Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Інститут телекомунікаційних систем
(повна назва інституту/факультету)

Кафедра телекомунікацій
(повна назва кафедри)

«На правах рукопису»
УДК ______________

До захисту допущено
В.о. завідувача кафедри

__________ Явіся В.С.___.
(підпис) (ініціали, прізвище)

“___”____________2019_р.

Магістерська дисертація

на здобуття освітнього ступеня «магістр»

Спеціальність 172 Телекомунікації та радіотехніка,
(код і назва)

За освітньо-професійною програмою Інженерія та програмування інфокомунікацій.

на тему: _ Метод оптимізації програмно-конфігурованих мереж із використанням _ SDN Optimization Layer____
____

Виконав (-ла): студент (-ка) _2_ курсу, групи ТЗ -з81 мп
(шифр групи)

_____ Сікач Тарас Олексійович ____________________________ ____________
(прізвище, ім’я, по батькові) (підпис)

Науковий керівник ___________________ Валуйський С.В. _____ ____________
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант ____ ____________
(назва розділу) (науковий ступінь, вчене звання, , прізвище, ініціали) (підпис)

Рецензент ____________________________ Скулиш М.А. _______ ____________
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____________
(підпис)

Київ – 2019 рік

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Інститут телекомунікаційних систем
(повна назва)

Кафедра телекомунікацій
(повна назва)

Спеціальність 172 Телекомунікації та радіотехніка
(код і назва)

Рівень вищої освіти – другий (магістерський) за освітньо-професійною програмою Інженерія та програмування інфокомунікацій.

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
__________ Явіся В.С.
(підпис) (ініціали, прізвище)
«___»____________ 2019 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

__________________ Сікачу Тарасу Олексійовичу __________________________
(прізвище, ім'я, по батькові)

1. Тема дисертації _ Метод оптимізації програмно-конфігурованих мереж із використанням SDN Optimization Layer____
____

науковий керівник дисертації _____ Валуйський С.В. __________________,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «_07_» «_11_» 2019р. № _3854-с_

2. Строк подання студентом дисертації ___________09______________________

3. Об'єкт дослідження _ Програмно-конфігуровані мережі _______________
____

4. Предмет дослідження _ Метод оптимізації програмно-конфігурованих мереж____

5. Перелік завдань, які потрібно розробити ______________________________
____
____
____
____

6. Орієнтовний перелік ілюстративного матеріалу ________________________
____
____

7. Орієнтовний перелік публікацій ____________________________________
____
____

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

9. Дата видачі завдання

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Строк виконання етапів магістерської дисертації	Примітка

Студент _____________ _____ Сікач Т.О. ______
(підпис) (ініціали, прізвище)

Науковий керівник дисертації _____________ __ Валуйський С.В.____
(підпис) (ініціали, прізвище)

Пояснювальна записка
до магістерської дисертації

на тему: ______ Метод оптимізації програмно-конфігурованих мереж із використанням SDN Optimization Layer____
____

Київ – 2019 року

РЕФЕРАТ

ТЕМА: «Метод оптимізації програмно-конфігурованих мереж із використанням SDN Optimization Layer»

Пояснювальна записка викладена на 85 сторінках та включає 22 ілюстрацій, 1 таблицю та 47 джерел за переліком посилань.

Об'єктом дослідження виступають програмно-конфігуровані мережі.

Предметом дослідження – метод оптимізації програмно-конфігурованих мереж.

Метою роботи є виведення/розробка оптимізаційного рішення для програмно-конфігурованих мереж.

В ході написання роботи було розглянуто рівні архітектури програмно-визначених мереж, мережеві програми та додатки які застосовуються в програмно-конфігурованих мережах, а також проблеми та вимоги які висуваються цими додатками для успішної та ефективної їх роботи із використанням якомога меншої кількості як мережевих ресурсів так і комп'ютерних.

Програмно-конфігуровані мережі допоможуть операторам керувати послугами мережі, коли функціонал управління відділений від нижнього рівня даних. Планування мережі і управління трафіком в при цьому відбувається програмним шляхом. Це дозволить операторам піти від використання дорогих спеціалізованих апаратно-програмних комплексів в сторону віртуалізованих програмних рішень. Всеохоплююче бачення в розробці SOL підняло рівень абстракції в розробці нових SDN додатків та особливо в зменшенні деяких з недосказанностей в розробці SDN базованих оптимізацій, роблячи їх більш доступними для розгортання мережевими менеджерами.

Ключові слова: програмно-конфігуровані мережі, віртуалізація, networks, SDN Optimization Layer, хмарні технології.

ПРОГРАМНО-КОНФІГУРОВАНІ МЕРЕЖІ, ВІРТУАЛІЗАЦІЯ, ВІРТУАЛЬНІ МАШИНИ, АРХІТЕКТУРА, МЕРЕЖА, МІГРАЦІЯ, ЦЕНТРАЛІЗАЦІЯ, SDN, SOL, VNF, NFV, GUROBI, DEFO, ET.

ABSTRACT

Topic: «Method Of Optimization Software-Defined Networks Using SDN Optimization Layer»

Explanatory memorandum presented on 85 pages and includes 22 illustrations, 1 table and 47 sources for references.

The object of the study is software defined networks.

The subject of the study is a method of optimization of SDN telecommunication networks.

The purpose of this work is to develop/implement an optimization solution for software-defined networks.

During the writing of the work were considered, levels of software-defined networks architecture, their components and their interactions within the level, applications which can be used in SDN networks, as well as requirements and problems which are implied by these applications for effective and successful their work using as little networks resources and computational resources as possible.

Software-configurable networks will help operators manage services when the management functionality is separated from the lower-level data. Planning of the network and traffic management in this case occurs programmatically. This will allow operators to avoid using expensive specialized hardware and software systems in the direction of virtualized software solutions. Overall vision in SOL implementation lift level of abstraction in development of new SDN applications and especially in reducing some of imperfections in development of SDN based optimization, making them more accessible for deployment by network managers

Keyword: software-defined networking, virtualization, networks, SDN Optimization Layer, cloud technologies.

SOFTWARE-DEFINED NETWORKING, VIRTUALIZATION, VIRTUAL MACHINES, ARCHITECTURE, NETWORK, MIGRATION, CENTRALIZATION, SDN, SOL, VNF, NFV, GUROBI, DEFO, ET.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ 10
ВСТУП 14
1. ПЕРЕДУМОВИ ТА МОТИВАЦІЯ ДЛЯ РІШЕНЬ SOL 22
1.1 Додатки/програм які можуть бути удосконаленими 22
1.1.1 Інженерія трафіку 22
1.1.2 Ланцюгування сервісів 24
1.1.3 Гнучке управління топологією. 26
1.1.4 Віртуалізація мережевих функцій 27
1.2 Мотивація для SOL 29
Висновки до розділу 31
2.SDN OPTIMIZATION LAYER ЯК РІШЕННЯ 32
2.1 Загальний огляд SOL 32
2.2 Деталізований дизайн SOL 37
2.2.1 Прелімінарії 37
2.2.2 Маршрутизаційні вимоги 39
2.2.3 Обмеження ресурсної ємності 41
2.2.4 Активаційні обмеження вузлів/зв’язків 43
2.2.5 Специфікація мережевих цілей 45
2.3 Генерація та вибір шляху. 46
Висновки до розділу 48
3. ПРАКТИЧНА РЕАЛІЗАЦІЯ SOL РІШЕННЯ 50
3.1 Загальний приклад 50
3.2. Детальний опис реалізації 52

