarrow \{N^s, E^s\}$
1: $\eta^{(g,a)} \leftarrow$ {Get number of ground-to-air IDLs from config}
2: $\eta^{(a,s)} \leftarrow$ {Get number of air-to-space IDLs from config}
3:
4: $A \leftarrow nx.Union(A^g, A^a, A^s)$ {Join graphs using networkx}
5:
6: for $\eta^{(g,a)}$ iterations do
7:     $n_i \leftarrow$ {Pick a random node from $A^g$}
8:     $n_j \leftarrow$ {Pick a random node from $A^a$}
9:
10:    $nx.add\_edge(A, n_i, n_j)$
11:    $e_{(n_i,n_j)} \leftarrow E_{(n_i,n_j)}^{(g,a)}$ {Get the newly created edge}
12:    $e_{(n_i,n_j)}^{BW} \leftarrow$ {Sample BW from ground-to-air config}
13:    $e_{(n_i,n_j)}^{DLY} \leftarrow$ {Sample DLY from ground-to-air config}
14: end for
15:
16: for $\eta^{(a,s)}$ iterations do
17:    $n_i \leftarrow$ {Pick a random node from $A^a$}
18:    $n_j \leftarrow$ {Pick a random node from $A^s$}
19:
20:    $nx.add\_edge(A, n_i, n_j)$
21:    $e_{(n_i,n_j)} \leftarrow E_{(n_i,n_j)}^{(a,s)}$ {Get the newly created edge}
22:    $e_{(n_i,n_j)}^{BW} \leftarrow$ {Sample BW from air-to-space config}
23:    $e_{(n_i,n_j)}^{DLY} \leftarrow$ {Sample DLY from air-to-space config}
24: end for
25:
26: return $A$
```

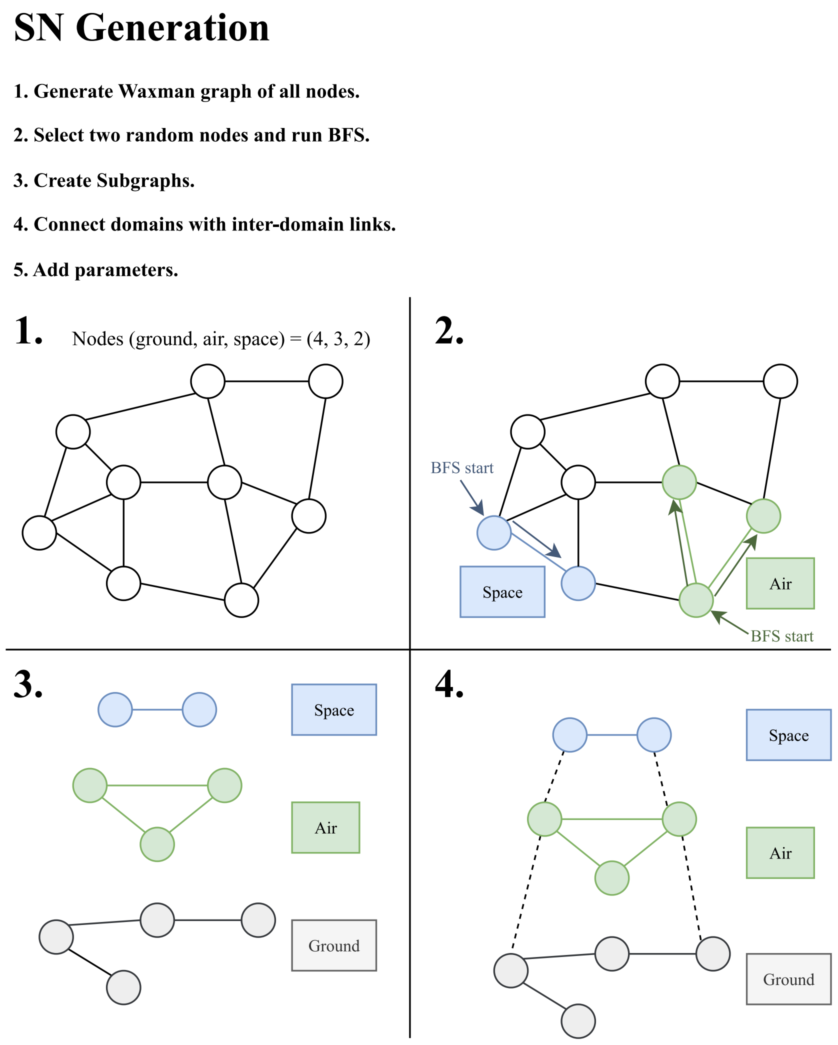
SN Generation
1. Generate Waxman graph of all nodes.
2. Select two random nodes and run BFS.
3. Create Subgraphs.
4. Connect domains with inter-domain links.
5. Add parameters.
1.
Nodes (ground, air, space) = (4, 3, 2)
2.
BFS start
Space
Air
BFS start
3.
Space
Air
Ground
4.
Space
Air
Ground

Figure 4.6: High-level visualization of SN generation.

domain and one from the air domain (lines 7-8). A new IDL is added to the graph A using networkx between the selected SNNs (line 10). The IDL is subsequently provided with bandwidth and delay sampled according to the configuration (lines 12-13). Once all ground-air IDLs are created, the air-space IDLs are added in a similar fashion (lines 16-24).

Figure 4.6 gives a high-level visualization of the abovementioned SN generation process.

Lastly, we provide a visualization of two generated SN topologies in figure 4.7 with the help of the networkx drawing method. The figure shows a small network to the left which includes only a few nodes and edges. On the right, a larger network is shown.

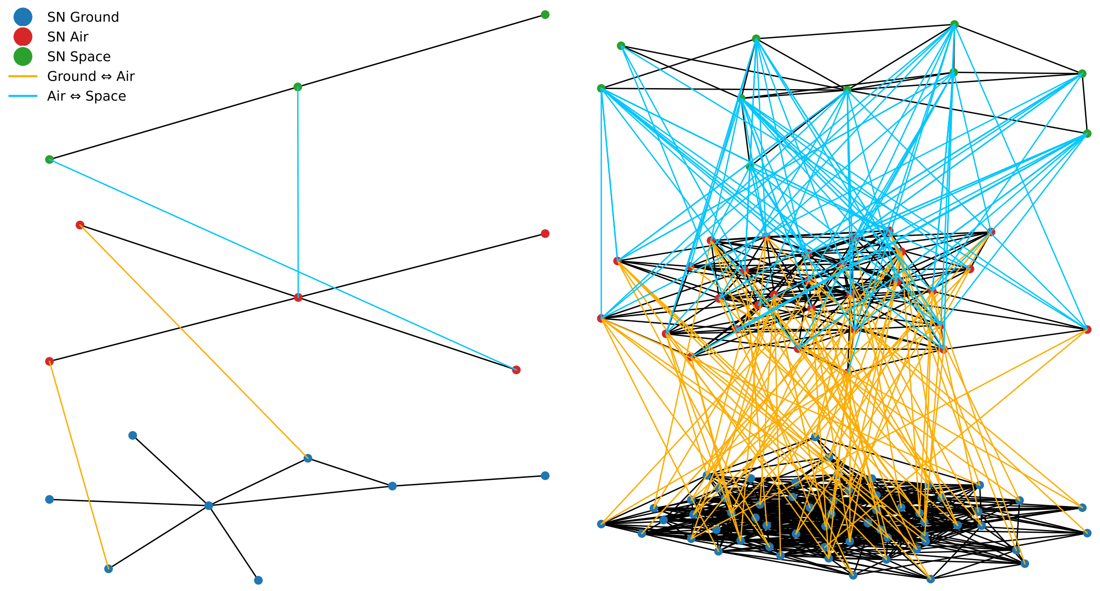

Figure 4.7: Visualization of two generated SN topologies.

4.0.4 Generating VNRs

A VNR consists of a VN, an arrival time, and a departure time. This section provides details on how the following parts are generated:

- How the VN is generated.
- VNR arrival and departure times.

Generating the VN

According to our VN definition in section 3.2.2, the VN should be represented as a linear, directed acyclic graph with parameters added to VNNs and VNLs. This graph is also created using networkx.

Algorithm 4.5 shows the steps used to generate a new VN. Initially, the length of the new VN is decided by sampling from the uniform distribution ranging from the minimum η_v^{min} and maximum η_v^{max} VN length, as specified in the configuration (lines 1-3).

Sampling from U(η_v^{min}, η_v^{max}) ensures that our VNs are of arbitrary length, which simulates the arbitrary number of VNNs required for different services.

To generate the graph, a list of VNN IDs is created by iterating from $i = 0$ up to the sampled VN length $i = VNLength$ and storing the VNN ID i at each step. Furthermore, a tuple $(i, i + 1)$ is stored to represent the VNL between the current and subsequent

Algorithm 4.5 GenerateVN

```
 1: $\eta_v^{min} \leftarrow$ {Get min VN length from config}
 2: $\eta_v^{max} \leftarrow$ {Get max VN length from config}
 3: $VNLength \leftarrow \mathrm{U}(\eta_v^{min}, \eta_v^{max})$ {Draw sample}
 4:
 5: $B \leftarrow nx.DiGraph()$ {Initialize empty graph}
 6:
 7: {Initialize empty Python lists for storing the SNNs and the SNLs}
 8: $nodes \leftarrow []$
 9: $edges \leftarrow []$
10:
11: for $i = 0$ to $i = VNLength - 1$ do
12:     nodes.append(i)
13:     edges.append((i, i+1))
14: end for
15: nodes.append($VNLength - 1$) {Add last node not included in the for-loop}
16:
17: {Add nodes and edges to $B$}
18: $B \leftarrow nx.addNodesFrom(B, nodes)$
19: $B \leftarrow nx.addEdgesFrom(B, edges)$
20: $B \leftarrow setVNParameters(B)$
21:
22: return B
```

VNN (lines 11-14). Using networkx, the list of VNN ID and VNL tuples can be added to the graph, as shown in lines 18-19. Figure 4.8 visualizes the process used to generate the VN graph.

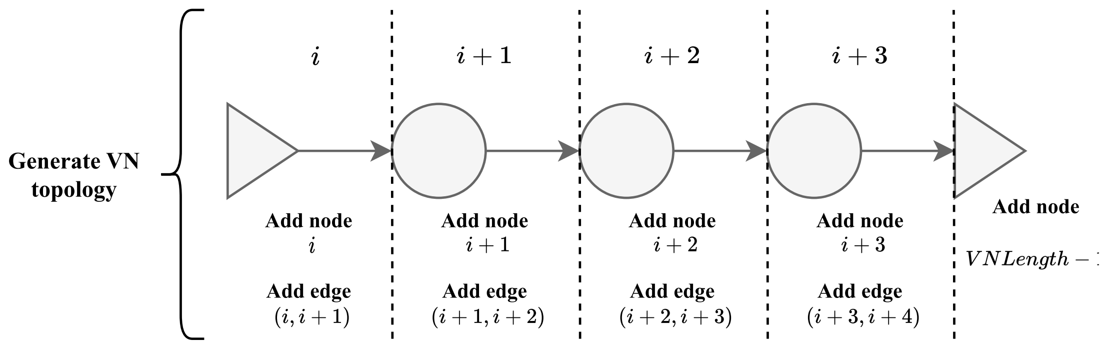

Figure 4.8: Visualization of VN generation.

The VN graph topology is finished at this stage, but its VNN and VNL parameter values are yet to be sampled and added. As such, we sample the parameter values of VNNs and VNL according to table 5.5 using algorithm 4.6.

Algorithm 4.6 takes as input the VN and iterates over all VNNs (line 2) and VNLs (line 18) and applies parameter values sampled according to the configuration.

Algorithm 4.6 setVNParameters

```
Input: VN graph $B \leftarrow \{V^B, L^B\}$

for $v_f \in V^B$ do
    if $f = 0$ or $f = |V^B| - 1$ then
        $v_f^{CPU} \leftarrow 0$
        $v_f^{SLD} \leftarrow 1$
    else
        $v_f^{CPU} \leftarrow$ {Sample CPU from VNN config}
        $v_f^{SLD} \leftarrow$ {Sample SLD from VNN config}
    end if

    if $f = 0$ then
        $v_f^{CAN} \leftarrow 1$ {Ingress candidate domain is ground only}
    else
        $v_f^{CAN} \leftarrow$ {Sample CAN from VNN config}
    end if
end for

for $l_{(v_f, v_h)} \in L^B$ do
    $v_{(v_f, v_h)}^{BW} \leftarrow$ {Sample BW from VNL config}
    $v_{(v_f, v_h)}^{DEL} \leftarrow$ {Sample DEL from VNL config}
end for

return B
```

Note that the first and last VNNs part of any VN represents the ingress and egress, respectively. Hence, we consider these VNNs to have no resource or security requirements. Moreover, ingress VNNs should only be located in the ground domain. Due to these factors, algorithm 4.6 sets $CPU = 0$ and $SLD = 1$ (line 4-5) for ingress and egress VNNs. Moreover, ingress VNNs are given a candidate domain in the ground domain (lines 11-13).

Arrival and Departure Times

Following the VNE online approach, each VNR should be associated with an arrival time and a departure time. As such, we include the following in the implementation:

1. A timer that keeps track of the simulation time. We refer to the current time at any point during training or testing as the *simulation time*.
2. A function $generateVNRs$ for generating a set of VNRs. The function should also ensure each VNR is given a simulation time for its arrival and a simulation time for its departure.

Algorithm 4.7 generateVNRs

```
Input: Number of requests to generate $\eta^R$
1: $\lambda_{Poisson} \leftarrow$ {Get arrival rate from config}
2: $\lambda_{Exp} \leftarrow$ {Get expected lifetime from config}
3: $R \leftarrow Map()$
4: $arrivalsAtTime \leftarrow Map()$
5: $departuresAtTime \leftarrow Map()$
6:
7: {Generate requests without arrival or departure times}
8: for $i = 0$ to $i = \eta^R$ do
9:     $r_i^B \leftarrow GenerateVN()$ {Create request $r_i$ and set its VN $B$}
10:    $R[i] \leftarrow r_i$
11: end for
12:
13: {Set arrival and departure times}
14: $i \leftarrow 0$
15: $time \leftarrow 0$
16: while $i < |R|$ do
17:    $nArrivals \leftarrow Poisson(\lambda_{Poisson}).sample()$
18:    $arrivalRequestIDs \leftarrow []$
19:    for $nArrivals$ do
20:        $arrivalTime \leftarrow time$
21:        $departureTime \leftarrow time + Exponential(\lambda_{Exp}).sample() + 1$
22:        $r_i \leftarrow R[i]$
23:        $r_i^{t^a} \leftarrow arrivalTime$
24:        $r_i^{t^d} \leftarrow departureTime$
25:
26:        $arrivingVNRs[time].append(r_i)$
27:        $departingVNRs[time].append(r_i)$
28:
29:        $i++$
30:    end for
31:    $time++$
32: end while
33: return $R, arrivingVNRs, departingVNRs$
```

The *generateVNRs* function is shown in algorithm 4.7, which initially creates three maps (lines 3-5):

1. R: A map using VNR ID as the key. Used for storing and retrieving a specific VNR.

2. *arrivalsAtTime*: A map using simulation time as the key. Used for storing and retrieving a list of VNR IDs arriving at a specific simulation time.

3. *departuresAtTime*: A map using simulation time as the key. Used for storing and retrieving a list of VNR IDs departing at a specific simulation time.

By using maps for storing requests, arrivals, and departures, we ensure a quick retrieval time of $\mathcal{O}(1)$ when storing and retrieving VNRs during training and testing.

Once the maps are created, algorithm 4.7 iteratively generates the VN of each VNR using algorithm 4.5 (lines 8-11). Each VN is stored in the R map with a key equal to the ID i of the current VNR.

To sample the arrival and departure times, we create two counters, one representing time (line 15) and one representing the current request (line 14). Next, we sample the Poisson distribution, which returns the number of arrivals for a single simulation time unit (line 17). Hence, if there are more than zero arrivals for the current time unit, a for-loop iteratively applies the current simulation time as the arrival time of each subsequent VNR (lines 19-27).

Moreover, whenever a VNR is given an arrival time, the departure time is also sampled using the Exponential distribution (line 21). The sample from the Exponential distribution represents the number of time units the VNR should be embedded in the SN, known as the lifetime. The departure time is calculated by adding the arrival time with the sampled lifetime.

An additional +1 is added to the departure time (line 21). This addition fixes an issue related to sampling zero lifetime from the exponential distribution, which caused some VNRs to arrive and depart at the same simulation time. These VNRs would therefore be removed immediately after being embedded and thus have no impact on the simulation. Hence, adding one ensures all VNRs have a nonzero lifetime.

4.0.5 Embedding

We refer to embedding as the process of allocating SNN and SNL resources based on VNN placement decisions made by the agent and VNL placement decisions made by a shortest path first algorithm.

By "allocating resources," we refer to reserving SN resources for a specific VNN or VNL placement.

We consider embedding in terms of the following stages:

- *The VNN embedding stage*: The DRL agent decides the SNN placement location of all VNNs, part of a VNR. If all placements are valid, the VNNs are subsequently embedded into the SNNs.

When we say "the placements are valid," we refer to whether the placements satisfy the constraints listed in section 3.2.4. We use algorithm 4.8 to verify whether a VNN placement decision is valid. The algorithm checks each constraint (lines 2-5) listed in section 3.2.4 and returns *true* if all constraints are satisfied.

- *The VNL embedding stage*: A shortest path first algorithm finds the shortest path of SNLs that can represent the VNL connections in a VN. The VNL is subsequently embedded into the chosen SNLs if a valid path is found.

Algorithm 4.8 verifyVNN

```
Input: SNN chosen by the agent $n_i$, VNN to embed $v_j$
1:
2: $CPU_{bool} \leftarrow n_i^{CPU} \geq v_j^{CPU}$
3: $STO_{bool} \leftarrow n_i^{STO} \geq v_j^{STO}$
4: $SLD_{bool} \leftarrow n_i^{SLA} \geq v_j^{SLD}$
5: $CAN_{bool} \leftarrow n_i^{DOM} = v_j^{CAN}$
6:
7: if $CPU_{bool}$ and $STO_{bool}$ and $SLD_{bool}$ and $CAN_{bool}$ then
8:     return true
9: end if
10: return false
```

The following subsections include specifics on how we handle each stage of the embedding process.

Embedding VNNs

We use algorithm 4.9 to embed valid VNN placements. The algorithm performs the following steps:

- The resources demanded by the VNN v_f are subtracted from the SNN chosen for placement (line 2). This ensures that subsequent placements cannot use the same SNN resources.

- Once the SNN resources are allocated to the VNN, we say the VNN is embedded. Algorithm 4.9 stores all embedded VNNs in a map called *storedVNNs*.

  The map stores which SNN has been used for embedding each VNN. This information is essential for the de-allocation stage later when the lifetime of the respective VNR expires. When a VNR expires, we can access each SNN used to host VNNs using the *storedVNNs* map and de-allocate the resources.

Algorithm 4.9 embedVNN

