MAXIMIZING SERVICE FUNCTION CHAIN AVAILABILITY IN

SOFTWARE DEFINED NETWORKS

A Thesis

Presented

to the Faculty of

California State University, Dominguez Hills

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Sterling Abrahams

Fall 2022

THESIS: MAXIMIZING SERVICE FUNCTION CHAIN AVAILABILITY IN SOFTWARE DEFINED NETWORKS

AUTHOR: STERLING ABRAHAMS

APPROVED:

____
Bin Tang, Ph.D
Thesis Committee Chair

____
Jack Han, Ph.D.
Committee Member

____
Mohsen Beheshti, Ph.D.
Committee Member

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor Bin Tang, for working with me so consistently and thoroughly, and teaching me a great deal about the process of research. In addition, I would like to thank Dr. Deng Pan of Florida International University for the many discussions. I would also like to thank Dr. Mohsen Beheshti and Dr. Han for always answering my questions and being my committee members.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	i
TABLE OF CONTENTS	ii
LIST OF TABLES	iii
LIST OF FIGURES	vi
ABSTRACT	v
1. INTRODUCTION	1
2. BACKGROUND AND RELATED WORKS	8
3. PROBLEM FORMULATION & SOFTWARE ARCHITECTURE	12
4. SGA AND RSGA ALGORITHMS	21
5. MCF ALGORITHM	26
6. NEURAL NETWORK APPROACH	34
7. CONCLUSION AND FUTURE WORKS	40
REFERENCES	41
APPENDIX: GITHUB REPOSITORY LINK	43

LIST OF TABLES

1. Problem Parameter…………………………………………………………………...13

2. Greedy Approach Example……………………………………………………………..16

3. Description of Residual Graphs………………………………………………………... 29

4. Description of Algorithm 5-tuple………………………………………………………33

LIST OF FIGURES

1. SFC Example Diagram………………………………………………………………….. 2

2. Example of Complete Weighted Bipartite Graph……………………………………….. 4

3. Conversion of WBG into Flow Network………………………………………………... 6

4. Example of Fat Tree Topology………………………………………………………….. 9

5. A description of how to maximize the availability of the Assignment Function…………… 15

6. A WBG Represented by a python dictionary…………………………………………… 17

7. A WBG and Corresponding CostMatrix………………………………………………..18

8. Pseudo Code for SGA Algorithm………………………………………………………… 21

9. Pseudo Code for RSGA Algorithm.……...…………………………………………….22

10. A direct comparison of SGA and RSGA……………………………………………… 23

11. A comparison of SGA vs. RSGA when r=3 and n=48………………………………… 24

12. Converting WBG into a Flow Network………………………………………………... 26

13. Flow Network Example…………………………………………………………………... 27

14. Pseudo Code for SSP…………………………………………………………………... 28

15. Original Graph…………………………………………………………………………… 29

16. Residual Graph 1…………………………………………………………………………..30

17. Residual Graph 2…………………………………………………………………………..30

18. Residual Graph 3…………………………………………………………………………..31

19. SGA vs. RSGA vs. MCF……………………………………………………………. 32

20. DNN Architecture……………………………………………………………………… 34

21. Training Feedback Loop…………………………………………………………….. 35

22. DNN layered Architecture……………………………………………………………... 37

23. DNN vs. MCF for r=1…………………………………………………………………..38

24. DNN vs. MCF for r=3…………………………………………………………………..39

ABSTRACT

In a Software Defined Network (SDN) there may be multiple Virtual Network Function (VNF) nodes within the network that comprise a Service Function Chain (SFC). Each one of these nodes may provide a specific function such as firewall, deep packet inspection (DPI), Network Address Translators (NAT), etc. If there is only a single instance of a VNF then there is a single point of failure in the network. For this reason there must be a backup server associated with each VNF. Assuming that each VNF and backup server have a failure probability, there is the problem of optimal backup server assignment to maximize availability of the SFC. This thesis explores multiple algorithms such as the Sorted Greedy Algorithm (SGA), Minimum Cost Flow (MCF), Combined Deep Neural Network, and how they respond to the static and dynamic case where the failure probabilities change and backup servers go down.

CHAPTER 1

INTRODUCTION

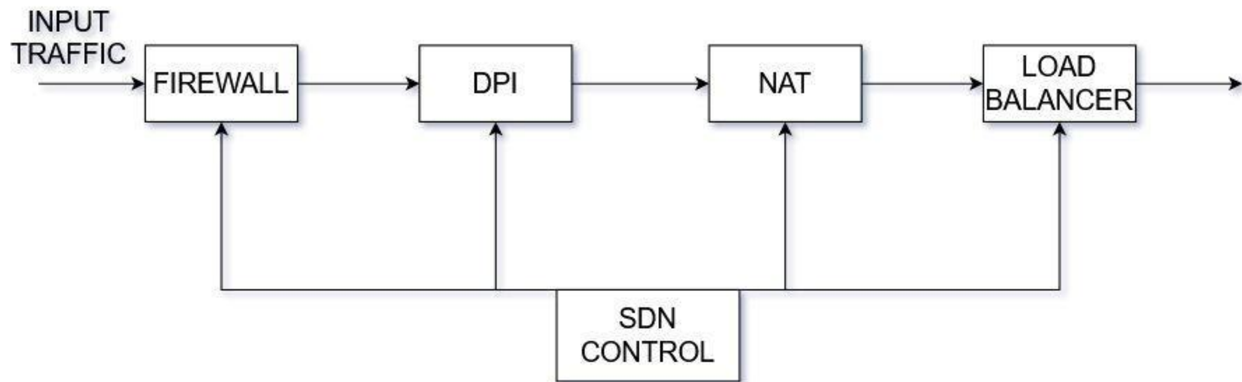
In traditional networking, all packets and data typically pass through a series of dedicated hardware devices called middleboxes (MBs). Large enterprise corporations rely on multiple middleboxes such as Network Address Translation (NAT), load balancing, traffic shaping, firewalls, and Deep Packet Inspection (DPI) (Kanizo et al., 2016). With the advent of Software Defined Networking (SDN), it is now possible to run Virtual Network Functions (VNFs) on commodity x86 or ARM hardware, or even deploy the VNFs into virtual machines or containers. SDN provides the ability to decouple the control and data plane (Xie et. al., 2019). The general concept of deploying VNFs on commodity hardware is known as Network Function Virtualization (NFV).

By the flexible nature of NFV, companies can considerably reduce operating expenses and also provide better utilization of resources. Multiple VNF instances chained together such as those mentioned comprise a single Service Function Chain (SFC).
All of the VNFs in the service chain can be deployed dynamically to any virtual machine or server since it is simply software. The dynamic nature in which the VNFs can be deployed means that there are various attributes of the service function chain that can be optimized. Figure 1 shows an example of a service chain with firewall, DPI, NAT, and load balancer. The breakdown of both physical machines (PM), virtual machines (VM) and software will inevitably cause VNF failure (He et al., 2019). If proper planning and resource allocation is not taken into consideration then the Quality of Service (QoS) of the SFC could be greatly diminished. A single VNF instance going down means that the entire SFC is unavailable because it is assumed that all

traffic in the network must pass through each VNF instance. The unavailability of a VNF will interrupt the SFCs that use that VNF and degrade the user experience (Kang et al., 2021).

Figure 1

SFC Example Diagram



In general, SDN decouples the control plane and the data plane (Xie et al., 2019) and the centralized controller will have a global view of the network by collecting real time data, as well as empirical data provided by the operators. This constant stream of data means that the network can be optimized for certain parameters. For instance, Khoshkholghi1 et al. (2020) aim to optimize both cost and latency. Cost in this scenario means the cost of redeployment. Since a virtual network function may be complex, it can take considerable time and energy to redeploy on a new VM. If the VNF is not already cached, it must be sent from the controller over the network which can be costly in terms of time. In addition, some corporations may have strict service function chain latency needs, especially if they are serving many customers. Finding an optimal placement for VNFs to optimize the total latency of the SFC is proven to be an NP hard problem (Khoshkholghi1 et al., 2020). Depending on the needs of the customer, they will have to prioritize certain parameters, such as latency, over other parameters. In addition, the most important priority to many corporations, is QoS and minimizing downtime, or minimizing the

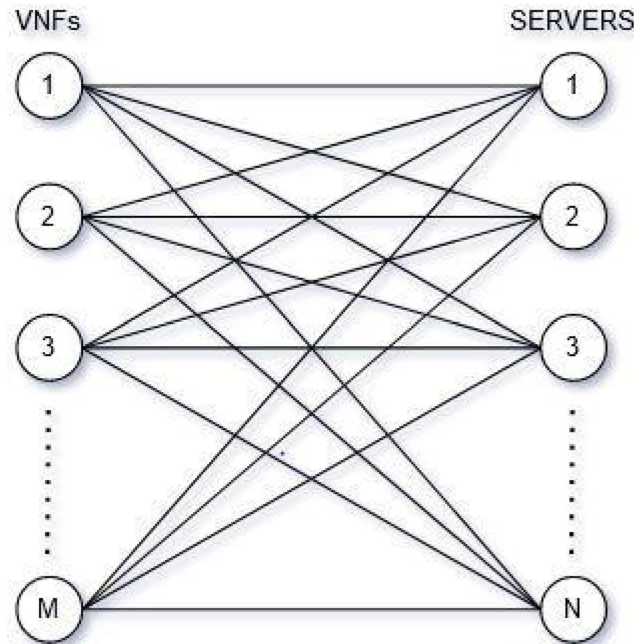
unavailability of the SFC. Operators must solve the VNF placement, the number and location of the instances required to deploy the service, and also take into account the chaining of the VNFs, taking into consideration the network resources available to each PM. We can refer to the combination of both problems as service function chain mapping (Ruiz et al., 2020).

Specifically, the problem of maximizing the service function chain availability is of upmost importance. With the growing ubiquity of 5g networks, and customer expectations for QoS, SFC availability is a factor that is extremely important for many large scale enterprises. The SFC availability is one piece of the puzzle in the SDN problem domain, but it is an important problem to solve. Taking into account multiple SFC parameters such as cost, latency, availability, is a rather large problem to solve all at once. By focusing on optimizing one of these parameters, the idea is that eventually the solutions will be able to be combined to provide good solutions to the entire problem domain.

The goal of this thesis is to focus on the availability of the service function chain as a whole. The idea is that in any given network, such as a fat tree network, the user may deploy some number of VNFs that form a service function chain. Each VNF must have an associated backup server. These two sets of nodes, VNFs and servers form a complete bipartite graph, in which both the VNFs and the servers have some failure probability. These failure probabilities represent the edge weights in the graph, forming a complete weighted bipartite graph. A depiction of the weighted bipartite graph (WBG) can be seen below in figure 2.

