ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

UNIVERSITY OF PIRAEUS

THESIS:

Algorithms for Network Functions Coordination and Placement in Network Function Virtualization (NFV) simulated environment

Author: Christos Kopsacheilis

Supervisor: Kostantinos Tsagkaris

Athens 2021

Abstract

Network Function Virtualization (NFV) is the current concept used from the majority of the operators that decouples network functions (such as firewalls, DNS, NATs, load balancers, etc.) from dedicated hardware devices (the traditional expensive middleboxes). This decoupling enables hosting of network services, known as Virtualized Network Functions (VNFs), on commodity hardware (such as switches or servers) and thus facilitates and accelerates service deployment and management by providers, improves flexibility, leads to efficient and scalable resource usage, and reduces costs. This paradigm is a major turning point in the evolution of networking, as it introduces high expectations for enhanced economical network services, as well as major technical challenges.

To the new generation

Contents

List of Figures

1 Introduction

Nowadays, Information Technology companies and academics use more and more about the Network Function Virtualization (NFV) concept. In fact, Communications Service Providers (CSPs) and Internet Service Providers (ISPs) are facing competition from Over-the-top (OTT) media services and web services, experiencing declining average revenue per user and feeling the pressure to innovate rapidly to respond to new trends such as 5G(preparing 6G as well), IoT (Internet of Things), and cloud edge computing.

Traditional network services are built by chaining together proprietary single-function boxes (middleboxes). The design of these services is custom, the thechnol-

ogy used is expensive, require lengthy prior analysis before implementing and most of the times cannot be shared with any other service. Once deployed, the operations and management of these services is largely non-automated, with each box presenting its own management interface. This technique of creating network services is very expensive, and offers no practical way of creating dynamic services.

Network Function Virtualization (NFV) is the trend-technology currently used by most of the operators, that can greatly assist in solving these business challenges. Once virtualized, the Virtual Network Functions (VNFs) can be hosted on an industry-standard server or commodity hardware. Virtualization does not stop at replacing physical boxes with virtual machines, but can go further by using microservices, containers, and cloud native architectures. Managing the Lifecycle of these services (such as initial deployment, configuration changes, upgrades, scale-out, scale-in, self-healing, etc.), can also be automated. These VNFs can also be chained and managed in a dynamic and automated fashion. All these advances enable the creation and management of agile network services.

This thesis addresses some baseline algorithms along with its results, of these virtualized network functions and services in a simulation-environment. That is the problem for coordination of service mesh consisting of multiple microservices. Includes Non-RL algorithms (Random Schedule, Shortest Path, and Load Balance). This topic is always under constant analysis and research from many operators , as the coordination of the services is a complicated a problem and proposals for better solutions are currently analyzed from many Research Departments.

1.1 Thesis organization

This thesis is organized into 4 core chapters (introduction not included). Besides the present chapter introducing the context , objectives and contributions of the thesis, the manuscript is organized as follows:

- **Chapter 2** provides the background information related to this thesis. It presents an overview of NFV and SDN. The chapter also gives an overview of the state of the art of the resource allocation problem in NFV and reviews networking challenges in cloud environments.
- **Chapter 3** provides the main topic of the thesis , which are the algorithms proposed for the VNF - Service function (SF) service coordination and flow scheduling problem along with the results by using the proposed algorithms.
- **Chapter 4** provided some conclusions regarding the chapter 3 analysis.

- **Chapter 5** provides some future directions, regarding the thesis Problem and what can be further researched.

2 State of the Art

2.1 Introduction

Software-Defined Networking (SDN) and Network Function Virtualization (NFV) are enabling network programmability and the automated provisioning of virtual networking services. Combining these new paradigms can overcome the limitations of traditional clouds and networks by enhancing their dynamic networking capabilities. Since these evolutions have motivated this thesis and our investigations, this chapter on the state of the art will provide an overview of NFV architecture, SDN, resource allocation challenges and reflect the convergence trend between cloud computing, software networks, and the virtualization of networking functions.

2.2 Network Function Virtualization (NFV)

2.2.1 Network Services before NVF

Communication Service Providers (CSPs) go beyond simply providing network connectivity for their enterprise customers. They also offer additional services and network functions like Network address translation (NAT), Firewall, Encryption, Domain Name Service (DNS), Caching, etc. Traditionally, these network functions were deployed using proprietary hardware at the customer premises. This approach provides additional revenue but deploying multiple proprietary devices is costly and makes upgrades difficult (i.e., every time a new network function is added to a service, a truck roll is required to install the dedicated new hardware device). Consequently, service providers began exploring ways to reduce cost and accelerate deployments through Network Function Virtualization (NFV).

2.2.2 What is NVF?

Network Function Virtualization (NFV) [1], [2], [3] is an innovative emerging way to design, deploy, and manage networking services by decoupling functions

(such as firewalls, DPIs, load balancers, etc.) from dedicated hardware and moving them to virtual servers. Several use cases of NFV are discussed in [4]. Note that manageability, reliability, stability, and security are considered in [4] as the key performance parameters in both physical and in software based virtualized networks.

Figure 1: Network Function Virtualization

- **Hardware Flexibility:** Because NFV uses regular Commercial-Off-The-Shelf (COTS) hardware, network operators have the freedom to choose and build the hardware in the most efficient way to suit their needs and requirements.
- **Faster Service Life Cycle:** New network services can now be deployed more quickly, in an on-demand and on-need basis, providing benefits for end users as well as the network providers.
- **Scalability and Elasticity:** New services and capacity-hungry applications keep network operators (especially cloud providers), on their toes to keep up with the fastincreasing demands of consumers.
- **Increased Revenue:** The combination of introducing new services faster and existing servers in a more dynamic fashion can jointly result in increased revenue.
- **Reduced Capital Expenditures (CAPEX):** The use of industry-standard services, increased hardware utilization and adoption of open source software results in reduced capital expenditures.
- **Reduced Operational Expenditures (OPEX):** Automation and hardware standardization can substantially slash operational expenditures.
- **Improved Customers' Satisfaction:** The combination of service agility and selfservice can result in greater customer satisfaction.

- **Reduced Power Consumption and Complexity:** Efficiencies in space, power, and cooling. Communications Service Providers (CSPs) may have finite physical space, electrical power, and cooling capacity in a data center, so they will carefully select equipment to efficiently consume those finite and/or costly resources. NFV provides a better energy efficiency resulting from the consolidation of resources, as well as their more dynamic utilization.

2.2.3 NFV Architecture

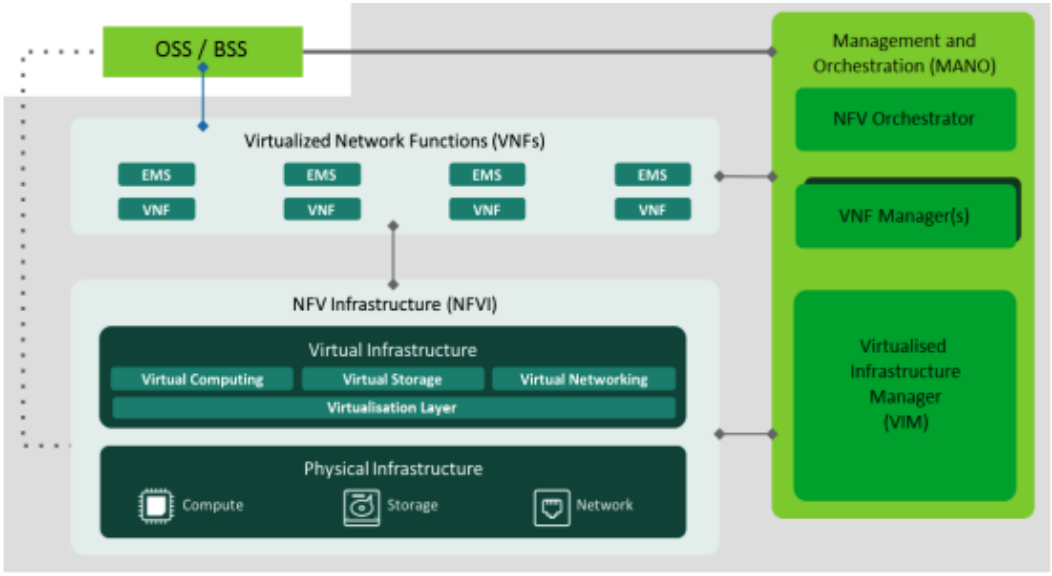


Figure 2: ETSI NFV reference architecture

The main components of the NFV architectural framework are:

1. NFV Infrastructure (NFVI): is a kind of cloud data center containing the totality of all hardware and software components that build up the NFV environment on which NFV services are deployed, managed and executed. NFVI includes:

- **Physical Hardware:** this includes computing hardware (such as servers, RAM), storage hardware (such as disk storage, Network Attached Storage (NAS)), and network hardware (such as switches and routers).
- **Virtualisation Layer:** abstracts the hardware resources and decouples the VNF software from the underlying hardware, thus ensuring a hardware inde-

pendent lifecycle for the VNFs. We can use multiple open source and proprietary options for the virtualisation layer (such as KVM, QEMU, VMware and Openstack).

- **Virtual Infrastructure:** this includes virtual compute (virtual machines or containers), virtual storage, and virtual networks (overlay networks).

2. **Virtualised Network Functions (VNFs):** run on top of the NFVI and represent virtualized instances of different network functions. Each VNF has a corresponding Element Management System (EMS) that provides management and control functionality for that VNF.

3. **NFV Management and Orchestration (MANO):** NFV MANO does not act in isolation. It interacts with the Operational and Business Support Systems (OSS/BSS) components of the operator to manage the operational and business aspects of the network. MANO includes:

- **Virtualized Infrastructure Manager (VIM):** or cloud management software, e.g. OpenStack or Kubernetes. It is responsible for controlling and managing the computing, storage, and network resources, as well as their virtualization.
- **VNF Manager(s):** it is responsible for VNF life cycle management, including VNF instantiation, update, query, scaling, and termination.
- **NFV Orchestrator:** it is in charge of the orchestration and management of NFV infrastructure and software resources, and realizing network services on NFVI. It utilizes resource allocation and placement algorithms to ensure optimal usage of both physical and software resources.

2.3 Integration of NFV with other technologies

Over the past years, the integration of NFV with other technologies, such as SDN, Cloud computing, and 5G [5] has attracted significant attention from both the academic research community and industry. NFV integration with SDN and Cloud computing is beneficial due to the complementary features and distinctive approaches followed by each technology toward providing solutions to today?s and future networks [6], [7]. For instance, NFV provides function abstraction (i.e., virtualization of network functions) supported by ETSI [8], SDN provides network

abstraction supported by Open Networking Foundation (ONF) [9], and Cloud computing provides computation abstraction (i.e., a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services)) supported by the Distributed Management Task Force (DMTF) [10]. Abstraction is one of the core features of cloud computing which allows abstraction of the physical implementation to hide the background (technical) details from users and developers. To summarize the relationships between NFV, SDN, and Cloud computing, we use Figure 3.

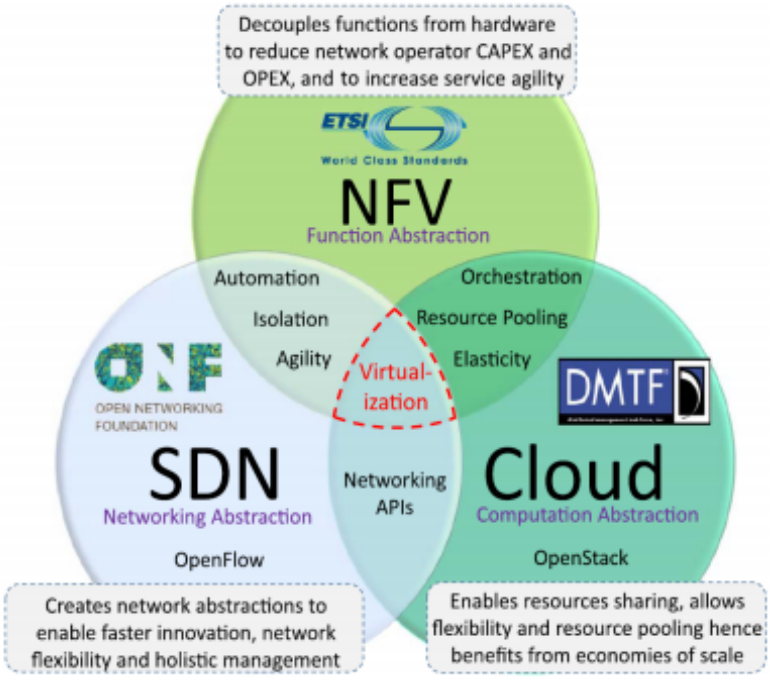


Figure 3: ETSI NFV reference architecture

SDN, NFV, and Cloud computing technologies are complementary to each other but are independent and can be deployed alone or together. A combination of these technologies together in a network architecture is more desirable [11]. In fact, the advantages that accrue from each of them are similar: agility, cost reduction, dynamism, automation, resource scaling, etc.

3 Virtual Service and flow coordination

3.1 Introduction

There is a rising demand for services consisting of multiple interconnected components, e.g., microservices in a service mesh or chained virtual network functions (VNFs) in network function virtualization (NFV) [12]. These services can scale flexibly by instantiating service components according to current demand. Such instances can run independently on any compute node in the network and process incoming flows requesting the service. The goal of service coordination is to ensure that these flows are processed successfully by traversing instances of all required service components. Additionally, flows should complete with short end-to-end delay to ensure good Quality of Service (QoS). To this end, the requested services need to be scaled and their instances placed in the network, i.e., we have to decide how many service components to instantiate where. Furthermore, incoming flows need to be routed from their ingress nodes through these deployed instances and finally to their egress nodes. In doing so, node and link capacities need to be respected and link delays should be considered.

To address the issue three algorithms proposed were evaluated, a simulation environment was deployed proposed at [12] Algorithm used: Random Schedule,Load Balance algorithm and Shortest Path algorithm (more details of these algorithms will be discussed in the next section)

Real graph network topologies were used as input with ”fake” traffic produced from the author, in order to simulate the mentioned algorithms.

3.2 Problem Formulation

We address the problem of coordinating services online over discrete time steps $t \in \mathbb{T}$. The problem can be formalized as follows:

A. Problem Inputs

We consider a substrate network $G = (V, L)$ of distributed nodes connected by non-directed links. Each node $u \in \mathbb{G}$ has a compute capacity $cap_u \in \mathbb{R}$ (e.g CPU). Each $link = (u, u') \in \mathbb{L}$ connects two nodes u and u' bidirectional with certaind delay $d_t \in \mathbb{R}$ and a maximum data rate $cap_t \in \mathbb{R}$ that is shared between both directions. We will also need to declare the Service Functions (SF) in order

to apply the algorithms with deterministic processing delays. Last we configure the NFV simulator, in order to have the fully solution deployed for testing - researching.

3.2.1 Algorithms proposed

In this section we will have a high level look on the algorithms that we are going to use.