```
Input: SN graph $A \leftarrow \{N, E\}$
Input: SNN $n_i$ to embed onto, VNN $v_f$ to embed
Input: Map of all VNN embeddings called $storedVNNs$
1:
2: $n_i^{CPU} \leftarrow n_i^{CPU} - v_f^{CPU}$ {Allocate SNN resources}
3: $storedVNNs[v_f] \leftarrow n_i$ {Store node embedding for later de-allocation}
```

Embedding VNLs

Once the VNN embedding stage has concluded successfully, we move on to the VNL embedding stage. In the VNL embedding stage, each VNL, part of some VNR, must be embedded into one or more SNLs that satisfy the bandwidth and delay constraints of the VNL. Furthermore, the SNLs chosen for embedding must connect the previously placed VNNs.

Figure 4.9 visualizes the VNL embedding process using a simplified SN graph structure:

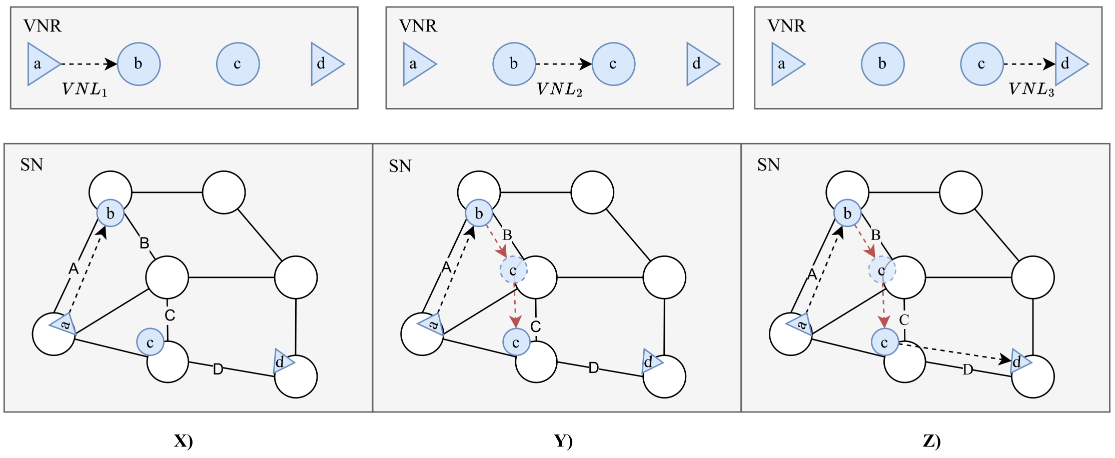

Figure 4.9: Visualization of how the VNL embedding stage can use multiple SNLs.

In part (X) of the figure, the VNN embedding stage has concluded successfully. Hence, all VNNs are embedded. Subsequently, the VNL embedding stage must now place VNL_1 to connect the ingress (a) and VNN (b). Part (X) shows a possible SNL path denoted as "A" that connects (a) and (b). We assume this is a valid path, and the VNL is embedded into SNL "A". In this case, a one-to-one embedding was found between the SNL and the VNL.

In part (Y), VNL_2 must be embedded such that VNN (b) and VNN (c) are connected. In this case, a one-to-one mapping is impossible since VNN (c) is placed so that at least two SNL links are required to embed VNL_2. In cases like this, we can use two SNLs, "B" and "C", to embed VNL_2.

Both "B" and "C" must satisfy the resource demands of VNL_2. Moreover SNLs "B" and "C" must both allocate the demanded bandwidth resources to VNL_2. Hence, the InP incurs additional costs for the extra SNL usage while no additional revenue is gained. Therefore, VNN placement decisions during the VNN embedding stage should always place VNNs closely grouped to avoid additional costs. Lastly, part (Z) of the figure embeds VNL_3 using a one-to-one mapping.

Regarding the choice of which SNLs should be used for embedding each VNL, several approaches could be applied:

- A shortest path first algorithm can be used to find a path between each pair of VNNs placed during the VNN embedding stage. This ensures the least amount of SNLs are used to embed the VNLs which minimizes the costs.
- The DRL agent could perform the placement decisions for both the VNNs and VNLs. With this approach, the agent could learn the optimal strategy for placing VNLs, potentially increasing LRC and ACR.

We choose to use the shortest path algorithm due to the following points:

- Although the DRL agent is not directly learning how to embed VNLs, it can still indirectly learn how VNLs impact performance. For instance, the agent could learn that placing multiple VNNs in close proximity can provide higher LTR and ACR. Hence, the agent should be able to learn VNN placements that reduce SNL usage and costs.
- Ease of implementation: The shortest path algorithm was chosen for its ease of implementation. This allowed for more work on optimizing the VNN placements.

We use the networkx implementation of the Dijkstra Shortest Path (DSP) algorithm to find valid SNL paths for VNLs.

The DSP algorithm takes the source and destination SNNs as input and iteratively explores the SNL weights starting from the source. The weights are stored in a priority queue which is updated based on the current shortest path. Once all SNLs are explored, the shortest path is returned.

The DSP algorithm assumes each SNL includes a weight represented by a single value. However, our SNLs are represented by two weights: the available bandwidth and the delay. Hence, we create a custom function for calculating the SNL weight.

We provide the DSP algorithm with the custom SNL weight function as shown in algorithm 4.10. The function takes the current VNL[10] $l_{(v_f,v_h)}$ being placed and the current SNL being explored $e_{(n_i,n_j)}$. Based on the resource demands of $l_{(v_f,v_h)}$, algorithm 4.11 returns *true* if the current SNL has sufficient resources or $false$ otherwise.

If the SNL has sufficient resources, then the custom SNL weight is set as one in algorithm 4.10 (line 2). Hence, the shortest path is the path with the least number of SNLs.

If the SNL has insufficient resources, then the custom SNL weight is returned as $None$ in algorithm 4.10 (line 4). This instructs the DSP algorithm to ignore the current SNL. Hence, all SNLs with insufficient resources are ignored, which ensures all SNL paths found are valid.

Algorithm 4.10 customSNLWeight

```
Input: SNL edge $e_{(n_i,n_j)}$, VNL edge $l_{(v_f,v_h)}$
1: if $verifyEdge(e_{(n_i,n_j)}, l_{(v_f,v_h)})$ then
2:     return 1
3: end if
4: return None
```

Algorithm 4.11 verifySNL

```
Input: SNL $e_{(n_i,n_j)}$, VNL $l_{(v_f,v_h)}$
1: $BW_{bool} \leftarrow e^{BW}_{(n_i,n_j)} \geq l^{BW}_{(v_f,v_h)}$
2: $DEL_{bool} \leftarrow e^{DEL}_{(n_i,n_j)} \leq l^{DEL}_{(v_f,v_h)}$
3:
4: if $BW_{bool}$ and $DEL_{bool}$ then
5:     return true
6: end if
7: return false
```

Lastly, we use algorithm 4.12 to perform the VNL embedding for an entire VNR. The algorithm iterates over each VNL part of the VNR and tries to find a valid path of SNLs using DSP. If a valid path is found for all VNLs, then the VNLs are embedded by allocating the demanded bandwidth resource from the SNLs, and we say the VNR is *accepted*. However, if no valid path is found for at least one VNL, the VNR is *rejected*, and no SNL resources are allocated.

[10]For example VNL_2 in figure 4.9.

Algorithm 4.12 embedVNLs

```
Input: SN graph $A \leftarrow \{N^A, E^A\}$, VN graph $B \leftarrow \{V^B, L^B\}$
Input: Map of all VNN embeddings called $storedVNNs$
Input: Map of all VNL embeddings called $storedVNLs$
1:
2: for $l_{(v_f,v_h)} \in L^B$ do
3:     $n_i \leftarrow storedVNNs[v_f]$ {Get SNN $n_i$ used to host VNN $v_f$}
4:     $n_j \leftarrow storedVNNs[v_f]$ {Get SNN $n_j$ used to host VNN $v_h$}
5:     $path \leftarrow nx.dijkstra\_path(A, n_i, n_j)$ {Find SNL path}
6:     if $path$ is empty then
7:         return false
8:     else
9:         for SNN source $n_a$ and SNN destination $n_b$ part of SNL in $path$ do
10:            $e_{(n_a,n_b)} \leftarrow E^A_{(n_a,n_b)}$ {Get SNL from SN}
11:            $e^{BW}_{(n_a,n_b)} \leftarrow e^{BW}_{(n_a,n_b)} - l^{BW}_{(v_f,v_h)}$ {Allocate SNL resources}
12:            $storedVNLs[l_{(v_f,v_h)}] \leftarrow path$ {Store path for later de-allocation}
13:        end for
14:    end if
15: end for
16: return true
```

4.0.6 Environment

In this section, we describe the environment used during training and testing.

Environment Interface

To implement our environment, we implement the environment interface provided by the Gym[11] RL library in Python.

Gym was chosen due to the following points:

- Gym provides a class interface for implementing the environment [32] which limits the scope of our environment implementation.
- By implementing the Gym environment interface, we can utilize the RL algorithms that are implemented in the Stable-Baselines3 (SB3)[12] Python library. The RL

[11]https://www.gymlibrary.dev/
[12]https://stable-baselines3.readthedocs.io/en/master/

algorithms, such as PPO, implemented by SB3 take an environment following the Gym interface as input. Hence, by using the Gym interface, we can utilize existing implementations of PPO and have more time to improve the model performance.

The environment interface provided by Gym requires the following environment methods to be implemented following the interface [32]:

1. *Environment initialization*: The environment initialization is used to prepare the environment for training or testing. This includes tasks such as generating the SN and VNRs.
2. *Environment step method*: For our VNE problem, each action the agent performs represents a VNN placement decision. The *step* method handles the placement decisions made by the agent. Furthermore, the method gives the agent the updated state based on the placement and the reward for the placement used during training.
3. *Environment reset method*: We iterate through a training set of VNRs and perform embedding during training. Once all VNRs are embedded, we must reset the environment state to resume training on the test set. This task is performed by the reset method [32].

Environment Initialization

When initializing the environment before training or testing, we perform the following operations:

- Generate the SN.
- Generate the VNRs.
- Initialize the current simulation time as zero.
- Get the first arriving VNR.
- Specify action space and observation space.

As specified in the last point, the Gym environment interface requires an *action space* and an *observation space* to be defined during the initialization [32].

The *action space* represents which actions can be performed by the agent [32]. For our problem, the possible actions are the possible placements of a VNN, which are all SNNs. Therefore, we define our action space as $Discrete(0, \eta^s + \eta^a + \eta^g)$ space [33]. Or in other words, each possible action is represented as a discrete value ranging from zero to the number of SNNs. Each discrete value represents the ID of an SNN.

The *observation space*[13] represents the dimensions of the observable state of the environment[14] that is provided to the agent during both training and testing to make placement decisions [32]. Any environment state provided to the agent during training or testing should adhere to the dimensions of the observation space.

Specifying the dimensions of the observation space depends on how much information we want to provide to the agent such that the agent can learn to make optimal placements. For our problem, we should at least provide the agent with information about the available SN resources and the demands of VNNs such that the agent can learn to make valid placements:

1. For every SNN, we include the available CPU, SLA, and domain to give the agent information about available resources. This extends the observation space with $4 * (\eta^s + \eta^a + \eta^g)$ entries.
2. We include the demanded CPU, SLD, and candidate domain of the next VNN to be placed, which gives the agent information regarding the placement demands. This extends the observation space by three additional entries.

Based on the observation space entries specified above, we initially have a total of $entries = 4 * (\eta^s + \eta^a + \eta^g) + 4$ entries. Hence, we define our *default* observation space as a Gym $Box(min = 0, max = 1, dim = (entries, 1))$ space [33]. Or in other words, we define our observation space as a column vector with $4 * (\eta^s + \eta^a + \eta^g) + 4$ normalized entries.

Note that we refer to this as the "default" observation space since we later experiment with adding additional entries to the observation space, which are described in the evaluation 5. However, for the purposes of describing the implementation, we stick with this "default" observation space for simplicity.

We visualize the default observation space and its connection with the policy network and action space in figure 4.10. The figure shows a state provided to the agent on the left, which has the dimensions of the observation space. The state includes three data entries for every SNN representing the available CPU, SLA, and domain. Moreover, the three final entries represent the CPU, SLD, and candidate domain of the next VNN to be placed.

[13]"Observation space" is a term defined by Gym [32][33].

[14]Which we refer to as the *state* of the environment.

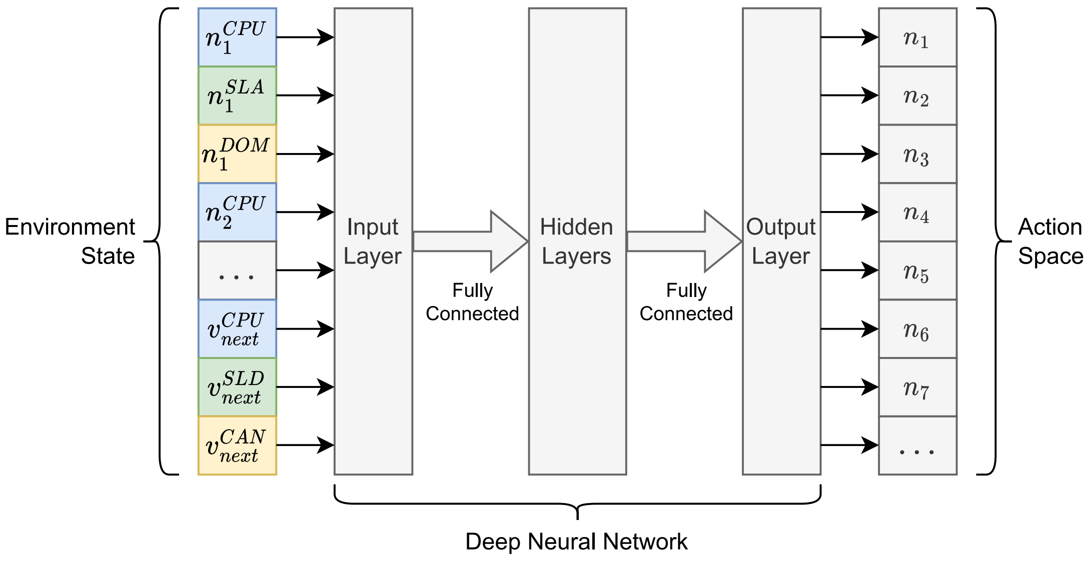

Figure 4.10: Connection between the observation space, policy network, and the action space.

The state is fed into the input layer of a deep NN, which represents the policy network, where the input layer has the exact dimensions as the observation space. Moreover, the output layer has the exact dimensions of the action space, defined as discrete values ranging from zero to the max number of SNNs. Each output value represents a possible action.

Lastly, we provide the pseudocode for initializing the environment in algorithm 4.13.

Algorithm 4.13 InitializeEnvironment

```
Input: Number of rows in observation space vector $nRows$
1: $PRNG.setSeed(config[seed])$
2: $A \leftarrow GenerateSN()$
3: $R, arrivingVNRs, departingVNRs = GenerateVNRs()$
4:
5: $actionSpace \leftarrow Discrete(0, \eta^s + \eta^a + \eta^g)$
6: $observationSpace \leftarrow Box(min = 0, max = 1, dim = calculateDim())$
7:
8: $timer \leftarrow 0$ {Simulation time}
9: $currentVNR \leftarrow getCurrentVNR()$
10: $currentVNN \leftarrow 0$
```

Algorithm 4.14 getCurrentVNR

```
while no arrivals in $arrivingVNRs[timer]$ do
    $timer++$

    {De-allocate the resources for all departing VNRs}
    if $timer$ in $departingVNRs$ then
        for departing VNR $r_i$ in $departingVNRs[timer]$ do
            $deAllocateResources(r_i)$
        end for
    end if

    {Check if the current simulation time exceeds the time of the last arrival}
    if $timer > maxKey(arrivalTimes)$ then
        return -1 {No more arrivals}
    end if
end while
$r_i \leftarrow arrivingVNRs[timer][0]$ {Get first arrival in current time}
$delete\ arrivingVNRs[timer][0]$
return $r_i$
```

Algorithm 4.13 sets the seed for all Pseudo Random Number Generators (PRNG) to the seed defined in the config and generates the SN and VNRs using this seed (lines 1-3). This seeding ensures we can re-generate identical SN and VNRs, which is useful when comparing different training setups. Next, the action and observation spaces are specified[15].

Lastly, algorithm 4.13 initializes three new variables:

- *timer*: Keeps track of the current simulation time.
- *currentVNR*: Keeps track of the current VNR. Represents the VNR that should be embedded next using the VNN and VNL embedding stages.
- *currentVNN*: The ID of the currently selected VNN. Represents the next VNN for which the agent should find a valid placement.

After the environment initialization, *currentVNR* should be the first VNR arrival. However, the first arrival is likely not at $timer = 0$. Hence, algorithm 4.14 is used to increment

[15]Note that the dimension of the observation space depends on the amount of information provided to the agent. This is specified in the configuration and is used to calculate the dimensions of the observation space.

the current simulation time until the time of the first arrival (lines 1-17). Once an arrival is found, the arrival is removed from the arrivals map (line 17). This ensures that future calls to *getCurrentVNR* will not return the same arrival.

A second important detail in algorithm 4.14 is that for every increment to the simulation time $timer++$, we check for any departing VNRs. If the current simulation time has any departing VNRs, then the resources of each VNR are deallocated by adding the allocated resources reserved by the VNR back to the SN.

Environment Step

During training and testing, the agent is given the state of the environment such that a new VNN placement is made. This results in an action being returned from the agent.

The main purpose of the *step* method is to process the action performed by the agent. By "process" we refer to the following points:

- Embed the VNN according to the placement decision.
- Embed VNLs.
- Calculate the reward that should be given to the agent based on its placement decision.
- Signal the start of a new trajectory. We consider each VNR to be a single trajectory.

We visualize the general flow of the step method in figure 4.11:

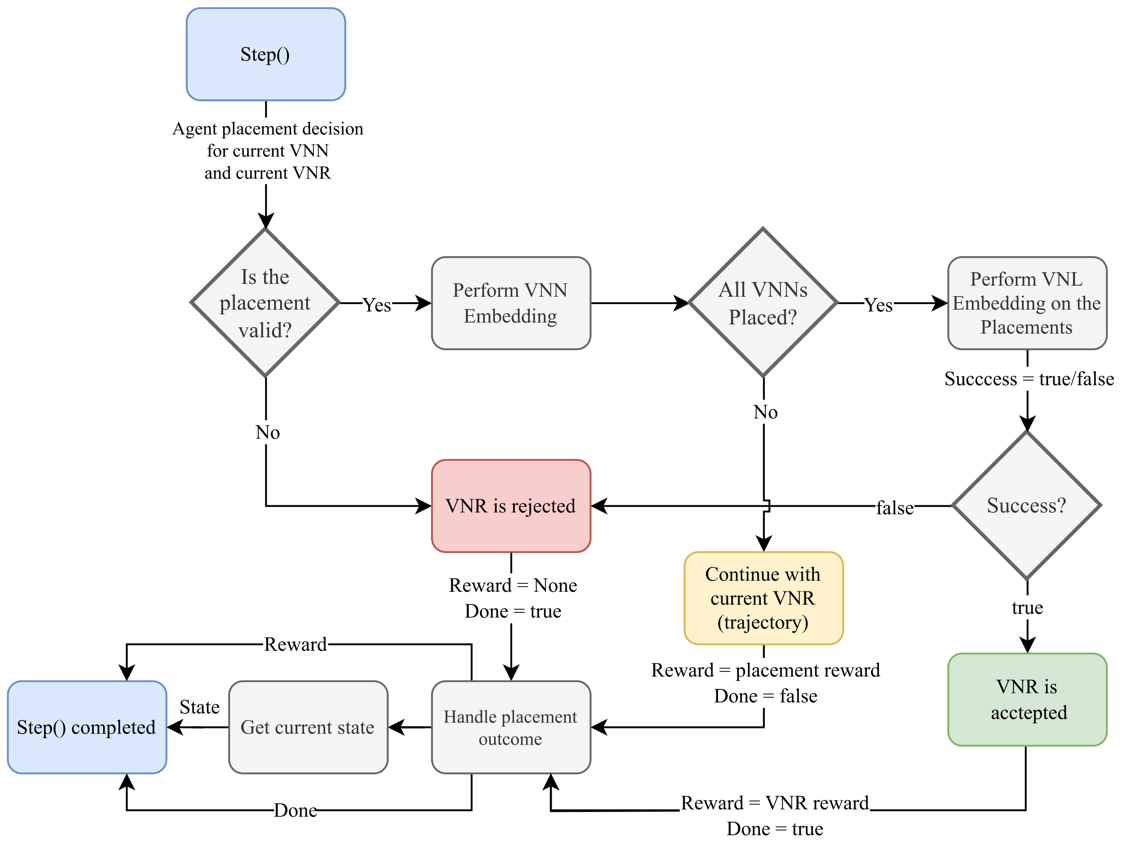

Figure 4.11: General flow of the step method.

Firstly, a *step* is triggered by the environment once the agent has performed an action. The action represents the ID of an SNN that should be used to place the next VNN part of the currently selected VNR.

The action chosen by the agent is sometimes invalid[16]. Hence, we perform a check validating whether the placement is valid:

- *The placement is invalid*: The current VNR is rejected. Get the next arriving VNR and get the first VNN part of that VNR. Generate a new state, give no reward for the invalid placement, and start a new trajectory by setting the *done* flag to true [32].
- *The placement is valid*: Proceed to VNN embedding.

Whenever we embed a valid VNN placement, we encounter the following two cases:

- *All VNNs part of the current VNR are placed*: Start the VNL embedding stage.

[16]Invalid means the VNN demands are not met by the selected SNN.

- *Not all VNNs part of the current VNR are placed*: Continue the current trajectory by setting the done flag to false and generate a new state for the next VNN part of the same current VNR. A small placement reward is given to incentivize further valid VNN placements during training.

In the case that all VNNs have valid placements, the VNL embedding stage is started. This results in two further cases:

- *One or more VNLs could not find a valid path of SNLs*: The current VNR is rejected. Get the next arriving VNR and get the first VNN part of that VNR. Generate a new state, give no reward, and start a new trajectory.
- *A valid path of SNLs was found for each VNL in the current VNR*: The VNR is accepted. Get the next arriving VNR and get the first VNN part of that VNR. Generate a new state, give a reward for the accepted VNR, and start a new trajectory.

The pseudocode for the step method is provided in algorithm 4.15. The step algorithm implements the logic described in figure 4.11.

Algorithm 4.15 step