Figure 2

Example of a Weighted Complete Bipartite Graph



Viewing the problem as a WBG has many benefits as WBGs are historically very important in graph theory. They are well studied and there exists many papers and literature which convert problems into WBGs, essentially turning the problems into variations of the assignment problem or weighted bipartite matching. The assignment problem falls under the category of combinatorial optimization problems. The most basic version of the problem is the case where one must assign one agent to each task, such that the total cost of the tasks is minimized. In the problem domain of SDN, the workers are the servers, and the tasks are the VNFs. We analyze the simple case, to assign one server to each VNF. We also analyze the case where each server can instantiate multiple VMs or containers that serve multiple VNF nodes in the service function chain. The latter being more realistic in the climate of today because of

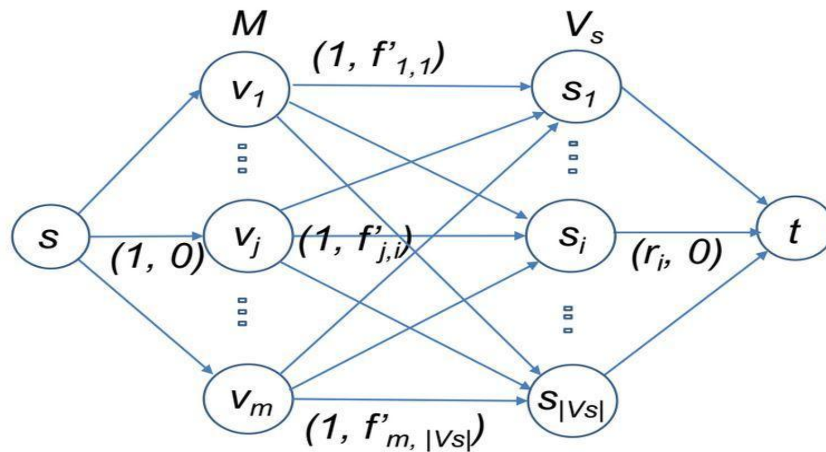
cloud computing architecture, and the ubiquity of both VMs and containers in the cloud environment.

We propose three different categories of algorithms to solve the problem. The first and most simple class of algorithms are two greedy algorithms: the Sorted Greedy Algorithm (SGA) and the Reverse Sorted Greedy algorithm (RSGA). These two algorithms are efficient because they simply sort VNF and server nodes according to their failure probabilities, then match them up according to Best-Best (BB) or Best-Worst (BW) order. We show that while these algorithms, specifically SGA, provide reasonable results, because of the greedy nature of the algorithm they are not guaranteed to provide an optimal solution. We next look at a much more robust solution, using Minimum Cost Flow (MCF). The MCF is well studied and can solve generalized flow network optimization problems. This is a novel approach as no current research has been shown to use MCF to solve SFC availability maximization in an SDN. The MCF algorithm works by first converting the Weighted Bipartite Graph (WBG), into a flow network.

A depiction of an example flow network can be seen below in figure 3.

Figure 3

Conversion of WBG into a Flow Network



Note: The edges indicate the capacities and costs with V being the set of VNF and S being the set of servers.

The general idea behind this approach is that we first convert the WBG into a flow network by adding a source and sink node. Once correctly converted, we can apply the MCF algorithm to our graph. Many modern graph theory libraries, such as Networkx, provide implementations of MCF. We implement our own solution using the Successive Shortest Paths (SSP) algorithm and compare our results to Networkx to prove that it is optimal. The last solution we propose consists of training Deep Neural Networks (DNNs) to solve the VNF-Backup Server assignment problem. We borrow some ideas from (Lee et al. 2018) as they used a DNN approach to solve the assignment problem. Although they solve the problem for the 1:1 task to worker ratio, we take it a step further by using a neural network approach to solve for the case where the capacity of each worker can be greater than 1.

The final implementation using neural networks is dynamic in the sense that it can respond well to change in the weights. There has been research on Dynamic Bipartite Matching,

such as that by Wang et al. (2019), but in this specific problem domain we are going to assume that the VNFs that compose the service function chain are static, while the servers can be dynamic. Depending on the DNNs trained, they can account for the case when the failure probability of a server goes to 1. This means that the server is down and unavailable. We show the progression of the algorithms as well as how they respond.

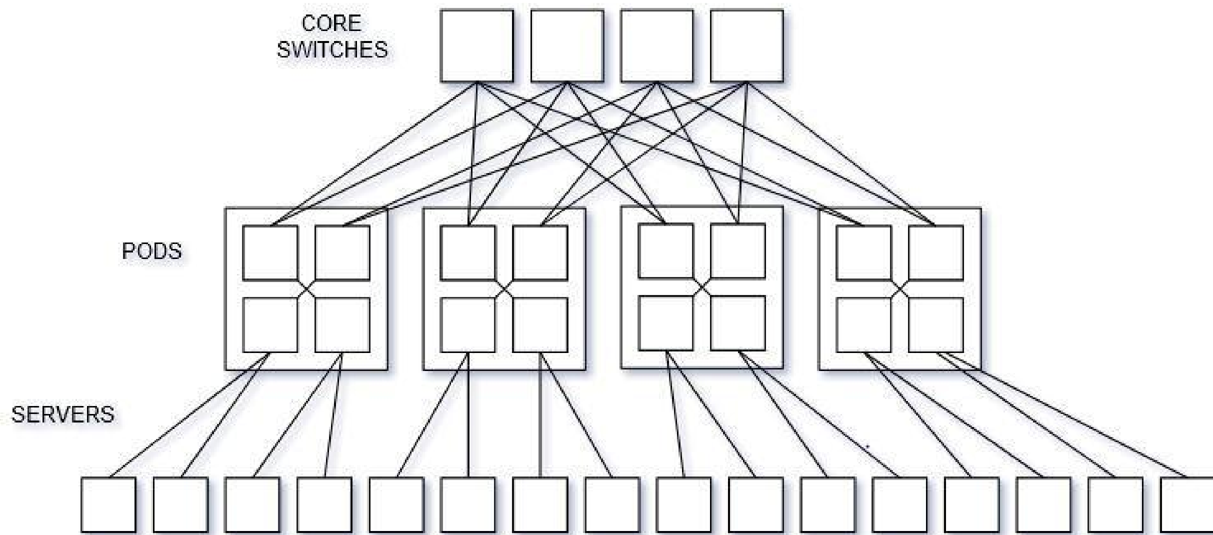
CHAPTER 2

RELATED WORKS

We are now going to survey some of the related works in the field that concern SDN, SFC, availability, as well as the assignment problem and bipartite matching. One piece of research that is very related to our work is by Kanizo et al. (2016). They go over virtual backup allocation for middleboxes and tackle the problem to maximize the probability that all VNFs have an operational instance. This will guarantee the survivability of the SFC. Their problem domain deals with maximizing the survivability of individual VNF nodes. Our approach differs in the sense that we are concerned with the availability of the Service Function Chain as a whole. In work by Khoshkholghi et al. (2020), they aim to optimize both the joint cost and latency in a SFC. The cost can be estimated by network resources, redeployment time, and energy consumption, as well as some other factors. The latency is calculated by the time it takes for traffic to pass through all of the VNF nodes in the chain. They show that finding an optimal path for service chain placement is an NP hard problem, and showcase some interesting approaches using the bee-colony and genetic algorithms. While both cost and latency are extremely important in enterprise applications, the work does not take into account the availability of the Service Function Chain. The work by Moualla et al. (2018) they go over SFC placement in fat tree data center topologies. This work concerns the insertion of an entire SFC into the fat tree topology. The scope is broader than the work we propose, since it concerns itself with SFC placement in a novel fat tree topology. The fat tree network serves as a good network topology to use for simulating service function chains. A general description of a fat tree is given by Mollah et al. (2018) in their paper on Multi-Commodity flows. See the diagram in Figure 4 for a general visual depiction of the fat tree topology.

Figure 4

Example of Fat Tree Topology



The fat tree topology consists of 3 layers of switches, with a bottom layer of hosts. In the initial research for the basis of the thesis we used a fat tree topology to simulate VNF placement. The diagram in figure 4 is for a topology with parameter k=4, where k is the number of pods in the network. In the initial phase of research, we use the fat tree topology with k=8, and assign some number of VNF to switches in the network. We then must choose backups for each VNF in the topology in order to maximize the availability. In the initial research using a fat tree network as basis, with k=8 there were 128 total nodes in the network. We randomly assign some of these nodes to be VNF hosts, and the rest can operate as backup servers. This is an example of a real-life deployment where backup servers must be assigned in the network.

Due to the ubiquity of both machine learning and reinforcement learning in the climate of today, it is necessary to perform due diligence in this area of algorithms. A tremendous reference

is the work by Xie et al. (2019) that goes over a survey of machine learning techniques applied in the SDN domain. The authors go over SDN architecture, splitting the architecture into three distinct and isolated planes. The control plane, application plane, and infrastructure plane. They then go on to give an overview of various work and algorithms used from four different domains. These domains are supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning. There are various problem spaces in SDN where learning algorithms can be applied. Some of these domains include traffic classification, routing optimization, resource management, and QoS. Both R. Shi et al. (2015) and S. I. Kim and H. S. Kim (2017) go over approaches for dynamic resource allocation in a NFV enabled environment. The latter uses a reinforcement learning approach to deploy service function chains in a cloud environment using open source technology from the OpenDaylight project. This is modern evidence that Machine Learning (ML) and Reinforcement Learning (RL) are already being used in open source SDN controllers. The OpenDaylight project is an open source initiative that has its first release in 2014. The project was started to accelerate adoption and innovation of SDN. The platform aims to provide a Model-Driven Service Abstraction layer (MD-SAL), where network devices and traditional Middle Boxes are models or objects. The project provides a mechanism to implement VNFs, and SDN controllers.

The advantage of using learning algorithms, albeit ML or RL, is that once trained, these models can provide extremely efficient optimal or near optimal solutions to many problems. Since there already exist algorithms such as the Hungarian algorithm, and minimum cost flow, that give optimal solutions in polynomial time, these techniques can be leveraged in ML and RL techniques. An RL approach is given by Wang et al. (2019) that uses a batch splitting strategy to solve the online dynamic bipartite matching problem. They model the problem as a sequential

Markov Decision Process (MDP), where at each time step, the model must choose to batch and run the Hungarian algorithm. The online version of the problem assumes that both workers and tasks can arrive or depart at any time. This is most suitable in highly dynamic environments such as taxi driver assignment, in real world applications like Uber or Lyft. For the purposes of our problem, it is assumed that the VNF nodes are constant, as the failure rate of software is constant. The server failure rates can be assumed to be variable.