Random Schedule:

- Places all VNFs on all nodes of the networks
- Creates random schedules for each source node, each SFC, each SF , each destination node
- All the schedules for an SF sum-up to 1

Load Balance algorithm:

- Always returns equal distribution for all nodes having capacities and SFs. Places all SFs on all nodes having some capacity.

Shortest path algorithm:

Based on network topology, SFC, and ingress nodes, calculates for each ingress node:

- Puts 1st VNF on ingress, 2nd VNF on closest neighbor, 3rd VNF again on closest neighbor of 2nd VNF and so on.
- Stores placement of VNFs and avoids placing 2 VNFs on the same node as long as possible. If all nodes are filled, continue placing a 2nd VNF on all nodes, but avoid placing 3 VNFs and so on.
- Avoids nodes without any capacity at all (but ignores current utilization).

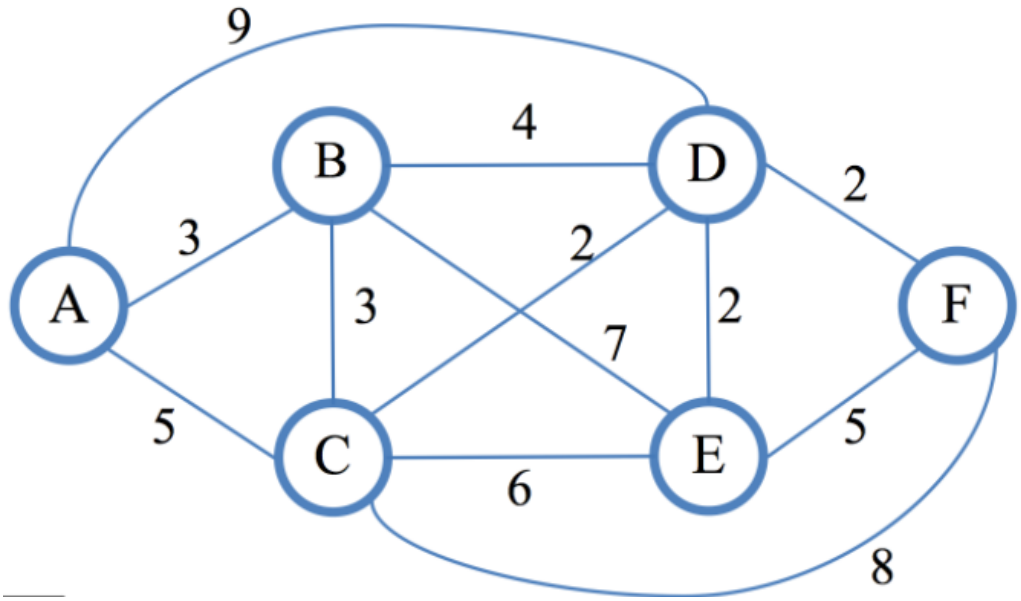


Figure 4: Shortest path Algorithm

3.2.2 Evaluation setup

We evaluate our three proposed algorithms(RS, LB, SP), using extensive simulations on the real-world Abilene network topology [13] with 11 nodes and 14 links. We consider the Abilene network to be a representative example of a small scale networks in practice. The simulator is based on SimPy and is tested with Python 3.8. We set randomly pick heterogeneous node capacities $capv \in (0, 7, 14)$.

Furthermore, we consider the SFCs (a, b, c) all with mean processing delay $processorDelayMean = 5$ and standard deviation $stdev = 0$

Furthermore, they incur a per-flow processing delay that is normally distributed with N (5 ms, 0 ms), where values are cut off at 0 ms to prevent negative delays. Flows arriving at the network's ingress nodes request one of the three services chosen uniformly at random. For each ingress, flow inter-arrival times mean is set to 12 , flow duration $df_mean = 1$, and flow data rate standard deviation is set to 0. The duration per experiment is $|T| = 100$ time steps.

The file structure of the simulator is as follows[14]: - docs (Folder): Contains the documentation files of the project. - params (Folder): Contains sample parameter files (network file and VNF file) for testing purposes. - src (Folder): Contains the source code for the simulator and the interface. - coordsim (Folder): contains the

source code of the simulator - metrics (Folder): contains the metrics module. - network (Folder): contains the network module. - reader (Folder): contains the params file reader module. - simulation (Folder): main simulator module. - main (Python): main executable for running the simulator from CLI - siminterface (Folder): contains the interface source code - tests (Folder): contains the unit tests for the simulator.

The simulator works as follows: The user (coordination algorithm or cli) provide two main inputs for the simulator: - Network file: GraphML file using the Zoo format. This file contains the network nodes and the edges. - VNF file: YAML file that includes the list of SFCs and the list of SFs under each SFC in an ordered manner. The file can also include a specified placement that can be used as a default placement. The SFs must include a processing delay mean and standard deviation values so that processing delays can be calculated for each flow passing through that SF.

Once the parameters are provided, the flow of data through the simulator is as follows:

- The input network and VNF files are parsed producing a NetworkX object containing the list of nodes and edges, and the shortest paths of the network (using Floyd-Warshall). The parsing also produces dictionaries that contain the list of SFCs and the list of SFs and their respective values. Additionally, the list of ingress nodes (nodes at which flows arrive) are also calculated from the GraphML file. These parameters are then passed to a SimulatorParams object, which holds all the parameters of the simulator, the simulator is then started using the FlowSimulator object's start() $function$.
- At each ingress node, the $function$ generateFlow() is called as a SimPy process, this $function$ creates Flow objects with exponentially distributed random inter arrival times. The flow's data rate and size are generated using normally distributed random variables. All of the inter arrival time, data rate, and flow size parameters are user configurable. The flow is also assigned a random SFC chosen from the list of available SFC given in the VNF file.
- Once the flow is generated, initFlow() is called as a SimPy process which initializes the handling of the flow within the simulator. The $function$ then calls passFlow(), which then handles the scheduling of the flow according to the defined load balancing rules (flow schedule). Once the next node has been determined, the forwarding of the flow is simulated by halting the flow for the path delay duration using the forwardFlow() $function$. Once that is done, the processing of the flow is simulated by calling processFlow() as a SimPy process. If the requested SF was not found at the next node, the flow is then dropped.

- In processFlow(), the processing delay for that particular SF is generated using given mean and standard deviation values using a normal distribution. The simulator checks the node's remaining processing capacity to check if the node can handle the data rate requested by the SF, if there is not enough capacity, then the flow is dropped. For the duration that the flow is being processed by the SF, the flow's data rate is deducted from the node's capacity, and returned after the flow finished processing completely.
- Once the flow was processed completely at each SF, departFlow() is called to register the flow's departure from the network. If the flow still has other SFs to be processed at in the network, processFlow() calls passFlow() again in a mutually recursive manner. This allows the flow to stay in the SF for processing, while the parts of the flow that were processed already to be sent to the next SF.

Input Parameters: The available input parameters that are configurable by the user are: - d: The duration of the simulation (simulates milliseconds). - s: The seed to use for the random number generator. - n: The GraphML network file that specifies the nodes and edges of the network. - sf: VNF file which contains the SFCs and their respective SFs and their properties. - iam: Inter arrival mean of the flows' arrival at ingress nodes. - fdm: The mean value for the generation of data rate values for each flow. - fds: The standard deviation value for the generation of data rate values for each flow. - fss: The shape of the Pareto distribution for the generation of the flow size values.

Figure 5: Abilene topology graph

3.3 Simulation

The simulations are based on a realistic network topology were run in a 1.60 GHz Quad Core machine with 16 GBytes of available RAM.

In order to run the test Ubuntu Operating system was install with WSL2 (Ubuntu running on Windows Kernel).

In our case we had the following as input:

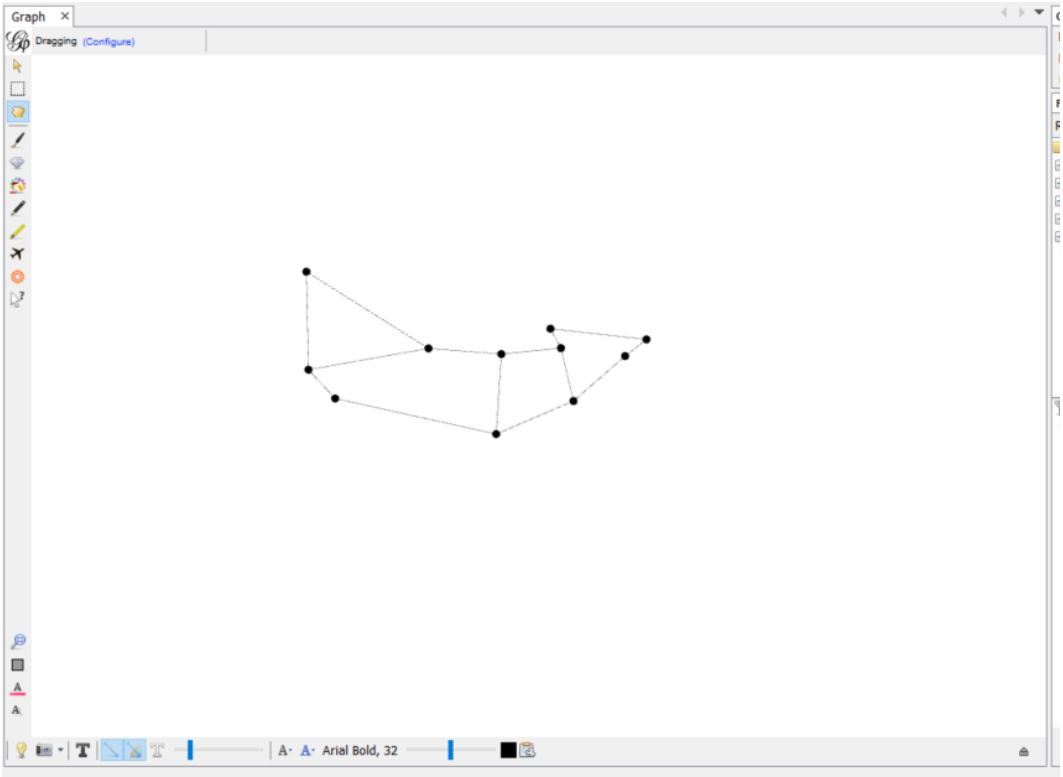


Figure 6: Abilene graphml as viewed in Gephi

Graph | Data Table

Nodes | Edges | Configuration | Add node | Add edge | Search/Replace | Import Spreadsheet | Export table | More actions | Filter: | Id

Id	Label	Interval	NodeCap	NodeType	Longitude	id	Country	Latitude	Internal
0	New York		0	Ingress	-74.00597	0	United States	40.71427	1
1	Chicago		7	Ingress	-87.65005	1	United States	41.85003	1
2	Washington DC		0	Ingress	-77.03637	2	United States	38.89511	1
3	Seattle		14	Ingress	-122.33207	3	United States	47.60621	1
4	Sunnyvale		7	Ingress	-122.03635	4	United States	37.36883	1
5	Los Angeles		14	Normal	-118.24368	5	United States	34.05223	1
6	Denver		0	Normal	-104.9847	6	United States	39.73915	1
7	Kansas City		0	Normal	-94.62746	7	United States	39.11417	1
8	Houston		14	Normal	-95.36327	8	United States	29.76328	1
9	Atlanta		7	Normal	-84.38798	9	United States	33.749	1
10	Indianapolis		7	Normal	-86.15804	10	United States	39.76838	1

Figure 7: Abilene network topology data(1)

Graph | Data Table

Nodes | Edges | Configuration | Add node | Add edge | Search/Replace | Import Spreadsheet | Export table | More actions | Filter: | Source

Source	Target	Type	Id	Label	Interval	Weight	LinkFwdCap	key	LinkNote	LinkLabel	LinkType
0	1	Undirected	0			1.0	1000	0	c	OC-192c	OC-192
0	2	Undirected	1			1.0	1000	0	c	OC-192c	OC-192
1	10	Undirected	2			1.0	1000	0	c	OC-192c	OC-192
2	9	Undirected	3			1.0	1000	0	c	OC-192c	OC-192
3	4	Undirected	4			1.0	1000	0	c	OC-192c	OC-192
3	6	Undirected	5			1.0	1000	0	c	OC-192c	OC-192
4	5	Undirected	6			1.0	1000	0	c	OC-192c	OC-192
4	6	Undirected	7			1.0	1000	0	c	OC-192c	OC-192
5	8	Undirected	8			1.0	1000	0	c	OC-192c	OC-192
6	7	Undirected	9			1.0	1000	0	c	OC-192c	OC-192
7	8	Undirected	10			1.0	1000	0	c	OC-192c	OC-192
7	10	Undirected	11			1.0	1000	0	c	OC-192c	OC-192
8	9	Undirected	12			1.0	1000	0	c	OC-192c	OC-192
9	10	Undirected	13			1.0	1000	0	c	OC-192c	OC-192

Figure 8: Abilene network topology data(2)

```yaml
# module for configuring the simulator
# configuration parameters are loaded and used both when using the simulator via the CLI and via the interface
# all parameters are required, defaults are in comments

inter_arrival_mean: 12.0           # default: 10.0
deterministic_arrival: False
flow_dr_mean: 1.0                  # default: 1.0
flow_dr_stdev: 0.0                 # default: 0.0
flow_size_shape: 0.001               # default: 0.001
deterministic_size: True
run_duration: 100                   # default: 100
ttl_choices: [100]

# Optional: Trace file trace relative to the CWD.
# Until values start in the trace file, the defaults from this file are used
# trace_path: res/traces/scenario09.csv

# States (two state markov arrival)
# Optional param: states: True | False
use_states: True
init_state: state_1

states:
  state_1:
    inter_arr_mean: 12.0
    switch_p: 0.05
  state_2:
    inter_arr_mean: 8.0
    switch_p: 0.05

```

Figure 9: Simulation configuration (yaml file)

```yaml
# simple chain of 3 SFs a->b->c with deterministic processing delays

# list of SFCs and involved SFs (order of SFs matters). names need to match dummy schedule and placement (dummy_data.py)
sfc_list:
  sfc_1:
    - a
    - b
    - c

# SF attributes (for now, processing delay)
sf_list:
  a:
    processing_delay_mean: 5.0
    processing_delay_stdev: 0.0
  b:
    processing_delay_mean: 5.0
    processing_delay_stdev: 0.0
  c:
    processing_delay_mean: 5.0
    processing_delay_stdev: 0.0

```

Figure 10: SFCs simulation configuration(abc.yaml)

3.3.1 Running the simulation

The simulator application is called rs for random schedule, lb for load balancer and sp for shortest path. To run the simulator, the following call may be executed:

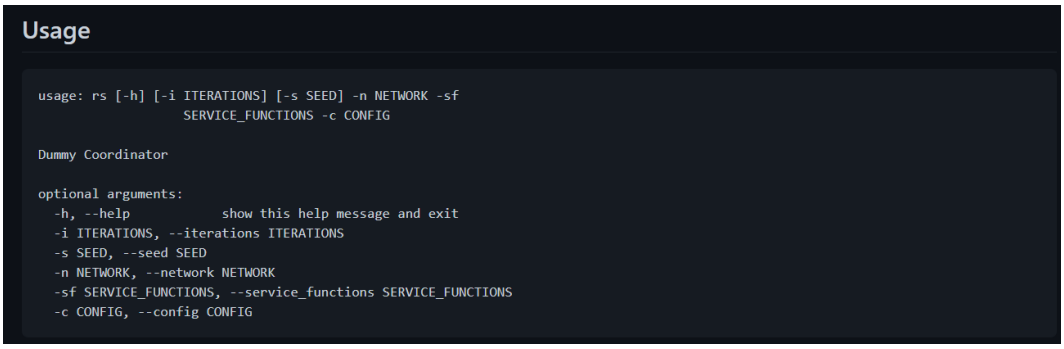


Figure 11: commands usage

In our case we ran the following:

```
8  rs -n "res/networks/abilene/abilene_5in_7x-cap.graphml" -sf "res/service_functions/abc.yaml" -c
   "res/config/rand-mmp-arrival12-8_det-size001_dur100.yaml" -i 1000
9
10 lb -n "res/networks/abilene/abilene_5in_7x-cap.graphml" -sf "res/service_functions/abc.yaml" -c
   "res/config/rand-mmp-arrival12-8_det-size001_dur100.yaml" -i 1000
11
12 sp -n "res/networks/abilene/abilene_5in_7x-cap.graphml" -sf "res/service_functions/abc.yaml" -c
   "res/config/rand-mmp-arrival12-8_det-size001_dur100.yaml" -i 1000
```

Figure 12: commands used in CLI

And this is the output of the command running. (LOG level set to INFO).