3.3 Оцінка та порівняння 54

3.3.1 Орієнтовне оцінювання розгортання 56

3.3.2 Оптимальність та Масштабованість. 58

3.3.3 Порівняння з Merlin та DEFO 62

3.4 Переваги для розробників 63

3.5 Чутливість 64

3.6 Виразність SOL 67

3.7 Зв'язані праці/роботи 68

Висновки до розділу 70

ЗАГАЛЬНІ ВИСНОВКИ ПО РОБОТІ 71

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 73

ДОДАТОК А 79

ДОДАТОК Б 83

ДОДАТОК В 85

ПЕРЕЛІК СКОРОЧЕНЬ

Англійські скорочення

API	Application Programming Interface	Прикладний програмний інтерфейс
AMPL	A Mathematical Programming Language	Математична мова програмування
BSS	Business Support System	Система підтримки бізнесу
CPI	Controller Plane Interface	Інтерфейс рівня контролеру
CPLEX	IBM ILOG CPLEX Optimization Studio	Симплекс метод/алгоритм оптимізації
CPU	Central Processing Unit	Центральний процесор
EEO	End-to End Orchestrator	Загальний оркестратор
DEFO	Optimization Framework	Оптимізаційний фреймворк
DPI	Deep Packet Inspection	Протокол глибокого дослідження пакетів
E2E	End To End	Повний цикл
EMS	Element Management System	Система управління мережевими елементами

EPC	Evolved Packet Core	Вдосконалене пакетне ядро
ET	ElasticTree	ElasticTree
ETSI	European Telecommunications Standards Institute	Європейський Інститут Телекомунікаційних Стандартів
FCAPS	Fault, Configuration, Accounting, Performance, Security	Відмови, конфігурація, облік, продуктивність, безпека
GUROBI	Optimization Solver	Оптимізаційний вирішувач
HA	High Availability	Висока доступність
IDS	Intrusion Detection System	Система розпізнавання втручань
IP	Internet Protocol	Інтернет протокол
IPS	Intrusion Prevention System	Система Запобігання Втручанням
MB	MiddleBox	Мережевий пристрій
NFV	Network Functions virtualization	Віртуалізація мережевих функцій
NMS	Network Management System	Система управління мережею
NSD	Network Service Descriptor	Дескриптор мережевих послуг

ONOS	Open Network Operating System	Відкрита Мережева Операційна Система
OSS	Operation Support System	Система підтримки операцій
PNF	Physical Network Function	Фізична мережева функція
PS	Policy Steering	Політики регулювання
LOC	Lines Of Code	Рядки коду
SDN	Software-defined network	Програмно-визначена мережа
SIMPLE	SDN-based policy enforcement layer middlebox-specific “traffic steering”	SDN політика для регулювання трафіку
SOL	SDN Optimization Layer	Рівень оптимізації SDN
SLA	Service-Level Agreement	Угоди про рівень обслуговування
REST	Representational State Transfer	Передача репрезентативного стану
TE	Traffic Engineering	Інженерія трафіку
TCAM	Telecommunication Access Method	Телекомунікаційний Метод Доступу
TZ	Topology Zoo	Топологія Zoo

VLAN	Virtual Local Area Network	Віртуальна локальна комп'ютерна мережа
VM	Virtual Machine	Віртуальна машина
VNF	Virtualized Network Function	Віртуалізована мережева функція
VNFM	Virtual Network Function Manager	Менеджер віртуальних мережевих функцій
VPN	Virtual Private Network	Віртуальна приватна мережа
WAN	Wide Area Network	Глобальна комп’ютерна мережа

ВСТУП

Загальний концепт програмно-конфігурованих мереж. Програмно-конфігурована мережа - мережа передачі даних, в якій рівень управління мережею відділений від пристроїв передачі даних і реалізується програмно, одна з форм віртуалізації обчислювальних ресурсів.

Три найбільш резонансні концепти SDN це можливість програмування, відділення рівнів контролю, та рівня даних, а також управління тимчасового стану мережі в централізованій моделі управління, незважаючи на ступінь централізації. В кінцевому рахунку, ці концепти втілені в ідеалізований SDN фреймворк. SDN контролер є втіленням ідеалізованого SDN фреймворку та в більшості випадків, є відображенням його.

Теоретично, контролер SDN надає служби, які може реалізувати розподілений рівень контролю, а також сприяє концепту тимчасового стану управління та централізації. Насправді, будь-який даний екземпляр контролера забезпечить зріз, або підмножину цієї функціональності, а також внесе щось своє в ці концепції.

Загальний опис контролера SDN являє собою систему програмного забезпечення , або сукупність систем що разом забезпечують:

1) Управління стану мережі і в деяких випадках управління та розподіл цього стану, може включати в себе базу даних.

2) Модель даних високого рівня, яка фіксує відносини між керованими ресурсами, політиками і іншими сервісами, що надаються контролером.

3) Сучасний, прикладний програмний інтерфейс (API) надає відкритий доступ служб контролеру до аплікації. Це полегшує більшу частину взаємодії контролер – аплікація. Цей інтерфейс ідеально наданий з моделі даних, що описує служби і особливості контролера.

4) Безпечний контроль TCP сеансами між контролером та асоційованими агентами в мережі елементів.

5) Заснований на стандартах протокол для надання програмно-спрямованої мережі стан на мережевих елементах.

6) Пристрій, топологія та механізм відкриття служб; система розрахунку шляху і потенційно інші мережево-орієнтовані або ресурсо-орієнтовані інформаційні служби.

Для програмних комутаторів / маршрутизаторів на хостах, контролер SDN являється критичним інтерфейсом управління. Контролери SDN пропонують деякі послуги по управлінню (на додаток до ініціалізації і виявлення), так як вони несуть відповідальність за асоційовані стани для своїх тимчасовий об'єктів мережі, таких як аналітика і повідомлення про події. У цьому аспекті, SDN має потенціал, щоб реконструювати наш погляд управління мережевого елемента (EMS).

Зазначимо ще раз що програмно-визначені мережі - це мережева архітектура (рисунок 1), де рівень управління мережею відокремлений від пристроїв передачі даних і реалізується програмно. «Інтелектуальні» функції мережі централізовані в основі програмного SDN контролеру, який підтримує глобальний вид мережі. В результаті, мережа відображається для додатків і систем політик як єдиний логічний комутатор. З SDN, підприємства можуть отримати незалежний від постачальника контроль над

всією мережею з єдиної логічної точки, що значно спрощує проектування мережі і її експлуатацію. SDN також значно спрощує мережеві пристрої самі по собі, так як вони більше не повинні розуміти і обробляти тисячі стандартів протоколів, а повинні просто приймати вказівки від контролеру SDN.