Lastly, we borrow central ideas from Lee et al. (2018), where they use deep neural networks to solve linear sum assignment problems. Being that assignment is a combinatorial optimization problem, it is generally not thought to use ML or RL. In the paper they decompose the problem into multiple DNNs, in which they then use a greedy collision avoidance rule for the final result. Each DNN represents a single assignment of one worker to one task. We take the approach one step further by applying it to the backup server assignment in an SFC. In addition, we show that the approach can be used when the capacity of each server can vary. Whereas Lee et al. (2018) use the Hungarian algorithm for training their DNNs, we use our own implementation of MCF to provide training data for the DNNs. The MCF algorithm is implemented via the Successive Shortest Paths algorithm (SSP). We first compare our implementation of MCF to the implementation in the networkx library. The network can be trained on data with many different failure rates. This will generate a model which is highly adaptive and shown to be more efficient than running a static algorithm such as MCF. This puts less computational load on the SDN controller.

CHAPTER 3

PROBLEM FORMULATION

We now aim to provide a concrete formulation of the problem we aim to solve, along with notation and assumptions. The general problem can be described by some as a complete bipartite graph, $G(V, E)$ that includes a set of VNFs, and a set of servers, denoted by V_{vnf} and V_s respectively. The total number of nodes in the graph is then $V = V_{vnf}\ U\ V_s$. There will be m number of VNF nodes and n number of server nodes. Each VNF and server will have an independent probability of failure. We simplify the problem by only taking into account the failure probabilities of VNF nodes and servers. It is also possible to take into account the failure probability of the network link. The failure probability of VNF node i is denoted by q_i and the failure probability of server j is denoted by p_j. Each server will also have some finite resource capacity denoted by r_j. In some formulations of the problem, we assume that the resource capacity of all the servers is the same, denoted by R. The service function chain can be represented as $SFC = \{VNF_1, VNF_2, \ldots, VNF_m\}$ where the incoming traffic must pass through each of the VNF nodes sequentially. This is considered an ordered policy. We also assume that all incoming traffic to the network must be routed through every node in the service chain. This implies that if one VNF node goes down, the entire chain is effectively down. We assume that the failure probabilities of the VNF nodes are static. This is a reasonable assumption as these nodes are software modules, hence unless there is a change to the software, the failure rate will remain constant. The failure rates of the servers can be variable. These failure rates can change according to physical parameters such as Mean Time Between Failure (MTBF) of the hard disk, overheating, or electrical failure. The servers can also fail for software related purposes such as a security breach, failed upgrade, or virtual machine failure. With these various parameters in mind

a failure rate model for a specific server or group of servers can be developed. Table 1 below displays all the relevant parameters to the problem. These are the common parameters that will be used throughout the problem. There will be other notations added for specific algorithmic solutions.

Table 1

A Summary of Various Parameters Used in the Problem Domain

Notation	Description
V_{vnf}	The set of VNF nodes
V_s	The set of available servers
m	The number of VNF nodes
n	The number of server nodes
i	Index for backup servers $1 \leq i \leq n$
j	Index for VNF nodes $1 \leq j \leq m$
G(V,E)	Complete bipartite graph $V_{vnf} \cup V_s$
M	An SFC consisting of m VNFs $v_j, 1 \leq j \leq m$
p_i	The failure probability of server $s_i \in V_s$
q_j	The failure probability of VNF $vnf_j \in V_{vnf}$
$f_{j,i}$	Unavailability of mapping (v_j, s_i)
a	The backup server assignment function
$s_{a(j)}$	Backup server of VNF v_j under assignment a
$A(a)$	SFC availability under backup server assignment a

The assignment function consists of *m* number of *mappings* in the form of a tuple (v_j, s_i), where each $v_j \in V_{vnf}$ and each $s_i \in V_s$. In this most simple case where the resource capacity r_i of each server is 1, the two sets V_{vnf} and V_s form an injection. In the other case, where $r_i > 1$, we have a situation where each server may backup multiple VNF nodes. An optimal solution for the former case can be provided by the Hungarian algorithm in polynomial time, with multiple variants of the algorithm yielding $O(n^3)$ time complexity. The latter more general case can be

solved with algorithms such as Bellman-Ford in $O(V^2 * E^2)$ complexity where V represents the number of vertices and E the number of edges.

We now define what the *Availability* of the SFC is, and how to transform the problem so that it can be solved via combinatorial optimization techniques and Deep learning algorithms. The *Unavailability* of some VNF v_j is calculated as $f_{j,i} = q_j \cdot p_i$ where q_j and p_i and both defined above as the independent failure probabilities of the VNF node and server. It can also be thought of as the probability that both the VNF and server fail simultaneously. Thus, the Availability of VNF v_j is equal to $1 - q_j \cdot p_i$. From this preliminary definition we can then go on to define the Availability of SFC *M*. This is the probability that all of the VNF nodes are available. This is defined as $A(a) = (1 - f_{1,a(1)}) \cdot (1 - f_{2,a(2)}) \cdot ... \cdot (1 - f_{m,\, a(m)})$. The objective of maximizing service function chain availability is to maximize *A(a)* under the constraint that each server can backup at most r_i VNF nodes. To put more distinctly, $|\{j \mid i \in s(j),\ 1 \leq j \leq m\}| \leq r_i,\ \forall i \in V_s$. A mathematical transformation must be made to the problem to convert it to a Linear Sum Assignment Problem (LSAP), (Lee et al., 2018). The cost of edge in the graph must be formulated as a logarithm which is proportional to the failure probability. We propose theorem 1 below. In addition, Weighted Bipartite Matching (WBM) algorithms can be optimized when each edge cost is an integer value. Some MCF algorithm implementations such as the Ford-Fulkerson algorithm, can have instances that do not terminate when all real valued edge weights are allowed. For this reason, we eventually multiply edge weights by a large constant to convert the floating point value to an integer. We also provide our own implementation of the Successive Shortest Paths (SSP) algorithm that will work on real valued edge weights. Theorem 1 below provides a concrete definition of the assignment.

Figure 5

A Description of How to Maximize the Availability of the Assignment Function

Theorem 1: Maximizing $\mathcal{A}(a)$ is equivalent to minimizing $\sum_{1<j\leq m} f'(j,a(j))$, where $f'(j,a(j)) = \log\frac{1}{(1-f_{j,a(j)})}$.
Proof: To maximize $\mathcal{A}(a)$ is to maximize $\log\mathcal{A}(a)$, where

$$\begin{aligned} \log\mathcal{A}(a) &= \log\big((1-f_{1,a(1)})\cdot ... \cdot(1-f_{m,a(m)})\big) \\ &= \log(1-f_{1,a(1)}) + ... + \log(1-f_{m,a(m)}). \end{aligned} \tag{1}$$

Maximizing $\log\mathcal{A}(a)$ is equivalent to minimizing $-\log\mathcal{A}(a)$, where

$$-\log\mathcal{A}(a) = \log\frac{1}{(1-f_{1,a(1)})} + ... + \log\frac{1}{(1-f_{m,a(m)})}. \tag{2}$$

Let $f'(j,a(j)) = \log\frac{1}{(1-f_{j,a(j)})}$. Therefore the goal of the SAM is to minimizing $\sum_{1\leq j\leq m} f'(j,a(j))$. ∎

For the simple case with n = m = 1, there are n¡ possible assignment. This problem is well studied, and although greedy algorithms may sometimes provide good results, they can also provide arbitrarily bad results. In this case the greedy algorithm will always pick the mapping with the lowest cost. To illustrate this, you can see table 2 below.

Table 2

An example to Show that the Greedy Algorithm can Provide Poor Results

	Job 1	Job 2
Worker 1	**1**	**2**
Worker 2	**3**	**10000**

Immediately it can be seen that Job 1 will be assigned to Worker 1, and job 2 will be assigned to worker 2. This yields a total cost of 10,001. Acting non-greedily in this scenario will provide much better results, with a total cost of 5.

At this point one might wonder what advantages can be gained if there already exist solutions that can provide the optimal assignment in polynomial time. Although the solutions are optimal, they are also static and any time there is a change in one of the weights the algorithm must run again. When the number of nodes and edges become large this approach can be wasteful. It is for this reason that using a DNN for function approximation is a good solution to provide adaptability and efficiency.

Software Architecture

In order to properly compare all of the algorithms, the interface and parameters to each algorithm must be the same. We use the Python programming language for all of the work. The input to each algorithm will be a bipartite graph. As noted above, this bipartite graph consists of two sets of nodes, $V_{vnf}\ U\ V_s$. In python, we choose to represent this bipartite graph as a dictionary. The dictionary will have two keys, represented by *'vnfs'* and *'servers'*. Each VNF and server can have various attributes. For the purposes of this problem, both the VNF and server

nodes will have a *'failure_prob'* attribute. The server nodes will have an additional attribute *'r'* which is for the resource capacity. An example of this WBG represented by a python dictionary can be seen below in figure 7.

Figure 6

A WBG Represented by a Python Dictionary

```
{'servers': {0: {'failure_prob': 0.05390929955823405, 'r': 1},
             1: {'failure_prob': 0.019413843640851212, 'r': 1},
             2: {'failure_prob': 0.06585989741621394, 'r': 1},
             3: {'failure_prob': 0.113977960226358, 'r': 1},
             4: {'failure_prob': 0.07266219731647308, 'r': 1}},
 'vnfs': {0: {'failure_prob': 0.010179591795113288},
          1: {'failure_prob': 0.025691797242963575},
          2: {'failure_prob': 0.027353799678074407},
          3: {'failure_prob': 0.07813361378926502},
          4: {'failure_prob': 0.04207816890863766}}}
```

The parameters for this example are n=3, m=3, and r=1. This architecture provides a consistent means to generate WBGs. To get the availability of VNF 0 and server 0 we do $1 - q_0 \cdot p_0$, where q_0 and p_0 are the failure probabilities of the VNF and server respectively. It is also possible to manipulate the WBG to represent it in other formats, such as a *cost matrix* which is the most common representation for combinatorial optimization problems. To convert to a cost matrix, we iterate through both the VNF and server nodes, where the cost of each entry in the matrix is $log\ (\frac{1}{(1 - f_{j,a(j)})})$. An example in figure 8 below where we show the WBG represented in dictionary format, along with the corresponding cost matrix.