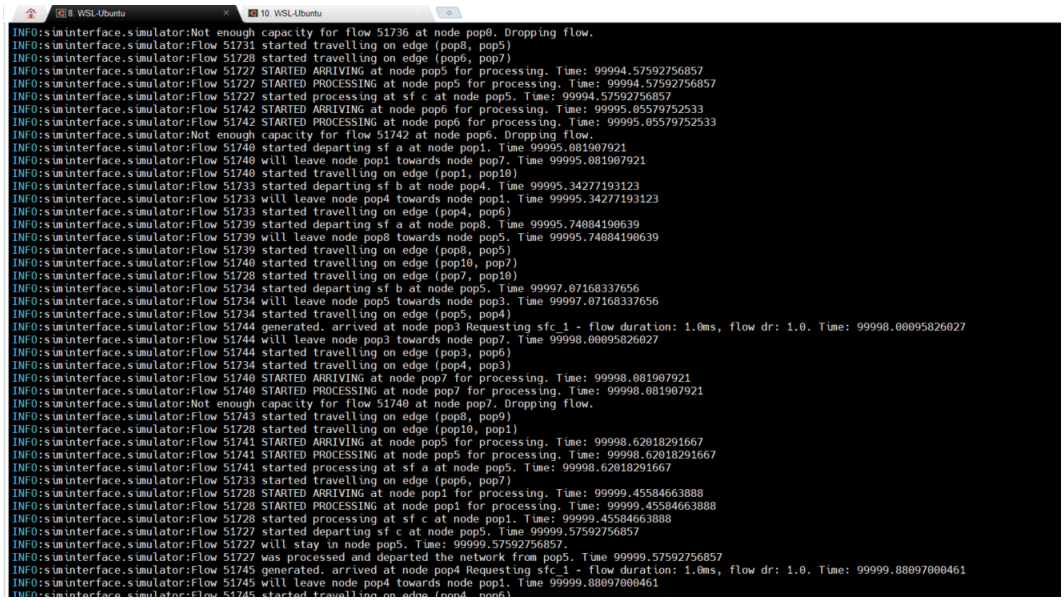


Figure 13: commands running

3.4 Results

The results are saved results in /usr/local/lib/python3.8/dist-packages/results/ in CSV format in order for the user to visualize them as he wants, using the export.

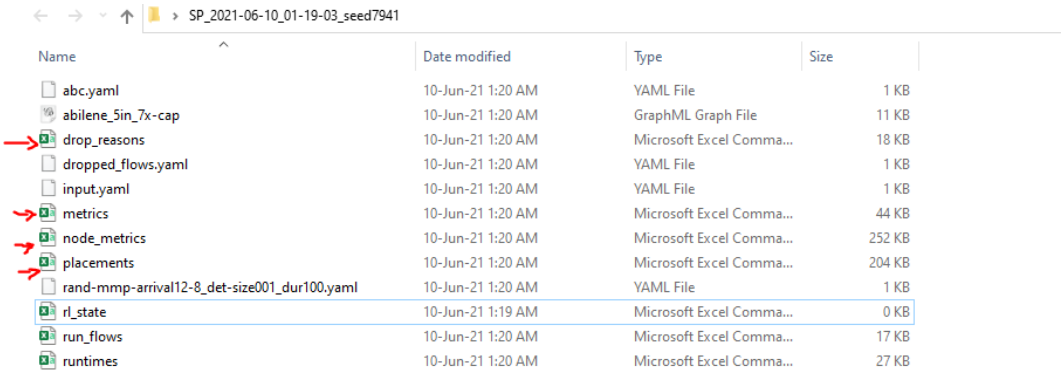


Figure 14: Results files

3.4.1 Random Schedule Results

Results After running the Random schedule algorithm:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	episode	time	total_flows	successful_flows	dropped_flows	in_network_flows	avg_end2end_delay							
2	1	0	5	0	0	5	0							
3	1	100	50	10	27	13	29.4							
4	1	200	87	20	62	5	30.85							
5	1	300	135	29	99	7	31.06896552							
6	1	400	193	40	143	10	32.5							
7	1	500	237	51	176	10	32.54901961							
8	1	600	276	63	204	9	31.47619048							
9	1	700	321	74	238	9	31.05405405							
10	1	800	370	85	279	6	31.22352941							
11	1	900	415	96	309	10	31.10416667							
12	1	1000	457	105	346	6	31.37142857							
13	1	1100	499	118	378	3	31.08474576							
14	1	1200	541	131	404	6	31.10687023							
15	1	1300	583	146	431	6	31.28767123							
16	1	1400	623	156	455	12	31.15384615							
17	1	1500	660	166	488	6	31.36746988							
18	1	1600	694	171	515	8	31.35672515							
19	1	1700	742	184	547	11	31.26630435							
20	1	1800	782	196	581	5	31.43367347							
21	1	1900	812	205	602	5	31.52195122							
22	1	2000	856	214	634	8	31.57476636							
23	1	2100	901	226	664	11	31.63716814							
24	1	2200	951	242	702	7	31.85123967							
25	1	2300	981	251	724	6	32.00796813							
26	1	2400	1039	261	764	14	32.06130268							
27	1	2500	1080	277	799	4	32.15884477							
28	1	2600	1121	284	831	6	32.14788732							
29	1	2700	1163	294	864	5	32.09863946							
30	1	2800	1220	309	902	9	32.21359223							
31	1	2900	1269	322	936	11	32.13043478							
32	1	3000	1316	338	973	5	32.29881657							
33	1	3100	1366	347	1014	5	32.26224784							
34	1	3200	1411	356	1048	7	32.31741573							
35	1	3300	1454	364	1083	7	32.36813187							
36	1	3400	1512	377	1123	12	32.41909814							
37	1	3500	1559	391	1157	11	32.48337596							
38	1	3600	1602	409	1189	4	32.44498778							
39	1	3700	1648	417	1220	11	32.441247							
40	1	3800	1694	432	1255	7	32.49768519							

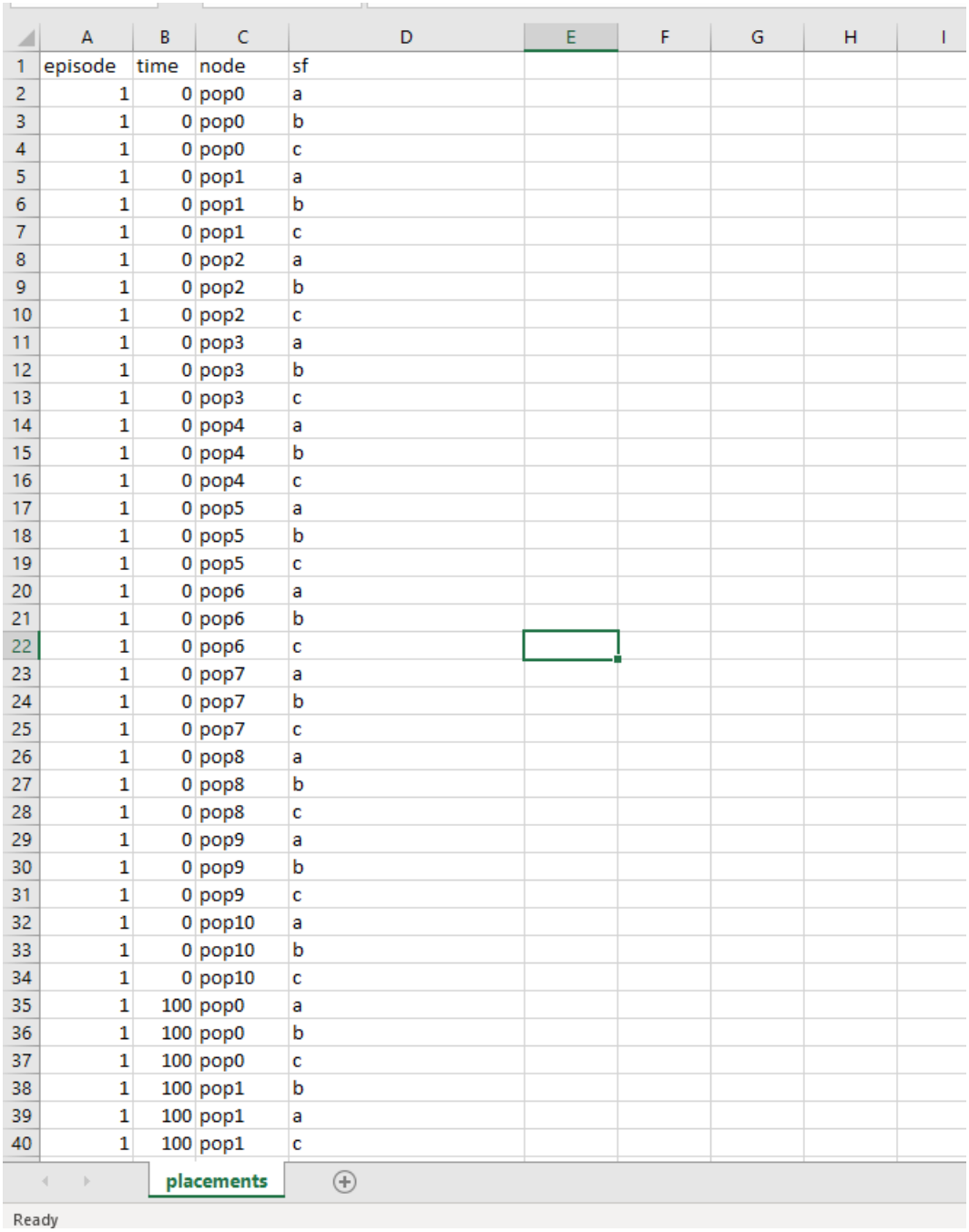
metrics

Figure 15: Metrics after running rs script

episode	time	node	node_capacity	used_resources	ingress_traffic
1	0	pop0	0	0	0
1	0	pop1	7	0	0
1	0	pop2	0	0	0
1	0	pop3	14	0	0
1	0	pop4	7	0	0
1	0	pop5	14	0	0
1	0	pop6	0	0	0
1	0	pop7	0	0	0
1	0	pop8	14	0	0
1	0	pop9	7	0	0
1	0	pop10	7	0	0
1	100	pop0	0	0	5
1	100	pop1	7	3	10
1	100	pop2	0	0	10
1	100	pop3	14	3	11
1	100	pop4	7	1	14
1	100	pop5	14	4	0
1	100	pop6	0	0	0
1	100	pop7	0	0	0
1	100	pop8	14	4	0
1	100	pop9	7	2	0
1	100	pop10	7	2	0
1	200	pop0	0	0	7
1	200	pop1	7	2	5
1	200	pop2	0	0	9
1	200	pop3	14	2	9
1	200	pop4	7	3	7
1	200	pop5	14	3	0
1	200	pop6	0	0	0
1	200	pop7	0	0	0
1	200	pop8	14	3	0
1	200	pop9	7	2	0
1	200	pop10	7	2	0
1	300	pop0	0	0	13
1	300	pop1	7	2	6
1	300	pop2	0	0	7
1	300	pop3	14	3	9
1	300	pop4	7	3	13
1	300	pop5	14	3	0

node_metrics

Figure 16: Node Metrics after running rs script



	A	B	C	D	E	F	G	H	I
1	episode	time	node	sf					
2	1	0	pop0	a					
3	1	0	pop0	b					
4	1	0	pop0	c					
5	1	0	pop1	a					
6	1	0	pop1	b					
7	1	0	pop1	c					
8	1	0	pop2	a					
9	1	0	pop2	b					
10	1	0	pop2	c					
11	1	0	pop3	a					
12	1	0	pop3	b					
13	1	0	pop3	c					
14	1	0	pop4	a					
15	1	0	pop4	b					
16	1	0	pop4	c					
17	1	0	pop5	a					
18	1	0	pop5	b					
19	1	0	pop5	c					
20	1	0	pop6	a					
21	1	0	pop6	b					
22	1	0	pop6	c					
23	1	0	pop7	a					
24	1	0	pop7	b					
25	1	0	pop7	c					
26	1	0	pop8	a					
27	1	0	pop8	b					
28	1	0	pop8	c					
29	1	0	pop9	a					
30	1	0	pop9	b					
31	1	0	pop9	c					
32	1	0	pop10	a					
33	1	0	pop10	b					
34	1	0	pop10	c					
35	1	100	pop0	a					
36	1	100	pop0	b					
37	1	100	pop0	c					
38	1	100	pop1	b					
39	1	100	pop1	a					
40	1	100	pop1	c					