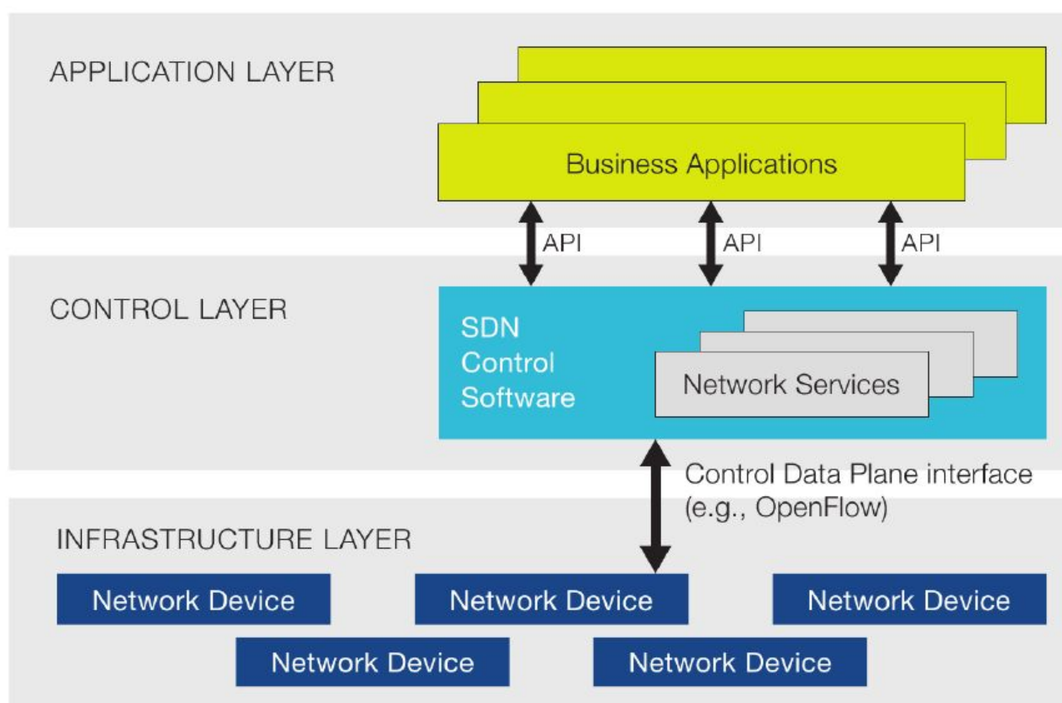


Рисунок 1: Спрощене відображення архітектури SDN

Найголовніше, що мережеві оператори і адміністратори можуть програмно налаштовувати цю спрощену мережеву абстракцію замість того, щоб вручну прописувати десятки тисяч рядків конфігурації. Крім того, використовуючи централізований інтелект контролера SDN можна змінити поведінку мережі в режимі реального часу і розгорнути нові програми та мережеві служби протягом декількох годин або днів. Крім того, вони можуть написати ці програми самі, а не чекати, поки можливості будуть

вбудовуванні в фірмові запатентовані та закриті програмні середовища в середині мережі.

Проблематика. Усвідомлення переваг SDN для багатьох мережевих додатків, програм (наприклад, інженерія трафіку, реконфігурація топологій, послідовне з'єднання сервісів) включає адресацію складних оптимізацій, які є центральними до цих проблем.

На жаль, такі проблеми оптимізації вимагають

- значних мануальних зусиль та експертизи для вираження
- нетривіальних обчислень та/або ретельно продуманих евристик для вирішення

Ціль – спростити розгортання SDN додатків, використовуючи загальні високорівневі абстракції для захоплення оптимізаційних вимог з яких ми можемо ефективно генерувати оптимальні рішення. Тому, SOL був презентований, фреймворк який демонструє що це можливо, одночасно досягти узагальнення та ефективності. Особливість, яка надає SOL можливість вирішити ці проблеми, це те що багато SDN додатків можуть бути перероблені в рамках уніфікованої оптимізаційної абстракції яка базується на метриках шляху. Використовуючи цю особливість, SOL може ефективно генерувати рішення близькі до оптимальних та конфігурації пристроїв для їх реалізації. SOL надає відносно порівнювану або кращу масштабованість ніж спеціалізовані оптимізаційні рішення для різноманітних додатків, дозволяє балансувати оптимальність та змішування шляхів для кожної переконфігурації. Та інтерфейси із сучасними SDN контролерами.

Програмно-конфігуровані мережі (SDN) - активізатор для мережевого управління додатками, який може бути досить складно реалізувати

використовуючи існуючі механізми рівня управління. Останні роботи які використовували механізми основані на SDN підході, були націлені на реалізацію конфігурації мережі для різноманітних завдань з управління:

- Розробка трафіку
- Ланцюгування (послідовне з’єднання) сервісів
- Енергоефективність
- Віртуалізація мережевих функцій (NFV)
- Вивантаження хмар
- Та інші.

Хоч ця робота була інструментальною для демонстрації потенційних переваг програмно-конфігурованих мереж, реалізація цих переваг вимагає значних зусиль. У більшості випадків, в основі багатьох SDN додатків лежить специфічні оптимізаційні проблеми для вирішення різноманітних обмежень та вимог які проявляються на практиці (детальні розділі 1). Наприклад, SDN додаток може потребувати: доступ до обмеженого TCAM, можливості посилань, можливостей мережевих пристроїв (MB), та інші. Розробка таких формулювань включає нетривіальні шляхи вивчення, глибокого розуміння теоретичних та практичних питань та значних мануальних зусиль. Крім того, коли результуюча оптимізаційна проблема не може бути вирішена навіть найсучаснішими інструментами/технологіями (CPLEX, Gurobi), евристичні алгоритми повинні бути створені для удостовірення в тому що нові конфігурації можуть бути згенеровані за потребою додатку як зміна релевантних вхідних даних. Крім того, без спільного фреймворку для репрезентації мережевих оптимізаційних задач,

досить складно перевикористати ключові ідеї між додатками, або об'єднати корисні рішення/особливості в деякий новий спеціалізований додаток.

Рисунок 2: Розробники використовують високорівневі SOL API для специфікації оптимізаційних цілей та обмежень. SOL генерує близькі до оптимальних рішення та виробляє конфігурації для пристроїв що є вхідними даними до рівня контролю SDN мережі.

Основна ціль в цій роботі – підняти рівень абстракції для написання мережевих оптимізаційних додатків SDN мереж. Тому, було представлено SOL, фреймворк що надає можливість розробникам SDN додатків виражати високорівневі абстрактні цілі та обмеження. Концептуально, SOL – це проміжний рівень що знаходиться між оптимізаційним додатком SDN та безпосередньо платформою контролю (рівнем контролю). Розробники додатків, які хочуть розробити нові оптимізаційні можливості для мереж – виражають вимоги використовуючи SOL API. SOL потім генерує

конфігурації що відповідають потрібним цілям, які можуть бути розгорнуті (встановлені) на платформи управління SDN.

Існують дві природні вимоги до такого фреймворку:

1. Загальність для вираження вимог до широкого спектру SDN додатків. (трафік інжиніринг, розподілення навантаження, управління топологіями, управління політиками policy steering)
2. Ефективність для створення майже оптимальної конфігурації в часовому масштабі, який відповідає вимогам додатків.