Figure 7

A WBG and its Corresponding Cost Matrix

```
Weighted Bipartite Graph:
{'servers': {0: {'failure_prob': 0.03507430011994193, 'r': 1},
             1: {'failure_prob': 0.03239778553611407, 'r': 1},
             2: {'failure_prob': 0.06757076870702061, 'r': 1},
             3: {'failure_prob': 0.07652990796682989, 'r': 1},
             4: {'failure_prob': 0.19089745491426852, 'r': 1}},
 'vnfs': {0: {'failure_prob': 0.025599556393773717},
          1: {'failure_prob': 0.04022603262419471},
          2: {'failure_prob': 0.04191938406006361},
          3: {'failure_prob': 0.04409148873326274},
          4: {'failure_prob': 0.08329748961932519}}}
Cost Matrix:
array([[0.00089829, 0.00082971, 0.00173128, 0.00196105, 0.00489887],
       [0.0014119 , 0.00130408, 0.0027218 , 0.00308324, 0.00770868],
       [0.00147137, 0.00135902, 0.00283654, 0.00321324, 0.00803449],
       [0.00154768, 0.00142949, 0.00298374, 0.00338002, 0.00845258],
       [0.00292588, 0.0027023 , 0.00564437, 0.00639515, 0.01602906]])
```

Converting the WBG to cost matrix format will eventually work better when training the DNNs in the neural network model, as matrices are very natural to work with in DNN algorithms. Similar to how we convert the WBG into a cost matrix to use as input in the DNNs, we can also convert the WBG into a *flow network* to use as input into the Minimum Cost Flow algorithm. To accomplish this the primary step must be to append a *source* and *sink* node to our graph. We rely upon the popular Networkx python library to accurately represent a flow network. This is a directed graph where each edge has some capacity and some flow through it. The amount of flow going through the edge cannot exceed its capacity. Both the source and the sink node must have a demand, where the sum of all demands in the graph must add up to zero. To

send flow indicates a negative demand, while to receive flow there must be a positive demand. This will be elaborated on further when we discuss MCF in detail. To summarize, we can represent the problem in three different ways depending on the algorithm we use. The three different formulations are a Weighted Bipartite Graph (WBG), Cost Matrix, and Flow Network.

CHAPTER 4

A GREEDY APPROACH: SGA & RSGA

We propose two similar greedy algorithms to the problem as a basis for comparison, the Sorted Greedy Algorithm (SGA), and Reverse Sorted Greedy Algorithm (RSGA). Although the algorithms SGA and RSGA are almost identical in terms of implementation they vary substantially in results, with SGA performing much better. This is because of Proposition 1 from Meng et al. (2010). This is possible because we are given extra information in the problem that is not given in many variants of the assignment problem. We are not just given the edge weights, but we are given the individual probabilities of failure for each VNF and server. In the SGA algorithm, we match the VNF of highest failure with the backup server of lowest failure. We call this Best-Worst, or BW matching. In RSGA we match the VNF of lowest failure with the server of lowest failure, and call the Best-Worst, or BW matching. Simulations show that the performance of SGA is considerably better especially as the number of VNF nodes increase and the capacity of each server is greater than 1. The advantage of these algorithms is that they are extremely efficient, relying on a simple sort. The disadvantage being that they do not guarantee an optimal solution. Despite the lack of optimality, a greedy approach to this problem is intuitive and can be used as a baseline to compare to more robust algoirithms. Pseudo code for both algorithms will be shown on the following 2 pages.

Figure 8

Pseudo Code for SGA Algorithm

```
   Algorithm 1: Sorted Greedy Algorithm for SAM.
Input: Data center G(V, E) with failure probability p_i for
   backup server s_i and failure probability q_j for VNF v_j;
Output: VNF backup server assignment a and SFC
   availability A(a).
0.    Notations:
      l: index of the resources for VNF backup servers;
      flag: assignment is done or not, initially false;
1.    Sort all the backup servers in non-descending order
         of q_i; WLOG, q_1 ≤ q_2 ≤ ... ≤ q_|V_s|;
2.    Sort all the VNFs in non-ascending order of p_j;
         WLOG, p_1 ≥ p_2 ≥ ... ≥ p_m;
3.    l = 1;
4.    for (i = 1; i ≤ |V_s|; i++)
5.       for (j = l; j < l + r_i; j++)
6.          a(j) = i; // Assign server s_i to v_j
7.          if (j ≥ m)
8.             flag = true;
9.             break;
10.         end if;
11.      end for;
12.      if (flag == true) break;
13.      end if;
14.      l = l + r_i; // Assign next backup server;
15.   end for;
16.   RETURN  a and A(a).
```

Figure 9

Pseudo Code for RSGA Algorithm

Algorithm 2: Reverse Sorted Greedy Algorithm for SAM.

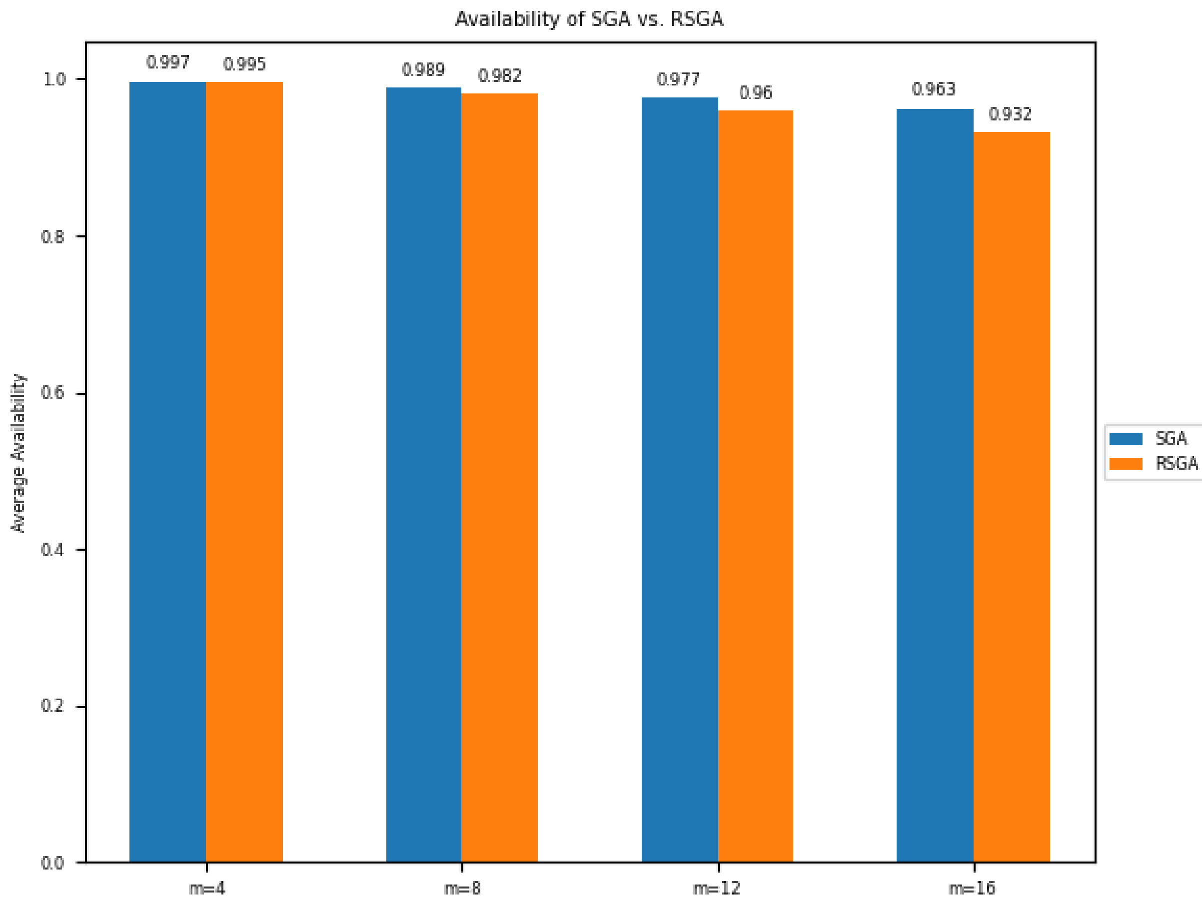
Input: Data center $G(V, E)$ with failure probability p_i for backup server s_i and failure probability q_j for VNF v_j;

Output: VNF backup server assignment a and SFC availability $\mathcal{A}(a)$.

0. **Notations**:
   l: index of the resources for VNF backup servers;
   $flag$: assignment is done or not, initially false;
1. Sort all the backup servers in non-descending order of q_i; WLOG, $q_1 \le q_2 \le ... \le q_{|V_s|}$;
2. Sort all the VNFs in non-descending order of p_j; WLOG, $p_1 \le p_2 \le ... \le p_m$;
3. $l = 1$;
4. **for** $(i = 1;\ i \le |V_s|;\ i{+}{+})$
5.   **for** $(j = l;\ j < l + r_i;\ i{+}{+})$
6. $a(j) = i$; // Assign server s_i to v_j
7.   **if** $(j \ge m)$
8. $flag = true$;
9.   **break**;
10.   **end if;**
11.   **end for;**
12.   **if** $(flag == true)$ **break**;
13.   **end if;**
14. $l = l + r_i$; // Assign next backup server;
15. **end for;**
16. **RETURN** a and $\mathcal{A}(a)$.