Figure 17: SF placements after running rs script

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	episode	time	successful	dropped_flo	total_flows									
2	1	0	0	0	5									
3	1	100	10	27	45									
4	1	200	10	35	37									
5	1	300	9	37	48									
6	1	400	11	44	58									
7	1	500	11	33	44									
8	1	600	12	28	39									
9	1	700	11	34	45									
10	1	800	11	41	49									
11	1	900	11	30	45									
12	1	1000	9	37	42									
13	1	1100	13	32	42									
14	1	1200	13	26	42									
15	1	1300	15	27	42									
16	1	1400	10	24	40									
17	1	1500	10	33	37									
18	1	1600	5	27	34									
19	1	1700	13	32	48									
20	1	1800	12	34	40									
21	1	1900	9	21	30									
22	1	2000	9	32	44									
23	1	2100	12	30	45									
24	1	2200	16	38	50									
25	1	2300	9	22	30									
26	1	2400	10	40	58									
27	1	2500	16	35	41									
28	1	2600	7	32	41									
29	1	2700	10	33	42									
30	1	2800	15	38	57									
31	1	2900	13	34	49									
32	1	3000	16	37	47									
33	1	3100	9	41	50									
34	1	3200	9	34	45									
35	1	3300	8	35	43									
36	1	3400	13	40	58									
37	1	3500	14	34	47									
38	1	3600	18	32	43									
39	1	3700	8	31	46									
40	1	3800	15	35	46									

run_flows

Figure 18: Run Flows after running rs script

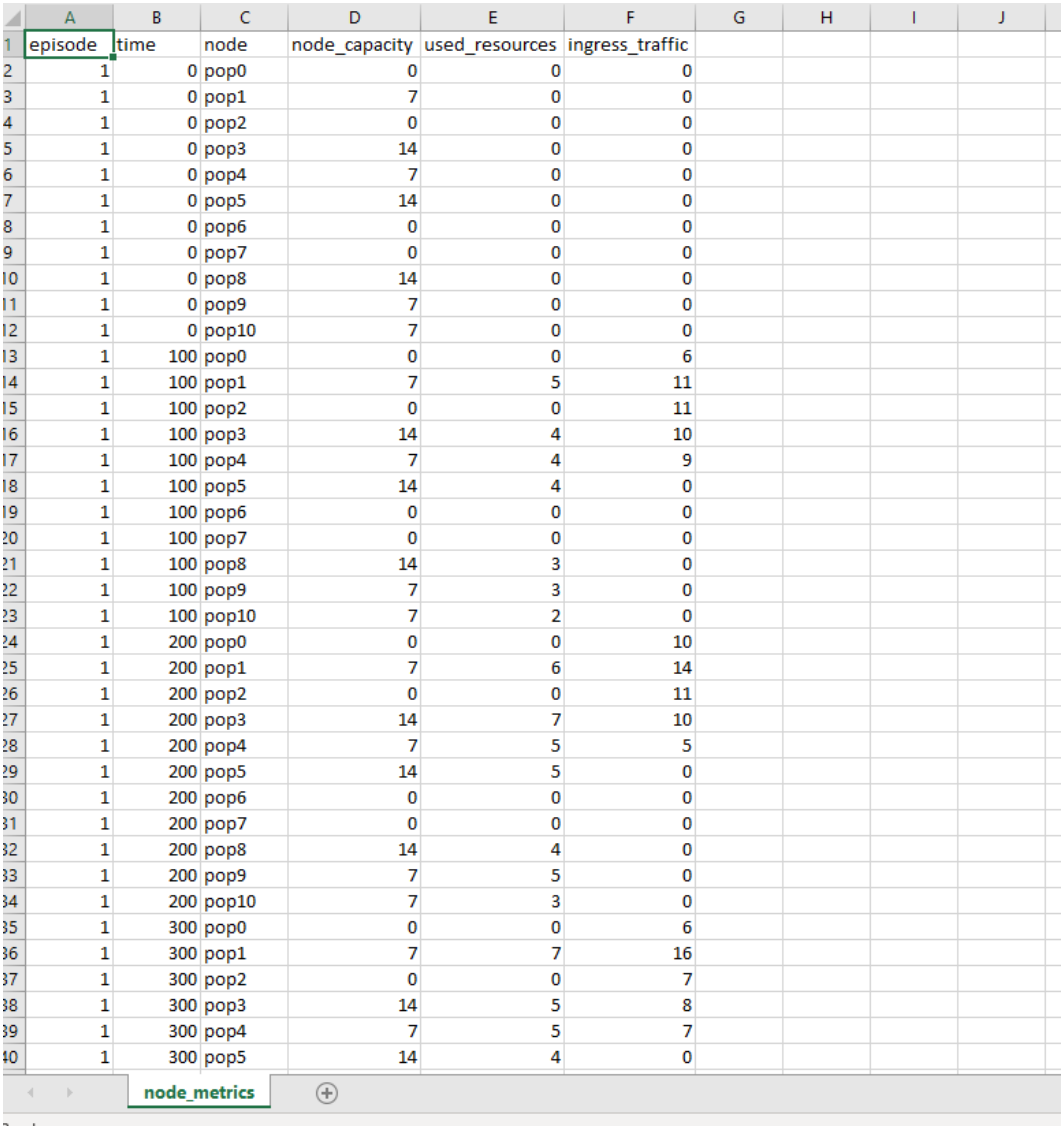
3.4.2 Load Balance Results

Results After running the Load Balance algorithm:

	A	B	C	D	E	F	G	H
1	episode	time	total_flows	successful_flows	dropped_flows	in_network_flows	avg_end2end_delay	
2	1	0	5	0	0	5	0	
3	1	100	47	33	0	14	32.75757576	
4	1	200	97	80	0	17	33.4125	
5	1	300	141	121	2	18	33.62809917	
6	1	400	201	176	2	23	33.36931818	
7	1	500	251	224	6	21	33.50892857	
8	1	600	302	276	9	17	33.60144928	
9	1	700	348	322	10	16	33.59627329	
10	1	800	392	365	11	16	33.55616438	
11	1	900	456	426	11	19	33.53755869	
12	1	1000	509	478	12	19	33.4665272	
13	1	1100	574	544	12	18	33.47242647	
14	1	1200	635	606	13	16	33.40924092	
15	1	1300	695	657	14	24	33.40943683	
16	1	1400	779	739	16	24	33.32611637	
17	1	1500	838	792	20	26	33.33459596	
18	1	1600	908	864	21	23	33.35416667	
19	1	1700	971	923	24	24	33.36294691	
20	1	1800	1032	992	25	15	33.29637097	
21	1	1900	1092	1047	26	19	33.2913085	
22	1	2000	1132	1085	26	21	33.26635945	
23	1	2100	1194	1149	27	18	33.29765013	
24	1	2200	1258	1213	27	18	33.30173124	
25	1	2300	1300	1259	28	13	33.28832407	
26	1	2400	1344	1300	30	14	33.28461538	
27	1	2500	1387	1344	30	13	33.33035714	
28	1	2600	1436	1388	30	18	33.3278098	
29	1	2700	1485	1438	33	14	33.36856745	
30	1	2800	1542	1488	35	19	33.36290323	
31	1	2900	1594	1541	35	18	33.4146658	
32	1	3000	1653	1595	36	22	33.42131661	
33	1	3100	1715	1650	37	28	33.43515152	
34	1	3200	1774	1720	40	14	33.42674419	
35	1	3300	1839	1783	40	16	33.3931576	
36	1	3400	1892	1834	40	18	33.39803708	
37	1	3500	1945	1892	40	13	33.42230444	
38	1	3600	2002	1935	40	27	33.41033592	
39	1	3700	2046	1994	44	8	33.44734203	
40	1	3800	2092	2028	44	20	33.43786982	

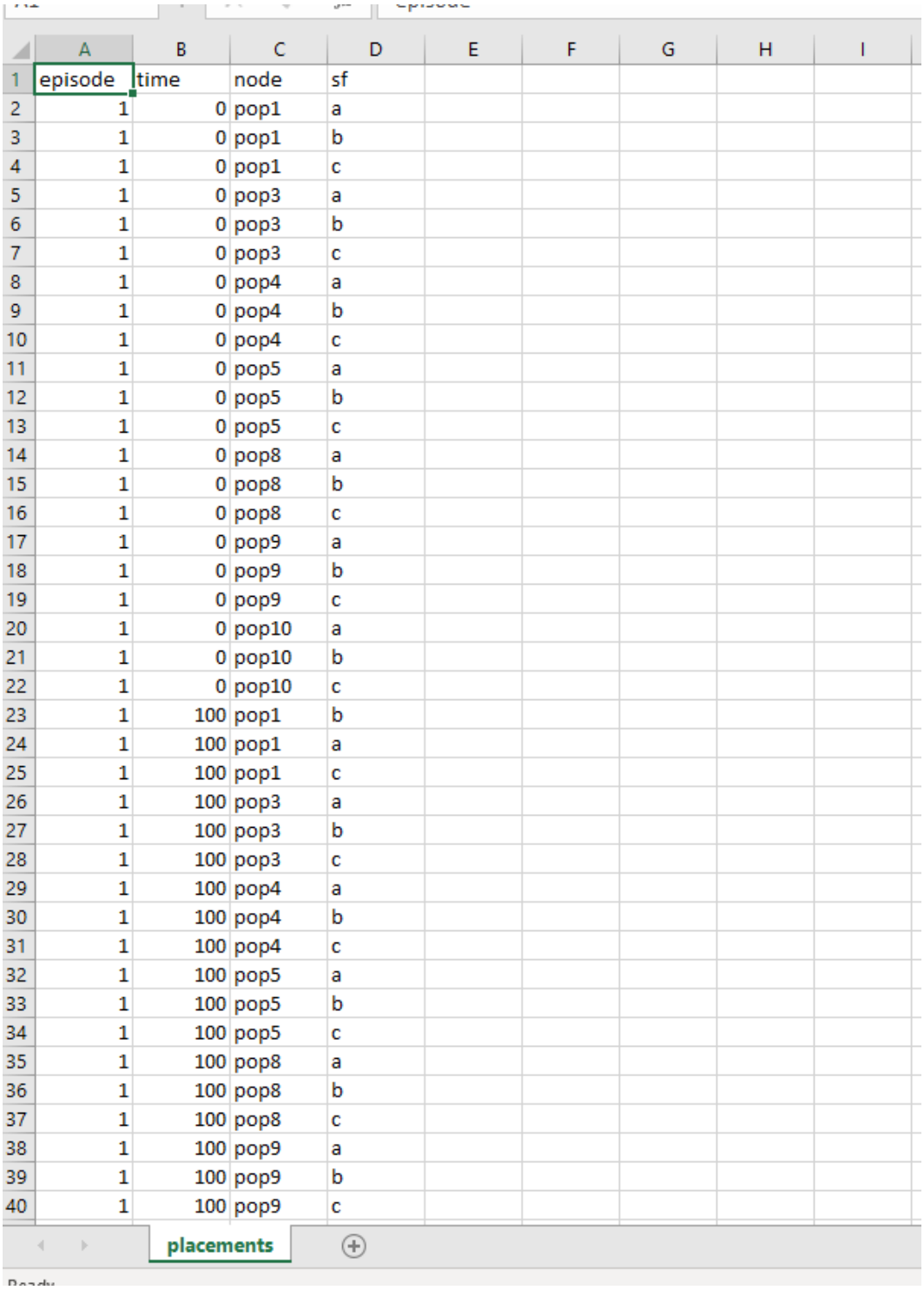
metrics

Figure 19: Metrics after running lb script



episode	time	node	node_capacity	used_resources	ingress_traffic
1	0	pop0	0	0	0
1	0	pop1	7	0	0
1	0	pop2	0	0	0
1	0	pop3	14	0	0
1	0	pop4	7	0	0
1	0	pop5	14	0	0
1	0	pop6	0	0	0
1	0	pop7	0	0	0
1	0	pop8	14	0	0
1	0	pop9	7	0	0
1	0	pop10	7	0	0
1	100	pop0	0	0	6
1	100	pop1	7	5	11
1	100	pop2	0	0	11
1	100	pop3	14	4	10
1	100	pop4	7	4	9
1	100	pop5	14	4	0
1	100	pop6	0	0	0
1	100	pop7	0	0	0
1	100	pop8	14	3	0
1	100	pop9	7	3	0
1	100	pop10	7	2	0
1	200	pop0	0	0	10
1	200	pop1	7	6	14
1	200	pop2	0	0	11
1	200	pop3	14	7	10
1	200	pop4	7	5	5
1	200	pop5	14	5	0
1	200	pop6	0	0	0
1	200	pop7	0	0	0
1	200	pop8	14	4	0
1	200	pop9	7	5	0
1	200	pop10	7	3	0
1	300	pop0	0	0	6
1	300	pop1	7	7	16
1	300	pop2	0	0	7
1	300	pop3	14	5	8
1	300	pop4	7	5	7
1	300	pop5	14	4	0

Figure 20: Node Metrics after running lb script



	A	B	C	D
1	episode	time	node	sf
2	1	0	pop1	a
3	1	0	pop1	b
4	1	0	pop1	c
5	1	0	pop3	a
6	1	0	pop3	b
7	1	0	pop3	c
8	1	0	pop4	a
9	1	0	pop4	b
10	1	0	pop4	c
11	1	0	pop5	a
12	1	0	pop5	b
13	1	0	pop5	c
14	1	0	pop8	a
15	1	0	pop8	b
16	1	0	pop8	c
17	1	0	pop9	a
18	1	0	pop9	b
19	1	0	pop9	c
20	1	0	pop10	a
21	1	0	pop10	b
22	1	0	pop10	c
23	1	100	pop1	b
24	1	100	pop1	a
25	1	100	pop1	c
26	1	100	pop3	a
27	1	100	pop3	b
28	1	100	pop3	c
29	1	100	pop4	a
30	1	100	pop4	b
31	1	100	pop4	c
32	1	100	pop5	a
33	1	100	pop5	b
34	1	100	pop5	c
35	1	100	pop8	a
36	1	100	pop8	b
37	1	100	pop8	c
38	1	100	pop9	a
39	1	100	pop9	b
40	1	100	pop9	c

placements

Figure 21: SF placements after running lb script

	A	B	C	D	E	F	G	H
1	episode	time	successful_flows	dropped_flows	total_flows			
2	1	0	0	0	5			
3	1	100	33	0	42			
4	1	200	47	0	50			
5	1	300	41	2	44			
6	1	400	55	0	60			
7	1	500	48	4	50			
8	1	600	52	3	51			
9	1	700	46	1	46			
10	1	800	43	1	44			
11	1	900	61	0	64			
12	1	1000	52	1	53			
13	1	1100	66	0	65			
14	1	1200	62	1	61			
15	1	1300	51	1	60			
16	1	1400	82	2	84			
17	1	1500	53	4	59			
18	1	1600	72	1	70			
19	1	1700	59	3	63			
20	1	1800	69	1	61			
21	1	1900	55	1	60			
22	1	2000	38	0	40			
23	1	2100	64	1	62			
24	1	2200	64	0	64			
25	1	2300	46	1	42			
26	1	2400	41	2	44			
27	1	2500	44	0	43			
28	1	2600	44	0	49			
29	1	2700	50	3	49			
30	1	2800	50	2	57			
31	1	2900	53	0	52			
32	1	3000	54	1	59			
33	1	3100	55	1	62			
34	1	3200	70	3	59			
35	1	3300	63	0	65			
36	1	3400	51	0	53			
37	1	3500	58	0	53			
38	1	3600	43	0	57			
39	1	3700	59	4	44			
40	1	3800	34	0	46			