```
Input: Action chosen by the agent represented as an ID $i$ for SNN $n_i$
1: $done \leftarrow$ false
2: $reward \leftarrow 0$
3:
4: {Is the placement valid?}
5: if $verifyVNN(n_i, v_f) =$ true then
6:   $embedVNN(A, n_i, v_f, storedVNNs)$
7:   $reward \leftarrow reward + config[VNNPlacementReward]$
8:
9:   {Are all VNNs in the current VNR embedded?}
10:  if $v_{currentVNN}$ is the last VNN in $r_{currentVNR}$ then
11:    $isValidPathFound \leftarrow embedVNL(A, r^B_{currentVNR})$
12:
13:    {Were the VNLs embedded successfully?}
14:    if $isValidPathFound$ then
15:      $reward \leftarrow reward + calculateVNRReward()$
16:    else
17:      $deAllocateResources(r_{currentVNR})$ {Release all allocated resources}
18:    end if
19:    $done \leftarrow$ true
20:  else
21:    $currentVNN = currentVNN + 1$
22:  end if
23: else
24:   $deAllocateResources(r_{currentVNR})$ {Release all allocated resources}
25: end if
26:
27: if done then
28:   $currentVNR \leftarrow getCurrentVNR()$
29:   $currentVNN \leftarrow 0$
30: end if
31: $obs \leftarrow getEnvironmentState()$
32: return reward, obs, done
```

Environment Reset

Whenever the step method completes a trajectory by returning $done = true$, the environment automatically calls the reset method [32].

The purpose of the reset, as defined by Gym, is to bring the environment state back to some starting state [32]. For example, if the environment were a chessboard part of some chess game, then the reset would be to reset the position of the chess pieces back to their starting location before starting a new game.

Similar to the chessboard example, we use the reset method to deallocate all allocated SN resources so that the SN returns to its initial state. However, we only want to reset the SN once all arrivals are accepted or rejected. Hence, the reset method serves two purposes:

1. *If not all arrivals are processed*: Retrieve next arrival and state.
2. *If all arrivals are processed*: Reset the state of the SN. Retrieve first arrival and state.

We visualize the flow of the reset method in figure 4.12. Every time the reset is called, the next arrival is retrieved. If no arrival was found, then the environment is reset to its initial state and the first arrival is retrieved. Note also that the reset method returns a new state. This is used by the agent when performing the first action in the new trajectory [32].

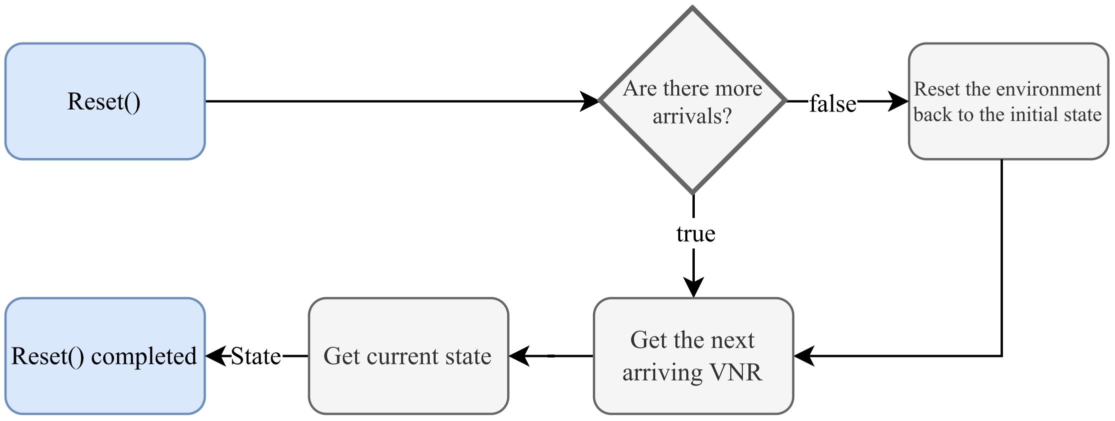

Figure 4.12: General flow of the reset method.

The pseudocode for the reset method is provided in algorithm 4.16.

Algorithm 4.16 reset

```
1: if no more arrivals in $arrivingVNRs$ then
2:     {Restore by performing a copy on the stored original copy}
3:     $arrivingVNRs \leftarrow restoreArrivals()$
4:     $A \leftarrow restoreSN()$
5: end if
6: $currentVNR \leftarrow getCurrentVNR()$ {Get next (or first) arrival}
7: return $getEnvironmentState()$
```

Calculating the State of the Environment

The state is calculated after completing every call to the step or reset methods in the environment. Subsequently, the state is provided as input to the agent's policy network, which outputs a new placement decision based on the provided state and the previous training on the policy network.

The pseudocode for retrieving the state is shown in algorithm 4.17.

Algorithm 4.17 getEnvironmentState()

```
1: $observationVector \leftarrow Vector(observationSpace)$
2: $k = 0$
3: for $n_i \in N^A$ do
4:     $observationVector[k] \leftarrow n_i^{CPU}$
5:     $observationVector[k+1] \leftarrow n_i^{SLA}$
6:     $observationVector[k+2] \leftarrow n_i^{DOM}$
7:     $k = k + 3$
8: end for
9:
10: $v_f \leftarrow currentVNN$
11: $observationVector[k] \leftarrow v_f^{CPU}$
12: $observationVector[k+1] \leftarrow v_f^{SLD}$
13: $observationVector[k+2] \leftarrow v_f^{CAN}$
```

Firstly, algorithm 4.17 creates a vector of length equal to the default observation space dimensions. Each index is subsequently filled with the available CPU, SLA, and domain of each SNN. Lastly, the demands of the next VNN are added as three additional inputs.

Note that additional entries can be provided to the state using the following steps:

1. Increase the dimensions of the observation space during environment initialization. For example, if an additional parameter is provided to each SNN, then the observation space should have a dimension of $dim = ((\eta^s + \eta^a + \eta^g) * (3 + 1) + 3)$.
2. Include the additional input in the observation space vector similar to algorithm 4.17[17].

Extending the Observation Space

One of the main issues with the "default" observation space described so far is that the agent is not provided any information regarding the available resources of SNLs. This may impact the performance negatively:

- We would like our agent to place VNNs part of the same VNR in relatively close proximity. This ensures fewer SNLs are used in the VNL embedding stage, thus reducing cost. However, the "default" observation space includes no spatial or SNL information the agent can use to learn this behavior.
- A lower ACR will likely be achieved if VNNs are placed far apart. This is because more SNLs must be used during the VNL embedding stage, increasing the probability of no valid path being found. For example, using a single SNL is more likely to meet the delay and bandwidth requirements of a VNL than using two SNLs.

Due to these issues, we experiment with adding additional information in the observation space. This may help the agent learn optimal placements in the VNN and VNL embedding stages. We list our experiments with an extended observation space in the appendix A. The optimal observation space is later used during the evaluation 5.

Calculating Rewards

A vital part of training an agent is to provide a good reward[18] to the agent after an action has been performed. The reward is vital since it defines the desired behavior of the agent. For example, if we want the agent to perform good VNN placements, where "good" refers to making only valid placements, then we give a positive reward every time the agent makes a valid embedding during training.

During our experimentation in the appendix A, we experiment with two types of rewards:

[17]Note that we implement variations of algorithm 4.17 during observation space experimentation instead of modifying the algorithm directly.

[18]Which is generated using a reward function.

1. *The VNN placement reward*: A small reward given when the agent performs a valid VNN placement. Incentivizes further valid placements. We refer to this as the "placement reward."

2. *The Accepted VNR reward*: A larger reward is given for every accepted VNR to incentivize the agent to place entire VNRs successfully. We refer to this as the "accepted reward."

4.0.7 Testing the Trained Model

After our experimentation in the appendix A, we choose the optimal model for evaluation in the evaluation section.

To test the optimal model, we run the testing algorithm 4.18.

Algorithm 4.18 testing