Figure 10

A Direct Comparison of SGA and RSGA

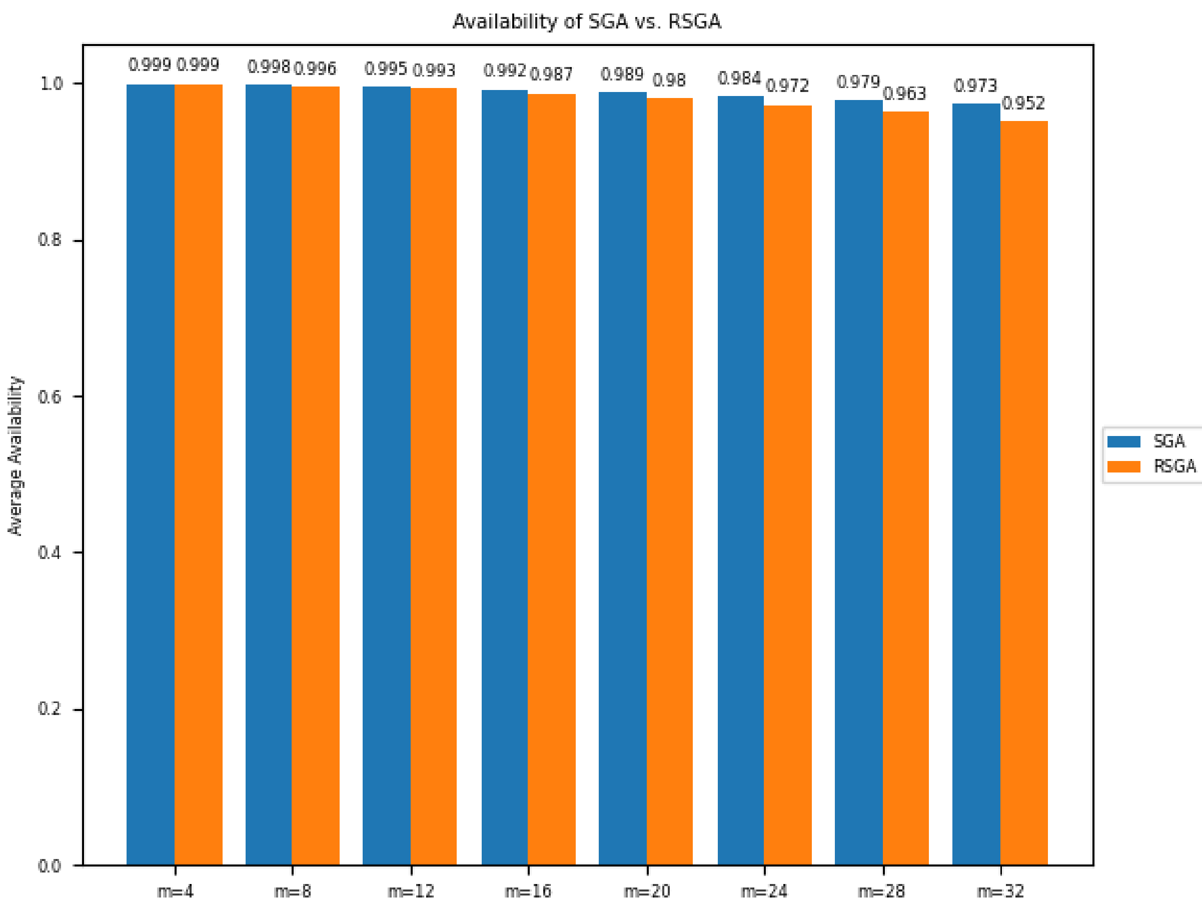


In the figure above, we see a direct comparison for SGA vs. RSGA where the number of VNF nodes varies from 4 to 16. In this simulation the other parameters are kept constant with n=24 and r=1. The failure probability for the VNF nodes is randomly assigned in the range (0, 0.1), and for the servers we have (0, 0.2). For each one of the algorithms, we ran 1000 simulations then took the average availability. It can be seen that on average the SGA algorithm performs approximately 3% better than SGA when the resource capacity of each server is 1. We show an additional comparison for the case of a resource capacity greater than one. This case shows that SGA starts to perform substantially better when the number of VNF nodes increases.

For this second comparison the number of servers is kept constant at 48, while the resource capacity is constant at 3. When there is n > 12, we see anywhere from a 6% to 15% better availability for SGA. Both algorithms have the same running time since they are essentially just sorting algorithms. From this empirical data, a good approach would be to use RSGA for small number of VNF n less than 12, and SGA for n > 12.

Figure 11

A Comparison of SGA vs. RSGA when r = 3 and n = 48



CHAPTER 5

MCF ALGORITHM

Now that we have discussed SGA and RSGA, we can see their limitations. Although extremely efficient, only bounded by the running time of modern sorting algorithms, there is no guarantee of an optimal solution. This is where we can convert the problem into a flow network, and apply a MCF algorithm to find the optimal assignment. We first start by elaborating on what a flow network is, and then show how a WBG can be converted into a flow network. A flow network is some directed graph $G = (V, E)$, with a source node $s \in V$ and a sink node $t \in V$. For our problem, $V = V_{vnf} \cup V_s$, and E consists of all edges in the complete bipartite graph. It should be noted that the edge costs from the source to each VNF node will be 0, and the edge costs from each server to the sink will also be 0. The only edges that have non-zero weights are the edges connecting each VNF to the servers. Each edge weight is given by $log\ (\frac{1}{(1 - f_{j, a(j)})})$. Another integral part of a flow network is that each edge must have some capacity. The capacity of each edge (v_j, s_i) is equal to the resource capacity of the associated server. We utilize the Networkx Python library to represent the DiGraph object. The procedure is given below for how to convert a general bipartite graph into a flow network for the purposes of applying a MCF algorithm.

Figure 12

Converting WBG into a Flow Network

Graph Transformation. Next, we transform the data center graph $G(V, E)$ into a flow network $G'(V', E')$ shown in Fig. 1 following below five steps.

Step I. $V' = \{s\} \cup \{t\} \cup M \cup V_s$. Here, s is the source node and t is the sink node in the flow network, $M = \{v_1, v_2, ..., v_m\}$ is the set of m VNFs, and V_s is the set of backup servers.

Step II. $E' = \{(s, v_j)\} \cup \{(v_j, s_i)\} \cup \{(s_i, t)\}$, where $v_j \in M$ and $s_i \in V_s$. Note that it is a complete bipartite graph between M and V_s.

Step III. For each edge (s, v_j), set its capacity as 1 and cost as 0. For each edge (s_i, t), set its capacity as r_i, the resource capacity of s_i, and cost as 0.

Step IV. For each edge (v_j, s_i), set its capacity as 1 and cost as $f'(j, i)$.

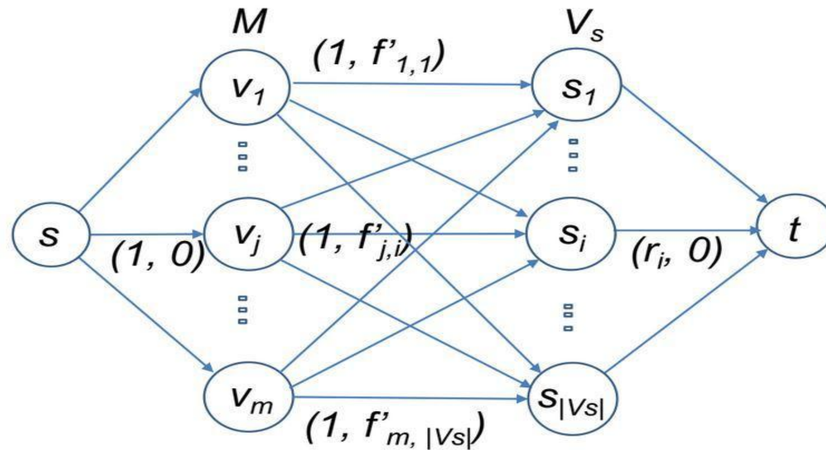
Step V. Set the supply at s and the demand at t as m, the number of VNFs in the SFC M.

Thus, $|V'| = m + |V_s| + 2$ and $|E'| = m + |V_s| + m \cdot |V_s|$.

The most important aspect of converting the WBG into a flow network is that the capacities and weights are initialized correctly. For any flow network to be solvable via a MCF algorithm, the overall demand in the network must sum to 0. Mathematically $\sum_{i=0}^{m+n+2} d_i = 0$, where d_i is the demand of any given node in the flow network. If this constraint is not met then the flow network cannot be solved via MCF. The graph consists of a WBG in the middle, with source and sink nodes added at the beginning and end of the graph. It is important to note that the source and sink nodes are simply placeholders to convert the WBG into a flow network, and do not represent real VNF or server nodes in the network. The procedure will produce a graph as shown in the diagram below

Figure 13

Flow Network Example



Next, we discuss the Successive Shortest Paths (SSP) algorithm. To understand SSP also requires understanding the concept of a *residual graph* because the SSP works by running some common shortest path algorithm, such as Dijkstra's algorithm successively, until only the edges which represent the maximal assignment are left with positive flow. It should be noted that the algorithm is guaranteed to terminate after *m* runs, where *m* is the number of VNF in the problem. To represent a visual example of the SSP algorithm, we show the following graph *G(V, E)* with source and sink node attached. At each step of the algorithm, we run some shortest path algorithm, such as Dijkstra's algorithm, we then convert the graph into its residual counterpart. We must repeat this process *m* times. The algorithm will terminate once there are no possible paths left. At this point we can then analyze the final residual graph and determine the edges that have net positive flow. These edges that are left with positive flow will give the optimal assignment. Below is the pseudocode for the SSP algorithm.

Figure 14

Pseudo Code for SSP

Algorithm 1 Successive Shortest Paths

```
F ← flowNetwork(G)
i ← m
A ← ()
  while i ≠ 0 do
    path ← Dijkstra(F)
    F ← Residual(F, path)
    i ← i − 1
  end while
  for edge in F do
    if flow > 0 then
      A ← A.append(edge)
    end if
  end for
  return A
```

The primary premise of the algorithm is that it makes use of the idea of a residual graph. When some edge has $flow \geq c_{i,j}$ where *c* is the capacity of the edge between nodes *(i, j)*, then it can no longer accept any flow. It is for this reason that the residual edge is added. The residual edge is added in the reverse direction of the original edge and the weight is negative. This is what makes the algorithm powerful because it essentially allows *backtracking*. In the examples below you will see 4 different plots. A summary of the plots is given above. The algorithm terminates after three runs because that is the number of VNF nodes in the graph.

Table 3

Summary of Residual Graphs

Plot 1	Original Graph
Plot 2	Residual Graph 1
Plot 3	Residual Graph 2
Plot 4	Residual Graph 3