run_flows

Figure 22: Run Flows after running lb script

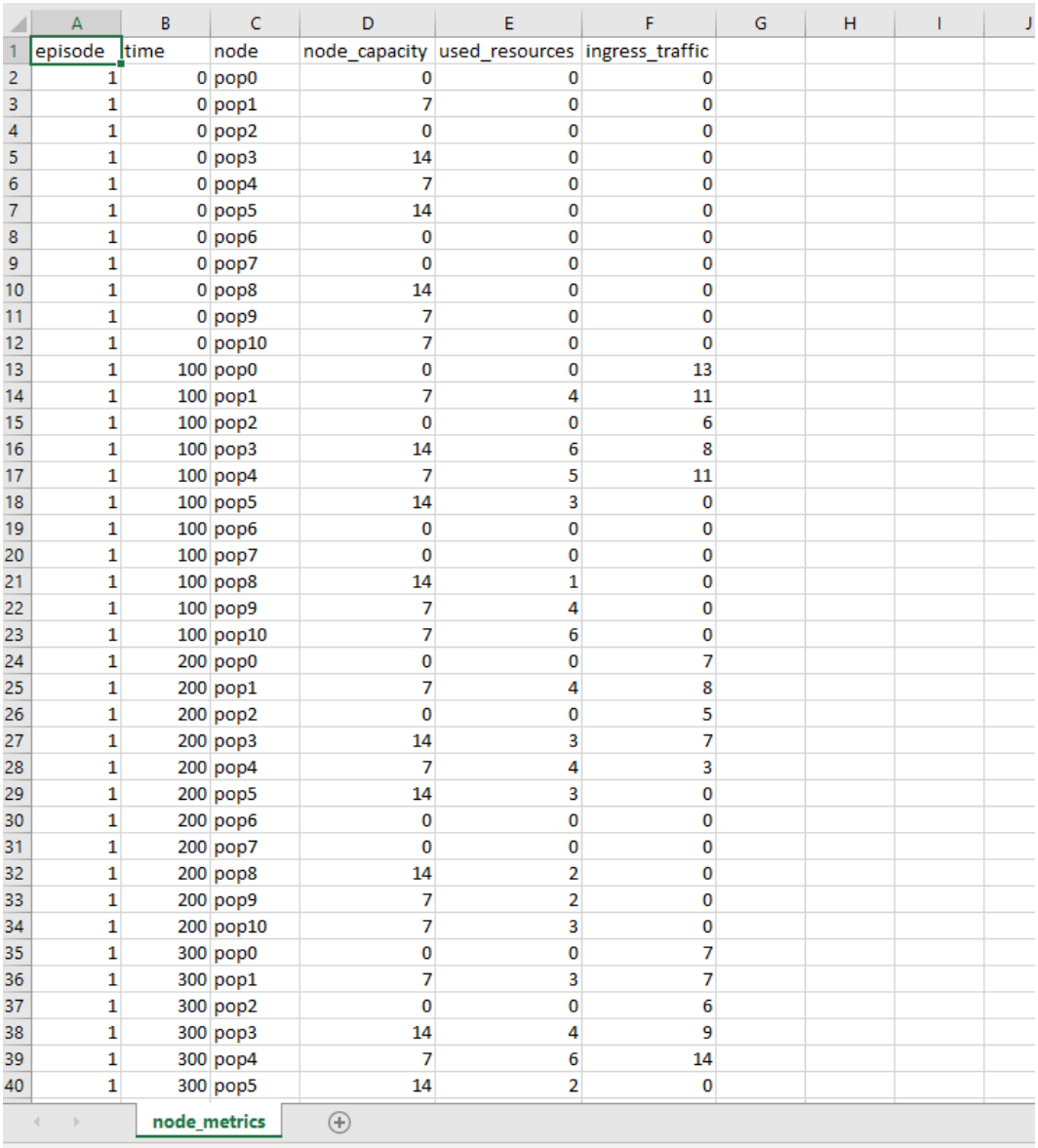
3.4.3 Shortest Path Results

Results After running the Shortest Path algorithm:

	A	B	C	D	E	F	G	H	I
1	episode	time	total_flows	successful_flows	dropped_flows	in_network_flows	avg_end2end_delay		
2	1	0	5	0	0	5	0		
3	1	100	49	37	0	12	26.2972973		
4	1	200	79	71	0	8	26.45070423		
5	1	300	122	112	0	10	26.33035714		
6	1	400	174	164	0	10	26.83536585		
7	1	500	217	200	0	17	26.74		
8	1	600	257	239	0	18	26.9832636		
9	1	700	305	290	0	15	26.99310345		
10	1	800	333	323	0	10	27.16718266		
11	1	900	384	372	0	12	27.31182796		
12	1	1000	413	406	0	7	27.18226601		
13	1	1100	456	442	0	14	27.30316742		
14	1	1200	503	493	0	10	27.22718053		
15	1	1300	540	530	0	10	27.2509434		
16	1	1400	589	577	0	12	27.34142114		
17	1	1500	616	605	0	11	27.3553719		
18	1	1600	664	655	2	7	27.3480916		
19	1	1700	714	697	2	15	27.32855093		
20	1	1800	756	741	2	13	27.25101215		
21	1	1900	797	787	2	8	27.25794155		
22	1	2000	839	827	2	10	27.22370012		
23	1	2100	892	878	2	12	27.15831435		
24	1	2200	935	920	2	13	27.17065217		
25	1	2300	975	964	2	9	27.19294606		
26	1	2400	1011	994	2	15	27.15191147		
27	1	2500	1060	1041	2	17	27.14505283		
28	1	2600	1105	1095	2	8	27.09954338		
29	1	2700	1149	1136	2	11	27.08010563		
30	1	2800	1200	1190	2	8	27.04453782		
31	1	2900	1230	1222	2	6	27.03355155		
32	1	3000	1286	1272	2	12	27.03144654		
33	1	3100	1335	1318	2	15	27.04172989		
34	1	3200	1389	1372	2	15	27.02623907		
35	1	3300	1447	1429	2	16	27.06438069		
36	1	3400	1497	1480	2	15	27.10540541		
37	1	3500	1553	1535	2	16	27.11596091		
38	1	3600	1621	1603	2	16	27.08671241		
39	1	3700	1671	1651	2	18	27.09206541		
40	1	3800	1732	1708	2	22	27.10772834		

metrics

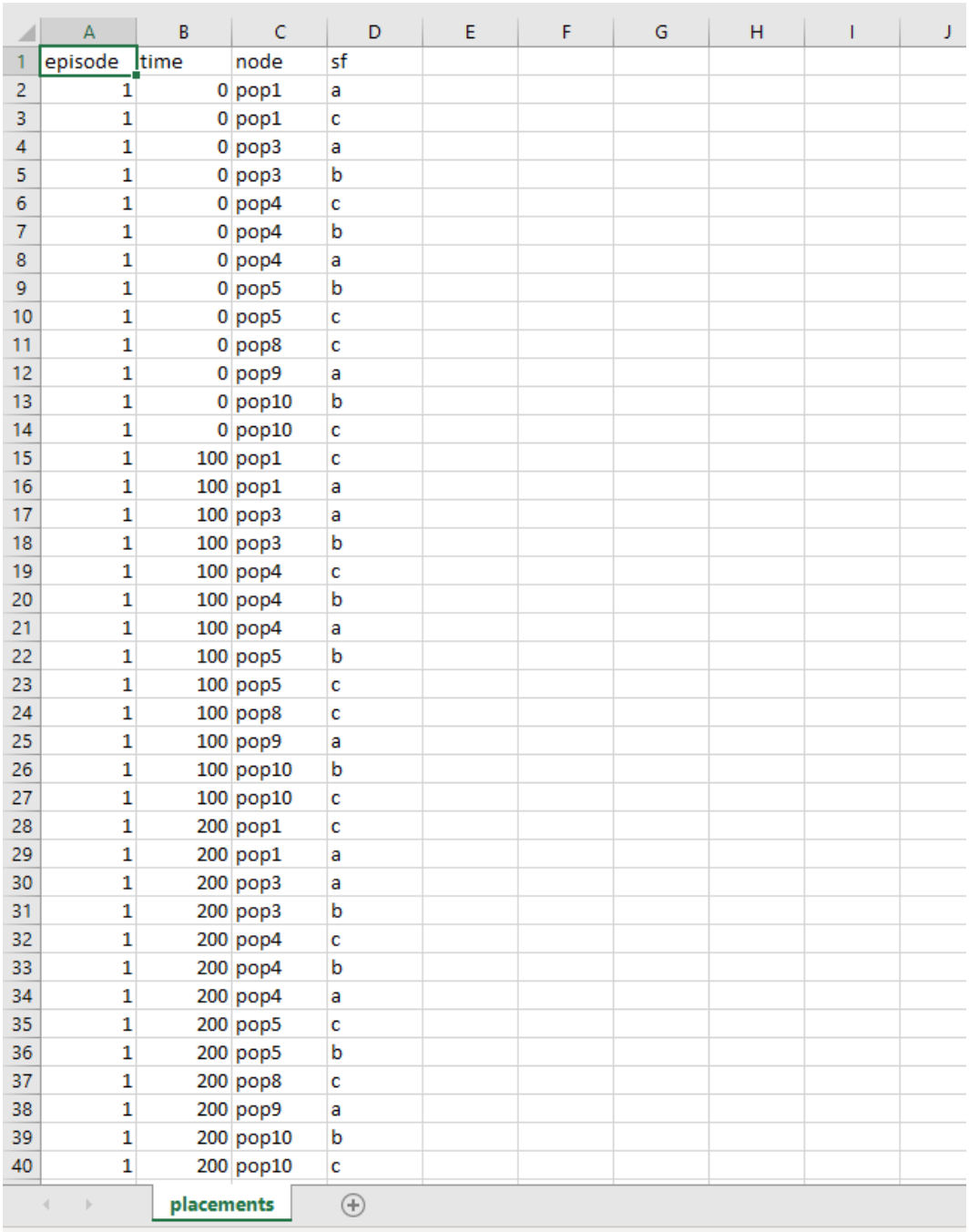
Figure 23: Metrics after running sp script



	A	B	C	D	E	F	G	H	I	J
1	episode	time	node	node_capacity	used_resources	ingress_traffic				
2	1	0	pop0	0	0	0				
3	1	0	pop1	7	0	0				
4	1	0	pop2	0	0	0				
5	1	0	pop3	14	0	0				
6	1	0	pop4	7	0	0				
7	1	0	pop5	14	0	0				
8	1	0	pop6	0	0	0				
9	1	0	pop7	0	0	0				
10	1	0	pop8	14	0	0				
11	1	0	pop9	7	0	0				
12	1	0	pop10	7	0	0				
13	1	100	pop0	0	0	13				
14	1	100	pop1	7	4	11				
15	1	100	pop2	0	0	6				
16	1	100	pop3	14	6	8				
17	1	100	pop4	7	5	11				
18	1	100	pop5	14	3	0				
19	1	100	pop6	0	0	0				
20	1	100	pop7	0	0	0				
21	1	100	pop8	14	1	0				
22	1	100	pop9	7	4	0				
23	1	100	pop10	7	6	0				
24	1	200	pop0	0	0	7				
25	1	200	pop1	7	4	8				
26	1	200	pop2	0	0	5				
27	1	200	pop3	14	3	7				
28	1	200	pop4	7	4	3				
29	1	200	pop5	14	3	0				
30	1	200	pop6	0	0	0				
31	1	200	pop7	0	0	0				
32	1	200	pop8	14	2	0				
33	1	200	pop9	7	2	0				
34	1	200	pop10	7	3	0				
35	1	300	pop0	0	0	7				
36	1	300	pop1	7	3	7				
37	1	300	pop2	0	0	6				
38	1	300	pop3	14	4	9				
39	1	300	pop4	7	6	14				
40	1	300	pop5	14	2	0				

node_metrics

Figure 24: Node Metrics after running sp script



	A	B	C	D	E	F	G	H	I	J
1	episode	time	node	sf						
2	1	0	pop1	a						
3	1	0	pop1	c						
4	1	0	pop3	a						
5	1	0	pop3	b						
6	1	0	pop4	c						
7	1	0	pop4	b						
8	1	0	pop4	a						
9	1	0	pop5	b						
10	1	0	pop5	c						
11	1	0	pop8	c						
12	1	0	pop9	a						
13	1	0	pop10	b						
14	1	0	pop10	c						
15	1	100	pop1	c						
16	1	100	pop1	a						
17	1	100	pop3	a						
18	1	100	pop3	b						
19	1	100	pop4	c						
20	1	100	pop4	b						
21	1	100	pop4	a						
22	1	100	pop5	b						
23	1	100	pop5	c						
24	1	100	pop8	c						
25	1	100	pop9	a						
26	1	100	pop10	b						
27	1	100	pop10	c						
28	1	200	pop1	c						
29	1	200	pop1	a						
30	1	200	pop3	a						
31	1	200	pop3	b						
32	1	200	pop4	c						
33	1	200	pop4	b						
34	1	200	pop4	a						
35	1	200	pop5	c						
36	1	200	pop5	b						
37	1	200	pop8	c						
38	1	200	pop9	a						
39	1	200	pop10	b						
40	1	200	pop10	c						

placements

Figure 25: SF placements after running sp script

	A	B	C	D	E	F	G	H	I
1	episode	time	successful_flows	dropped_flows	total_flows				
2	1	0	0	0	5				
3	1	100	37	0	44				
4	1	200	34	0	30				
5	1	300	41	0	43				
6	1	400	52	0	52				
7	1	500	36	0	43				
8	1	600	39	0	40				
9	1	700	51	0	48				
10	1	800	33	0	28				
11	1	900	49	0	51				
12	1	1000	34	0	29				
13	1	1100	36	0	43				
14	1	1200	51	0	47				
15	1	1300	37	0	37				
16	1	1400	47	0	49				
17	1	1500	28	0	27				
18	1	1600	50	2	48				
19	1	1700	42	0	50				
20	1	1800	44	0	42				
21	1	1900	46	0	41				
22	1	2000	40	0	42				
23	1	2100	51	0	53				
24	1	2200	42	0	43				
25	1	2300	44	0	40				
26	1	2400	30	0	36				
27	1	2500	47	0	49				
28	1	2600	54	0	45				
29	1	2700	41	0	44				
30	1	2800	54	0	51				
31	1	2900	32	0	30				
32	1	3000	50	0	56				
33	1	3100	46	0	49				
34	1	3200	54	0	54				
35	1	3300	57	0	58				
36	1	3400	51	0	50				
37	1	3500	55	0	56				
38	1	3600	68	0	68				
39	1	3700	48	0	50				
40	1	3800	57	0	61				

run_flows

Figure 26: Run Flows after running sp script

3.4.4 Graph Visualization

In this section we will visualize the CSV files into graphs (using pandas and matplotlib), so we can compare and evaluate the proposed algorithms.