З огляду на різноманітність вимог додатків та траєкторію початкової роботи в розробці спеціалізованих рішень, загальність та ефективність виявляються досить складними навіть окремо, не говорячи навіть про поєднання. Але не зважаючина це, досягти одночасно узагальненості та ефективності, дійсно можливо. Ключова особливість, яка надає можливість SOL-у досягати узагальненості – це те, що більшість проблем мережевої оптимізації можуть бути виражені як формулювання побудовані на основі шляху. Шляхи – це природні абстракції для розробників додатків для міркувань про очікувану поведінку мереж, а також для вираження політики вимог. Наприклад, ми можемо використовувати шляхи для визначення вимог ланцюга служб (наприклад, кожен шлях включає брандмауер та систему виявлення вторгнень, у такому порядку) або надмірність (наприклад, кожен включає дві системи для запобігання вторгненням, для підвищення надійності у випадку відмови однієї). І на кінець, досить легко змоделювати пристрій (простір TCAM, CPU middlebox) та з'єднати ресурс який споживає дані, основуючись на обсяг трафіку який проходить через шляхи даних пристроїв.

Природне запитання – чи узагальнення формулювань на основі шляху включає ефективність. Дійсно, якщо реалізація була здійснена не на необхідному рівні, оптимізаційні проблеми виражені через шляхи, якими трафік може бути переданий, призведуть до проблем з ефективністю так як кількість шляхів зростає експоненційно з ростом розміру мережі. Ключове розуміння – поєднуючи нечасті, офлайн обрахунки з простими онлайн алгоритмами вибору шляху (наприклад найкоротші або випадково обрані шляхи), можна досягти близькі до оптимальних рішення на практиці для всіх додатків які були включені до первинного розгляду.

Більше того, SOL зазвичай набагато ефективніший ніж рішення оптимізаційних проблем які використовувались раніше для вираження вимог додатків.

SOL був реалізований як бібліотека на основі Python, що взаємодіє з ONOS та OpenDaylight. Також було створено численні прототипи SDN додатків в SOL, включаючи SIMPLE, ElasticTree, Panopticon та інші конструкції власного дизайну. SOL є рішенням з відкритим вихідним кодом, він містить модулі створені для різноманітних додатків створених в SOL, та як ONOS розширення.

Базуючись на аналізі, широкого діапазону топологій показав:

1. SOL перевершує декілька оригінальних оптимізаційних алгоритмів на порядок або більше і навіть є конкурентоспроможним з їхньою власною спеціалізованою евристикою.
2. Масштабування SOL є простішим в порівнянні з іншими інструментами управління такими як Merlin наприклад.

3. SOL істотно зменшує зусилля необхідні (рядки коду як приклад) для імплементації нових SDN додатків на порядок та більше.
4. Необов’язкові SOL розширення можуть істотно зменшити змішування маршрутів (route churn), шлхом переконфігурації з незначним впливом на оптимальність.

1. ПЕРЕДУМОВИ ТА МОТИВАЦІЯ ДЛЯ РІШЕНЬ SOL

У цьому розділі, буде описано представницькі мережеві програми/додатки які можуть отримати перевати/бути удосконалені за допомогою такого фреймворку як SOL.

Потрібно звернути увагу на необхідність ретельного формулювання та розробки алгоритму, що має місце в зусиллях які необхідні на початкових етапах розробки, а також різноманітність вимог які вони включають в себе.

1.1 Додатки/програми які можуть бути удосконаленими

1.1.1 Інженерія трафіку

Інженерія трафіку (TE) – це канонічний додаток, який був одним із перших рішень для виконання додатків для SDN мереж. Рисунок 1.1 показує приклад в якому класи трафіку C1 та C2 повинні бути перенаправлені повністю в той же час, мінімізуючи навантаження на найбільш завантажені посилання. Додаток TE приймає на вхід вимоги до трафіку (наприклад, матрицю трафіку між WAN сайтами), специфікацію класів трафіку та пріоритетів трафіку, та топологію мережі і ємність посилань (потужність зв’язків). Так визначається як/куди направити кожен клас для досягнення цілей в масштабах мережі (наприклад, мінімізація перевантаженості, зважена максимальна та мінімальна чесність (fairness)) [24,23].

Задовільнення вимог класів C1, C2 100%;

Мінімізація максимальної утилізації зв'язків;

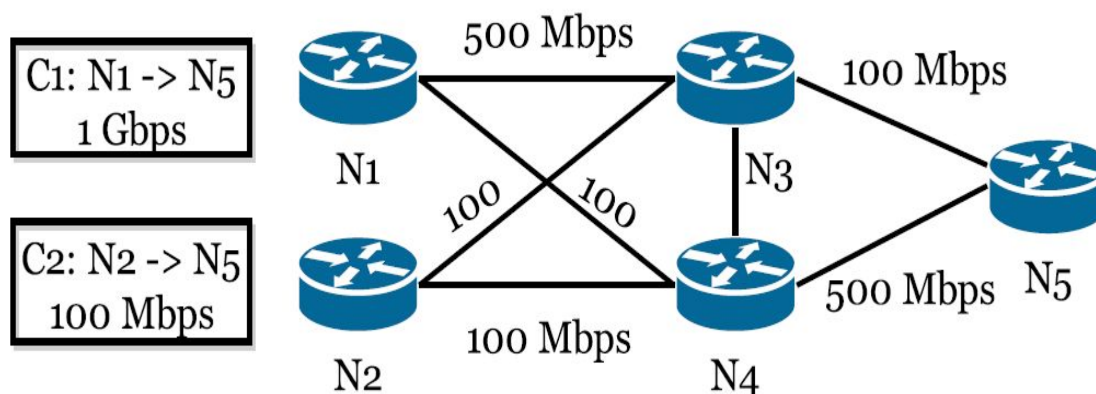


Рисунок 1.1: Додатки інжинірингу трафіку.

Виклики: Прості цілі, такі як перевантаження зв'язків, можуть бути представлені та вирішені за допомогою формулювання максимальних потоків. Однак виразність та ефективність досить швидко зменшується для більш складних цілей, таких як мін-макс чесність(fairness), для яких були проведені численні дослідження [23, 9, 24]. Коли такі формулювання як формулювання максимального потоку зазнають невдачі, розробники/дизайнери неминуче повертаються до "низькорівневих" технік таких як LP (лінійне програмування), або комбінаторні алгоритми. Ні один із вищезгаданих не є ідеальним - використовуючи/змінюючи LP рішення - досить проблематично. так як вони надають дуже низькорівневі інтерфейси. А комбінаторні алгоритми вимагають значної теоретичної експертизи. І на кінець, транслювання/переведення/трактування вихід алгоритмів в/до

реальних правил роутингу, вимагає великої уважності до встановлення правил які враховують об’єми, для того щоб по-справжньому отримати переваги в оптимізації.

1.1.2 Ланцюгування сервісів

На сьогодні. мережі покладаються/залежать від широкого спектру MB (middleboxes - мережеві пристрої) (наприклад IDS, проксі, фаєрвол) для продуктивності, безпеки та зовнішньої узгодженості/відповідності (наприклад [44]). Ціль ланцюгування сервісів - удостовіритись, що кожен клас трафіку направлений через необхідні/бажані послідовності мережевих функцій. Наприклад, на рис. 1.2, клас C1 повинен бути переданий через фаєрвол та проксі послідовно(в порядку). Така політика правил роутингу повинна бути закодована в доступних TCAM на SDN комутаторах (switch) [39]. Виходячи з того що MD (middleboxes) досить часто виконують інтенсивні обрахунки, вони можуть бути просто перевантажені, а отже оператори хотіли б збалансувати завантаження цих застосунків. Ключові вхідні дані для таких додатків - вимоги ланцюгування сервісів для різних класів трафіку їх потреб. а також доступність обчислювальні ресурси MD (middleboxes). Потім додатки встановлюють правила перенаправлення так, щоб вимоги ланцюгування сервісів були задоволені у той же час ресурси TCAM коммутатора та MD були утилізовано ефективно. Крім того, враховуючи те що велика кількість цих MD зберігають в собі стан, ці правила повинні гарантувати цілісність потоку.