Figure 15

Original Graph

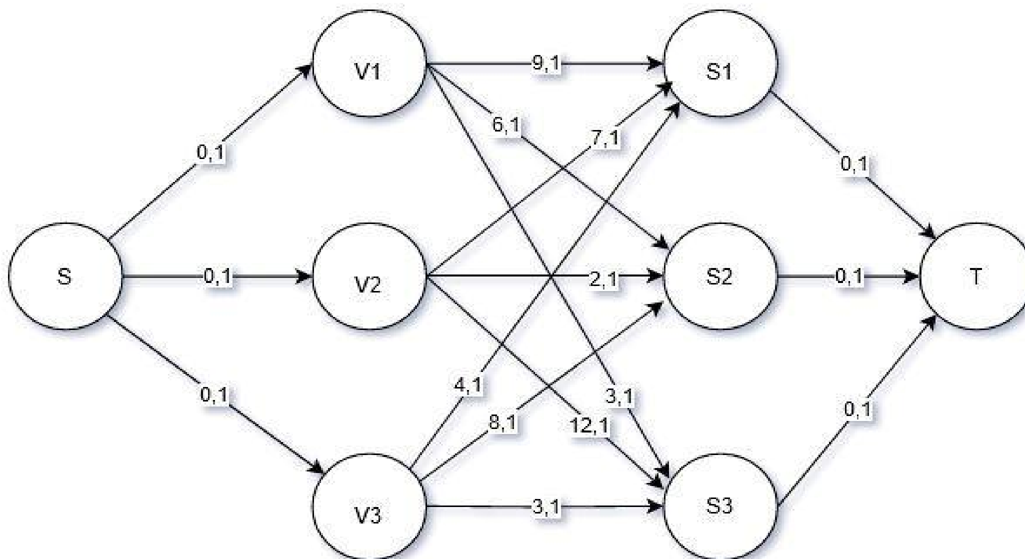


Figure 16

Residual Graph 1

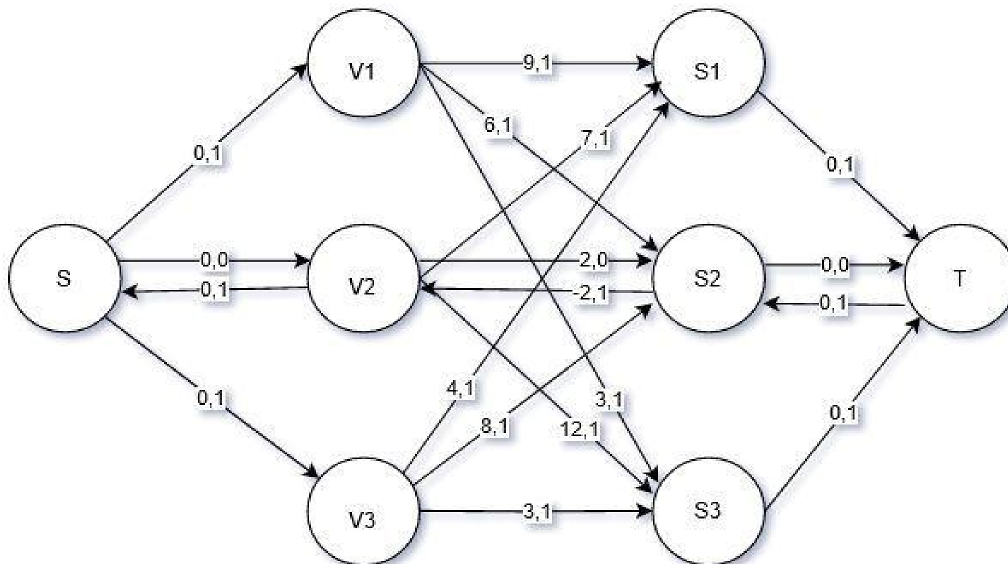


Figure 17

Residual Graph 2

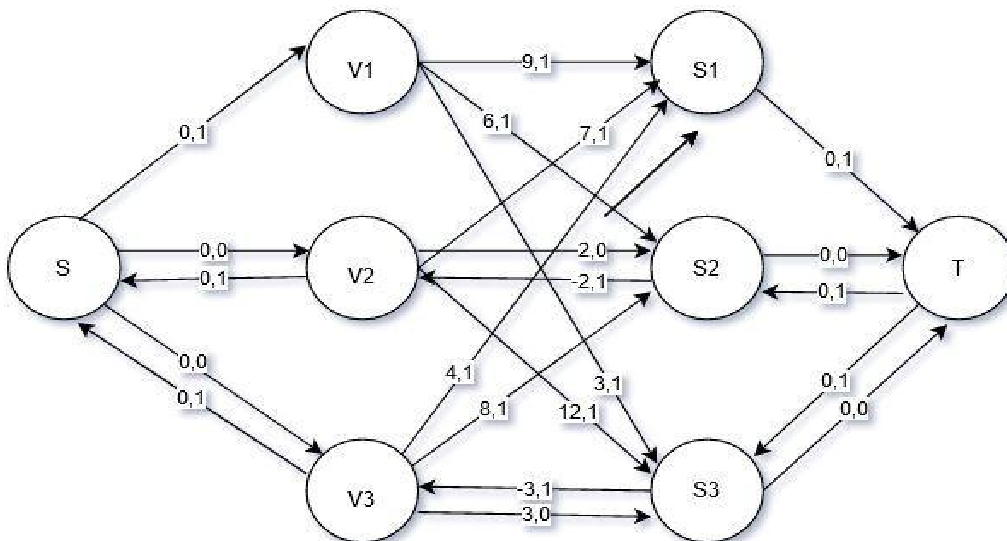
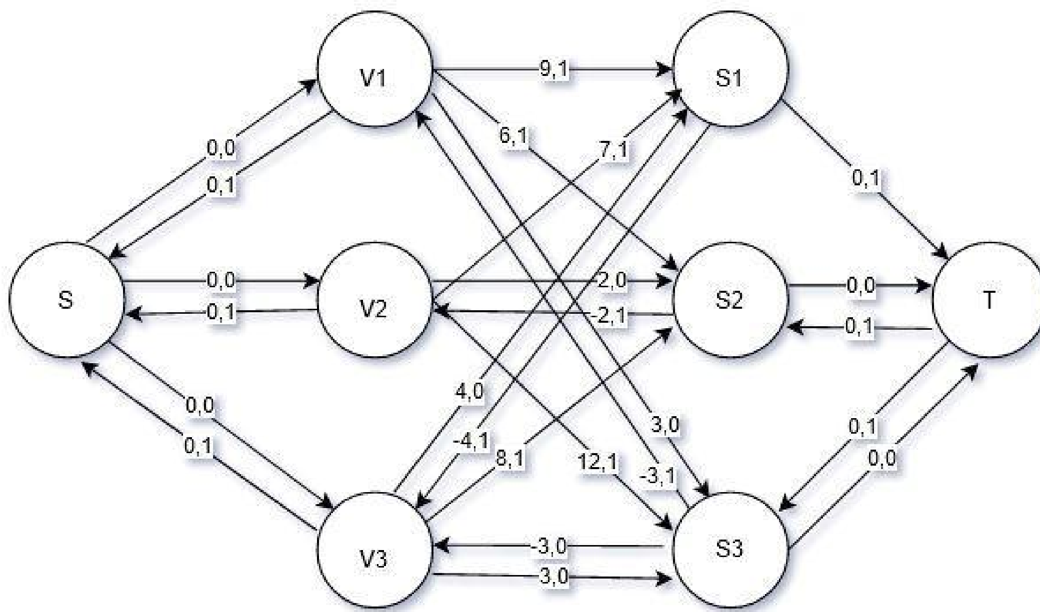


Figure 18

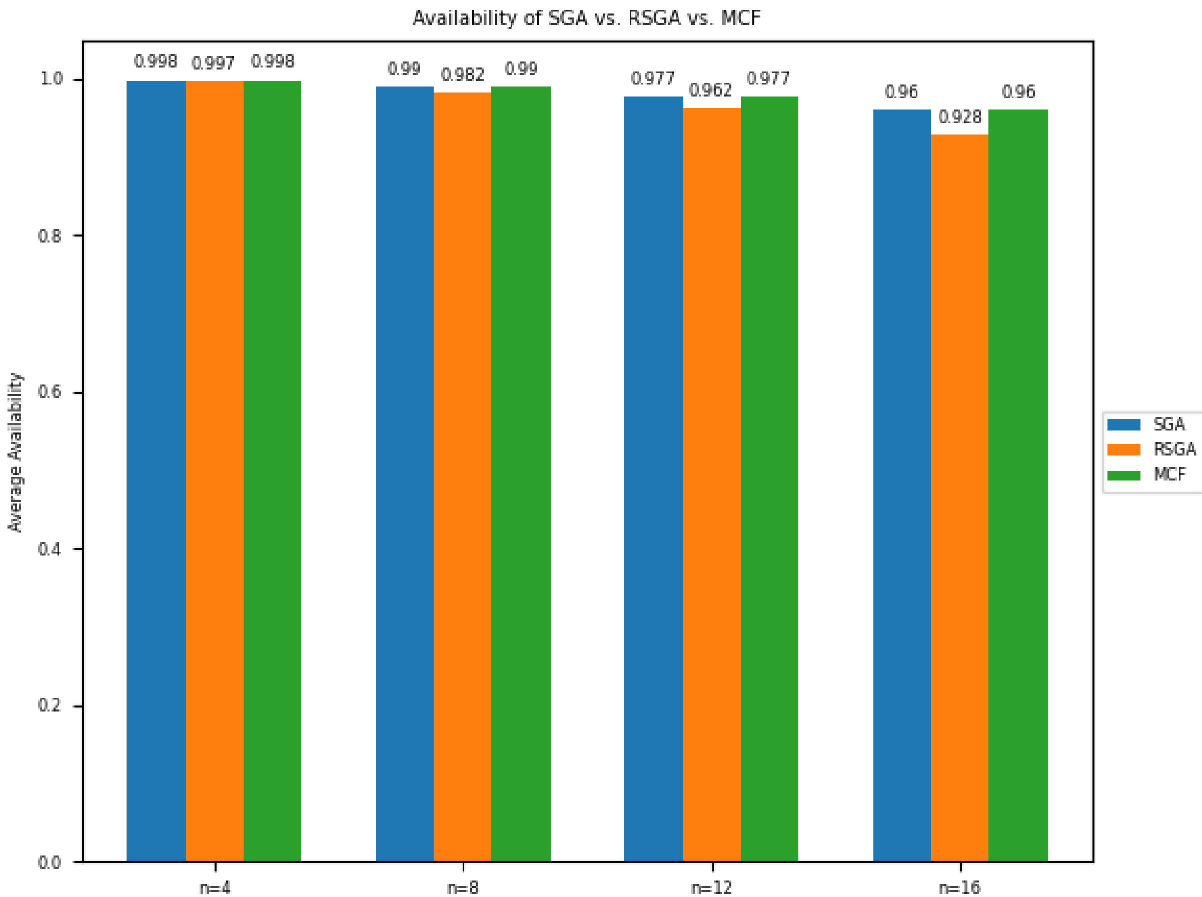
Residual Graph 3



Now that the idea behind the algorithm is clear, we analyze both the time complexity and the results of SGA, RSGA, and MCF using the SSP algorithm. The time complexity of SGA and RSGA are simply bounded by that of modern sorting algorithms. We assume a complexity $O((n + m)log(n + m))$. As for the successive shortest paths algorithm, we know that the algorithm will terminate in *n* runs, where in each run a shortest path algorithm is used. Since Dijkstra's algorithm is used for shortest path, it has $O(nm)$ complexity. This gives a complexity of $O(n^2m)$. There are two cases in the problem domain, when r=1, and when r>1. Since both cases are distinct, we provide simulation data for each. Below is a comparison of all three algorithms for the case of r=1, m=24, VNF failure range = (0, 0.1), and server failure range (0, 0.2).

Figure 19

SGA vs. RSGA vs. MCF Plot 1



Note: Simulation data for r=1, n=24

The algorithms perform very similar when the number of VNF nodes is relatively small. As the number of VNF increase, RSGA becomes obsolete as both SGA and MCF perform similarly. There is really no reason to use RSGA at this point since SGA performs better and has the same time complexity.