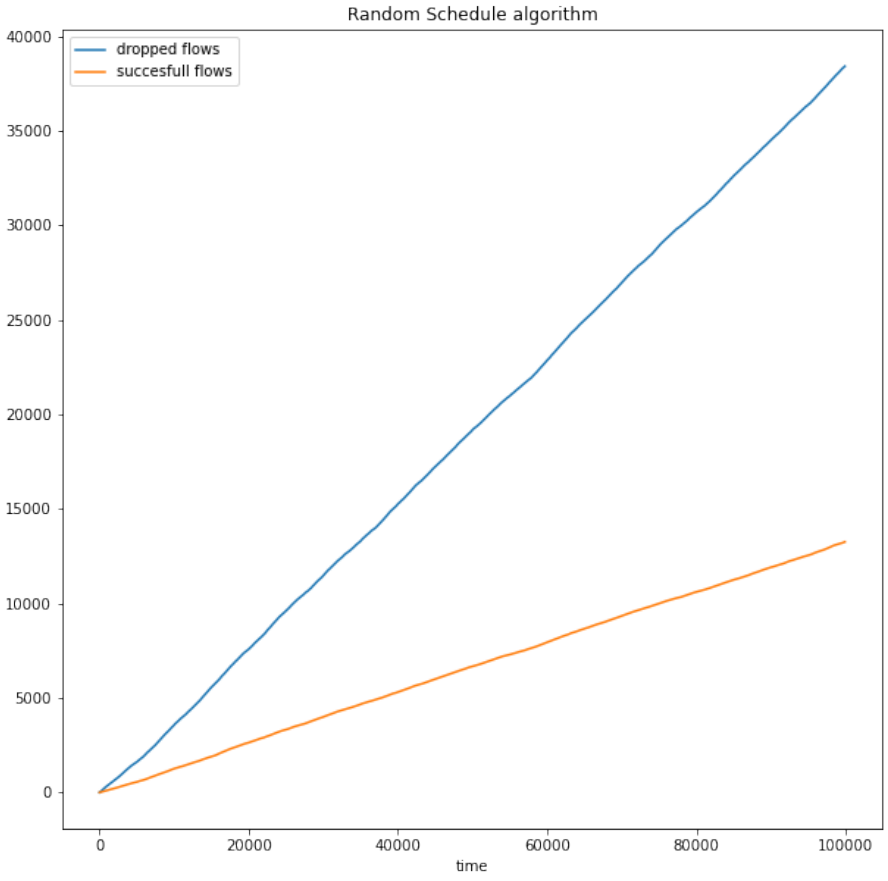


Figure 27: RS successful flows over dropped flows

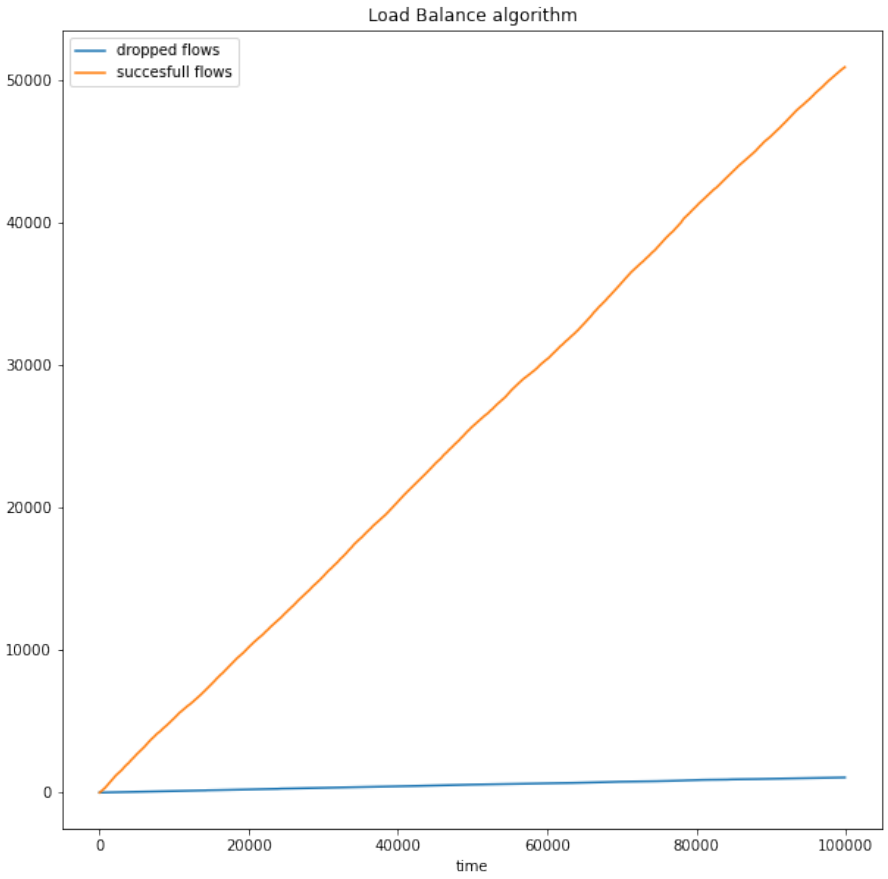


Figure 28: LB successful flows over dropped flows

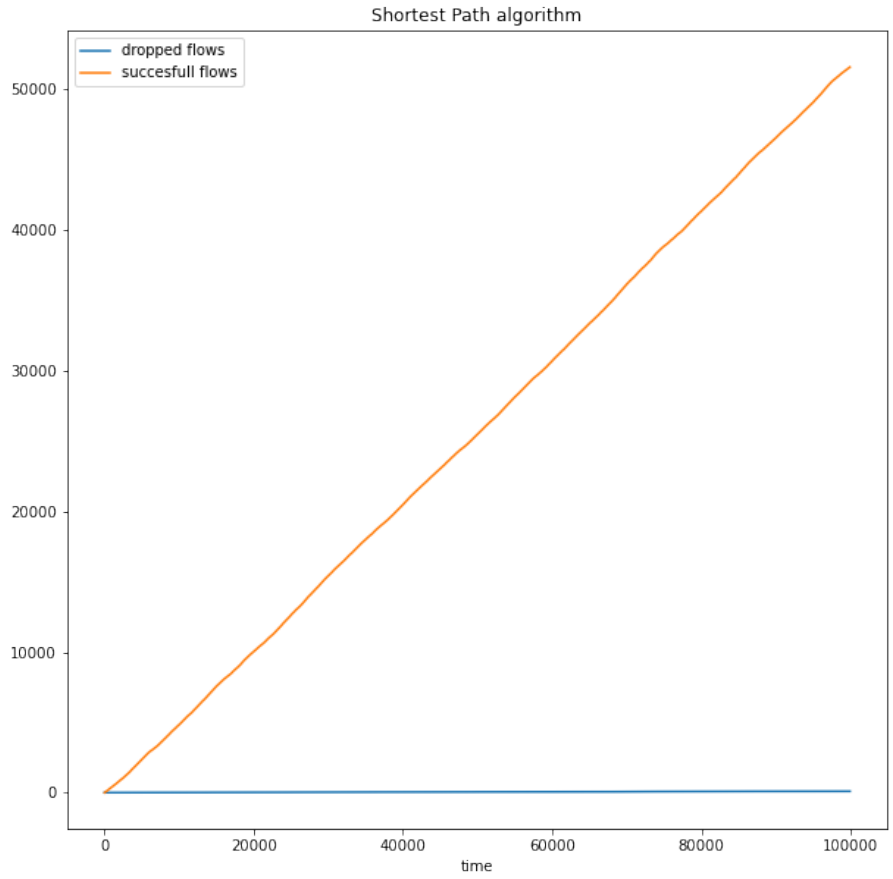


Figure 29: SP successful flows over dropped flows

1. Figures 27,28,29 show the successful flows over dropped flows achieved by three different algorithms. As expected Random Schedule algorithm has the worst results -although the successful flows are more than the dropped ones- and shows that randomness is not a good solution for the thesis subject. With the other two algorithms the results are closer. From the successful over the dropped flows we cannot deduce much. So let's see only the dropped flows for these two.

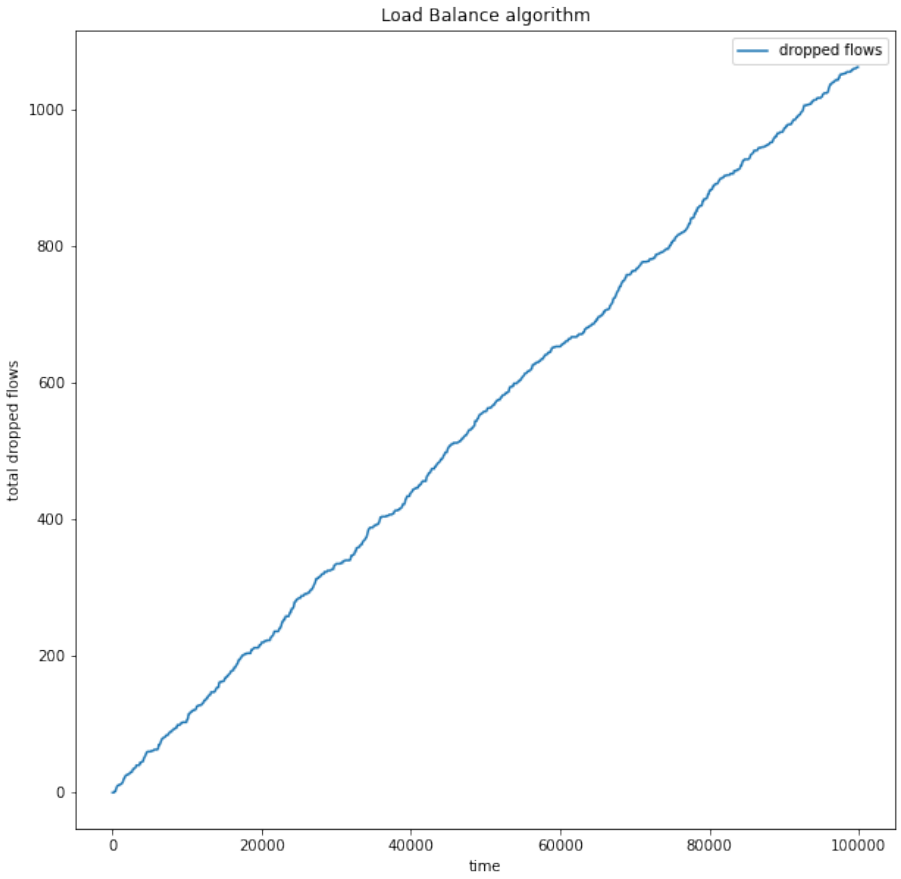


Figure 30: LB dropped flows over time

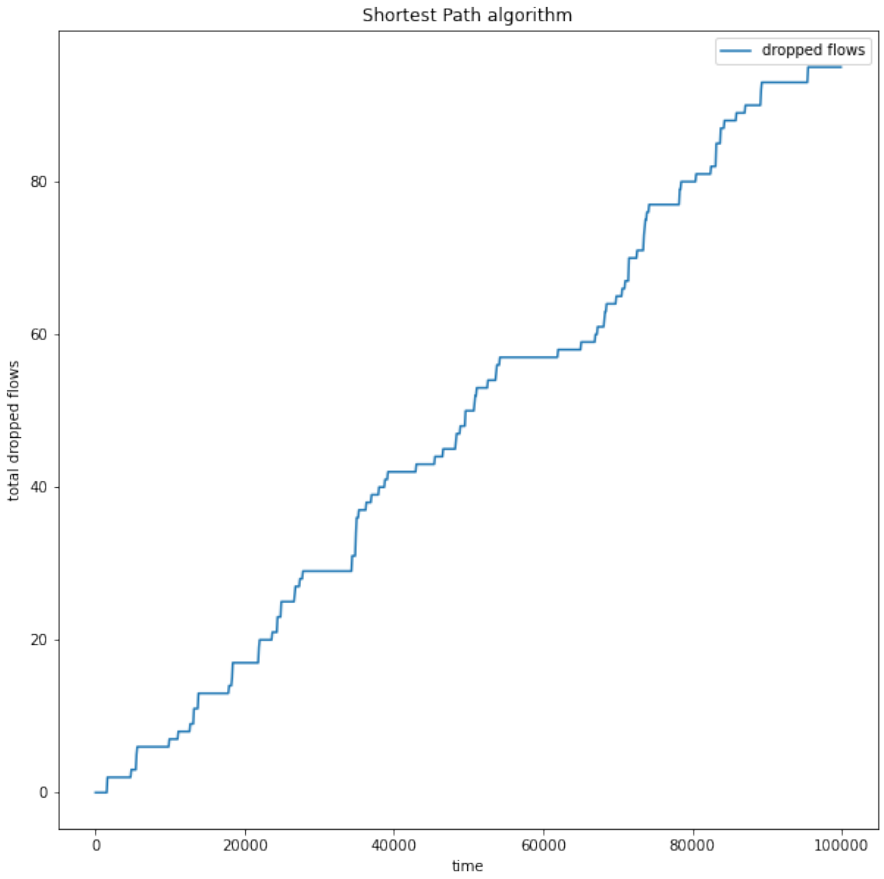


Figure 31: SP dropped flows over time

2. Figures 30, 31 provide a better look on which of these two is more efficient. With the Load Balance algorithm we can see that the total dropped flows over time reached to over 1000, whereas with the Shortest Path the dropped flows do not exceed 100. So we see deduce that regarding the dropped flows , Shortest path algorithm achieves the better results for the simulated environment.

Now let's have a look to the End-to-End delay of the successfully processed flows.