Виклики: Ланцюгування сервісів представляє більш складні вимоги в порівнянні з ТЕ додатками.

Задовільнення політик маршрутизації;

Без перевантаження МВ;

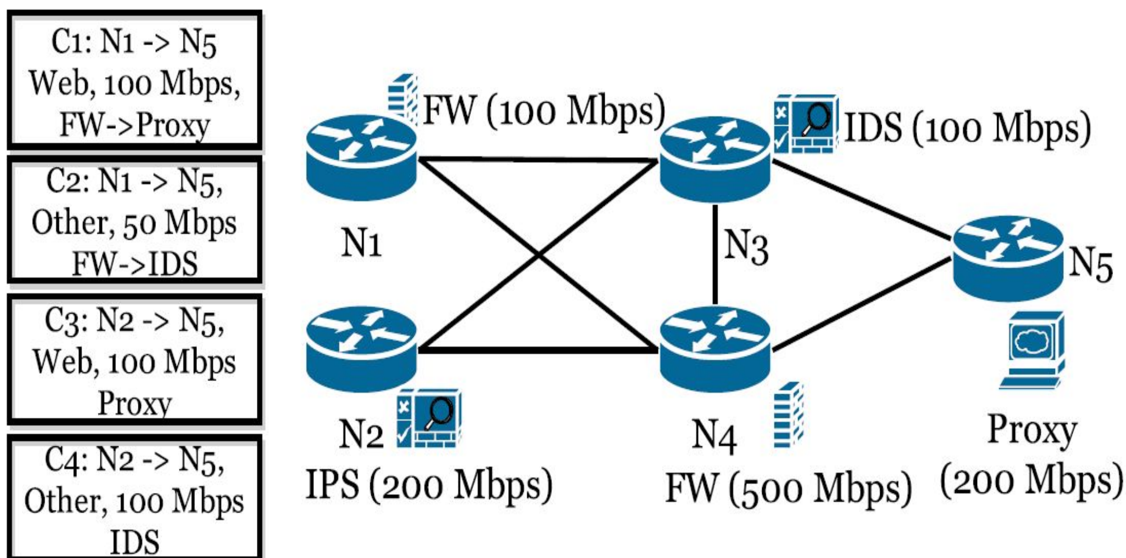


Рисунок 1.2: Додатки ланцюгування (послідовного включення/з'єднання сервісів)

Перше, моделювання споживання TCAM коммутатора надає дискретні компоненти для оптимізації, які впливають на масштабування. Друге, такі вимоги до роботи сервісів не потрапляють до області існуючих мережевих потокових абстракцій. Третє, ланцюгування сервісів підкреслює складність комбінування різних вимог; наприклад, міркування з приводу взаємодії між розподілювачем навантаження та TCAM коммутатором має свої обмеження які не є тривіальними [25]. Існуючі рішення ланцюгування сервісів послугували фундаментом для появи специфікованих евристичних [7] або

нових теоретичних розширень [8]. Також, як було вже згадано раніше, підтвердження цілісності потоку може бути досить складною задачею [21, 20].

1.1.3 Гнучке управління топологією.

SDN надає можливість модифікацій топології, що було б складно реалізувати з існуючими можливостями рівня управління. Наприклад, ElasticTree [19] та Response [46] використовує SDN для динамічного перемикання мережевих посилань та вузлів для забезпечення кращої енергоефективності дата-центрів. На рис. 1.3 ці додатки можуть вимкнути вузол N3 в періоди низького рівня використання ресурсів (утилізації), якщо класи C1 та C2 можуть бути направлені через N4 без значного впливу на кінцеву продуктивність. Реконфігурацію топології - особливо легко досягти у "багатих" топологіях з численними шляхами між кожним джерелом та пунктом призначення. Такі додатки приймають як вхідні дані - матрицю потреб (схожу до TE завдань) і потім відбувається обрахунок вузлів та зв'язків які повинні бути активними traffic-engineered routes для забезпечення продуктивності SLAs.

Вирішення які зв'язки/вузли деактивувати та направити трафік з мінімізацією витрат/енергії.

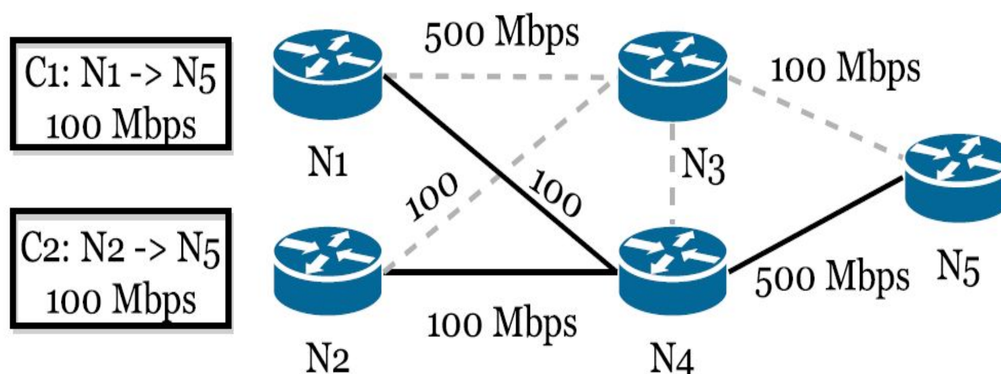


Рисунок 1.3: Додатки реконфігурації топологій

Виклики: Ввімкнення-вимкнення вимоги для комутаторів/зв'язків знову представляють дискретні обмеження, нестійкі лінійно-цілочисельні оптимізації які теоретично не злагоджені та які важко виразити використовуючи абстракції типу максимального потоку. Вирішення таких проблем вимагає значних обчислень навіть для малих топологій, а отже змушує розробників моделювати нові евристичні стратегії рішень; наприклад, ElasticTree використовує пожадливий алгоритм бінування пакетів[19].

1.1.4 Віртуалізація мережевих функцій

Попередня робота використовувала SDN можливості такі як: вивантаження або аутсорсинг мережевих функцій для задіяння кластерів або хмар [44, 16, 40]. Це є дійсно корисно для дорогих сервісів DPI [20]. Ключове

рішення тут - це вирішити скільки виконання(обчислень) на кожен шлях передати на віддалений дата-центр - наприклад на рис. 1.4, скільки трафіку C1 класу потрібно направити до дата-центру між N4 та N5 для IPS обробки, в порівнянні з обробкою цих же даних на N3. Вивантаження може підвищити затримку, яку користувач сприймає, та накласти додаткове навантаження на мережеві зв'язки.Тим паче, деякі активні функції (наприклад WAN оптимізація, або IPS) спонукають зміни до об'ємів трафіку через їхні активності. Отже, оптимізація такого вивантаження повинна враховувати скупчення що може бути присутнім в такій же мірі як вплив затримки та будь-які зміни до об'єму трафіку викликані таким аутсорсингом функцій. Подальші узагальнення враховують не тільки вивантаження MB (middlebox) сервіси, але також гнучке їх масштабування [36, 14, 34, 6] загострюють ці питання.