```
Input: Environment $Env$, Set of training VNRs $R$
  while VNR arrivals remaining in R do
    $r_x \leftarrow getCurrentVNR()$
    for VNN $v_f$ part of $r_x$ do
      $state \leftarrow Env.getEnvironmentState()$
      $sortedProbs \leftarrow model.getProbabilityDistribution(state).sort()$
      $n_i \leftarrow$ Get first valid placement based on sorted probabilities
      $embedVNN(n_i, v_f, \ldots)$
    end for
    if All VNNs part of $r_x$ have valid placements then
      $validEdgesFound \leftarrow embedVNL(Env.A, r_x^B)$
      if $validEdgesFound$ then
        VNR is accepted. Proceed to next VNR
        $r_x^{Accepted} = 1$
      end if
    end if
    VNR is not accepted. Proceed to next VNR
    $r_x^{Accepted} = 0$
  end while
```

The algorithm takes in an environment that is initialized with an identical observation space and action space as the trained model. Furthermore, a set of VNRs is given as an input. These VNRs represent our test set.

Next, the algorithm iterates over each arrival and prompts the agent for each VNN placement. The agent returns the action probabilities based on the current state, and we embed the first valid VNN part of the action probabilities. If all VNNs and VNLs are embedded successfully, we count the VNR as accepted. Otherwise, the VNRs count as rejected.

Chapter 5

Evaluation

This section provides the evaluation and covers the following:

1. SN configuration.
2. VNR configuration.
3. Environment configuration for training and testing.
4. Evaluation methodology.
5. Evaluation results.

5.1 Solution Approaches

The two solution approaches tested in this thesis are described in the following sections.

5.1.1 Proximal Policy Optimization

Section 1.3 on RL introduced the policy gradient technique for training DRL agents. This section describes the specific policy gradient algorithm used as the primary solution approach for training our DRL model, namely the DRL algorithm.

DRL is a policy gradient method. Hence, it uses a similar training method as described by equations 1.8 and 1.9 in section 1.3. However, DRL incorporates a modified score function compared to equation 1.9, which is defined as [20]:

$$\begin{aligned} L^{CLIP}(\theta) &= \mathbb{E}\left[min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}\hat{A}_t, clip(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}\hat{A}_t, 1+\epsilon, 1-\epsilon)\right)\right] \\ &= \mathbb{E}\left[min\left(r(\theta)\hat{A}_t, clip(r(\theta)\hat{A}_t, 1+\epsilon, 1-\epsilon)\right)\right] \end{aligned} \tag{5.1}$$

Equation 5.1 handles the well-known gradient ascent issue of overshooting local optimums. For example, the policy update may jump past the optimum if learning too quickly instead of reaching a maximum.Hence, equation 5.1 includes *clipping* to limit the max $1+\epsilon$ and min $1-\epsilon$ updates [21]:

$$clip\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}, 1+\epsilon, 1-\epsilon\right) \tag{5.2}$$

The new score function $L^{CLIP}(\theta)$ replaces the previous score function $J(\theta)$ in equation 1.10.

The SB3 Python library provides the implementation of DRL used in this thesis [34]. The SB3 DRL implementation takes our environment from section 4.0.6 as input and trains the agent using the environment and the DRL algorithm [35].

During the experimentation in the appendix A, we tune the SB3 DRL algorithm by changing the *hyperparameters* of the DRL algorithm. A hyperparameter refers to a training parameter that alters how the DRL algorithm performs training updates to the model. Hence, we tune the hyperparameters such that optimal training can be achieved[1]. A brief description of the hyperparameters tuned during the experimentation is listed below [20]:

1. *Timesteps total*: Every time the *step* method is called by the environment[2], an internal step counter is incremented. If this step reaches the *timesteps total*, then the current training *iteration* stops, and we save the current version of the model [34]. Afterward, we start training for another iteration to improve the model further.

   By saving the model every iteration, we can retrieve earlier versions of the same model if the model performance decreases in future iterations. We keep this parameter static throughout the experimentation to save all models in the same interval.

[1]By "optimal," we refer to a high ACR and LRC.
[2]See algorithm 4.15.

2. *Learning rate*: This is the α used in equation 1.10. We can increase this value to learn faster by making big updates to the policy[3]. However, from the experimentation in appendix A, we found that increasing the learning rate is often problematic due to large negative updates. This agent might initially show good results but subsequently makes a large negative update which ruins the performance.

3. *Training steps*: The DRL algorithm will embed VNRs during training. Each embedded VNR represents a single trajectory consisting of one step per VNN. When DRL trains on the reward received from these steps and trajectories, it uses the *training steps* to decide how many steps to select for training [34][36]. From our experimentation, we found that increasing the number of steps increased the "smoothness"[4] of the training.

4. *Training epochs*: The steps collected during the *training steps* represent a lot of computation spent embedding VNRs. Hence, we want to extract as much information as possible to learn the agent from this data. This can be achieved by increasing the DRL *number of epochs*, which makes the DRL algorithm train multiple times using the same data [20][34]. Our experiments found that increasing the epochs increases training instability[5].

5. *Training Batch size*: The number of epochs of training is not performed on the entirety of the data at once; instead, the DRL algorithm uses smaller chunks, also known as *batches* for training [36]. From our experimentation, we found that the default batch size of 64 worked best for our scenario [34]. Increasing or decreasing the batch size resulted in equal or worse ACR and LTR performance.

6. *Entropy*: When the agent chooses an action during training, the chosen action is based on the probabilities the policy network outputs. For example, if the policy network returns a maximum embedding probability on some SNN n_i, then the agent will likely choose this SNN for embedding. However, we want the agent to explore all embedding strategies to find the optimal strategy. Hence, we can lower the probability of the best action by increasing the DRL *entropy coefficient* [36].

7. *Clip*: This is the clipping ϵ shown in equation 5.2. We found the default clipping of 0.2 to give the best results [34].

8. *Gamma*: The discount factor mentioned in section 1.3.

9. *Generalized Advantage Estimation (GAE)*: Parameter used when estimating $\hat{A}$ in equation 5.2 [34][37].

[3]Note that the DRL clipping limits the updates.
[4]Policy updates are smaller.
[5]The policy changes sporadically.

The optimal configuration of the hyperparameters found during the experimentation[6]is shown in table 5.1. This configuration is used when training our testing model.

PPO Hyperparameters	
Parameter Name	**Parameter Value**
Timesteps total	10000
Learning rate	0.0001
Training steps	4096
Training epochs	8
Training batch size	64
Entropy coef.	0
Clip	0.2
Gamma	0.99
GAE	0.99

Table 5.1: PPO hyperparameters after tuning.

5.1.2 Global Resource Capacity

As a comparison against our DRL-based solution, we implement the Global Resource Capacity (GRC) heuristic algorithm proposed by Gong et al. [11].

The GRC algorithm differs from our DRL solution by being a heuristic rather than a DRL solution. Specifically, our solution bases its VNN embedding decisions on the output of a policy network. In contrast, GRC bases its VNN embedding decisions on a measure called GRC [11].

The main benefit of the GRC solution is that GRC can operate immediately on any SN topology without any training. In contrast, our solution requires extensive training and tuning of hyperparameters for each unique SN topology.

However, regarding embedding performance, our DRL solution should be able to learn how one VNR embedding can impact future embeddings. Hence, our solution should achieve better results after sufficient training. This will be tested during our comparison in subsequent sections.

The GRC measurement is defined by equation 5.3 which is provided by [11]:

$$GRC(n_i) = (1-d) \cdot X_1 + d \cdot X_2 \quad where \quad 0 \leq d \leq 1 \tag{5.3}$$

[6]See the appendix A.

$$X_1 = \frac{c_{n_i}}{\sum_{j=0}^{|N^A|} c_{n_j}} \tag{5.4}$$

$$X_2 = \sum_{e_{(n_i,n_j)} \in N_{n_i}^{adj}} \frac{e_{(n_i,n_j)}^{BW}}{\sum_{e_{(n_l,n_j)} \in N_{n_j}^{adj}} e_{(n_l,n_j)}^{BW}} \cdot GRC(n_j) \tag{5.5}$$

In equation 5.3, the notation $N_{n_i}^{adj}$ is specified as a set of SNNs that have a direct connection to n_j using an SNL denoted as $e_{(n_i,n_j)}$. Next, the notation c_{n_i} is specified as the remaining resources in the SNN n_i [11].

The main goal of equation 5.3 is to calculate a score called GRC for the available resources in each SNN n_i, which is based on the GRC of its adjacent SNNs $N_{n_i}^{adj}$ in a recursive fashion. Hence, the amount of SNN and SNL resources in the entire SN is part of the $GRC(n_i)$ calculation. Once GRC is calculated for all SNNs, the SNN with the maximum GRC value can be used to make the next VNN embedding decision [11].

The first part of equation 5.3, denoted as X_1 for simplicity, finds the available resources in n_i relative to the available SNN resources in the entire SN [11].

The second part of the equation is denoted as X_2. This part contributes to the total GRC value by summing the available resources of each SNL $e_{(n_i,n_j)}^{BW}$ between n_i and $n_j \in N_{n_i}^{adj}$. Furthermore, the GRC of the adjacent SNN n_j is found with a recursive call $GRC(n_j)$ and is used to scale the contribution of SNL $e_{(n_i,n_j)}^{BW}$ to the GRC calculation [11].

This scaling ensures that adjacent SNNs with few available resources contribute less to the GRC value of n_i. As such, if n_i is surrounded by SNNs with too few resources to support an entire VNR, n_i also gets a lower score and is less likely to be picked [11].

Lastly, we provide the pseudocode for the GRC implementation in the appendix as algorithm A.4 and algorithm A.5, which are based on *algorithm 1* and *algorithm 2* in the paper by [11].

5.2 Simulator Configuration

This section provides tables of configuration settings used during the training and testing of the DRL model.

Table 5.2 highlights the simulation configuration used by several of the related works. The listed works propose parameter value ranges suitable for simulating VNE but also state the recommended SN and VN size.

We base our simulation settings on the values proposed by the [23], [27], [2], and [17] works as listed in table 5.2.

Attributes	[23]		[27]		[2]				[17]			
	SN	**VN**	**SN**	**VN**	**SN Ground**	**SN Air**	**SN Space**	**VN**	**SN Ground**	**SN Air**	**SN Space**	**VN**
CPU	U[50,100]	U[0,50]	U[50,100]	U[0,50]	U[50,100]	U[20,40]	U[20,40]	U[1,20]	U[50,100]	U[50,80]	U[50,80]	U[1,50]
STO	-	-	U[50,100]	U[0,50]	-	-	-	-	U[50,100]	U[50,80]	U[50,80]	U[1,50]
RAM	-	-	-	-	-	-	-	-	-	-	-	-
BW	U[20,50]	U[0,50]	U[50,100]	U[0,50]	U[50,100]	U[50,100]	U[50,100]	U[1,20]	U[50,100]	U[50,80]	U[50,80]	U[1,50]
DEL	-	-	-	-	U[1,20]	U[10,30]	U[20,40]	U[1,50]	-			
CAN	-	-	-	-	-	-	-	-	-	-	-	$\{1, 2, 3\}$
SLA - SLD	0 - 3	0 - 3	0 - 3	0 - 3	-	-	-	-	-	-	-	-
Simulation												
Networks	1	2000	1	2000	1	1	1	2000	1	1	1	-
Nodes	100	U[2,10]	-	U[2,10]	60	30	10	U[2,10]	60	30	10	
Links	570	-	-	-		600		-	-	-	-	-
Connectivity	-	50%	-	-	-	-	-	-	-	-	-	50%

Table 5.2: Parameter and simulation configuration of RL- and DRL-based related works.

5.2.1 SN Configuration

We generate a single SN graph used for training and testing. The SN includes a total of 100 SNNs distributed in the ground, air, and space domains as follows:

Ground Nodes η^g	**Air Nodes η^a**	**Space Nodes η^s**	**Nodes Total**
60	30	10	100

Table 5.3: SN nodes config.

The SN is generated using algorithm 4.1, which uses the Waxman graph generator configured with $n = \eta^g + \eta^a + \eta^s$, $\alpha = 0.5$, and $\beta = 0.5$ in equation 4.1. The graph is subsequently split into the three SAGIN domains following algorithm 4.1.

Furthermore, the domains are interconnected using $\eta^{(g,a)} = 100$ ground-to-air IDLs and $\eta^{(a,s)} = 100$ air-to-space IDLs.

When the SN graph is generated successfully using algorithm 4.1, the domain-specific parameter values for the SNNs and SNLs are sampled using the distributions and value ranges specified in table 5.4. Note that the parameter value ranges for the IDLs are best characterized by the most challenging of the two connected domains. Hence, the ground-air IDLs use the air-domain SNL parameter value ranges, while the air-space IDLs use the space-domain SNL value ranges.

	Parameter	Value Range		
		Ground	Air	Space
Node	CPU	U[50,100]	U[30,70]	U[30,70]
	SLA	[1,2,3]	[1,2,3]	[1,2,3]
	DOM	[1,2,3]	[1,2,3]	[1,2,3]
Link	BW	U[50,100]	U[50,80]	U[50,80]
	DLY	U[1,20]	U[10,30]	U[20,40]

Table 5.4: SN parameter value ranges.

5.2.2 VNR Configuration

We generate a set of 1000 VNRs for training the agent called the *training set*. Another set called the *test set* of 1000 VNRs is generated for testing the trained agent. The same test set will be used to compare with GRC.

Each set is generated with unique Pseudo Random Number Generator (PRNG) seeds that ensure the uniqueness of each set. By using two unique sets of VNRs, we prove the agent is not just trained to embed a specific set of VNRs but can embed any set of VNRs optimally.

Each VNR is provided with an arrival time sampled according to the Poisson distribution with an expectation of $\lambda = 0.04$ [3][4]. This arrival rate ensures enough spacing between the arrivals so that network congestion can slowly increase[7].

Moreover, we give each VNR a departure time sampled according to an Exponential distribution with expectation $\lambda = 100$ [4]. By specifying this departure time, we ensure that VNRs have a long enough lifetime, so network congestion can slowly build during training and testing.

Network congestion is a desired effect during training and simulation. This forces the agent to use the SN resources effectively to maximize the objective. If no congestion is encountered during training, the agent might perform poorly if given a scenario with congestion that could occur in a real-world scenario.

[7]By network congestion, we refer to the lack of available SN resources when many VNRs are embedded.

In addition to the arrival time, each VNR is given a lifetime using an exponential distribution with expectation $\lambda = 100$ [4]. This ensures that some VNRs will have a long lifetime, facilitating VNR congestion over time.

Once the lifetime of a VNR expires, the VNR is de-embedded from the SN, thus releasing the allocated SN resources. Note that the expectation can be adjusted for longer or shorter VNR lifetimes. A longer lifetime can be helpful to simulate a more congested SN.

Lastly, when generating the VN for each VNR, we sample the VNN and VNL parameter values according to the value ranges shown in table 5.2.

	Parameter	**Value Range**
Node	*CPU*	U[1, 40]
	CAN	U[1,3]
	SLD	U[1,3]
Link	*BW*	U[1, 40]
	DLY	U[5, 50]

Table 5.5: VN parameter value ranges.

5.2.3 Observation Space Configuration

The observation space[8] setup used during testing is based on the best setup found during the experimentation[9]. This setup is visualized in figure 5.1.

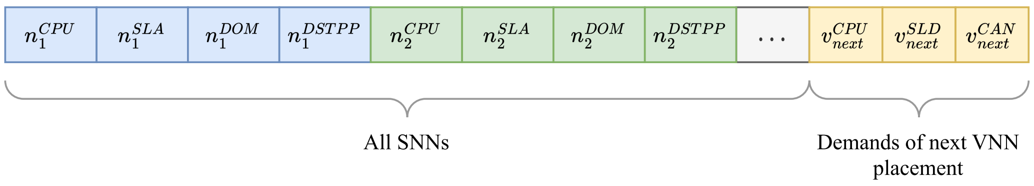

Figure 5.1: Observation space setup used for testing. Visualized as a row vector instead of a column vector for simplicity.

Figure 5.1 shows the observation space configured with the following:

- SNN available resources in each SNN required by the agent to make valid placements.

[8]See the section 4.0.6.
[9]See the appendix A.

- Demanded resources of the next VNN to be placed v_{next}. Gives the agent knowledge of the demands of the next placement.
- The Distance from the Previous Placement (DSTPP). DSTPP measures the distance to the previous VNN placement using the number of hops in the SN. The observation space is extended with a DSTPP entry for each SNN. Hence, the total number of observation space entries is $entries = (\eta^s + \eta^a + \eta^g) * 4 + 3 = (10 + 30 + 60) * 4 + 3 = 403$.

  By including the DSTPP for each SNN, the agent can learn to reduce costs by placing VNNs closer together.

  DSTPP is calculated as shown in equation 5.6. If the current VNN being placed is the first ingress VNN, then there are no previous placements, and a DSTPP of zero is returned. However, if a previous VNN placement exists, denoted as $n_{previous}$, then we return the normalized number of hops from n_i to $n_{previous}$.

$$DSTPP(n_i) = \begin{cases} \frac{SNL\_Hops(n_i, n_{previous})}{SNL\_Max\_Possible\_Hops(A)}, & n_{previous} \text{ exists} \\ 0, & n_{previous} \text{ is None} \end{cases} \tag{5.6}$$

Neural Networks Configuration

The DRL implementation by SB3 includes two NNs for which we must define the NN architecture: The policy network and the *value* network, used to predict the state value of each possible state in the environment [18][34].

Firstly, we must specify the number of input neurons for each NN, which depends on the dimensions of the observation space. For instance, the state[10] is used as an input to both the agent's policy and value estimation NNs during training [34]. Hence, we must configure the NN architectures such that the input layer has the correct dimensions to allow the state to be used as an input. Consequently, both NNs are given an input layer with 403 neurons, as shown in figure 5.2.

Furthermore, we set the number of hidden layer neurons as 128 for the policy NN and a larger value of 512 for the value-estimation NN [38].

Lastly, we set the number of output neurons in each NN according to the desired behavior of each NN:

- *Policy NN output*: The policy NN output layer outputs the probabilities of selecting each SNN in the SN using the softmax activation function. Hence, the number of output nodes must equal $\eta^g + \eta^a + \eta^s = 60 + 30 + 10 = 100$.

[10]Which has the dimensions of the observation space.