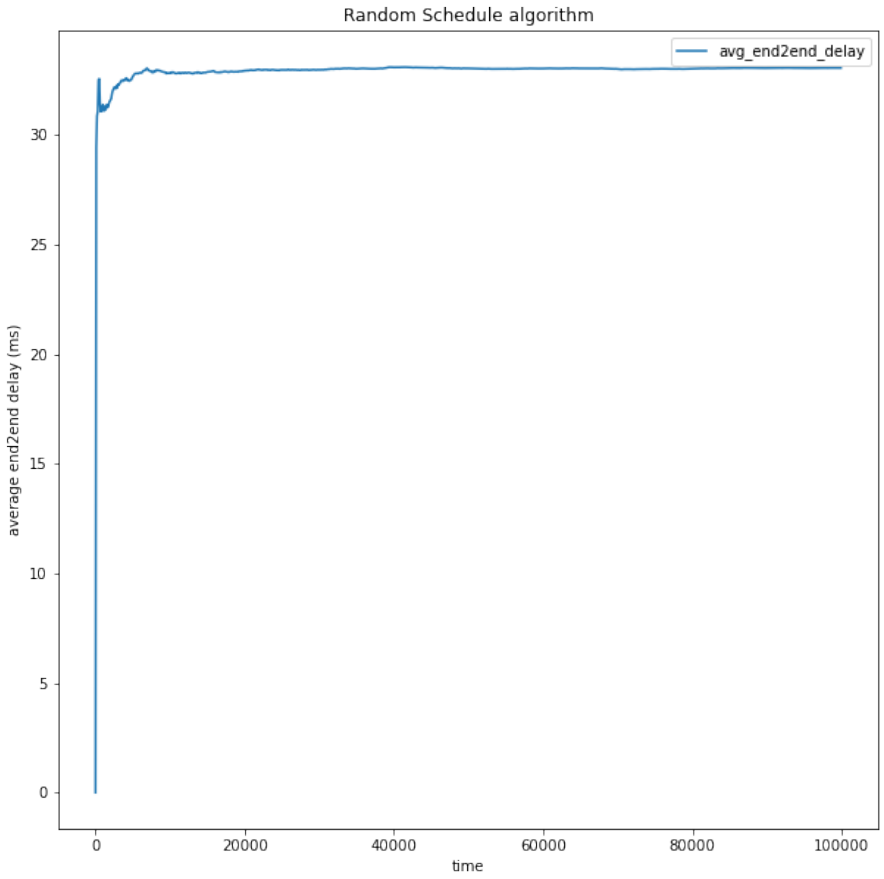


Figure 32: RS successful flows over dropped flows

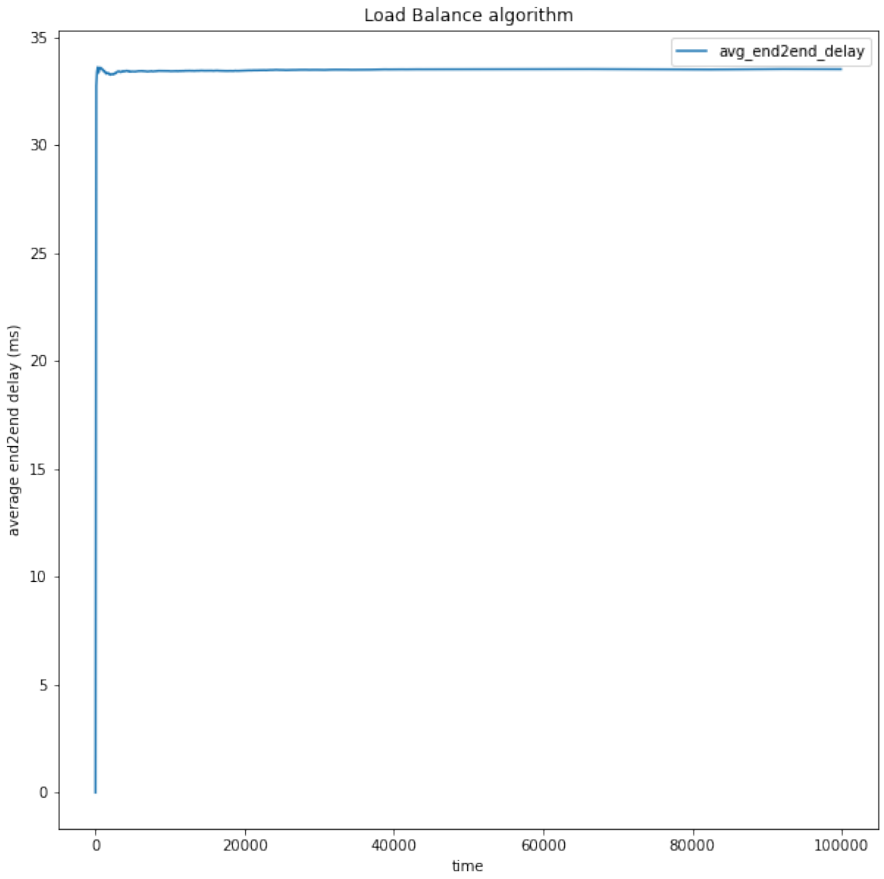


Figure 33: LB successful flows over dropped flows

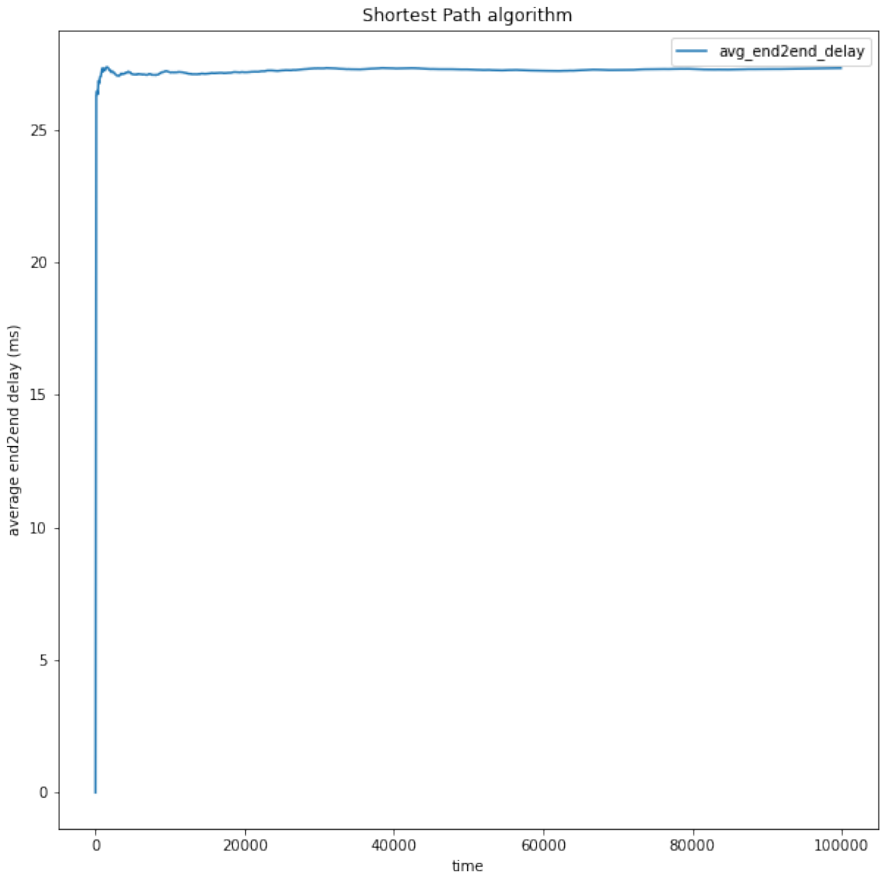


Figure 34: SP successful flows over dropped flows

3. From Figures, 32,33, 34 we see the avg. end-to-end delay of successfully process flows. In this case again Shortest path achieves better results, but the interesting fact is that Random Schedule in this case is more efficient than the Load Balance.

So all in all, Shortest path algorithm achieves far better results for this simulated environment than the other 2.

4 Conclusions

The proposed scheme (developed algorithms on a simulation environment) coordinate services well and achieve service quality comparable to a state of-the-art centralized coordination goal. At the same time, they require less or no global network information, are faster, can be massively parallelized, and are more robust to failures. We believe that an algorithmic approach can significantly improve service coordination and resulting QoS in practice.

The experimental results, which have been carried out through , demonstrate how the proposed method can have an impact in a more automated SF coordination without manual actions.

We developed a simulated NFV network on a real network topology. We simulated the VNF connectivity and we provided the test results.

5 Future directions

In future work, we can investigate more algorithms to implement in this environment and even an AI-oriented (Artificial Intelligence) decision making approach. As well, we can look in hybrid approaches, where some coordination decisions are made centrally and others in a distributed manner.

But one thing is for sure , automation can help us find solution to the VNF related problems , so that in the future the QoS that Operators need to achieve, to be optimized.

References

[1] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. Network function virtualization: State-of-the-art and research challenges. volume 18, pages 236–262, 2016.

[2] Etsi, network functions virtualisation - introductory white paper. in sdn and openflow world congress. pages 1–16, 2012. https://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[3] Bo Yi, Xingwei Wang, Keqin Li, Sajal k. Das, and Min Huang. A comprehensive survey of network function virtualization. *Computer Networks*, 133:212–262, 2018.

[4] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, 2015.

[5] Sherif Abdelwahab, Bechir Hamdaoui, Mohsen Guizani, and Taieb Znati. Network function virtualization in 5g. *IEEE Communications Magazine*, 54(4):84–91, 2016.

[6] Kelvin Lopes Dias Michel S. Bonfim and Stenio F. L. Fernandes. Integrated nfv/sdn architectures: A systematic literature review. 2018.

[7] Van-Giang Nguyen, Anna Brunstrom, Karl-Johan Grinnemo, and Javid Taheri. Sdn/nfv-based mobile packet core network architectures: A survey. *IEEE Communications Surveys Tutorials*, 19(3):1567–1602, 2017.

[8] Etsi. network functions virtualisation (nfv).

[9] Onf. software defined standards. https://www.opennetworking.org/software-defined-standards/overview/.

[10] Dmtf. standards and technology. https://www.dmtf.org/.

[11] Yong Li and Min Chen. Software-defined network function virtualization: A survey. *IEEE Access*, 3:2542–2553, 2015.

[12] Stefan Schneider, Lars Dietrich Klenner, and Karl Holger. Every node for itself: Fully distributed service coordination. In *International Conference on Network and Service Management (CNSM)*. IFIP/IEEE, 2020.

[13] Simon Knight, Hung X. Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.

[14] Realvnf coordination simulation. https://coordination-simulation.readthedocs.io/en/latest/.

A´ Code used

Key code used for running the proposed scripts: can be found at https://github.com/kopsa95/baseline-algorithms/tree/master/src/algorithms

RandomSchedule.py

```
import argparse
import logging
import os
import random
from collections import defaultdict
from datetime import datetime
from pathlib import Path
from random import uniform

from common.common_functionalities import
    normalize_scheduling_probabilities , \
    get_ingress_nodes_and_cap, copy_input_files , create_input_file
# for use with the flow-level simulator https :// github.com/RealVNF/
    coordination-simulation ( after installation )
from siminterface . simulator import Simulator
from spinterface import SimulatorAction
from tqdm import tqdm

log = logging.getLogger(__name__)
DATETIME = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
PROJECT_ROOT = str(Path(__file__).parent.parent. parent )


def get_placement( nodes_list , sf_list ):
    """ places each sf in each node of the network
    Parameters:
        nodes_list
        sf_list
    Returns:
        a Dictionary with:
```

```
            key = nodes of the network
            value = list of all the SFs in the network
    """
    placement = defaultdict ( list )
    for node in nodes_list :
        placement[node] = sf_list
    return placement


def get_schedule( nodes_list , sf_list , sfc_list ):
    """ return a dict of schedule for each node of the network
    for each node in the network, we generate floating point random
        numbers in the range 0 to 1
        '''
        Schedule is of the following form:
            schedule : dict
                {
                    'node id' : dict
                    {
                        'SFC id' : dict
                        {
                            'SF id' : dict
                            {
                                'node id' : float ( Inclusive of zero
                                    values )
                            }
                        }
                    }
                }
        '''
    Parameters:
        nodes_list
        sf_list
        sfc_list
    Returns:
        schedule of the form shown above
    """
    schedule = defaultdict (lambda: defaultdict (lambda: defaultdict (
        lambda: defaultdict ( float ))))
    for outer_node in nodes_list :
        for sfc in sfc_list :
            for sf in sf_list :
```

```
                    # this list may not sum to 1
                    random_prob_list = [uniform(0, 1) for _ in range(len(
                        nodes_list))]
                    # Because of floating point precision (.59 + .33 + .08)
                        can be equal to .99999999
                    # So we correct the sum only if the absolute diff. is
                        more than a tolerance (0.00000001490116119384765625)
                    random_prob_list = normalize_scheduling_probabilities(
                        random_prob_list)
                    for inner_node in nodes_list:
                        if len(random_prob_list) != 0:
                            schedule[outer_node][sfc][sf][inner_node] =
                                random_prob_list.pop()
                        else:
                            schedule[outer_node][sfc][sf][inner_node] = 0
    return schedule


def parse_args():
    parser = argparse.ArgumentParser(description="Dummy Coordinator")
    parser.add_argument('-i', '--iterations', required=False, default=10,
        dest="iterations", type=int)
    parser.add_argument('-s', '--seed', required=False, default=9999,
        dest="seed", type=int)
    parser.add_argument('-n', '--network', required=True, dest='network')
    parser.add_argument('-sf', '--service_functions', required=True, dest
        ="service_functions")
    parser.add_argument('-c', '--config', required=True, dest="config")
    return parser.parse_args()


def main():
    # Parse arguments
    args = parse_args()
    if not args.seed:
        args.seed = random.randint(1, 9999)
    logging.basicConfig(level=logging.INFO)
    logging.getLogger("coordsim").setLevel(logging.WARNING)

    # Creating the results directory variable where the simulator result
        files will be written
    network_stem = os.path.splitext(os.path.basename(args.network))[0]
```

```
    service_function_stem = os.path.splitext(os.path.basename(args.
        service_functions))[0]
    simulator_config_stem = os.path.splitext(os.path.basename(args.config
        ))[0]

    results_dir = f"{PROJECT_ROOT}/results/{network_stem}/{
        service_function_stem}/{simulator_config_stem}" \
                  f"/{DATETIME}_seed{args.seed}"

    # creating the simulator
    simulator = Simulator(os.path.abspath(args.network),
                          os.path.abspath(args.service_functions),
                          os.path.abspath(args.config), test_mode=True,
                              test_dir = results_dir)
    init_state = simulator.init(args.seed)
    log.info("Network Stats after init(): %s", init_state.network_stats)
    nodes_list = [node['id'] for node in init_state.network.get('nodes')]
    sf_list = list(init_state.service_functions.keys())
    sfc_list = list(init_state.sfcs.keys())
    ingress_nodes = get_ingress_nodes_and_cap(simulator.network)
    # we place every sf in each node of the network, so placement is
        calculated only once
    placement = get_placement(nodes_list, sf_list)
    # iterations define the number of time we wanna call apply()
    log.info(f"Running for {args.iterations} iterations ...")
    for i in tqdm(range(args.iterations)):
        schedule = get_schedule(nodes_list, sf_list, sfc_list)
        action = SimulatorAction(placement, schedule)
        _ = simulator.apply(action)

    # We copy the input files (network, simulator config ....) to the
        results directory
    copy_input_files(results_dir, os.path.abspath(args.network), os.path.
        abspath(args.service_functions),
                     os.path.abspath(args.config))
    # Creating the input file in the results directory containing the
        num_ingress and the Algo used attributes
    create_input_file(results_dir, len(ingress_nodes), "Rand")
    log.info(f"Saved results in {results_dir}")


if __name__ == '__main__':
```