Розвантаження "дорогих" функцій виведення в хмарні обчислення

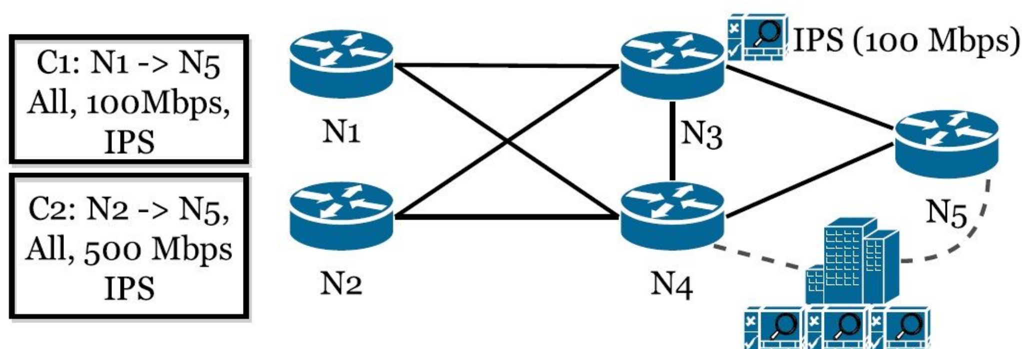


Рисунок 1.4: Вивантаження мережевих функцій.

Виклики: Таке вивантаження та гнучке масштабування можливостей представляє нові виміри(розмірення) для оптимізації які важко охопити (відслідкувати/виміряти).

Наприклад, вивантаження вимагає перенаправлення трафіку, а отже оптимізація повинні моделювати вплив на завантаження зв’язків низовинних вузлів, та TE цілей. Якщо все було виконано без належної точності та скурпульозності, це може представити на лінійні залежності з того моменту як дії назинних вузлів залежать від контролю рішень які були зроблені вище. Активні зміни до об’ємів трафіку якоюсь функцією (стиснення для зниження надлишковості, або зменшення IPS-ом) також представляють нелінійні залежності в оптимізації. І на кінець, еластичне масштабування представляє дискретний аспект до проблеми схожої до додатків модифікації топологій, далі погіршуючи проблему тягостійкості.

1.2 Мотивація для SOL

Продовжуючи дискусію яка була почата раніше (а також основуючись на досвіді), можна підсумувати декілька ключових пунктів:

- Мережеві додатки мають різні та складні оптимізаційні вимоги; наприклад, ланцюгування сервісів вимагає розмірковувати про валідність шляхів в той же час як модифікація топології потребує ввімкнення/вимкнення вузлів.

- Моделювання цих додатків потребує значних зусиль у вираженні та відладці цих проблем використовуючи низькорівневі оптимізаційні бібліотеки.

- Це може брати не тривіальну експертизу для забезпечення того, що проблеми можуть бути вирішені у відносно короткий часовий проміжок, щоб залишатись релевантним для операційної діяльності, наприклад, перерахунок TE кожні декілька хвилин, або періодично вирішувати великі лінійно-цілочисельні програми (ILPs) підтримуючи реконфігурацію топології (наприклад, [19]).

Висновки до розділу

Як можна помітити є досить велика кількість мережевих завдань/додатків які можуть бути або ж потребують певного удосконалення та оптимізації, серед яких додатки з інженерії трафіку, ланцюгування сервісів, побудови гнучкої топології мережі та віртуалізація мережевих функцій. Так як кожна задача в сукупності являється досить великою по об’єму задачею, для привнесення певних змін, тим паче основуючись на існуючих/динамічних даних, може виявитись що вирішення цих проблем окремо та спеціалізовано хоч і безсумнівно принесе результати, але саме рішення буде аж занадто не гнучке та спеціалізоване на вирішення задач для вузького кола проблем.

2.SDN OPTIMIZATION LAYER ЯК РІШЕННЯ

2.1 Загальний огляд SOL

Всеохоплююче бачення в розробці SOL підняло рівень абстракції в розробці нових SDN додатків та особливо в зменшенні деяких з недосказанностей в розробці SDN базованих оптимізацій, роблячи їх більш доступними для розгортання мережевими менеджерами. Щоб досягти цього, SOL абстрагується від низькорівневих деталей оптимізаційних рішень та SDN контролерів, дозволяючи розробникам фокусуватись на високорівневих цілях додатків (див. рис 1). SOL приймає як вхідні параметри - топологію мережі, патерни трафіку та оптимізаційні вимоги в SOL API. Потім відбувається переведення цих обмежень для оптимізації рішень таких як CPLEX або Gurobi. І на кінець, SOL інтерфейси з існуючими платформами управління SDN, такі як ONOS, для встановлення правил форвардингу на SDN комутаторах. SOL не вимагає модифікацій до існуючих компонентів рівнів управління або рівня даних мережі. Бачення для SOL на відміну від стану справ на сьогоднішній день, в яких розробник стикається з програмуванням нової оптимізації SDN або безпосередньо, для узагальнених та низькорівневих оптимізаційних рішень, таких як CPLEX, або використовуючи евристичні алгоритми розроблені вручну, після чого, необхідне трактування змінних рішення в змінні конфігурацій пристрою для оптимізації.

Абстракція шляху: Для SOL, що бути корисним та надійним, нам необхідно уніфікувати абстракцію яка може охопити вимоги різноманітних класів оптимізаційних додатків SDN мереж описаних в попередній секції. SOL побудований з використанням шляхів через мережу як основна абстракція для вираження проблем оптимізації мережі. Це суперечить тому, скільки оптимізацій зформульовано в літературі - використовуючи більш стандартні крайово-центричні підходи[1]. З досвіду відомо, проте, крайово-центричний підхід змушує складність, коли представлений з додатковими вимогами, особливо при спробах охопити властивості шляху [29, 19].

На відміну, формулювання на основі шляху, охоплюють ці вимоги більш природним шляхом. Наприклад, значна частина складності в моделюванні ланцюгування сервісів або вивантаження мережевих функцій з розділу 1 полягає в захопленні властивостей шляхів, які повинні бути задовільнені. З абстракціями на основі шляху, ми можемо досить просто визначити умови які визначають валідність шляхів - наприклад, ті що включають певні шляхи

або ті що уникають певних вузлів (очікуючи що вузол вийде з ладу). Також, можна змоделювати використання ресурсів на основі шляху досить просто. Наприклад використання TCAM простору в комутаторі відповідає шляху через який проходить трафік, перетинаючи цей комутатор (а отже правило для розміщення даного шляху). Без абстракції шляху, моделювання таких обмежень є непростою задачею (приклад, [39]). І на кінець, вираження обмежень на вузлах та гранях не підвищує складність додатково, в порівнянні з підходом який зосереджується на гранях.

Масштабування: У сценаріях чистої потокової маршрутизації, формуляції основані на гранях визнають просты алгоритми, що гарантують поліноміальний час виконання. Формулювання на основі шляху, на відміну, часто відміняються через їхню очевидну неефективність - зрештою, в найгіршому випадку, кількість шляхів в мережі, є експоненціальним в розмірі мережі - або через складність алгоритмів для вирішення формулювань на основі шляхів (декомпозиція, генерування колонок, і т.д. [1]). Однак, в багатьох практичних випадках, кількість валідних шляхів (що визначено додатком) більш за все буде значно меншою. Крім того, мульти-шляхова маршрутизація може надати тільки стільки мережевої різноманітності до того як цінність стане незначною[30]. Тому, набір шляхів який повинен бути розглянутий не є великим.