- The value-estimation NN has only a single output neuron representing the estimated state value.

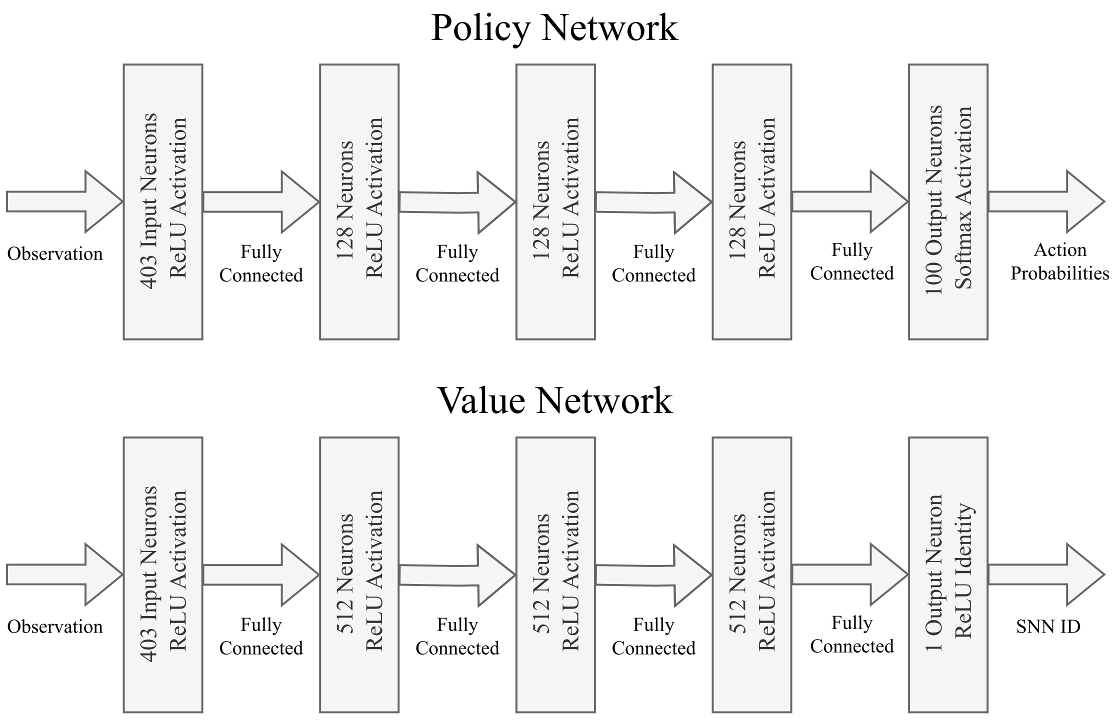
Policy Network
Observation
403 Input Neurons ReLU Activation
Fully Connected
128 Neurons ReLU Activation
Fully Connected
128 Neurons ReLU Activation
Fully Connected
128 Neurons ReLU Activation
Fully Connected
100 Output Neurons Softmax Activation
Action Probabilities
Value Network
Observation
403 Input Neurons ReLU Activation
Fully Connected
512 Neurons ReLU Activation
Fully Connected
512 Neurons ReLU Activation
Fully Connected
512 Neurons ReLU Activation
Fully Connected
1 Output Neuron ReLU Identity
SNN ID

Figure 5.2: NN architecture of policy and value networks.

5.2.4 Reward Configuration

The *reward* setup used during testing is based on the best setup found during the experimentation[11]. We use the following reward setup for testing:

- *Placement reward*: We give the DSTPP of the selected SNN as a placement reward. This incentivizes the agent to place VNNs closer together, reducing SNL usage and costs. Furthermore, the placement reward incentivizes the agent to make as many valid placements as possible since more valid placements can achieve a higher reward. This contributes to an overall higher ACR.

  The placement reward is calculated according to equation 5.7. The DSTPP is subtracted from one to reward close placements. Furthermore, the reward is divided by ten to get a maximum placement reward of 0.1. This was found to be an appropriate scale for the placement reward from the experimentation.

- *Accepted VNR reward*: We give the current LRC as a reward whenever a VNR is accepted. This incentivizes the agent to utilize the SNL resources effectively to achieve a higher LRC. The equation for LRC is shown in equation 3.25.

[11]See the appendix A.

$$\text{Placement Reward for } n_i = \frac{1 - n_i^{DSTPP}}{10} \tag{5.7}$$

5.3 Evaluation Methodology

The evaluation is divided into two main stages:

1. Evaluating the agent training performance.
2. Evaluating the trained agent.

In point one, which is listed in the appendix A, we extensively experiment with different training configurations[12], including changes to the environment and the DRL hyperparameters. The experimentation was conducted using the training set:

- Experiment with providing different rewards to the agent. The goal is to find a reward setup that encourages the agent to learn ACR and LRC.
- Experiment by adding additional information to the observation space. The goal is to find which information that is required by the agent to learn optimal placements for maximizing ARC and LRC.
- Hyperparameter tuning of the DRL hyperparameters.

Once an optimal training configuration is found, we evaluate the trained model in section 5.4. We evaluate the optimally trained model using our test set and algorithm 4.18. The trained model predicts the placement of VNNs for each VNR, and a shortest path first algorithm finds the VNL placements. Once all VNRs are accepted or rejected, we plot the ACR, LRC, and embedding time. Additionally, we evaluate whether our model outperforms the placements made by running GRC on the same problem and test set.

Lastly, we evaluate the performance of our model in a high-congestion scenario by increasing both the arrival rate and the lifetime of VNRs. This scenario is also compared with GRC.

[12]No changes are made to the SN or VNR configuration during experimentation.

5.4 Experimental Results

In the following sections, we present the results of testing our optimally trained model on our environment and a test set of 1000 VNRs. Moreover, we run the GRC heuristic on the same environment and test set, and compare the results.

5.4.1 ACR

The ACR performance is shown in figure 5.3.

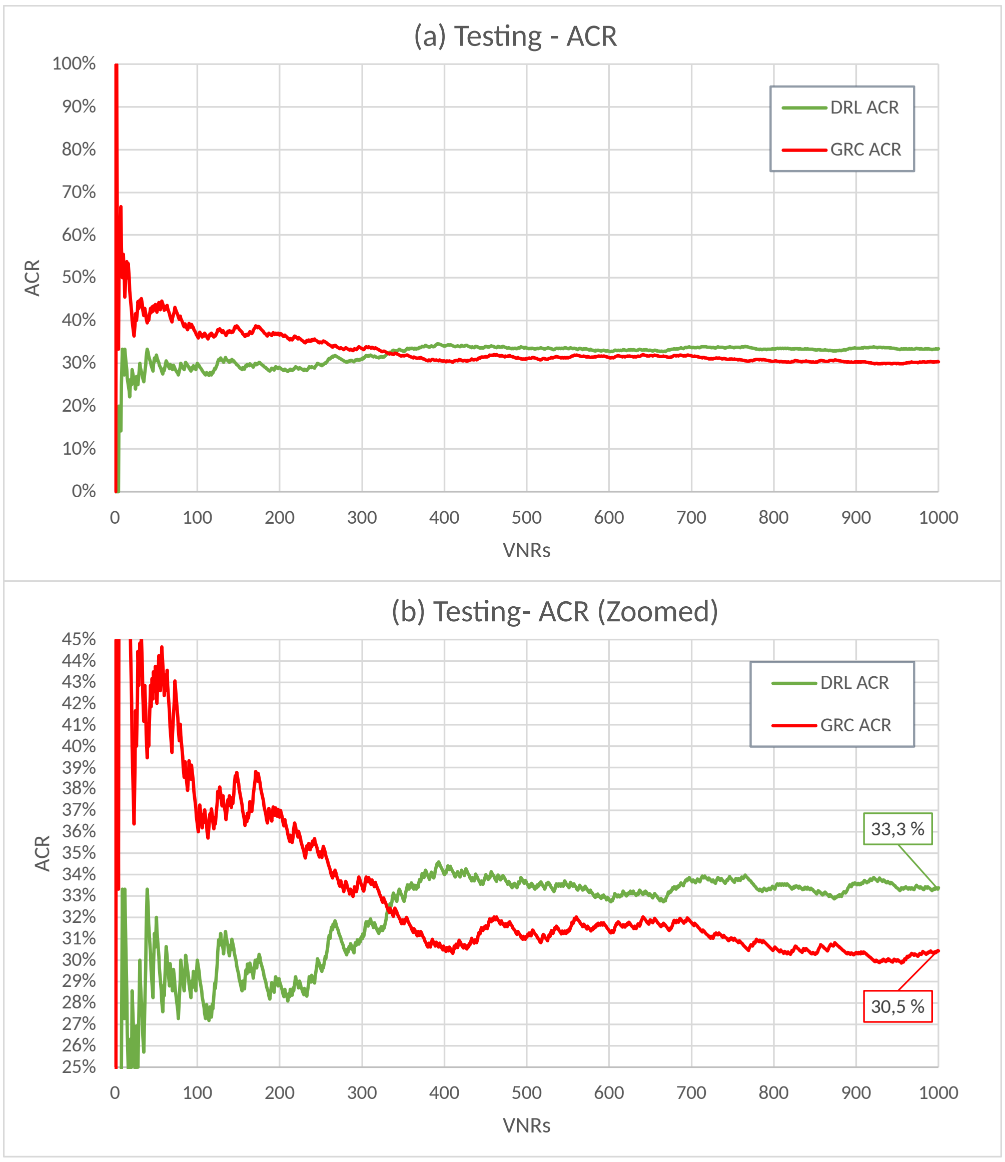

Figure 5.3: The ACR of our DRL-based solution compared to the ACR of the GRC heuristic solution.

Figure 5.3 shows the ACR of DRL in green and the GRC in red. ACR measures the ratio between accepted VNRs and arrived VNRs up to the current simulation time.

Initially (0-300 VNRs), both DRL and GRC experience instability[13] in terms of ACR. This happens because ACR is calculated from the number of accepted VNRs divided by the sum of VNRs arrived at the current simulation time. Hence, when only a few VNRs have arrived, this will result in ACR instability before more VNRs arrive.

Secondly, we notice that our DRL model initially performs worse than GRC and thereafter starts improving. This behavior is difficult to explain since the strategy utilized by the agent is based on a NN. However, we can analyze what causes the initially high VNR rejection rate using figure 5.4.

ACR Analysis - Embedding Failures

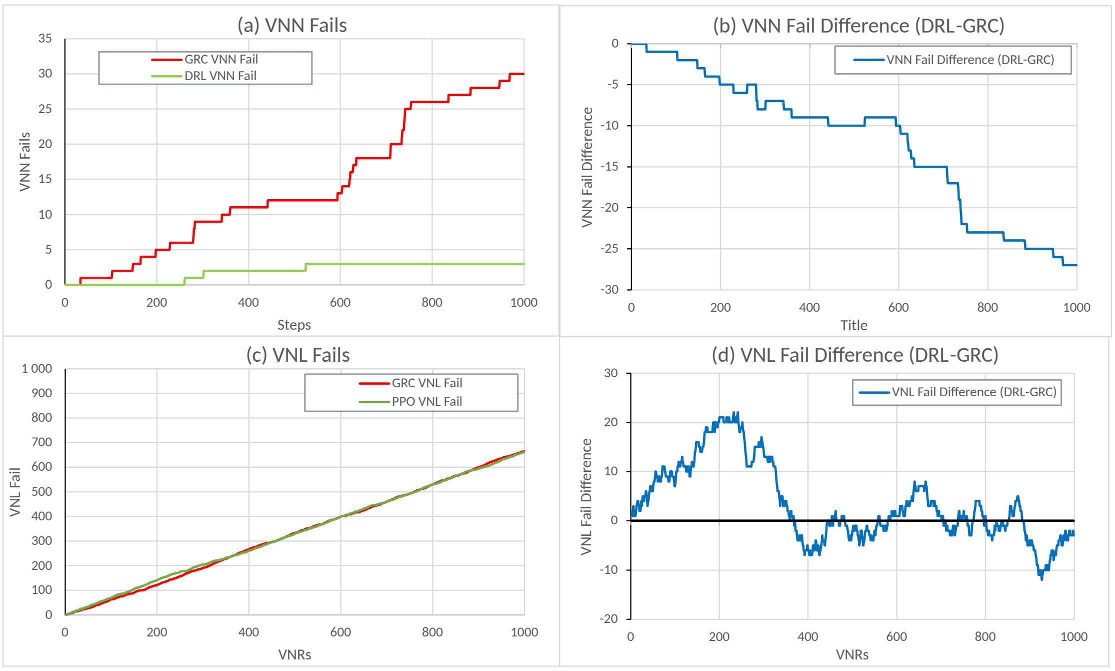

Figure 5.4: Visualization of failures caused by insufficient available resources during testing.

Figure 5.4 shows four figures (a,b,c,d):

(a) Shows the number of failures in the VNN embedding stage caused by insufficient available SNN resources.

(b) Shows the VNN fail difference (DRL - GRC) between DRL and GRC. A smaller value indicates GRC has a higher failure rate.

[13]ACR changes sporadically.

(c) Shows the number of failures in the VNL embedding stage caused by insufficient available SNL resources.

(d) Shows the VNL fail difference (DRL - DRL) between DRL and GRC. A smaller value indicates GRC has a higher failure rate. A larger value indicates DRL has a higher failure rate.

These figures give an idea of how optimally DRL and GRC place VNNs such that the resources are utilized efficiently, which allows for a higher ACR. Each VNN or VNL failure indicates a VNR has been rejected. Hence, the failure rate should ideally be as low as possible.

Figure (a) shows that DRL achieves more effective VNN resource utilization than GRC due to a lower VNN failure rate. This can also be seen in Figure (b), which shows the difference increases throughout the entire test set. Hence, VNN failures do not contribute to the initially low ACR in Figure 5.3[14].

Next, the VNL failures are shown in Figures (c) and (d). Figure (d) shows that DRL initially has a significantly higher VNL failure rate than GRC, which causes the initially low ACR in Figure 5.3.

A potential cause of the high initial VNL failure rate is that the DRL model has been overtrained on the training set. For instance, the model might favor a specific set of SNNs for embedding the VNRs in the training set. However, when given the test set, the VNR resource demands change, leading to reduced performance.

A potential cause for the subsequent decrease in VNL failures could be the following: As the resources of the favored SNNs start to decrease, the agent begins picking other SNNs. If these SNNs have more SNL connections with high bandwidth and low delay, then there will be fewer VNL failures.

ACR Analysis - VNR Rejections

Another interesting detail that can be seen in Figures 5.4 (a) and (c) is that VNL failures contribute the most to VNR rejections compared to VNN failures. After completing the test set, the number of VNR rejections caused by VNL failures is approximately 660 (66%),[15] for both DRL and GRC.

The reason for this high VNL failure rate was found to be due to challenging embedding constraints:

[14]On the contrary, it contributes to a higher ACR.
[15]See figure (c).

1. The candidate domain and ingress constraints limit VNN placements to a specific domain. Hence, when multiple VNNs are chained in a VN and are placed in different domains, they will require several IDLs during VNL embedding. This increases the challenge of finding a valid SNL path.

2. The delay parameter significantly increases the challenge of finding a valid path in the VNL embedding stage.

5.4.2 LTR

Figure 5.5 shows the LTR performance of DRL and GRC.

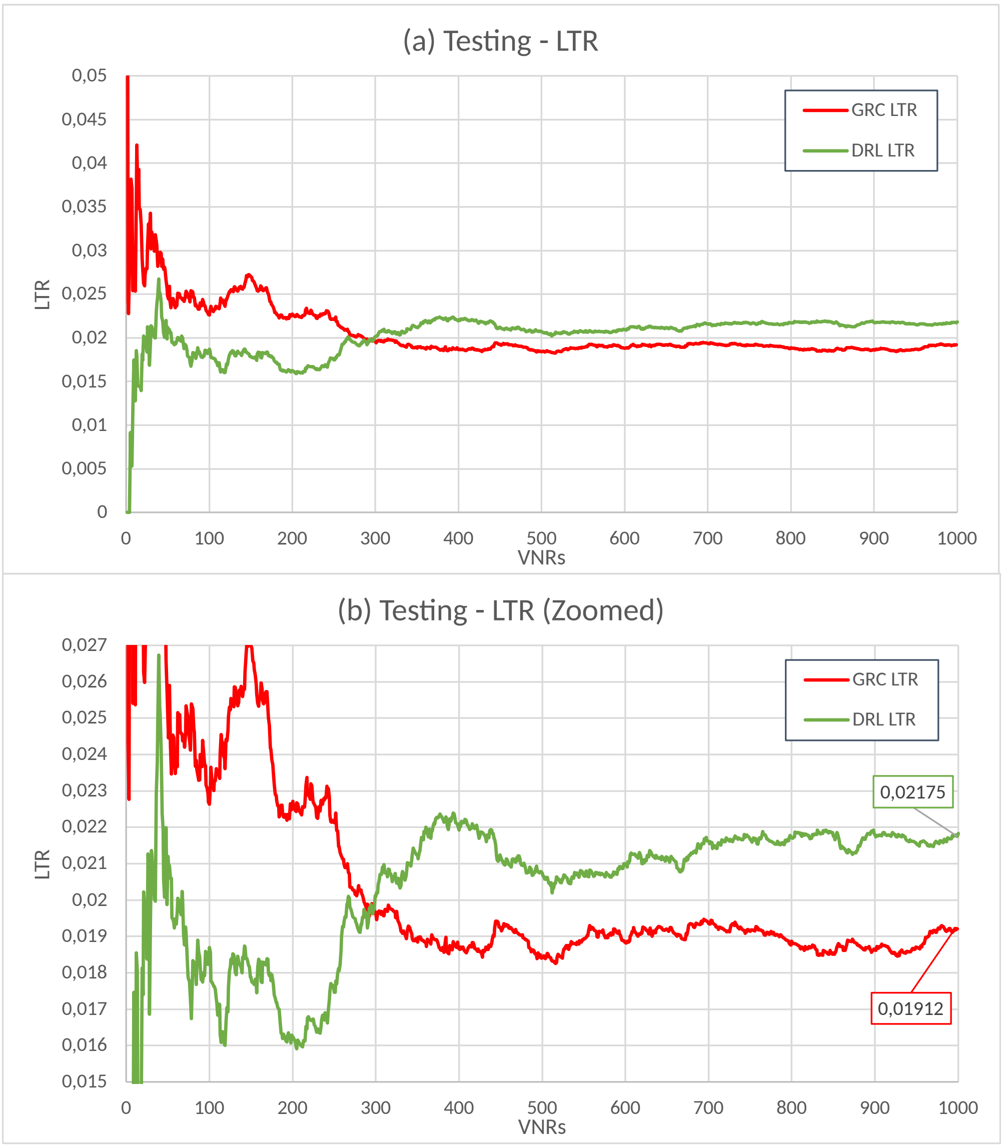

Figure 5.5: The LTR of our DRL-based solution compared to the LTR of the GRC heuristic solution.

LTR measures the revenue gained from successful VNR embeddings in the long term.

The performance of LTR follows the trend of the ACR since each accepted VNR also increases the revenue. Hence, we see that DRL eventually generates more revenue than GRC due to the higher ACR.

We also note the high initial instability of LTR, which is caused by the following:

1. Initial instability in ACR.
2. LTR varies depending on the number of VNNs in each VNR and the amount of demanded resources in each VNN. For example, a VNR with eight VNNs and high resource demands will generate more revenue than a VNR with three VNNs and low resource demands.

Overall, our DRL solution can outperform GRC regarding LTR due to the higher ACR achieved by the DRL solution.

5.4.3 LTC

Figure 5.6 shows the LTC performance of DRL and GRC.

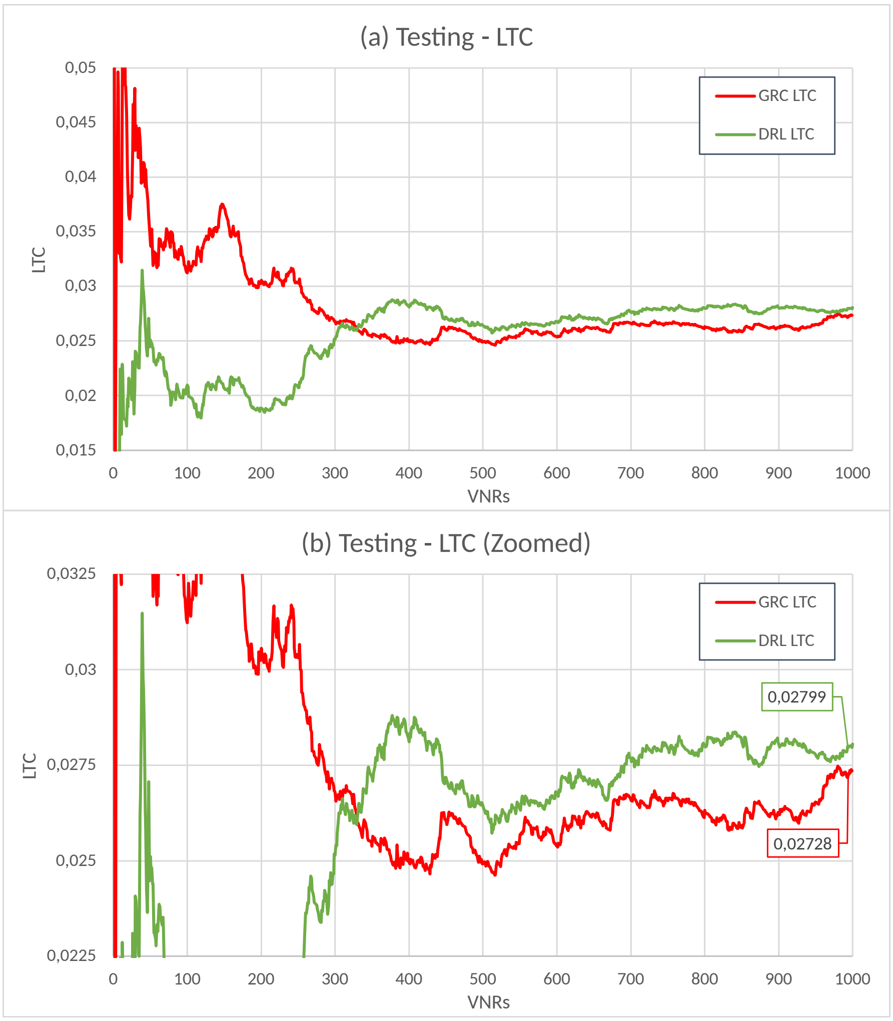

Figure 5.6: The LTC of our DRL-based solution compared to the LTC of the GRC heuristic solution.

LTC measures the operational costs associated with accepted VNRs. A cost reduction can be achieved using fewer SNLs in the VNL embedding stage.

Similar to LTR, the LTC metric also depends on the ACR. Hence, we see that PPO has slightly more costs than GRC.

At the end of the test set (900-1000 VNRs), GRC and DRL experience an increase in cost, likely due to the reduction in available SN resources causing longer SNL paths during VNL embedding and more costs. However, we also see that DRL only experiences a slight increase while GRC has a more significant increase in cost. Hence, this indicates DRL is better at resource allocation to reduce costs in the long term.

5.4.4 LRC

Figure 5.7 shows the LRC performance of DRL and GRC.

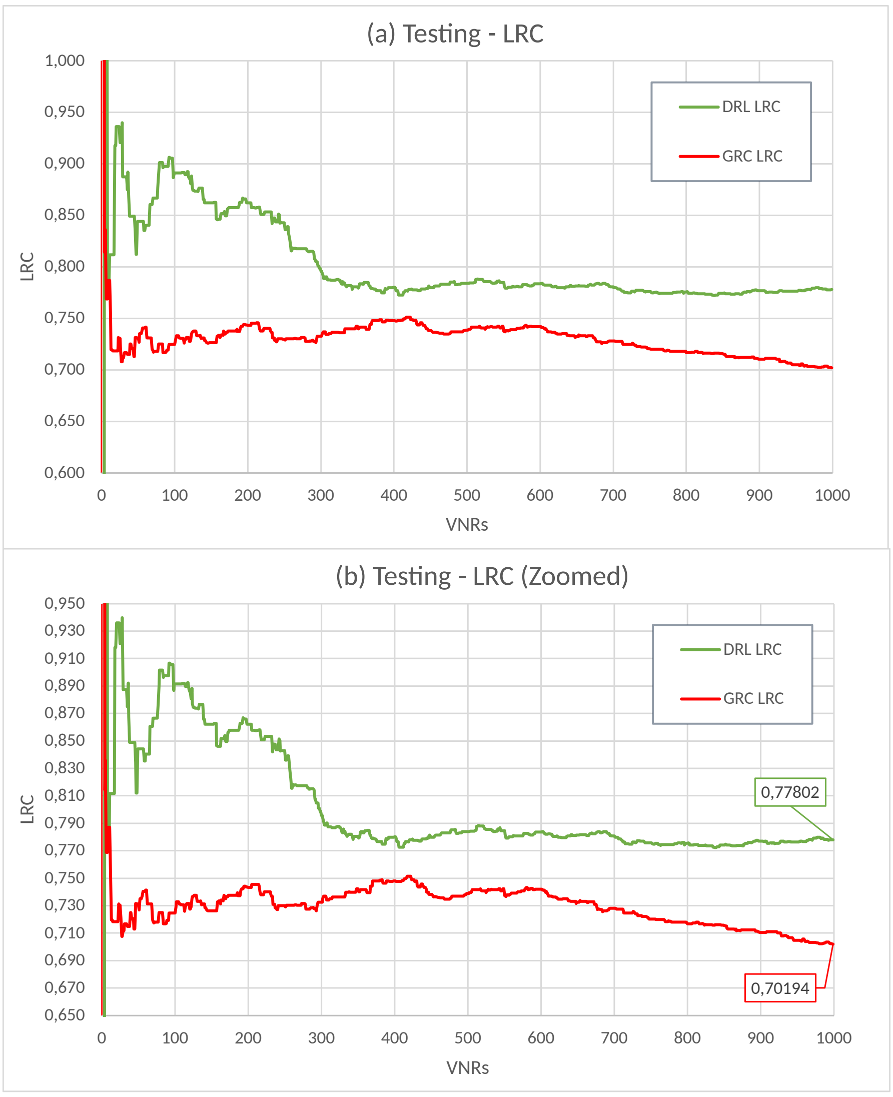

Figure 5.7: The LRC of our DRL-based solution compared to the LRC of the GRC heuristic solution.

LRC measures the ratio between the LTR and LTC and is effective for inferring which approach achieves the maximum profits by reducing SNL usage in the VNL embedding stage.

The following can be used as an indicator of what the LRC value represents:

- $LRC = 1$: Indicates an on-average one-to-one mapping between VNLs and SNLs in the VNL embedding stage. This is an improbable LRC value for our problem due to the candidate domain and ingress constraints that cause increased SNL usage.

For example, an ingress v_1 must be placed in the ground domain, and a subsequent VNN v_2 arbitrarily has space as its candidate domain. In this scenario, at least two IDLs are required to represent the VNL $l_{(v_1,v_2)}$.

- $LRC > 1$: Indicates that, on average, the VNL embedding stage can represent VNLs using no SNLs. This results from placing multiple VNNs, part of the same VNR, onto the same SNN. However, achieving an $LRC > 1$ is also very unlikely for our problem due to the reasons described in the previous point.
- $LRC < 1$: Indicates that, on average, the VNL embedding stage uses more than one SNL to represent each VNL.

Figure 5.7 shows that DRL initially achieved a very high LRC at around 0.9. This indicates the agent favors low-DSTPP placements. By placing VNNs closely together, the number of SNLs used in the VNL embedding stage is reduced, thus reducing costs.

However, DRL subsequently stabilizes at a reduced LRC of around 0.77. This decrease is likely due to the resource exhaustion of SNNs with a low DSTPP which causes the agent to use SNNs with a higher DSTPP. Hence, more SNLs must be used to represent VNLs, thus increasing the cost.

Overall, our DRL solution is able to outperform GRC in terms of LRC consistently. Moreover, while GRC experiences a negative LRC trend in the long term, DRL is able to maintain a consistently high LRC.

5.4.5 VNR Embedding Time

Figure 5.8 shows the time spent embedding each VNR for the DRL and the GRC approach in milliseconds. Note that the figure does not include the time spent on rejected VNRs. Hence, if a VNR is rejected, the plot will have a flat area on the curve for that VNR.

For DRL, a timer is started when a new VNR is selected. The timer is stopped once the DRL model has made all embedding predictions and both the VNN and VNL embedding stages have concluded successfully.

For GRC, a timer is started when a new VNR is selected. The timer is stopped once GRC has calculated the GRC value for all SNNs and both the VNN and VNL embedding stages have concluded.

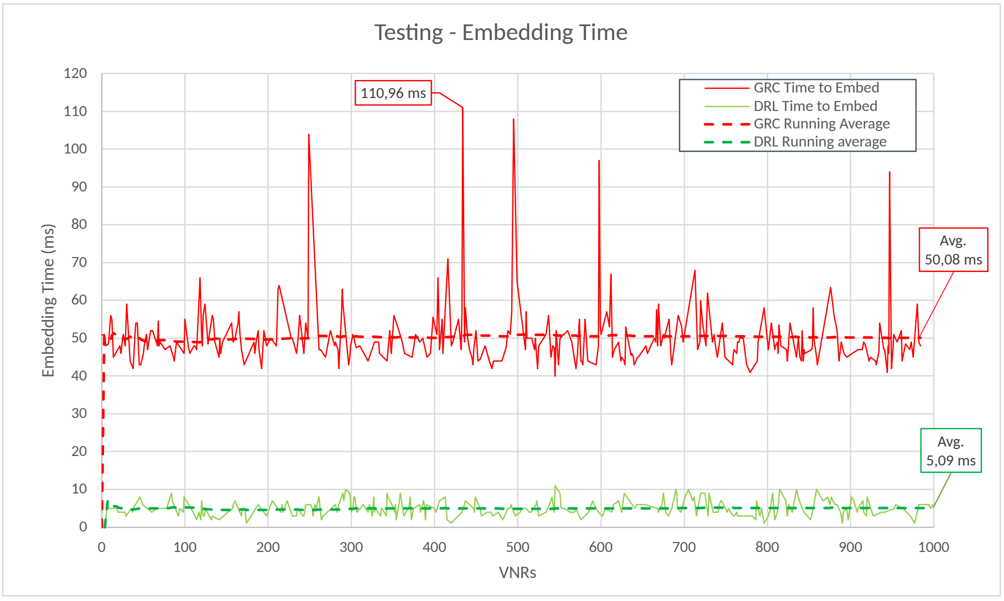

Figure 5.8: The time spent embedding each VNR in the test set for both solution approaches. The dashed line shows the running average.

Figure 5.8 shows that our DRL solution can consistently outperform GRC regarding running time. We achieve an average of $\frac{50.08}{5.09} = 9.83$ times faster embedding time.

Whereas GRC requires significant computation to find the GRC value of all SNNs, our solution only requires one forward pass of the policy NN to find a suitable placement.

5.4.6 Increased Congestion

To evaluate the performance with heavy VNR congestion, we vary the VNR lifetime and arrival rate as shown in table 5.6.

We evaluate using the same model used in the previous evaluations[16]. Moreover, we limit our analysis to ACR and LRC for simplicity and since the LRC metric captures LTR and LTC.

Arrival Rate λ	Lifetime λ			
0.04	100	150	200	250
0.05	100	150	200	250

Table 5.6: Evaluated VNR congestion scenarios.

[16]Trained on an arrival rate of 0.04 and an expected VNR lifetime of 100.

ACR Scenario 1

We show the ACR performance of arrival rate $\lambda = 0.04$ with all four lifetimes in figure 5.9. Moreover, we plot the converged[17] ACR values of GRC and DRL in figure 5.10.

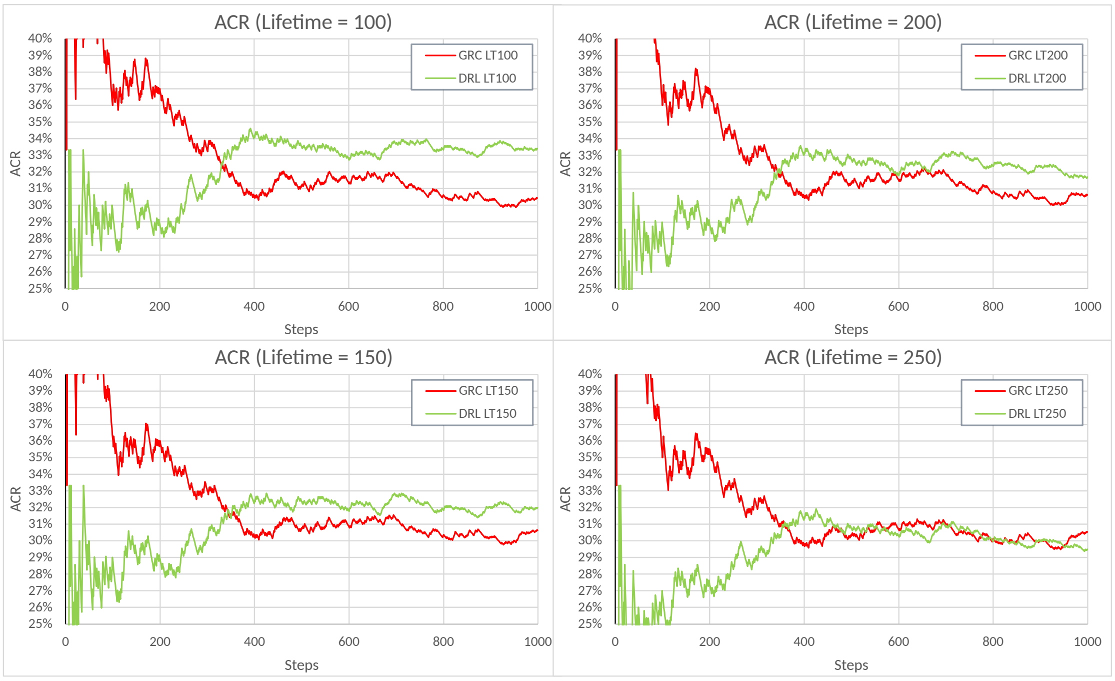

Figure 5.9: ACR for DRL and GRC with arrival rate $\lambda = 0.04$ and expected VNR lifetime of 100, 150, 200, and 250.

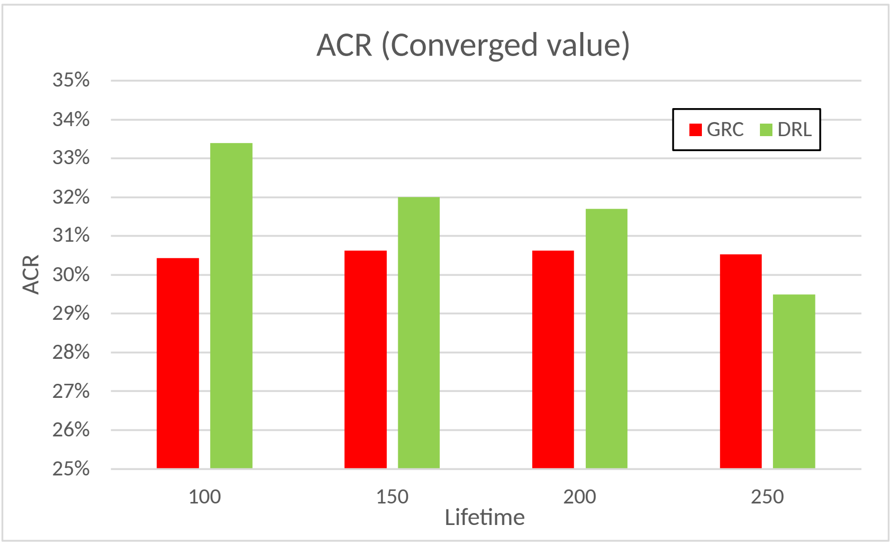

Figure 5.10: ACR for DRL and GRC with arrival rate $\lambda = 0.04$ and expected VNR lifetime of 100, 150, 200, and 250.

[17] By "converged," we refer to the last value in the 5.6 line plots.

Figures 5.9 and 5.10 show that the relative ACR performance of DRL is high in scenarios with 100, 150, and 200 VNR lifetime. However, when the VNR lifetime is set to 250, GRC starts to outperform DRL in terms of ACR.

This decrease in performance is likely due to the DRL model being trained on the scenario with an arrival rate of 0.04 and a lifetime of 100. Hence, when the lifetime is increased, the agent is placed in an environment that differs from the training, resulting in decreased performance.

LRC Scenario 1

We show the LRC performance of arrival rate $\lambda = 0.04$ with all four lifetimes in figure 5.11. Moreover, we show the converged LRC values in Figure 5.12.

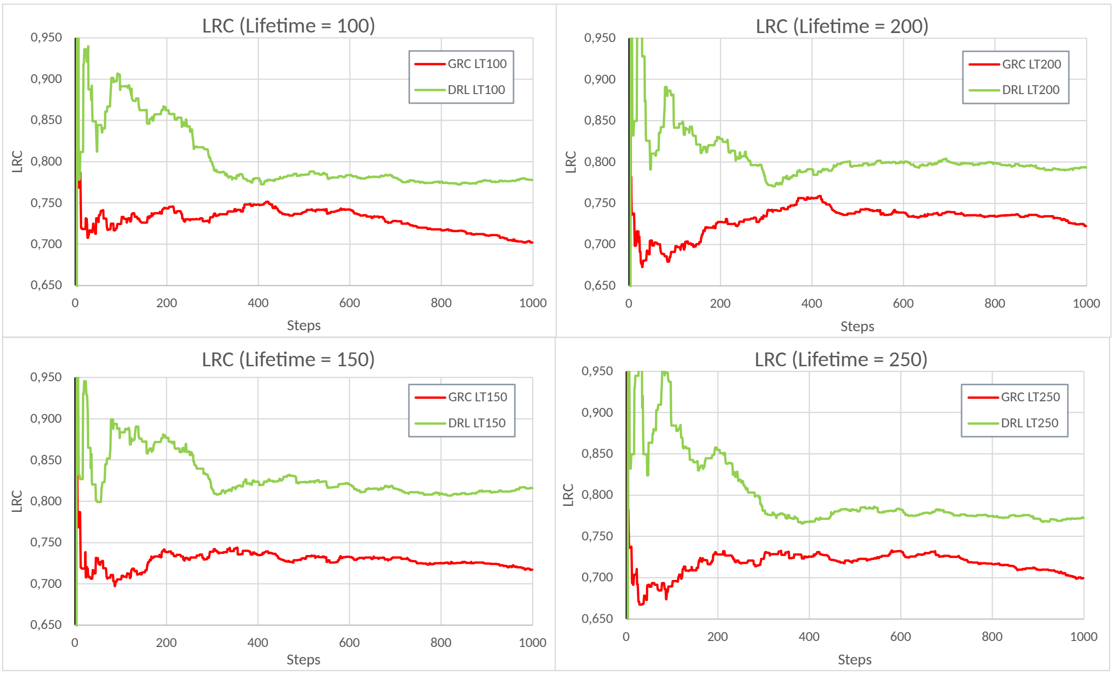

Figure 5.11: LRC for DRL and GRC with arrival rate $\lambda = 0.04$ and expected VNR lifetime of 100, 150, 200, and 250.

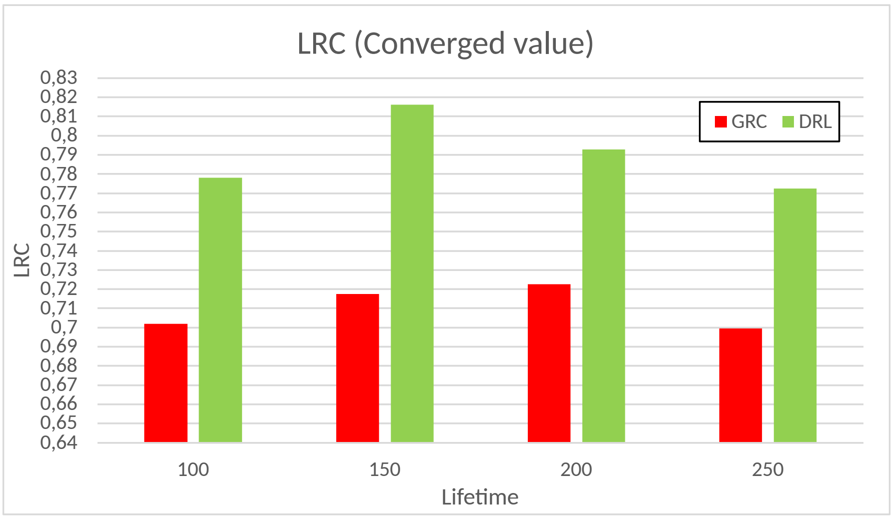

Figure 5.12: LRC for DRL and GRC with arrival rate $\lambda = 0.04$ and expected VNR lifetime of 100, 150, 200, and 250.

Figures 5.11 and 5.12 show that DRL achieves a higher relative LRC performance than GRC for all congestion scenarios. This indicates that DRL can make better placements by limiting additional SNLs and reducing costs.

ACR Scenario 2

We show the ACR performance of arrival rate $\lambda = 0.05$ with all four lifetimes in figure 5.13. Moreover, we show the converged ACR values in Figure 5.14.

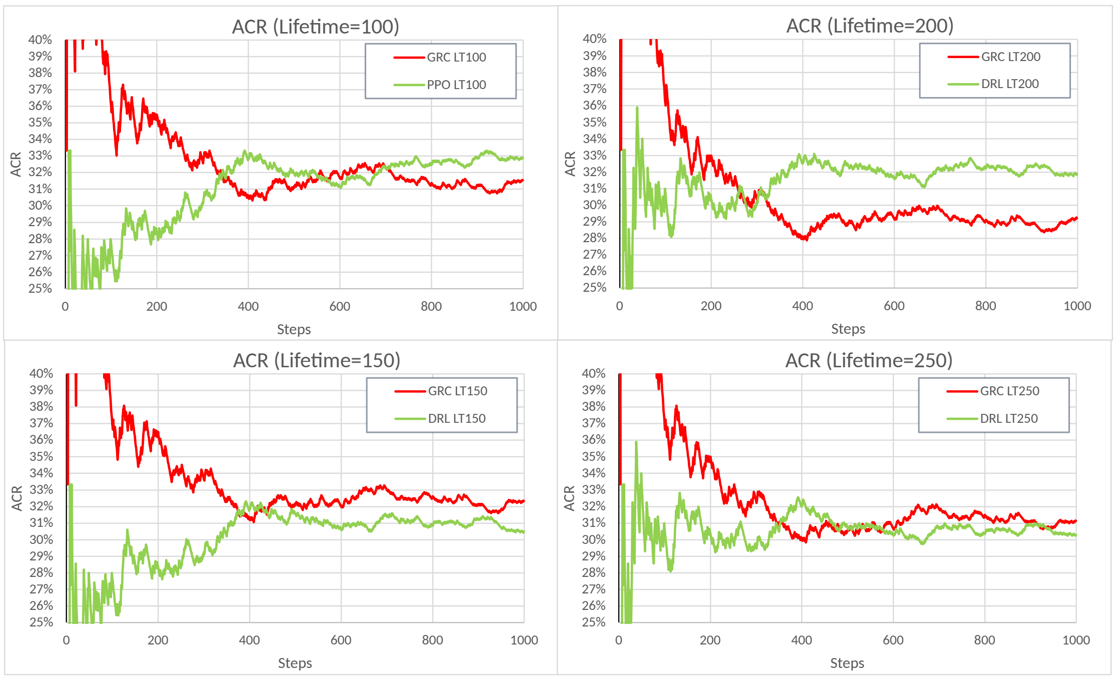

Figure 5.13: ACR for DRL and GRC with arrival rate $\lambda = 0.05$ and expected VNR lifetime of 100, 150, 200, and 250.

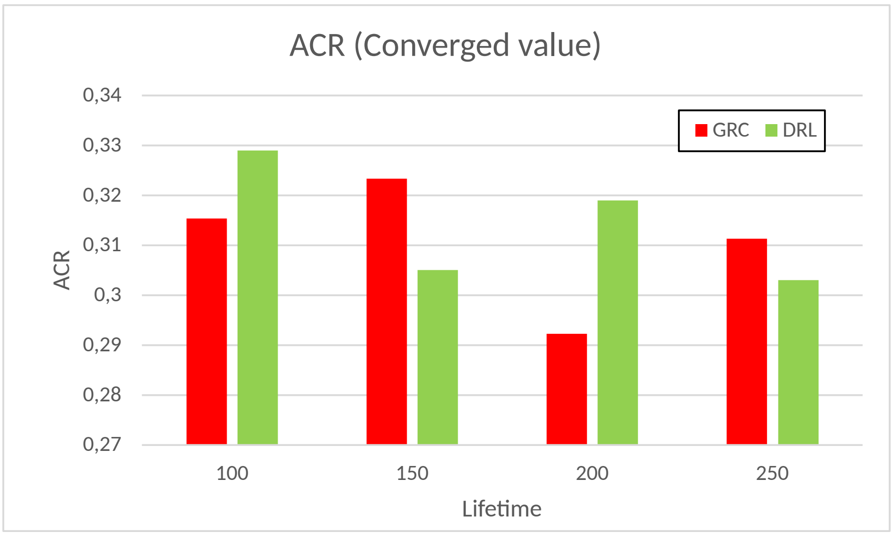

Figure 5.14: LRC for DRL and GRC with arrival rate $\lambda = 0.05$ and expected VNR lifetime of 100, 150, 200, and 250.

Figures 5.13 and 5.14 show that DRL experiences varied ACR performance relative to GRC when the arrival rate varies. This is likely due to the model being trained on an arrival rate of 0.04, which causes a decrease in performance when placed in an environment with an arrival rate of 0.05.

LRC Scenario 2

We show the ACR performance of arrival rate $\lambda = 0.05$ with all four lifetimes in figure 5.15. Moreover, we show the converged ACR values in Figure 5.16.

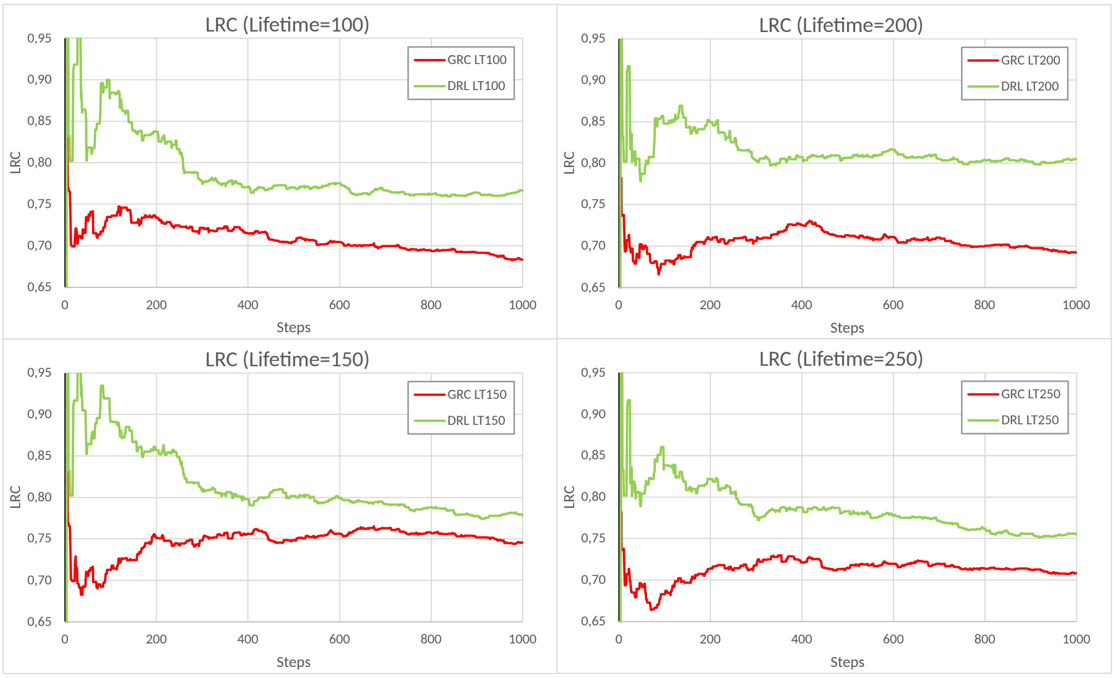

Figure 5.15: LRC for DRL and GRC with arrival rate $\lambda = 0.05$ and expected VNR lifetime of 100, 150, 200, and 250.

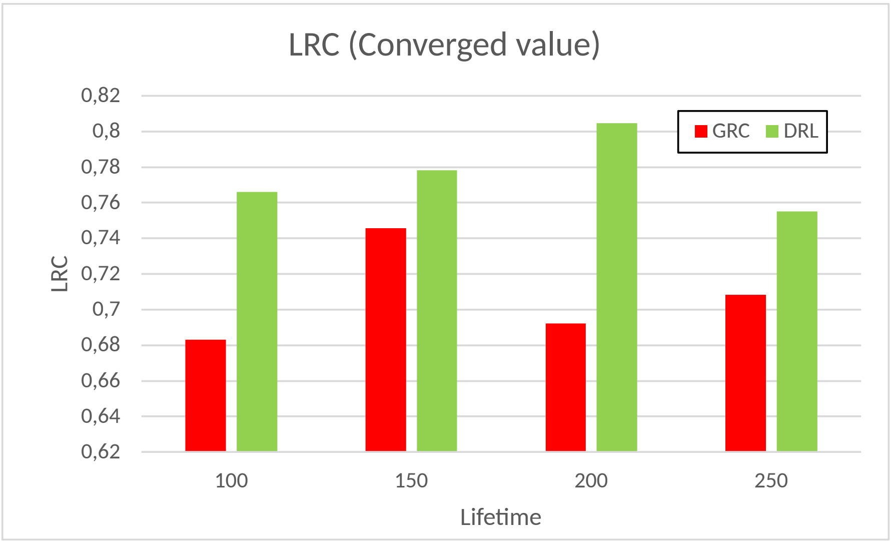

Figure 5.16: LRC for DRL and GRC with arrival rate $\lambda = 0.05$ and expected VNR lifetime of 100, 150, 200, and 250.

Figures 5.15 and 5.16 show that DRL achieves good LRC performance compared to GRC, even when the arrival rate and lifetime are varied.

Chapter 6

Conclusions

In this thesis, we have provided a solution to the security-aware VNE problem in SAGIN using a DRL-based solution.

We have provided the background on NFV, a technology enabling fast deployment of network services using VNFs, offering high scalability, high resiliency, and numerous other benefits. Furthermore, we have provided the necessary background for understanding the potential security vulnerabilities faced by InPs and VNFs, introduced by NFV. Additionally, we introduced the concept of SAGIN, where NFV is combined with multi-domain hardware to form an integrated network capable of providing global network communications services.

We have formulated our problem as a VNE problem which includes the following considerations:

- We considered three distinct domains interconnected by inter-domain links representing a SAGIN SN. Each domain includes different CPU and bandwidth resources that restrict VNN placements. Furthermore, we considered that each domain has a different amount of delay on SNLs and delay demands on VNLs. We found this to increase the difficulty of the VNL embedding stage considerably.
- We considered candidate domains, which allow VNRs to specify the desired domains for each VNF.
- We considered the placement of ingress and egress VNNs to represent the ingress and egress points of traffic in network services.
- We considered security using three levels of SLA in SNNs and three levels of SLD in VNNs.

To solve the VNE problem, we utilized the DRL approach, which is used to provide fast and highly optimal solutions to complex problems such as security-aware SAGIN VNE.