loadBalance.py

```
import argparse
import logging
import os
import random
from collections import defaultdict
from datetime import datetime
from pathlib import Path

from common.common_functionalities import
    normalize_scheduling_probabilities , create_input_file ,
    copy_input_files , \
    get_ingress_nodes_and_cap
from siminterface . simulator import Simulator
from spinterface import SimulatorAction
from tqdm import tqdm

log = logging . getLogger(__name__)
DATETIME = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
PROJECT_ROOT = str(Path(__file__).parent.parent. parent )


def get_placement( nodes_list , sf_list ):
    """ places each sf on each node of the network with some capacity
    Parameters:
        nodes_list
        sf_list
    Returns:
        a Dictionary with:
            key = nodes of the network
            value = list of all the SFs on the node
    """
    placement = defaultdict ( list )
    for node in nodes_list :
        placement[node] = sf_list
    return placement


def get_schedule( nodes_list , nodes_with_cap, sf_list , sfc_list ):
    """ return a dict of schedule for each node of the network
        '''
        Schedule is of the following form:
```

```
            schedule : dict
                {
                    'node id' : dict
                    {
                        'SFC id' : dict
                        {
                            'SF id' : dict
                            {
                                'node id' : float (Inclusive of zero
                                    values)
                            }
                        }
                    }
                }
        '''
    Parameters:
        nodes_list
        sf_list
        sfc_list
    Returns:
        schedule of the form shown above
    """
    schedule = defaultdict(lambda: defaultdict(lambda: defaultdict(
        lambda: defaultdict(float))))
    for outer_node in nodes_list:
        for sfc in sfc_list:
            for sf in sf_list:
                # all 0's list
                uniform_prob_list = [0 for _ in range(len(nodes_with_cap)
                    )]
                # Uniformly distributing the schedules between all nodes
                uniform_prob_list = normalize_scheduling_probabilities(
                    uniform_prob_list)
                for inner_node in nodes_list:
                    if inner_node in nodes_with_cap:
                        schedule[outer_node][sfc][sf][inner_node] =
                            uniform_prob_list.pop()
                    else:
                        schedule[outer_node][sfc][sf][inner_node] = 0
    return schedule
```

```
def parse_args():
    parser = argparse.ArgumentParser(description="Load Balance Algorithm")
    parser.add_argument('-i', '--iterations', required=False, default=10,
        dest="iterations", type=int)
    parser.add_argument('-s', '--seed', required=False, dest="seed", type=int)
    parser.add_argument('-n', '--network', required=True, dest='network')
    parser.add_argument('-sf', '--service_functions', required=True, dest="service_functions")
    parser.add_argument('-c', '--config', required=True, dest="config")
    return parser.parse_args()


def main():
    # Parse arguments
    args = parse_args()
    if not args.seed:
        args.seed = random.randint(1, 9999)
    logging.basicConfig(level=logging.WARNING)
    logging.getLogger("coordsim").setLevel(logging.WARNING)

    # Creating the results directory variable where the simulator result files will be written
    network_stem = os.path.splitext(os.path.basename(args.network))[0]
    service_function_stem = os.path.splitext(os.path.basename(args.service_functions))[0]
    simulator_config_stem = os.path.splitext(os.path.basename(args.config))[0]

    results_dir = f"{PROJECT_ROOT}/results/{network_stem}/{service_function_stem}/{simulator_config_stem}" \
                  f"/{DATETIME}_seed{args.seed}"

    # creating the simulator
    simulator = Simulator(os.path.abspath(args.network),
                          os.path.abspath(args.service_functions),
                          os.path.abspath(args.config), test_mode=True,
                          test_dir=results_dir)
    init_state = simulator.init(args.seed)
    log.info("Network Stats after init(): %s", init_state.network_stats)
    nodes_list = [node['id'] for node in init_state.network.get('nodes')]
```

```
    nodes_with_capacity = []
    for node in simulator.network.nodes(data=True):
        if node[1]['cap'] > 0:
            nodes_with_capacity.append(node[0])
    sf_list = list(init_state.service_functions.keys())
    sfc_list = list(init_state.sfcs.keys())
    ingress_nodes = get_ingress_nodes_and_cap(simulator.network)
    # we place every sf on each node of the network with some capacity,
        so placement is calculated only once
    placement = get_placement(nodes_with_capacity, sf_list)
    # Uniformly distributing the schedule for all Nodes with some
        capacity
    schedule = get_schedule(nodes_list, nodes_with_capacity, sf_list,
        sfc_list)
    # Since the placement and the schedule are fixed, the action would
        also be the same throughout
    action = SimulatorAction(placement, schedule)
    # iterations define the number of time we wanna call apply()
    log.info(f"Running for {args.iterations} iterations ...")
    for i in tqdm(range(args.iterations)):
        _ = simulator.apply(action)
    # We copy the input files (network, simulator config ....) to the
        results directory
    copy_input_files(results_dir, os.path.abspath(args.network), os.path.
        abspath(args.service_functions),
                     os.path.abspath(args.config))
    # Creating the input file in the results directory containing the
        num_ingress and the Algo used attributes
    create_input_file(results_dir, len(ingress_nodes), "LB")
    log.info(f"Saved results in {results_dir}")


if __name__ == '__main__':
    main()
```

shortestPath.py

```
import argparse
import logging
import os
import random
from collections import defaultdict
from datetime import datetime
from pathlib import Path

from common.common_functionalities import
    normalize_scheduling_probabilities , create_input_file , \
    copy_input_files , get_ingress_nodes_and_cap
from siminterface . simulator import Simulator
from spinterface import SimulatorAction
from tqdm import tqdm

log = logging . getLogger(__name__)
DATETIME = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
PROJECT_ROOT = str(Path(__file__).parent.parent. parent )


def get_closest_neighbours (network, nodes_list ):
    """
    Finding the closest neighbours to each node in the network. For each
        node of the network we maintain a list of
    neighbours sorted in increasing order of distance to it .
    params:
        network: A networkX graph
        nodes_list : a list of nodes in the Network
    Returns:
        closest_neighbour : A dict containing lists of closest neighbour
            to each node in the network sorted in
                increasing order to distance .
    """

    all_pair_shortest_paths = network.graph[' shortest_paths ']
    closest_neighbours = defaultdict ( list )
    for source in nodes_list :
        neighbours = defaultdict (int)
        for dest in nodes_list :
            if source != dest :
```

```
                delay = all_pair_shortest_paths [(source, dest)][1]
                neighbours[dest] = delay
        sorted_neighbours = [k for k, v in sorted(neighbours.items(), key
            =lambda item: item[1])]
        closest_neighbours[source] = sorted_neighbours
    return closest_neighbours


def next_neighbour(index, num_vnfs_filled, node, placement,
    closest_neighbours, sf_list, nodes_cap):
    """
    Finds the next available neighbour to the index node
    Args:
        index: closest neighbours of 'node' is a list, index tells which
            closest neighbour to start looking from
        num_vnfs_filled: Tells the number of VNFs present on all nodes e.
            g: every node in the network has atleast 1 VNF,
                some might have more than that. This tells us
                    the minimum every node has
        node: The node whose closest neighbour is to be found
        placement: placeement of VNFs in the entire network
        closest_neighbours: neighbours of each node in the network in the
            increasing order of distance
        sf_list: The VNFs in the network
        nodes_cap: Capacity of each node in the network
    Returns:
            The next closest neighbour of the requested node that:
            - has some capacity
            - while some of the nodes in the network have 0 VNFs it
                returns the closest neighbour that has 0 VNFs,
              If some nodes in the network has just 1 VNF, it returns the
                  closest neighbour with just 1 VNF and so on
    """
    while len(placement[closest_neighbours[node][index]]) >
        num_vnfs_filled[0] or \
            nodes_cap[closest_neighbours[node][index]] == 0:
        index += 1
        if index == len(closest_neighbours[node]):
            num_vnfs_filled[0] += 1
            index = 0
        if num_vnfs_filled[0] > len(sf_list):
            index = 0
```

```
            break
    return index


def get_placement_schedule(network, nodes_list, sf_list, sfc_list,
    ingress_nodes, nodes_cap):
    """
        '''
        Schedule is of the following form:
            schedule : dict
                {
                    'node id' : dict
                    {
                        'SFC id' : dict
                        {
                            'SF id' : dict
                            {
                                'node id' : float (Inclusive of zero
                                    values)
                            }
                        }
                    }
                }
        '''
    Parameters:
        network: A NetworkX object
        nodes_list: all the nodes in the network
        sf_list: all the sf's in the network
        sfc_list: all the SFCs in the network, right now assuming to be
            just 1
        ingress_nodes: all the ingress nodes in the network
        nodes_cap: Capacity of each node in the network
    Returns:
        - a placement Dictionary with:
                key = nodes of the network
                value = list of all the SFs in the network
        - schedule of the form shown above
    """
    placement = defaultdict(list)
    schedule = defaultdict(lambda: defaultdict(lambda: defaultdict(
        lambda: defaultdict(float))))
    # Initializing the schedule for all nodes, for all SFs to 0
```

```
for src in nodes_list :
    for sfc in sfc_list :
        for sf in sf_list :
            for dstn in nodes_list :
                schedule[ src ][ sfc ][ sf ][ dstn ] = 0
# Getting the closest neighbours to each node in the network
closest_neighbours = get_closest_neighbours (network, nodes_list )

# - For each Ingress node of the network we start by placing the
    first VNF of the SFC on it and then place the
#  2nd VNF of the SFC on the closest neighbour of the Ingress , then
    the 3rd VNF on the closest neighbour of the node
#  where we placed the 2nd VNF and so on.
# - The closest neighbour is chosen based on the following criteria :
#   - while some nodes in the network has 0 VNFs , the closest
    neighbour cannot be an Ingress node
#   - The closest neighbour must have some capacity
#   - while some of the nodes in the network have 0 VNFs it chooses
    the closest neighbour that has 0 VNFs,
#     If some nodes in the network has just 1 VNF, it returns the
    closest neighbour with just 1 VNF and so on
for ingress in ingress_nodes:
    node = ingress
    # We choose a list with just one element because a list is
        mutable in python and we want 'next_neighbour'
    # function to change the value of this variable
    num_vnfs_filled = [0]
    # Placing the 1st VNF of the SFC on the ingress nodes if the
        ingress node has some capacity
    # Otherwise we find the closest neighbour of the Ingress that has
        some capacity and place the 1st VNF on it
    if nodes_cap[ ingress ] > 0:
        if sf_list [0] not in placement[node]:
            placement[node].append( sf_list [0])
        schedule[node][ sfc_list [0]][ sf_list [0]][ node] += 1
    else :
        # Finding the next neighbour which is not an ingress node and
            has some capacity
        index = next_neighbour(0, num_vnfs_filled , ingress , placement
            , closest_neighbours , sf_list , nodes_cap)
        while num_vnfs_filled[0] == 0 and closest_neighbours [ ingress
            ][ index ] in ingress_nodes :
```

```
                if index + 1 >= len( closest_neighbours [ ingress ]) :
                    break
                index = next_neighbour(index + 1, num_vnfs_filled,
                    ingress , placement, closest_neighbours ,
                                        sf_list , nodes_cap)
            node = closest_neighbours [ ingress ][ index]
            if sf_list [0] not in placement[node]:
                placement[node].append( sf_list [0])
            schedule[ ingress ][ sfc_list [0]][ sf_list [0]][ node] += 1

        # For the remaining VNFs in the SFC we look for the closest
            neighbour and place the VNFs on them
        for j in range(len( sf_list ) - 1):
            index = next_neighbour(0, num_vnfs_filled, node, placement,
                closest_neighbours , sf_list , nodes_cap)
            while num_vnfs_filled[0] == 0 and closest_neighbours [node][
                index] in ingress_nodes :
                if index + 1 >= len( closest_neighbours [node]):
                    break
                index = next_neighbour(index + 1, num_vnfs_filled, node,
                    placement, closest_neighbours ,
                                        sf_list , nodes_cap)
            new_node = closest_neighbours[node][index]
            if sf_list [j + 1] not in placement[new_node]:
                placement[new_node].append( sf_list [j + 1])
            schedule[node][ sfc_list [0]][ sf_list [j + 1]][ new_node] += 1
            node = new_node

    # Since the sum of schedule probabilities for each SF of each node
        may not be 1 , we make it 1 using the
    # ' normalize_scheduling_probabilities ' function .
    for src in nodes_list :
        for sfc in sfc_list :
            for sf in sf_list :
                unnormalized_probs_list = list (schedule[ src ][ sfc ][ sf ].
                    values () )
                normalized_probs = normalize_scheduling_probabilities (
                    unnormalized_probs_list )
                for i in range(len( nodes_list )):
                    schedule[ src ][ sfc ][ sf ][ nodes_list [ i ]] =
                        normalized_probs[ i ]
    return placement, schedule
```

```
def parse_args():
    parser = argparse.ArgumentParser(description="Load Balance Algorithm"
        )
    parser.add_argument('-i', '-- iterations ', required=False, default=10,
         dest=" iterations ", type=int)
    parser.add_argument('-s', '--seed', required=False, dest="seed", type
        =int)
    parser.add_argument('-n', '--network', required=True, dest='network')
    parser.add_argument('-sf', '-- service_functions ', required=True, dest
        =" service_functions ")
    parser.add_argument('-c', '--config', required=True, dest="config")
    return parser.parse_args()


def main():
    # Parse arguments
    args = parse_args()
    if not args.seed:
        args.seed = random.randint(1, 9999)
    logging.basicConfig(level=logging.WARNING)
    logging.getLogger("coordsim").setLevel(logging.WARNING)

    # Creating the results directory variable where the simulator result
        files will be written
    network_stem = os.path.splitext(os.path.basename(args.network))[0]
    service_function_stem = os.path.splitext(os.path.basename(args.
        service_functions))[0]
    simulator_config_stem = os.path.splitext(os.path.basename(args.config
        ))[0]

    results_dir = f"{PROJECT_ROOT}/results/{network_stem}/{
        service_function_stem}/{simulator_config_stem}" \
                  f"/{DATETIME}_seed{args.seed}"

    # creating the simulator
    simulator = Simulator(os.path.abspath(args.network),
                          os.path.abspath(args.service_functions),
                          os.path.abspath(args.config), test_mode=True,
                              test_dir=results_dir)
    init_state = simulator.init(args.seed)
```

```
    log.info("Network Stats after init(): %s", init_state.network_stats)
    nodes_list = [node['id'] for node in init_state.network.get('nodes')]
    sf_list = list(init_state.service_functions.keys())
    sfc_list = list(init_state.sfcs.keys())
    ingress_nodes, nodes_cap = get_ingress_nodes_and_cap(simulator.
        network, cap=True)
    # getting the placement and schedule
    placement, schedule = get_placement_schedule(simulator.network,
        nodes_list, sf_list, sfc_list, ingress_nodes,
                                                 nodes_cap)
    # Since the placement and the schedule are fixed, the action would
        also be the same throughout
    action = SimulatorAction(placement, schedule)
    # iterations define the number of time we wanna call apply(); use
        tqdm for progress bar
    log.info(f"Running for {args.iterations} iterations ...")
    for i in tqdm(range(args.iterations)):
        _ = simulator.apply(action)
    # We copy the input files (network, simulator config ....) to the
        results directory
    copy_input_files(results_dir, os.path.abspath(args.network), os.path.
        abspath(args.service_functions),
                     os.path.abspath(args.config))
    # Creating the input file in the results directory containing the
        num_ingress and the Algo used attributes
    create_input_file(results_dir, len(ingress_nodes), "SP")
    log.info(f"Saved results in {results_dir}")


if __name__ == '__main__':
    main()
```

B´ Author Resume

Christos Kopsacheilis, Msc Digital Communications and Networks, University of Piraeus

- **Personal information**
  Birth date: 1995 | Birth Place: Larissa, Greece

- **Studies**
  2013 , Lyceum Diploma ,1st High school of Giannouli Larissas
  2014-2019: Bachelor of Science (BSc), Mathematics, NKUA
  2019–present: Master of Science (MSc), Digital Communications and Networks, University of Piraeus

- **Work Experience**
  2018- 2020: Cloud Application Support Engineer, Atos
  2020-present: IMS Integration Engineer, Ericsson