SOL виконує офлайн етап генерування шляху для визначення валідних шляхів (етап 1 на рисунку 2.1). Базуючись на тому що для більшості додатків, набір валідних шляхів здебільшого статичний і не потребує переобрахування при кожному перезапуску оптимізації, можна припустити що цей етап/крок не буде виконуватись часто. Далі, SOL вибирає піднабір цих шляхів (етап/крок 2) використовуючи стратегію вибору (див. розділ 2.3) та запускає оптимізацію тільки для обраних шляхів як вхідні параметри (крок 3) для удостовірення того, що оптимізація виконається швидко. Як показано в розділі 3.3, що ця стратегія все ще дозволяє включення достатньо великої кількості шляхів для оптимізації для звуження до (близько) оптимального значення. Тому, поки ефективність оптимізації яка основується на шляху є валідним теоретичним концептом, на практиці буде показано, що є практичні евристики які призначені для цього питання.

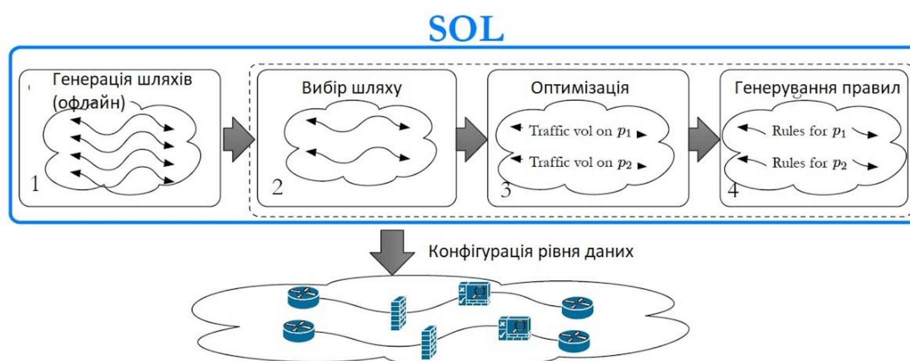


Рисунок 2.1: SOL архітектура, огляд робочого процесу.

Генерування конфігурації пристрою: SOL переводить змінні рішення з SOL оптимізації до конфігурації мережевих пристроїв для реалізації потрібного потоку маршрутизації (крок 4 на рис 6). Алгоритм утилізований в SOL для виконання цього переведення і базується на роботі [47, 20]. Однак, через те що оптимізація основується на шляху, алгоритми є більш прямолінійними і вимагають менше кроків.

`nodes`: Set of all nodes, part of the topology

`links`: Set of all links, part of the topology

`classes`: Set of all traffic classes

`paths`(c): Paths available for class $c \in$ `classes`; output by path-selection stage (§5)

Рисунок 2.2: Вхідні дані для мережі

	Var.	Description
Decision	$x_{c,p}$	Fraction of class-c flows allocated to path $p \in \texttt{paths}(c)$; non-integer
	b_p	Is path p used; binary
	b_v	Is node v used; binary
	b_l	Is link l used; binary
	$capvar_v^r$	Capacity allocated for resource r at node v; non-integer
Derived	a_c	Fraction of c's "demand" routed; non-integer
	$load_l^r$	Amount of resource r consumed by flows routed over link l; non-integer
	$load_v^r$	Amount of resource r consumed by flows routed via node v; non-integer

Рисунок 2.3: Змінний інтервал для оптимізації

2.2 Деталізований дизайн SOL

В цій секції. буде представлено деталізований дизайн SOL. Фокус буде на високорівневому API, який розробники SDN додатків будуть використовувати щоб виразити додаток через SOL, та вплив цих API викликів на внутрішнє представлення оптимізаційної проблеми SOL. Проте потрібно відмітити, що розробники "мислять" в термінах високорівневого API, аніж низькорівневих деталей роботи зі змінними рівня рішення, як шляхи ідентифікуються і т.д.

Розробник починає нову оптимізацію в SOL інстанціюючи об'єкт opt через getOptimization функцію і потім будує оптимізацію використовуючи шаблони обмежень, які будуть детальніше розглянуті нижче.

2.2.1 Прелімінарії

Вхідні дані: Є два типи базових вхідних даних які розробник повинен надати для оптимізації будь-якої мережі. Перший, топологія мережі - обов’язковий параметр, зазначений у вигляді графу з вузлами та зв’язками. Він також містить метадані для типів вузлів/зв’язків або властивостей; наприклад, вузол може мати виділену функцію типу “комутатор” або “MB”. Другий, SOL потребує специфікацію класів трафіку, де кожен клас C має асоційовані вхідні та вихідні вузли та певний очікуваний об’єм трафіку. Кожен клас може(опціонально) бути асоційований зі специфікацією “обробки” яка є необхідною для трафіку в цьому класі, наприклад, ланцюгування сервісів. І на кінець, для кожного класу трафіку C є асоційований набір Paths (C) доступний для маршрутизації потоків в класі C; Paths (C) є вихідним результатом для кроку попередньої обробки відбору шляху, описаного в розділі 2.3.

Внутрішні змінні: SOL всередині визначає набір змінних які узагальнено на рисунку 2.3. Реітеруючи/повторюючи те що розробнику не потрібно розмірковувати про ці змінні , він лише повинен використовувати високорівневу модель, як було обговорено раніше.

Є два головних види цих змінних:

- Змінні рішення які ідентифікують ключові оптимізаційні рішення контролю. Найфундаментальнішою змінною рішенняє Xc,p, яка зберігає рішення маршрутизації трафіку та позначає частини потоку для класу трафіку C що шлях переноситься.
-

$$p \in paths(c) \quad (2.1)$$

Ця змінна є центральною для різних типів додатків управління ресурсами, що ми побачимо пізніше. Для захоплення вимог топології (наприклад, розділ 1.1.3), було представлено три бінарні змінні рішення bp, bv, and bl що позначають для кожного шляху чи зв'язку(відповідно) чи є вони ввімкненими (= 1) чи вимкненими (= 0). Змінна capvar(r, v) назначений SOL ресурс r до вузла v.

Похідні змінні - функції визначені над змінними рішення що слугують як зручні скорочення. Ac позначає кінцеву частину потоку для класу C який передається через всі шляхи. Змінні завантаження, load(r,v) та load(v,l) моделюють споживання ресурсів r на вузлі v, та зв'язку l, відповідно.

2.2.2 Маршрутизаційні вимоги

Обмеження маршрутизації контролює назначення потоку в мережі. addAllocateFlowConstraint створює необхідну структуру для маршрутизації трафіку через набір шляхів для кожного класу трафіку. Деякі мережеві додатки намагаються задовольнити максимально можливу кількість вимог потоку (наприклад, я max-flow) коли інші (наприклад, TE) хочуть "наситити" вимоги. Наприклад, розробник TE додатку (розділ 1.1.1) хотів би направити весь трафік через мережу, а отже для цього буде створене новий шаблон високорівневого обмеження маршрутизації до пустого opt об'єкту:

```
opt.addAllocateFlowConstraint()
opt.addRouteAllConstraint()
```

На відміну, в простому max-flow потрібно буде тільки addAllocateFlowConstraint так як в цьому випадку немає вимог по насиченню вимог(потреб).