We trained our DRL agent using the PPO algorithm implemented by SB3 [34] on a custom environment. We designed the custom environment to contain the multi-domain SN, generated using algorithm 4.6. Furthermore, the environment simulates online VNR arrivals following the online approach using arrival times and departure times.

We experimented with different training setups, including PPO hyperparameters and environment configurations, to find the optimal setup for maximizing the ACR, LTR, LTC, and LRC metrics. Once an optimal configuration was found, we tested and evaluated the model by comparing the evaluation metrics against the well-known GRC heuristic solution approach.

The trained model achieved increased performance compared to GRC regarding ACR, LTR, LTC, and LRC. The model also achieved competitive performance when evaluated in various high-congestion scenarios. In addition to the increased performance, our solution achieved these results with an approximately ten times reduction in the running time compared to the GRC solution approach.

6.1 Future Directions

Although our trained model can achieve increased performance over GRC, there is still more room for improving the model. Hence, one area of future research is to experiment with providing additional or different rewards for the agent to increase the ACR and LRC performance. Moreover, the information in the state provided to the agent could also be experimented with further. The better the reward and observable state are at describing the environment and embedding goals, the better the training and subsequent testing performance will be.

Another area of future research is to simulate security-aware SAGIN VNE using a realistic cost and reward scaling based on real-world data. For instance, our table 3.8 considers an equal cost and reward for all SN resources since these may be set differently for each InP. Hence, future work could try to simulate the problem using the actual revenue and cost scalings used by an InP.

Appendix A

Improving the Training Performance

This section gives a brief overview of the experimentation and testing phase. In this phase, we train our DRL model using PPO on a training set of 1000 VNRs and vary the following settings:

- Rewards used to train the model.
- Information provided in the observation space.
- Hyperparameters part of the PPO training algorithm.

A.1 Initial Setup

Table A.1 shows the initial environment configuration.

Environment Config	
Config Name	**Config Value**
VNN Placement Reward	0.001
VNR Accepted Reward	LRC
State Vector Inputs	Available resources in all SNNs Demanded resources of next VNN SNN Average Distance

Table A.1: Environment configuration.

We start with a small VNN placement reward and LRC as the accepted VNR reward.

Furthermore, the environment observation space includes the SNN available resources, and the next VNN demanded resources, as specified in section 4.0.6. Additionally, we experiment by including an Average Distance (AVD) in the observation space, as suggested by [17].

Wang et al. [17] proposes to include the AVD in the state and defines the AVD as follows:

$$AVD(n_i) = \frac{\sum_{j=0}^{|N^A|-1} SNL\_Hops(n_i, n_j)}{|N^A|} \tag{A.1}$$

The AVD measures how "central" each SNN is compared to all other SNNs, where "central" refers to having few hops to other SNNs [17]. The goal of AVD is that the agent will learn to use SNNs that are more central. A central SNN will require fewer SNLs during the VNL embedding stage since few hops are required to place the subsequent VNN. This reduces the cost of the VNL embedding stage.

The pseudocode for calculating the AVD is shown in algorithm A.1.

PPO Hyperparameters	
Parameter Name	**Parameter Value**
Timesteps total	10000
Learning rate	0.0001
Training steps	4096
Training epochs	10
Training batch size	64
Entropy coef.	0.001
Clip	0.2
Gamma	0.99
Gamma GAE	0.95

Table A.2: PPO hyperparameters configuration.

Table A.2 shows the PPO initial hyperparameters. These are mostly default values set by SB3 [34].

Starting with this initial setup, we experiment with the following to find which setup gives the highest ACR and LRC during training:

- **Experiment 1**: We experiment with the VNN placement reward scale.

- **Experiment 2**: We experiment with the placement reward function and the accepted VNR reward function. Furthermore, we experiment with providing additional inputs to the observation space.
- **Hyperparameter tuning**: We experiment with the PPO hyperparameters. Each parameter is adjusted slightly. The subsequent performance is evaluated against a baseline.

A.2 Experiment 1

The VNN placement reward provides feedback to the agent after every placement decision.

A valid placement gives a positive reward, while an invalid placement gives no reward. This should incentivize the agent to learn to maximize reward by performing valid placements.

Additionally, we provide the agent with a second reward whenever a VNR is accepted. We use LRC for this reward. This should incentivize the agent to learn to maximize LRC.

This experiment aims to find the best scale for the placement reward such that the agent learns both valid placements and maximizes LRC.

If the placement reward is relatively large compared to the accepted reward, then the agent might prioritize valid placements over LRC due to the larger reward. Furthermore, if the placement reward is relatively small compared to the accepted reward, then the agent might not learn how to make valid placements.

Therefore, we experiment with the placement reward scale to find the best balance between valid placements and maximizing LRC. We test the following static placement rewards:

1. VNN placement reward of 0.0001.
2. VNN placement reward of 0.001.
3. VNN placement reward of 0.01.
4. VNN placement reward of 0.1.
5. VNN placement reward of 0.3.
6. VNN placement reward of 0.6.
7. VNN placement reward of 1.0.

A.2.1 Results

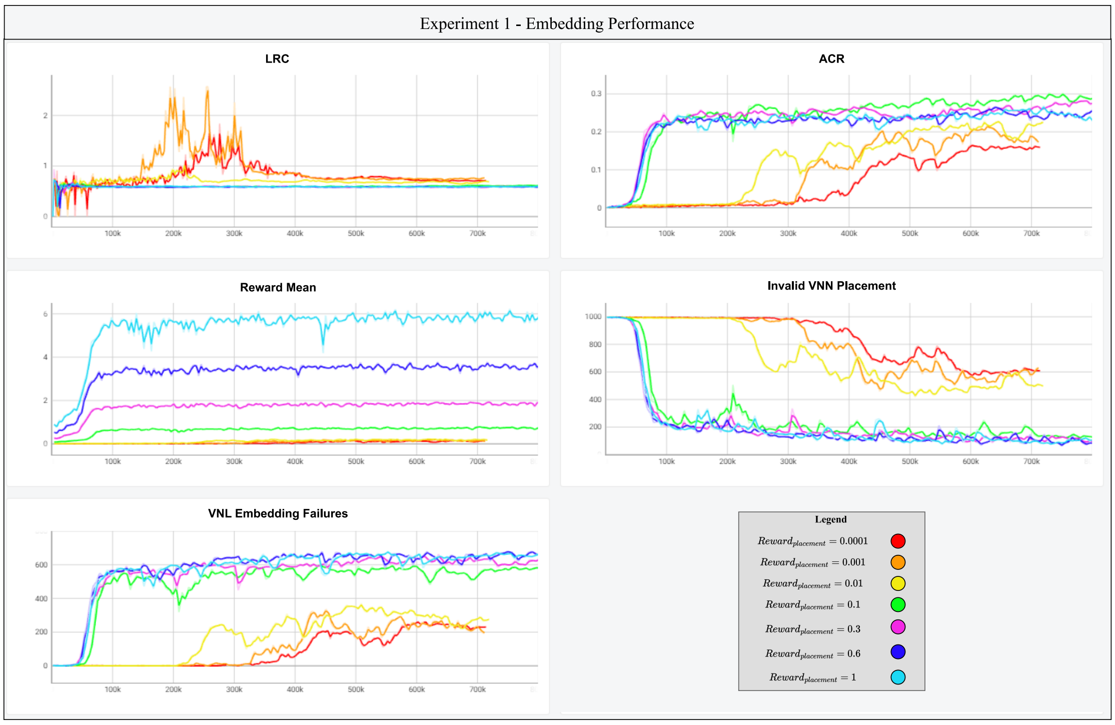

Figure A.1: Experiment 1 results.

We discuss the results shown in figure A.1:

- **ACR**: Any placement reward above or equal to 0.1 gives the best result. Runs with a high placement reward learn valid placements faster, resulting in a fast increase in ACR.

- **Invalid VNN placements**: Invalid VNN placements represent how many VNNs the agent failed to place. In the worst case, the agent may fail 1000 placements. This means the agent fails the first VNN of every VNR. When the value decreases, the agent can make fewer invalid placements[1].

  The plot shows that a placement reward of 0.1, 0.3, or 0.6 causes the agent to fail fewer placements. However, the agent struggles to perform valid placements for the lower rewards.

- **VNL embedding failures**: Measures the number of failed VNL embeddings. A VNL embedding failure may only occur if the VNN embedding stage has succeeded without failures. The available VNL resources and the VNN placement decisions made by the agent impacts the failure rate of the VNL embedding.

[1]If the placement is impossible (no valid placement exists), the agent will always make an invalid placement. Such placements are not counted as a part of invalid placements.

In figure A.1, some runs have a low VNL failure rate. This is due to most failures occurring in the VNN embedding stage.

However, we see a high failure rate for the runs with rewards of $0.1 - 1$ for VNL embeddings. There are two reasons for this high failure rate.

Firstly, these runs have a low VNN embedding failure rate. Secondly, the VNL delay parameter is difficult to satisfy. This is due to the candidate domains, ingress, and egress VNN placement restrictions which force the agent to place VNNs in multiple domains. This increases the number of SNLs used during embedding, which increases the probability of one or more SNLs being unable to satisfy the delay constraint.

- **LRC**: The agent converges on 0.6 LRC for most runs. Runs with a smaller placement reward have a larger LRC.
- **Reward mean**: Measures the trajectory reward averaged over the last 100 trajectories. The reward mean indicates that the reward has converged, meaning the agent is not improving regarding the placement reward or LRC.

A.2.2 Learning Outcome

A VNN placement reward of 0.1 seems to strike the best balance between learning valid placements and maximizing LRC.

In subsequent experiments with placement rewards, we use 0.1 as the maximum placement reward.

A.3 Experiment 2

We want to increase the ACR and LRC from experiment 1. Therefore, we experiment with providing different rewards to the agent. We set the maximum placement reward as 0.1 for all experiments with placement rewards.

Furthermore, the agent might need additional information to learn the optimal embedding strategy. Hence, we experiment by providing additional information in the observation space.

A.3.1 AVD in Placement Reward

In addition to AVD being included in the observation space, we also include the AVD in the placement reward.

The goal is for the agent to pick SNNs with a shorter average distance to other nodes. This reduces the probability of failure in the VNL embedding stage and increases LTR.

The reward of placement is calculated as follows, where n_i is the host SNN of the VNN being placed:

$$reward_{placement} = \frac{1 - n_i^{AVD}}{10} \tag{A.2}$$

A smaller distance should give a better reward. Hence, the AVD is subtracted from one. Moreover, we divide by ten such that the max reward is $\frac{1}{10} = 0.1$, which was found to work best in experiment 1.

The AVD is calculated using algorithm A.1.

This experiment extends the setup in tables A.1 and A.2.

A.3.2 Distance from Previous Placement in Placement Reward

We extend the observation space with the distance from the previous VNN placement (DSTPP). This represents the number of hops from any SNN to the previously placed VNN[2].

$$DSTPP(n_i) = \frac{SNL\_Hops(n_i, n_{previous})}{SNL\_Max\_Possible\_Hops(A)} \tag{A.3}$$

The goal is for the agent to pick SNNs close to the previously placed VNN. This reduces the probability of failure in the VNL embedding stage and increases LTR.

The reward of placement is calculated as follows, where n_i is the host SNN of the VNN being placed:

$$reward_{placement} = \frac{1 - n_i^{DSTPP}}{10} \tag{A.4}$$

The distance from the last placement is calculated using algorithm A.3.

This experiment extends the setup in tables A.1 and A.2.

[2]Note that we also normalize the DSTPP

A.3.3 Edge Information in the Observation Space

In this experiment, we extend the observation space with bandwidth and delay. For each SNN part of the state, we include the normalized average bandwidth of the connected SNLs and the normalized average delay of the connected SNLs. Moreover, we include the normalized bandwidth and delay requirements of the next VNN being placed.

$$AVG_{bandwidth}(n_i) = \frac{\text{Normalized bandwidth sum of SNL connected to } n_i}{\text{Number of connected SNLs to } n_i} \tag{A.5}$$

$$AVG_{delay}(n_i) = \frac{\text{Normalized delay sum of SNL connected to } n_i}{\text{Number of connected SNLs to } n_i} \tag{A.6}$$

The goal is for the agent to learn that the resources of connected VNLs should also be satisfied for a VNR to be accepted.

This experiment extends the setup in tables A.1 and A.2.

A.3.4 No Placement Reward

We experiment by removing the placement reward altogether. Hence, the only reward the agent receives is the accepted VNR reward which is the LRC, as specified in table A.1.

The goal is to verify whether including a placement reward contributes to the overall performance or whether the placement reward confuses the agent.

This experiment extends the setup in tables A.1 and A.2.

A.3.5 Revenue to Cost

We experiment by changing the accepted VNR reward from LRC to Revenue over Cost (RC). RC does not consider the revenue to cost over the long term but only for a single accepted VNR r_i.

$$RC(r_i) = Revenue(r_i)/Cost(r_i) \tag{A.7}$$

$$reward_{accepted} = RC(r_i) \tag{A.8}$$

This experiment extends the setup in tables A.1 and A.2.

A.3.6 Results

The following names in the legend represent the experiments of the above sections:

- AVD in Placement Reward (AVDP).
- Distance from Previous Placement (DSTPP).
- Edge Information in the Observation (EOBS).
- No Placement Reward (NPP)
- No Placement Reward with RC (NPPRC).

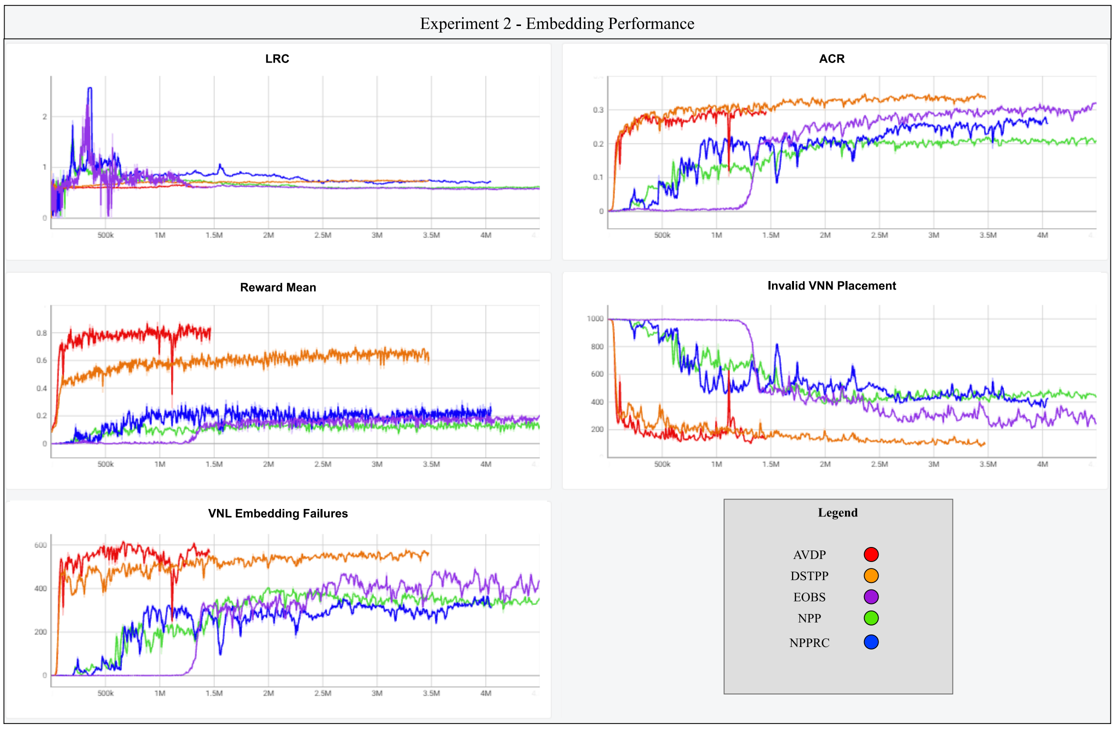

Figure A.2: Experiment 2 results.

We discuss the results shown in figure A.2:

- **ACR & Invalid Placements**: We see that DSTPP outperforms every other experiment regarding the ACR. AVDP had an initial high increase in ACR but later worsened compared to DSTPP. This worsening was due to AVDP's inability to satisfy the VNL constraints in the VNL embedding stage. This can be seen from the "VNL embedding failures," where AVDP often fails due to VNL failures compared to DSTPP. Both AVDP and DSTPP have a similar failure rate for VNN placements.

- **LRC**: Regarding LRC, the NPPRC, EOBS, and NPP initially had a very high LTR. This was due to the low ACR causing high instability in the LRC. As the ACR increases, the LRC converges toward its actual value.

  The DSTPP has a reliably high LRC compared to the other experiments.

- **Reward mean**: The mean reward is not comparable between the experiments due to different reward functions being used. However, we see that AVDP generally rewards more than the other solutions. This is due to the high connectivity in the SN, which causes SNNs to have a high AVD. Thus, the placement rewards are also higher for AVDP.

A.3.7 Learning Outcome

The above experiments show that the DSTPP is the best-performing solution regarding our ACR, LTR, LTC, and LRC objectives. The ACR is high, while LRC is stable and relatively high. The LRC measure captures the LTR and LRC measures. We proceed to the hyperparameter tuning using the DSTPP setup.

A.4 Hyperparameter Tuning

In this section, we tune the hyperparameters used by PPO on the best-performing setup from experiments 1 and 2. This setup is shown in tables A.3 and A.4 below.

Simulation Config	
Config Name	**Config Value**
VNN Placement Reward	DSTPP
VNR Accepted Reward	LRC
State Vector Inputs	Available resources in all SNNs Demanded resources of next VNN DSTPP in all SNNs

Table A.3: Environment configuration.

PPO Hyperparameters	
Parameter Name	**Parameter Value**
Timesteps total	10000
Learning rate	0.0001
Training steps	4096
Training epochs	10
Training batch size	64
Entropy coef.	0.001
Clip	0.2
Gamma	0.99
Gamma GAE	0.95

Table A.4: PPO hyperparameters before tuning.

A.4.1 Learning Rate

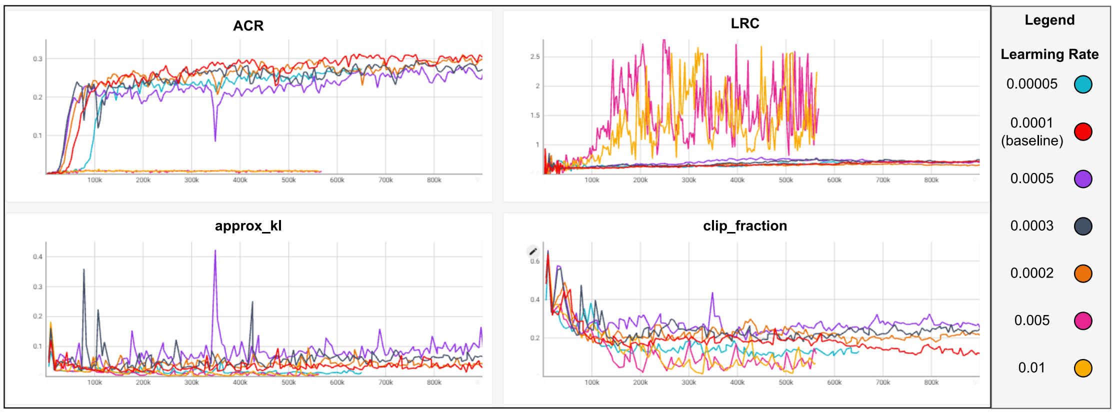

Figure A.3: Learning rate experiments. The baseline is the default learning rate.

We show the ACR, LRC, approx_kl, and clip fraction in figure A.3.

1. *ACR*: The baseline (red) gives the best ACR. The agent cannot learn VNN placements with a learning rate of 0.005 or 0.01.
2. *LRC*: The baseline gives the best LRC in the long run. The learning rate of 0.005 or 0.01 is unstable due to their low ACR.
3. *approx_kl*: A large value in the approx_kl represents a large change in the policy [39]. This statistic is useful for visualizing agent learning.

   With a learning rate of 0.005 and 0.01, the approx_kl is consistently close to zero, and the ACR is also close to zero. This indicates the agent is unable to learn when using these learning rates.

The learning rate of 0.0005 has a large spike in the approx_kl between 300k-400k steps. This overall negatively affected ACR performance, seen between 300k-400k steps.

4. *clip_fraction*: PPO clipping as discussed in section 5. The PPO clip helped reduce the impact of the large policy change around 300k-400k steps by clipping the update.

   A larger clip range could be used for the 0.0005 learning rate to ensure more stable policy changes.

We continue using the default learning rate of 0.0001 due to its high ACR and LRC.

A.4.2 PPO Clip

Although the chosen learning rate from the learning rate tuning has stable policy updates, we experiment with slight changes to the clipping factor to view how it impacts the policy updates.

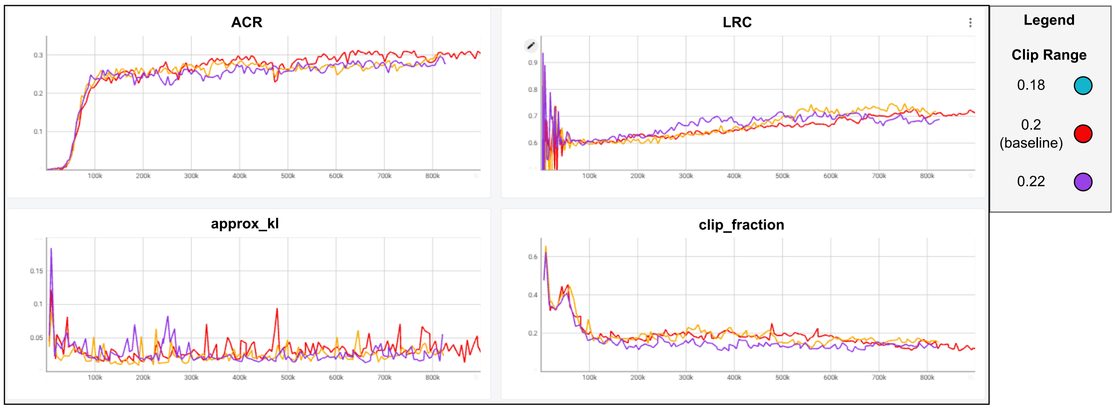

Figure A.4: Clip rate experiments. The baseline is the default clip range.

A.4.3 Training Epochs

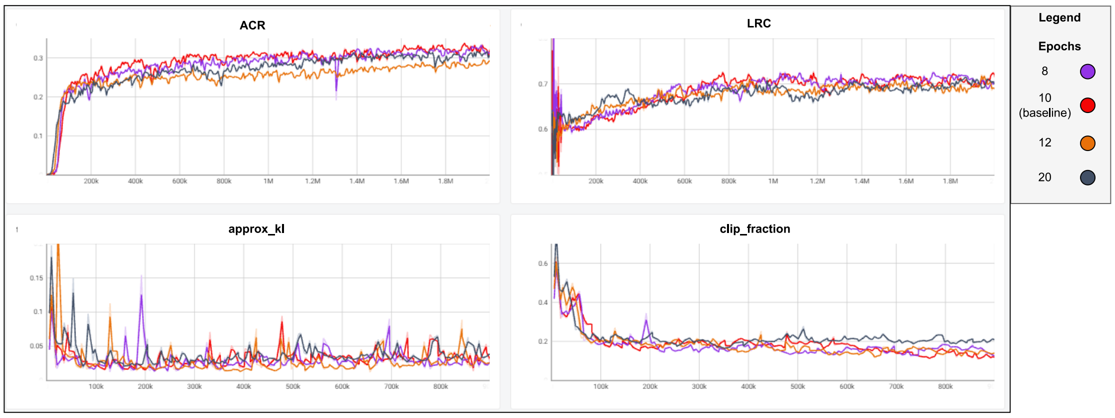

Figure A.5: Experimentation with the number of epochs. The baseline is the default number of epochs.

Figure A.5 shows very little difference between the epoch experiments. However, the experiment with eight epochs has a slightly positive trend in LRC relative to the baseline. Hence, we use eight epochs going forward.

A.4.4 Training Batch Size

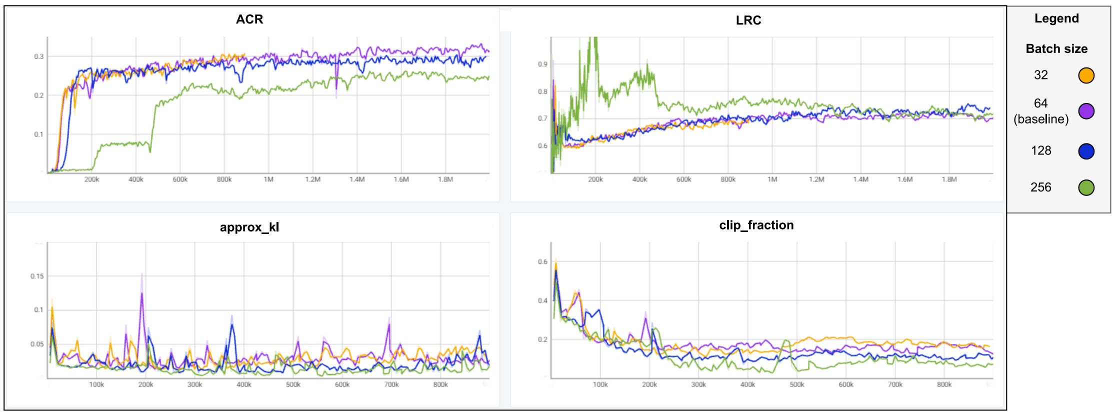

Figure A.6: Experimentation with the batch size. The baseline is the default batch size.

Figure A.6 shows little difference between the experiments. The experiment with a batch size of 256 initially gives a high LRC, however, this is due to instabilities caused by the low ACR. We continue using the default batch size of 64 since this experiment has the best overall ACR and LRC performance.

A.4.5 Training Steps

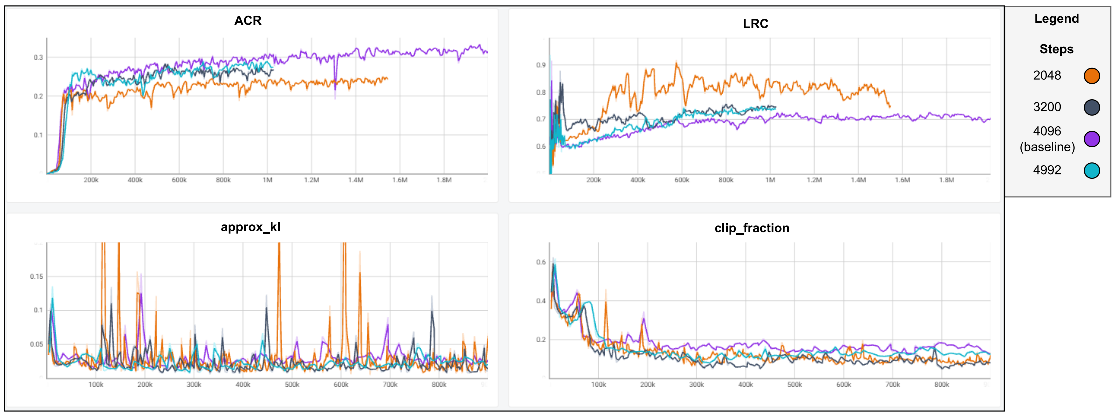

Figure A.7: Experimentation with the number of steps. The baseline is 4096 steps, which is used in earlier experimentation.

Figure A.7 shows the experiments with various number of training steps. The experiment with 2048 steps gives a high LRC but only a moderately high ACR. Furthermore, we see from the approx_kl that this experiment is experiencing large policy updates which also causes spikes in the clipping. Overall, the baseline of 4096 gives the best performance.

A.4.6 GAE Gamma

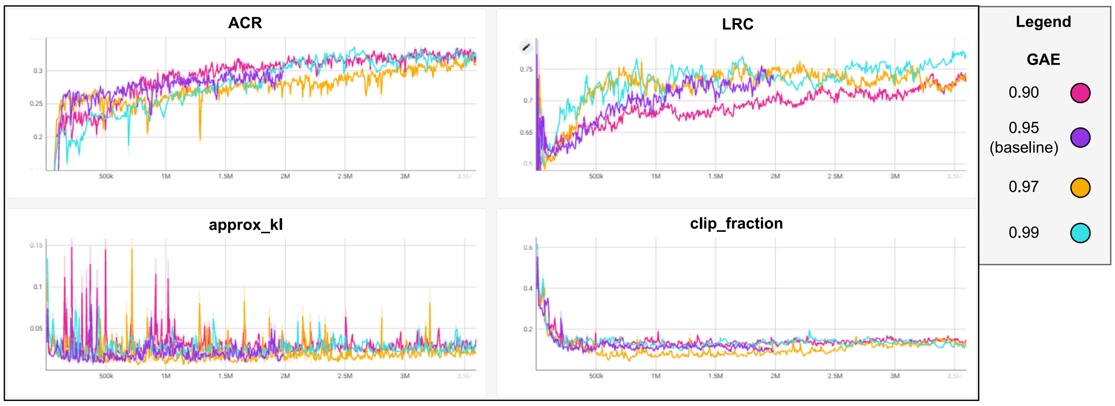

Figure A.8: GAE parameter experiments. The baseline is the default GAE value.

Figure A.8 shows the experimentation with the GAE parameter. We see the baseline ($GAE = 0.95$) is outperformed by $GAE = 0.90$. However, $GAE = 0.90$ performs poorly in terms of LRC.

However, $GAE = 0.99$ improves the LRC while maintaining a relatively high ACR. Hence, we continue using $GAE = 0.99$ going forward.

A.4.7 Entropy

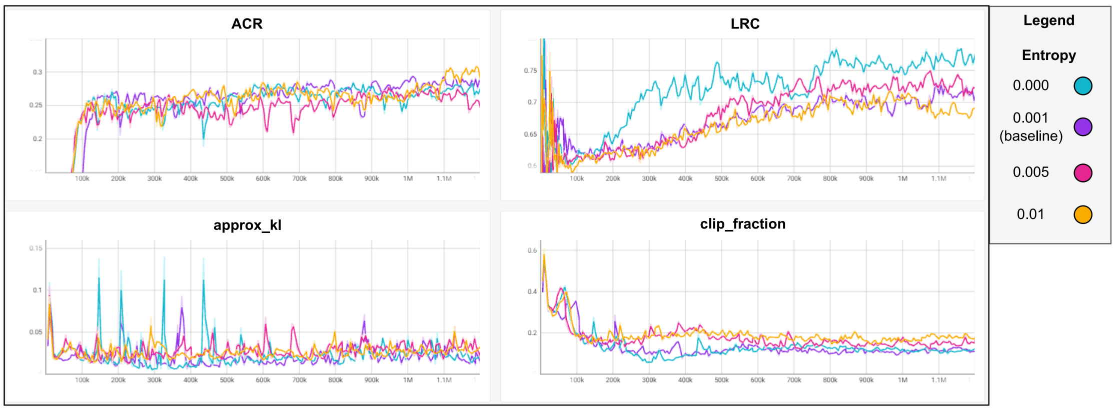

Figure A.9: Entropy experiments. The baseline is the default entropy value.

Figure A.9 shows that $entropy = 0$ gives a large boost to the LRC. This could be because the agent favours less exploration with no entropy, causing the agent to utilize the optimal strategy without exploration. We use $entropy = 0$ going forward.

A.5 Additional Pseudocode

A.5.1 AVD

Algorithm A.1 calculateAD