CHAPTER 6

NEURAL NETWORK APPROACH

Given that we already have algorithms that can yield optimal solutions for both the assignment problem and the generalized assignment problem, it is natural that we can generate training data to train a Neural network. We borrow some ideas from Lee et al. (2018) as they were able to use a deep learning approach to solve Linear Sum Assignment Problems (LSAP). The general idea is that given some WBG, we can run either the Hungarian algorithm or a MCF algorithm to produce the assignment function a and $A(a)$, the SFC availability for said assignment. At first glance, one might think to use the cost matrix $C = \{ c_{ij} \}$ as input, and an assignment matrix $Y = \{ y_{ij} \}$ as output, but this would lead to a large network and is not easily parallelizable. Rather, we can us a Combined Deep Neural Network approach to generate the optimal assignment a WBG with 5-tuple $(m, n, R, (q_{min}, q_{max}), (p_{min}, p_{max}))$ where the values of the 5-tuple are described below.

Table 4

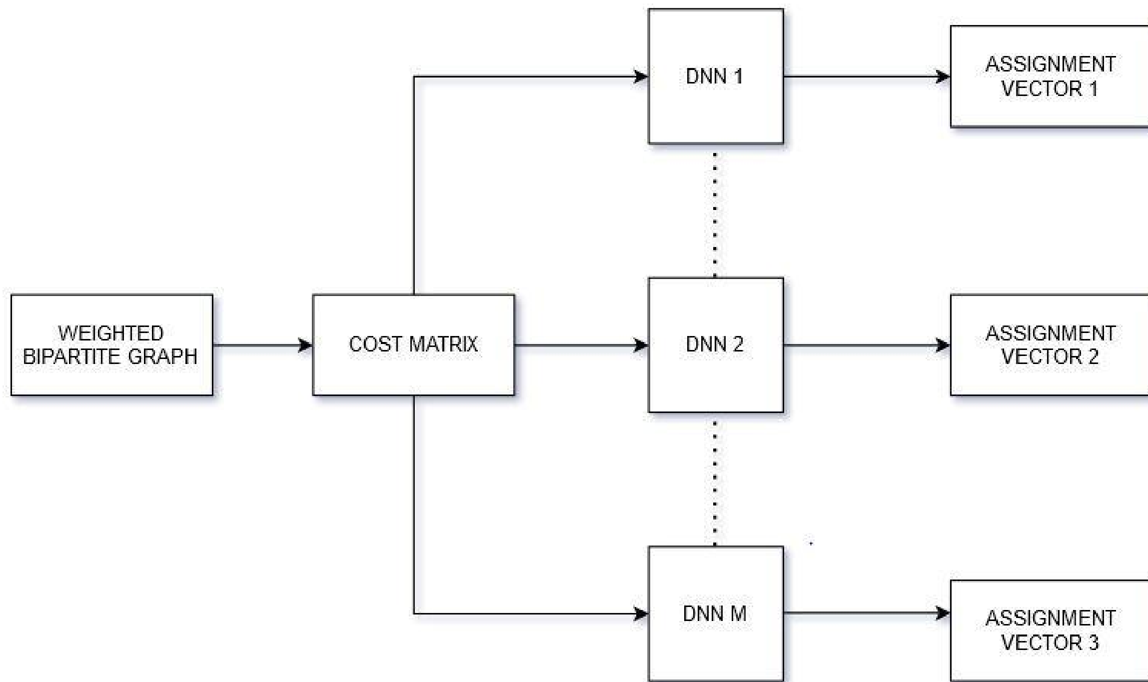
Description of Algorithm 5-tuple

m	Number of VNF
n	Number of servers
R	Resource capacity
$(q_{min,} q_{max})$	VNF failure probability range
(p_{min}, p_{max})	Server failure probability range

With this 5-tuple, we can generate a set of *training instances* to train the DNNs on. There will be one DNN for every VNF we have in the graph. Once we train the DNN networkx, each network will output a *one-hot* n-vector which signifies *a(j,i)*, or that VNF *i* has been mapped to server *j*. A visual representation of the output can be seen below.

Figure 20

DNN Architecture



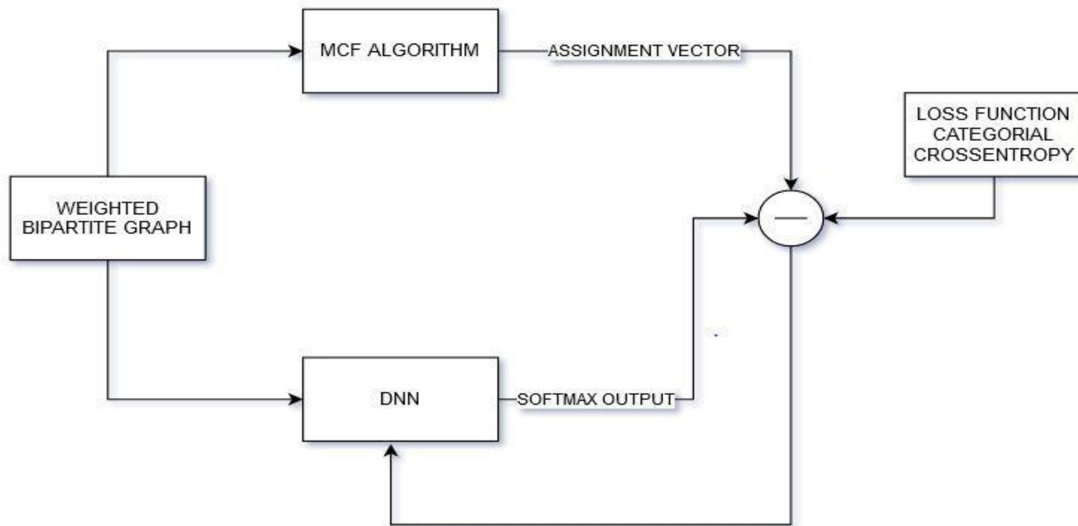
Because of the robustness of Deep Neural Networks in their ability to model function approximators for complex tasks, this same approach will also work on networks that have resource capacity $R > 1$. It can also work for networks where the resource capacity of each server may vary. This is beneficial in environments like cloud computing.

Now that we have discussed the general network architecture, we will go into depth to explain the training phase. As the data set, we generate 10000 training instances of WBGs. Each WBG will have a corresponding cost matrix. If the parameters are $n = m = 3$, with $r = 1$, then there will be a corresponding 3x3 cost matrix. As input to each DNN, we flatten the matrix, so a 3x3 cost matrix will be flattened to a 9x1 vector. To train the network, we provide the input vector, as well as a one-hot vector which represents the assignment for that particular cost

matrix. For instance, for VNF node 1 we might have a cost vector $[c_{00,} c_{01,} \dots, c_{33}]$, with a corresponding output vector *[0, 0, 1]*. This means that VNF 1 is mapped to server number 3, or in other words $s_{a(j)} = 3$. The training data is all generated by running the MCF algorithm, since MCF provides the optimal assignment. For a depiction of the training process, see the diagram below.

Figure 21

Training Feedback Loop

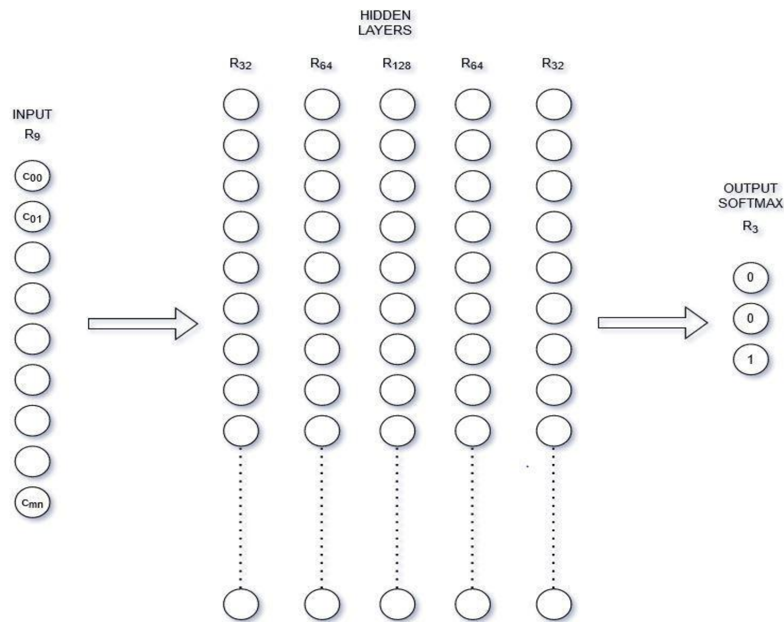


The general premise is that we compute the optimal solution with MCF, and then use this data to train the network with categorical cross entropy loss function. We use this choice of loss function because this is a *Multi-Class Classification* problem. This area is well studied in machine learning and involves choosing the correct class from some input data. Since each assignment vector is a one-hot vector, the softmax activation function is best suited because the output vector will form a probability distribution. Then, finding the correct assignment simply

involves running the *argmax()* function on the output vector. The server that has the highest probability will be the server that is assigned to that particular VNF. The general network architecture will have some variation depending on input parameters, but in general we have *Feed-Forward Neural Network* FFNN, with an input layer, output layer, and 5 hidden layers. The hidden layers consist of 32, 64, 128, 64, and 32 neutrons respectively. The activation function chosen for each hidden layer is *relu* activation function since this activation function has very fast results and is standard in training feed forward networks. The most important aspect is the activation function for the output layer. This is why the softmax function is chosen. Since we want the results to be mutually exclusive, softmax is integral to obtaining a proper solution. A few other parameters are the learning rate, loss function, batch size, and epochs. After some testing it seemed that a learning rate $\alpha = 0.003$ was the best choice for relatively fast convergence and adequate accuracy. The networks were able to reach ~95% accuracy after 2000 epochs of training. After 5000 epochs each DNN had reached over 99% accuracy which is sufficient for this problem. Categorical Crossentropy is the de facto loss function standard for multi-class classification problems. The training process must be repeated for each DNN, for n=m=3 we must train 3 separate DNN networks then use each generated model to predict the server that is assigned for that particular VNF. Below you can see a diagram that shows the architecture of each DNN. The DNN shown in this example is for the parameters n=m=3, and r=1.