```
1:  $distanceVector \leftarrow Vector(|N^A|)$
2:  {Calculate the distances between any SNL pair}
3:  $distanceMap \leftarrow nx.shortest_path_length(A)$
4:  for $n_i \in N^A$ do
5:      $sumDistances = 0$
6:      for $n_j \in distanceMap[n_i]$ do
7:          $sumDistances = sumDistances + distanceMap[n_i][n_j]$ {Distance $n_i$ to $n_j$}
8:      end for
9:      $distanceVector[i] = sumDistances/|N^A|$
10: end for
11: return $distanceVector$
```

A.5.2 Normalized Average Bandwidth and Delay

Algorithm A.2 minimumNormalizedBW

```
1: $BWVector \leftarrow Vector(|N^A|)$
2: $DELVector \leftarrow Vector(|N^A|)$
3: $maxPossibleBW \leftarrow config[maxPossibleBW]$
4: $maxPossibleDEL \leftarrow config[maxPossibleDEL]$
5:
6: for $n_i \in N^A$ do
7:     $BWVector[i] \leftarrow getMaxBW(n_i)/maxPossibleBW$
8:     $DELVector[i] \leftarrow getMinDEL(n_i)/maxPossibleDEL$
9: end for‘
10: return $BWVector, DELVector$
```

A.5.3 Distance from Previous VNN Embedding

Algorithm A.3 calculateDistanceFromPreviousVNN