Figure 22

DNN Layered Architecture



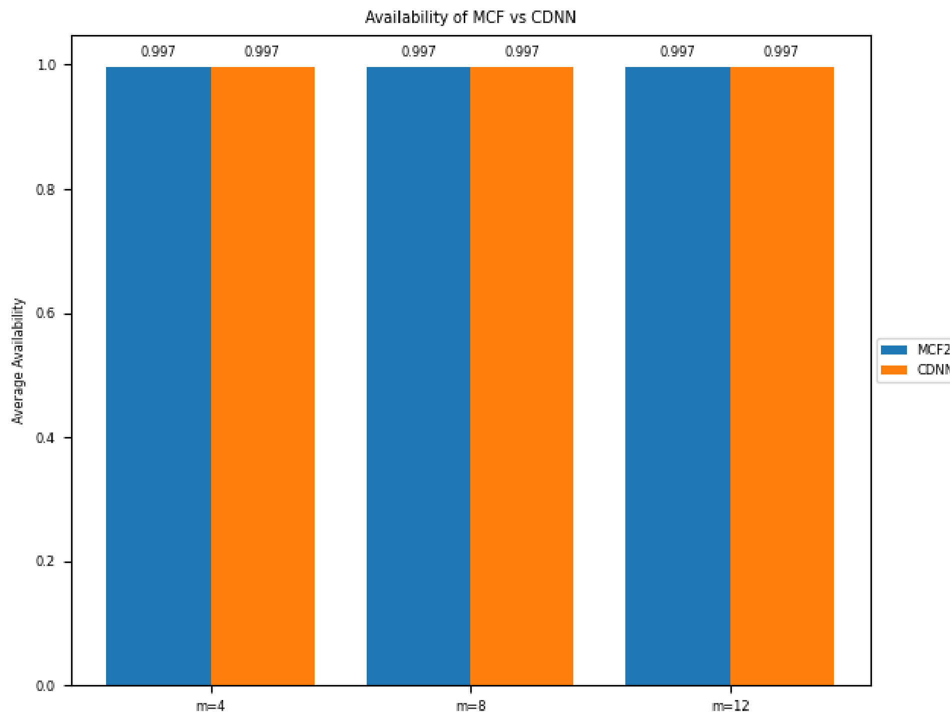
At each hidden layer we use the *relu* activation function, and for the output we apply the softmax activation function. The model can be trained for an arbitrary number of VNF and server nodes, as well as different capacities. For larger inputs, the number of neurons may need to be adjusted. Say for n=m=6, we would have a R^{36} input layer, since the cost matrix would be 6x6. In this scenario the first hidden layer would need to be adjusted to say, 64 neurons instead of 32.

The primary advantage of using DNNs over the static algorithms like MCF and the Hungarian algorithm, is that the network can be trained over a wide range of failure probabilities, with 99% accuracy. Once the model is trained, it will be able to provide accurate results extremely close to MCF. As an initial check, we perform a calculation for n=m=3, with r=1. The failure probability range for the VNF nodes is (0, 0.1), and for servers (0, 0.2). After running

both algorithms on 100 different graphs, we compute the average availability of both. The algorithms each performed the same having a 0.988 average availability. This is proof that the DNN approach works just as well as MCF when trained properly. The only caveat is that when one of the primary network parameters (m, n, r) change, then an entire neural network must be trained. Below are results for the case where we vary m from the range 4 to 12, while keeping n constant at 24, with the same failure probabilities. Since the DNN networks have been trained to over 99% accuracy, they match the optimal solution of MCF. The advantage is that the running time of the DNN is 31% faster. The plot below shows results for resource capacity 1, while keeping the servers constant at 24.

Figure 23

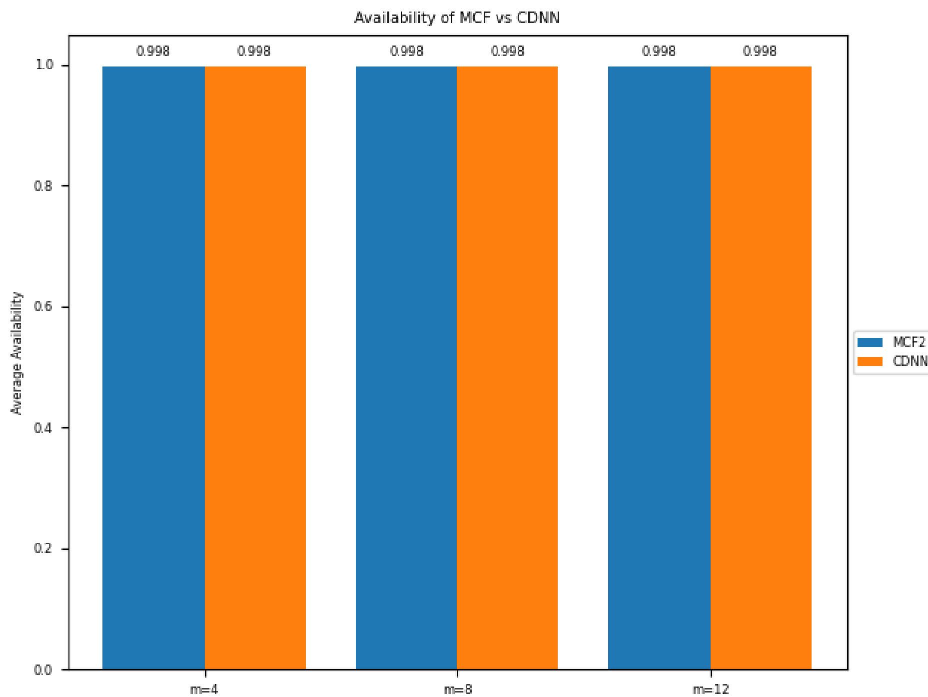
DNN vs. MCF



These results were calculated as the average availability over 1000 simulations, the availability for MCF and CDNN were nearly identical, with small discrepancies only appearing due to floating point errors. For this specific simulation the running time over 1000 simulations was 31% faster for the DNN, while also producing optimal solutions. We provide another example with the same parameters, except that we change the resource capacity to three.

Figure 24

DNN vs. MCF for r=3



Note: Running time over 1000 simulations for MCF-11.1 seconds, DNN-7.7 seconds

The running time for MCF was 11.1 seconds, while the running time for DNN was about 7.7 seconds. These results are a clear indicator that using DNNs to solve already known problems can provide a boost in efficiency.

CHAPTER 7

CONCLUSION AND FUTURE WORKS

In this thesis we analyzed the problem of SFC availability in the growing field of Software Defined Networking. We analyzed multiple algorithms of increasing complexity which consist of SGA, RSGA, MCF, and a Neural Network approach. We showed that while the greedy algorithms SGA and RSGA provide respectable results in most scenarios, they do not guarantee optimal solutions and can provide arbitrarily bad solutions. We then implemented solved the assignment problem by implementing the MCF algorithm via Shortest Successive Paths. While these algorithms have been around for some time, they have yet to be used in this problem domain to our knowledge. To take it a step further we trained Neural Networks to solve the problem and provide optimal solutions over 99% of the time. The advantage of Neural Networks is that they provide up to a 31% increase in time efficiency compared to the MCF algorithm. Future work will certainly involve incorporating more parameters, such as the cost to repopulate a server with a given VNF. We can also incorporate latency, and simultaneously attempt to optimize both availability and latency through the service chain. This may require using reinforcement learning or borrowing ideas from game theory. These algorithms and ideas may be put to use in real life through projects like the OpenDaylight project.

REFERENCES

Kang, R., He, F., & Oki, E. (2021). Virtual network function allocation in service function chains using backups with availability schedule. *IEEE Transactions on Network and Service Management*, *18*(4), 4294-4310. https://doi.org/10.1109/tnsm.2021.3096254

Kanizo, Y., Rottenstreich, O., Segall, I., & Yallouz, J. (2016). Optimizing virtual backup allocation for middleboxes. *2016 IEEE 24th International Conference on Network Protocols (ICNP)*. https://doi.org/10.1109/icnp.2016.7784411

Khoshkholghi, M. A., Gokan Khan, M., Alizadeh Noghani, K., Taheri, J., Bhamare, D., Kassler, A., Xiang, Z., Deng, S., & Yang, X. (2020). Service function chain placement for joint cost and latency optimization. *Mobile Networks and Applications*, *25*(6), 2191-2205. https://doi.org/10.1007/s11036-020-01661-w

Kim, S. I., & Kim, H. S. (2017). A research on dynamic service function chaining based on reinforcement learning using resource usage. *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*. https://doi.org/10.1109/icufn.2017.7993856

Lee, M., Xiong, Y., Yu, G., & Li, G. Y. (2018). Deep neural networks for linear sum assignment problems. *IEEE Wireless Communications Letters*, *7*(6), 962-965. https://doi.org/10.1109/lwc.2018.2843359

Mollah, M. A., Yuan, X., Pakin, S., & Lang, M. (2015). Fast calculation of Max-MIN fair rates for multi-commodity flows in fat-tree networks. *2015 IEEE International Conference on Cluster Computing*. https://doi.org/10.1109/cluster.2015.56

Moualla, G., Turletti, T., & Saucez, D. (2018). An availability-aware SFC placement algorithm for fat-tree data centers. *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*. https://doi.org/10.1109/cloudnet.2018.8549338

Ruiz, L., Duran, R. J., De Miguel, I., Merayo, N., Aguado, J. C., Fernandez, P., Lorenzo, R. M., & Abril, E. J. (2020). Comparison of different protection schemes in the design of VNF-mapping with VNF resiliency. *2020 22nd International Conference on Transparent Optical Networks (ICTON)*. https://doi.org/10.1109/icton51198.2020.9203066

Shi, R., Zhang, J., Chu, W., Bao, Q., Jin, X., Gong, C., Zhu, Q., Yu, C., & Rosenberg, S. (2015). MDP and machine learning-based cost-optimization of dynamic resource allocation for network function virtualization. *2015 IEEE International Conference on Services Computing*. https://doi.org/10.1109/scc.2015.19

Wang, M., Cheng, B., Zhao, S., Li, B., Feng, W., & Chen, J. (2019). Availability-aware service chain composition and mapping in NFV-enabled networks. *2019 IEEE International Conference on Web Services (ICWS)*. https://doi.org/10.1109/icws.2019.00028

Wang, Y., Tong, Y., Long, C., Xu, P., Xu, K., & Lv, W. (2019). Adaptive dynamic bipartite graph matching: A reinforcement learning approach. *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. https://doi.org/10.1109/icde.2019.00133

Xie, J., Yu, F. R., Huang, T., Xie, R., Liu, J., Wang, C., & Liu, Y. (2019). A survey of machine learning techniques applied to software defined networking (SDN): Research issues and challenges. *IEEE Communications Surveys & Tutorials*, *21*(1), 393-430. https://doi.org/10.1109

APPENDIX A: GITHUB REPOSITORY

https://github.com/gunslingster/backup_server_assignment