```
1: $distanceVector \leftarrow Vector(|N^A|)$
2: {Calculate the distances between any SNL pair}
3: $distanceMap \leftarrow nx.shortest_path_length(A)$
4: for $n_i \in N^A$ do
5:     $sumDistances = 0$
6:     $j \leftarrow IDofPreviousVNN()$
7:     $distanceVector[n_i] \leftarrow distanceMap[n_i][n_j]$
8: end for
9: $distanceVector \leftarrow distanceVector/max(distanceVector)$ {Normalize}
10: return $distanceVector$
```

A.5.4 GRC Pseudocode

Algorithm A.4 getSortedGRCVector

```
Input: Any graph $G(N^G, E^G)$, Dampening $d$, Threshold $th$
1:  grc = []
2:
3:  {Calculate equation 5.4}
4:  $resourceVector = []$
5:  for VNN $v_f$ do
6:      $resourcesAvailable = v_i^{CPU}$
7:      $resourceVector[i] = resourcesAvailable$
8:  end for
9:  $resourceVector = resourceVector/Sum(resourceVector)$
10:
11: {Calclate equation 5.5}
12: $M = G.adjacencyMatrix$
13: for edge $e_{(n_{row_i}, n_{col_i})}$ in M do
14:     $BWSum = 0$
15:     for edge $e_{(x, n_{col_i})}$ adjacent to $n_{col_i}$ do
16:         $BWSum+ = e^{BW}_{(x, n_{col_i})}$
17:     end for
18:     $M[row_i, col_j] = BWSum$
19: end for
20:
21: {Find the GRC vector of graph G according to alg. 1 in [11]}
22: $c = resourceVector$
23: $k = 0$
24: $grc_k = resourceVector$
25: $\Delta = \inf$
26: while $\Delta \geq th$ do
27:     $grc_{k+1} = (1-d)c + dM \cdot grc_k)$
28:     $\Delta = ||grc_{k+1} - grc_k||$
29:     $grc_k = grc_{k+1}$
30: end while
31: return $grc_k.sortDecending()$
```

Algorithm A.5 GRCEmbedVNR

```
Input: SN graph $A$, VNR $r_i$
 1: $grc_{SN} = getSortedGRCVector(A)$
 2: $grc_{VN} = getSortedGRCVector(r_i^B)$
 3:
 4: for sorted node $v_f$ in order $grc_{VN}$ do
 5:     $nodeIsPlaced$ = false
 6:     for sorted node $n_i$ in order $grc_{SN}$ do
 7:         if $hasSuffficcientResources(A, n_i, v_f)$ then
 8:             $embedVNN(A, n_i, v_f)$
 9:             $nodeIsPlaced$ = true
10:         end if
11:     end for
12:
13:     if $nodeIsPlaced$ then
14:         $edgesArePlaced = embedVNLs(A, r_i^B)$
15:     end if
16:
17:     if $nodeIsPlaced$ and $edgesArePlaced$ then
18:         return true
19:     end if
20: end for
21:
22: deAllocateResources()
23: return false
```

Bibliography

[1] Peiying Zhang, Pan Yang, Neeraj Kumar, and Mohsen Guizani. Space-Air-Ground Integrated Network Resource Allocation Based on Service Function Chain. *IEEE Transactions on Vehicular Technology*, 71(7):7730–7738, July 2022. ISSN 0018-9545, 1939-9359. doi: 10.1109/TVT.2022.3165145. URL https://doi.org/10.1109/TVT.2022.3165145. Number: 7.

[2] Peiying Zhang, Chao Wang, Neeraj Kumar, and Lei Liu. Space-Air-Ground Integrated Multi-Domain Network Resource Orchestration Based on Virtual Network Architecture: A DRL Method. *IEEE Transactions on Intelligent Transportation Systems*, 23(3):2798–2808, March 2022. ISSN 1524-9050, 1558-0016. doi: 10.1109/TITS.2021.3099477. URL https://doi.org/10.1109/TITS.2021.3099477. Number: 3.

[3] Peiying Zhang, Yi Zhang, Neeraj Kumar, and Mohsen Guizani. Dynamic SFC Embedding Algorithm Assisted by Federated Learning in Space-Air-Ground Integrated Network Resource Allocation Scenario. *IEEE Internet of Things Journal*, pages 1–1, 2022. ISSN 2327-4662, 2372-2541. doi: 10.1109/JIOT.2022.3222200. URL https://doi.org/10.1109/JIOT.2022.3222200.

[4] Peiying Zhang, Yi Zhang, Neeraj Kumar, and Ching-Hsien Hsu. Deep Reinforcement Learning Algorithm for Latency-Oriented IIoT Resource Orchestration. *IEEE Internet of Things Journal*, pages 1–1, 2022. ISSN 2327-4662, 2372-2541. doi: 10.1109/JIOT.2022.3229270. URL https://doi.org/10.1109/JIOT.2022.3229270.

[5] Jiajia Liu, Yongpeng Shi, Zubair Md. Fadlullah, and Nei Kato. Space-Air-Ground Integrated Network: A Survey. *IEEE Communications Surveys & Tutorials*, 20(4):2714–2741, 2018. ISSN 1553-877X. doi: 10.1109/COMST.2018.2841996. URL https://doi.org/10.1109/COMST.2018.2841996. Conference Name: IEEE Communications Surveys & Tutorials.

[6] Frederico Schardong, Ingrid Nunes, and Alberto Schaeffer-Filho. NFV Resource Allocation: a Systematic Review and Taxonomy of VNF Forwarding Graph Embedding. *Computer Networks*, 185:107726, February 2021. ISSN 13891286. URL https://doi.org/10.1016/j.comnet.2020.107726.

[7] Ghasem Mirjalily and Zhiquan Luo. Optimal Network Function Virtualization and Service Function Chaining: A Survey. *Chinese Journal of Electronics*, 27 (4):704–717, 2018. ISSN 2075-5597. doi: 10.1049/cje.2018.05.008. URL https://doi.org/10.1049/cje.2018.05.008.

[8] Jie Sun, Yi Zhang, Feng Liu, Huandong Wang, Xiaojian Xu, and Yong Li. A survey on the placement of virtual network functions. *Journal of Network and Computer Applications*, 202:103361, June 2022. ISSN 10848045. URL https://doi.org/10.1016/j.jnca.2022.103361.

[9] ETSI 3rd Generation Partnership Project (3GPP). Network functions virtualisation (nfv); architectural framework. Technical Report 3GPP TS 128.530 V16.2.0 Release 16, ETSI, December 2014. URL https://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.02.01_60/gs_nfv002v010201p.pdf. Accessed: 2023-03-21.

[10] Song Yang, Fan Li, Stojan Trajanovski, Ramin Yahyapour, and Xiaoming Fu. Recent Advances of Resource Allocation in Network Function Virtualization. *IEEE Transactions on Parallel and Distributed Systems*, 32(2):295–314, February 2021. ISSN 1045-9219, 1558-2183, 2161-9883. doi: 10.1109/TPDS.2020.3017001. URL https://doi.org/10.1109/TPDS.2020.3017001. Number: 2.

[11] Long Gong, Yonggang Wen, Zuqing Zhu, and Tony Lee. Toward profit-seeking virtual network embedding algorithm via global resource capacity. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, pages 1–9, April 2014. URL https://doi.org/10.1109/INFOCOM.2014.6847918. ISSN: 0743-166X.

[12] Andreas Fischer, Juan Felipe Botero, Michael Till Beck, Hermann de Meer, and Xavier Hesselbach. Virtual Network Embedding: A Survey. *IEEE Communications Surveys & Tutorials*, 15(4):1888–1906, 2013. ISSN 1553-877X. doi: 10.1109/SURV.2013.013013.00155. URL https://doi.org/10.1109/SURV.2013.013013.00155. Number: 4 892 citations (Crossref) [2022-12-30].

[13] Shuiqing Gong, Jing Chen, Conghui Huang, Qingchao Zhu, and Siyi Zhao. Virtual network embedding through security risk awareness and optimization. *KSII Transactions on Internet and Information Systems*, 10(7):2892–2913, July 2016. URL https://doi.org/10.3837/tiis.2016.07.002.

[14] Shankar Lal, Tarik Taleb, and Ashutosh Dutta. NFV: Security Threats and Best Practices. *IEEE Communications Magazine*, 55(8):211–217, August 2017. ISSN 1558-1896. URL https://doi.org/10.1109/MCOM.2017.1600899. Conference Name: IEEE Communications Magazine.

[15] Max Alaluna, Luís Ferrolho, José Rui Figueira, Nuno Neves, and Fernando M.V. Ramos. Secure multi-cloud virtual network embedding. *Computer Communications*, 155:252–265, April 2020. ISSN 01403664. doi: 10.1016/j.comcom.2020.03.023. URL https://doi.org/10.1016/j.comcom.2020.03.023.

[16] OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d. O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning, 2019. URL https://doi.org/10.48550/arXiv.1912.06680.

[17] Chao Wang, Lei Liu, Chunxiao Jiang, Shangguang Wang, Peiying Zhang, and Shigen Shen. Incorporating Distributed DRL Into Storage Resource Optimization of Space-Air-Ground Integrated Wireless Communication Network. *IEEE Journal of Selected Topics in Signal Processing*, 16(3):434–446, April 2022. ISSN 1932-4553, 1941-0484. doi: 10.1109/JSTSP.2021.3136027. URL https://doi.org/10.1109/JSTSP.2021.3136027. Number: 3.

[18] Yuxi Li. Deep Reinforcement Learning, October 2018. URL https://doi.org/10.48550/arXiv.1810.06339. Issue: arXiv:1810.06339 Issue: arXiv:1810.06339 arXiv:1810.06339 [cs, stat].

[19] Lilian Weng. A (long) peek into reinforcement learning. *lilianweng.github.io*, 2018. URL https://lilianweng.github.io/posts/2018-02-19-rl-overview/.

[20] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, August 2017. URL https://doi.org/10.48550/arXiv.1707.06347. arXiv:1707.06347 [cs].

[21] Lilian Weng. Policy gradient algorithms. *lilianweng.github.io*, 2018. URL https://lilianweng.github.io/posts/2018-04-08-policy-gradient/.

[22] João Nogueira, Márcio Melo, Jorge Carapinha, and Susana Sargento. Virtual network mapping into heterogeneous substrate networks. In *2011 IEEE Symposium on Computers and Communications (ISCC)*, pages 438–444, June 2011. doi: 10.1109/ISCC.2011.5983876. URL https://doi.org/10.1109/ISCC.2011.5983876. ISSN: 1530-1346.

[23] Peiying Zhang, Chao Wang, Chunxiao Jiang, and Abderrahim Benslimane. Security-Aware Virtual Network Embedding Algorithm Based on Reinforcement Learning. *IEEE Transactions on Network Science and Engineering*, 8(2):1095–1105, April

2021. ISSN 2327-4697, 2334-329X. doi: 10.1109/TNSE.2020.2995863. URL https://doi.org/10.1109/TNSE.2020.2995863.

[24] Xian Zhang, Xiuzhong Chen, and Chris Phillips. Achieving effective resilience for QoS-aware application mapping. *Computer Networks*, 56(14):3179–3191, September 2012. ISSN 1389-1286. doi: 10.1016/j.comnet.2012.06.015. URL https://doi.org/10.1016/j.comnet.2012.06.015.

[25] Montida Pattaranantakul, Ruan He, Qipeng Song, Zonghua Zhang, and Ahmed Meddahi. NFV Security Survey: From Use Case Driven Threat Analysis to State-of-the-Art Countermeasures. *IEEE Communications Surveys & Tutorials*, 20(4): 3330–3368, 2018. ISSN 1553-877X. URL https://doi.org/10.1109/COMST.2018.2859449. Conference Name: IEEE Communications Surveys & Tutorials.

[26] Dongcheng Zhao, Long Luo, Hongfang Yu, Victor Chang, Rajkumar Buyya, and Gang Sun. Security-SLA-guaranteed service function chain deployment in cloud-fog computing networks. *Cluster Computing*, 24(3):2479–2494, September 2021. ISSN 1386-7857, 1573-7543. doi: 10.1007/s10586-021-03278-4. URL https://doi.org/10.1007/s10586-021-03278-4.

[27] Peiying Zhang, Chao Wang, Chunxiao Jiang, Neeraj Kumar, and Qinghua Lu. Resource Management and Security Scheme of ICPSs and IoT Based on VNE Algorithm. *IEEE Internet of Things Journal*, 9(22):22071–22080, November 2022. ISSN 2327-4662, 2372-2541. doi: 10.1109/JIOT.2021.3068158. URL https://doi.org/10.1109/JIOT.2021.3068158. Number: 22.

[28] Peiying Zhang, Chunxiao Jiang, Xue Pang, and Yi Qian. STEC-IoT: A Security Tactic by Virtualizing Edge Computing on IoT. *IEEE Internet of Things Journal*, 8 (4):2459–2467, February 2021. ISSN 2327-4662, 2372-2541. doi: 10.1109/JIOT.2020.3017742. URL https://doi.org/10.1109/JIOT.2020.3017742.

[29] Soroush Haeri and Ljiljana Trajković. Virtual Network Embedding via Monte Carlo Tree Search. *IEEE Transactions on Cybernetics*, 48(2):510–521, February 2018. ISSN 2168-2275. URL https://doi.org/10.1109/TCYB.2016.2645123. Conference Name: IEEE Transactions on Cybernetics.

[30] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[31] waxman_graph — networkx 3.0 documentation. https://networkx.org/documentation/stable/reference/generated/networkx.generators.geometric.waxman_graph.html. (Accessed on 02/25/2023).

[32] Make your own custom environment - Gym Documentation, . URL https://www.gymlibrary.dev/content/environment_creation/.

[33] Spaces - Gym Documentation, . URL https://www.gymlibrary.dev/api/spaces/.

[34] PPO — Stable Baselines3 2.0.0a10 documentation, . URL https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html.

[35] Using Custom Environments — Stable Baselines3 2.0.0a8 documentation, . URL https://stable-baselines3.readthedocs.io/en/master/guide/custom_env.html.

[36] Shengyi Huang, Rousslan Fernand Julien Dossa, Antonin Raffin, Anssi Kanervisto, and Weixun Wang. The 37 implementation details of proximal policy optimization. In *ICLR Blog Track*, 2022. URL https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/.

[37] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation, October 2018. URL https://doi.org/10.48550/arXiv.1506.02438. arXiv:1506.02438 [cs].

[38] Siddharth Mysore, Bassel Mabsout, Renato Mancuso, and Kate Saenko. Honey, I Shrunk The Actor: A Case Study on Preserving Performance with Smaller Actors in Actor-Critic RL, June 2021. URL https://doi.org/10.48550/arXiv.2102.11893. arXiv:2102.11893 [cs].

[39] Logger — Stable Baselines3 2.0.0a10 documentation. URL https://stable-baselines3.readthedocs.io/en/master/common/logger.html.