Обмеження addEnforceSinglePath(C) вносить такі обмеження як назначення єдиного шляху для перенесення класу

$$c \in C \quad (2.2)$$

попереджуючи мульти-шляхову маршрутизацію та ділення потоку.

Внутрішні особливості: addAllocateFlowConstraint запевнює що сумарний потік трафіку через всі обрані шляхи для класу C відповідають змінній Ac

$$\forall c \in classes : \sum p \in paths(c) \quad (2.3)$$

$$xc,p = ac$$

Group	Function	Description

Routing ($C \subseteq$ `classes`)	`addAllocateFlowConstraint`
	`addRouteAllConstraint`
	`addEnforceSinglePath` (C)
		Allocate flow in the network
		Route all traffic demands
		For each $c \in C$, at most one $p \in$ `paths`(c) is enabled.

Capacities	`addLinkCapacityConstraint` (r, *lnCap*, `linkCapFn`) `addNodeCapacityConstraint` (r, *ndCap*, `nodeCapFn`) `addNodeCapacityPerPathConstraint` (r, *ndCap*, `nodeCapFn`) `addCapacityBudgetConstraint` (r, N, *totCap*)	If l is in *lnCap*, then limit utilization of link resource r on link l to *lnCap*$[l]$. If v is in *ndCap*, then limit utilization of node resource r on node v to *ndCap*$[v]$. If v is in *ndCap*, then limit utilization of node resource r on node v by enabled paths to *ndCap*$[v]$. Limit total type-r resources allocated to nodes in $N \subseteq$ `nodes` to *totCap*. Used when SOL is allocating capacities.
Topology control ($C \subseteq$ `classes`)	`addRequireAllNodesConstraint` (C) `addRequireSomeNodesConstraint` (C) `addRequireAllEdgesConstraint` (C) `addPathDisableConstraint` (C) `addBudgetConstraint` (`nodeBudgetFn`, k)	For each $c \in C$ and each $p \in$ `paths`(c), p can be enabled iff all nodes on p are enabled. For each $c \in C$ and each $p \in$ `paths`(c), p can be enabled iff some node on p is enabled. For each $c \in C$ and each $p \in$ `paths`(c), p can be enabled iff all links on p are enabled. For each $c \in C$ and each $p \in$ `paths`(c), p can carry traffic only if it is enabled. Total cost of enabled nodes, as computed using `nodeBudgetFn`, is at most k.
Objective	`setPredefinedObjective` (*name*)	Set one of the predefined functions as the objective (see Fig. 11).

Рисунок 2.4: Вибрані шаблонні функції обмежень для побудови оптимізації; Див. Рисунок 2.5 для linkCapFn, nodeCapFn, та nodeBudgetFn

Схоже, addAllRouteConstraint означає,

$$\forall c \in \text{classes} : ac = 1 \quad (2.4)$$

через просторові обмеження, формальна основа для addEnforceSinglePath не надається.

2.2.3 Обмеження ресурсної ємності

Як ми бачимо в першому розділі, при SDN оптимізації необхідно мати справу з різними обмеженнями по ємності для мережевих ресурсів, таких як пропускна здатність зв'язків, правила комутації, MB CPU та пам'ять.

SOL дозволяє користувачам писати спеціальну логіку для менеджменту ресурсів визначаюченням декількох функцій "вартості" зображених на рис 10. Ці функції прописують як обраховувати вартість маршрутизації трафіку через певні зв'язки, вузли. шляхи. SOL надає імплементацію за замовчуванням для спільних завдань, але дозволяє користувачам вказувати їхню власну логіку також, як буде показано в (розділі 3.1).

Ці функції вартості можуть потім бути передані в шаблони обмежень. Наприклад, щоб додати обмеження що обмежує використання зв'язку, користувач може викликати шаблонну функцію addLinkCapacityConstraint з ресурсом в який ми хочемо внести обмеження (наприклад, пропускна здатність), а також колекцію відношень зв'язків та їх ємностей, а також як необов'язковим параметром, спеціалізований linkCapFn для обрахунку вартості трафіку на зв'язку.

Цей вираз індикує що пропускна здатність не повинна перевищувати 10 Мбіт/сек для зв'язків 1-2 та 2-3. потрібно відмітити, що функції по замовчуванню створені чисто для ілюстрації; розробник може написати свої власні linkCapFn (див. рис. 2.5).

addNodeCapacityPerPathConstraint генерує обмеження на вузлах які не залежать від трафіку, але залежать від шляху маршрутизації. Тому, вартість маршрутизації для вузла не залежить на об'єм та тип трафіку який

передається; вона залежить від шляху та його властивостей. Найкращий приклад такого використання - це облік для обмеженого місця для правил на комутаторі (наприклад, розділ 1.1.2). Якщо шлях активний, правило повинно бути встановлене на кожному комутаторі, для підтримки шляху.

`linkCapFn`(l, c, p, r): Amount of resource type r consumed if all class-c traffic is allocated to path $p \ni l$ for link l

`nodeCapFn`(v, c, p, r): Amount of resource r consumed if all class-c traffic is allocated to path $p \ni v$ for node v

`nodeBudgetFn`(v): Cost of using node v; required with `addBudgetConstraint`

`routingCostFn`(p): Cost of routing along path p; required with `minRoutingCost`

`predicate`(p): Determine whether any given path is valid by returning `True` or `False`

Рисунок 2.5: Функції що піддаються кастомізації (модифікації)

Внутрішні аспекти: addLinkCapacityConstraint та addNodeCapacityConstraint покладаються на linkCapFn та nodeCapFn, відповідно, для обрахунку вартості використання певного ресурсу на зв’язку або вузлі, якщо весь трафік класу C був переданий через цей елемент. Внутрішньо, завантаження помножено на Xc,p змінну для отримання точного значення завантаження, потім завантаження обмежується параметром який задається користувачем lnCap(ndCap), який являє собою відповідність

зв’язків(вузлів) до ємності для заданого ресурсу r. (Схожий вираз для ємності вузлів не приводився)

addNodeCapacityPerPathConstraint функція трішки відрізняється, так як вона залежить від ввімкнений шляхів.

2.2.4 Активаційні обмеження вузлів/зв’язків

Наступний набір обмежень, коли використовується, дозволяє розробникам логічно моделювати процес “вмикання” та “вимикання” вузлів, зв’язків, та шляхів; наприклад, для управління енергією або іншими витратами (наприклад, розділ 1.1.3). Можна ідентифікувати дві можливі моделі взаємодії між цими модифікаторами топологій, та оптимізаційний розробник може обирати ту модель, яка більш підходить для конкретного випадку, їхнього контексту.

1) addRequireAllNodesConstraint захоплює властивість що деактивує вузол, що в свою чергу деактивує всі шляхи які містять цей вузол.
2) addRequireSomeNodesConstraint захоплює властивість що активує вузол, дозволяючи будь-якому шляху який містить даний вузол, бути активним також.

Більш пізня версія підходить коли, наприклад, вузол може призупиняти маршрутизований трафік навіть якщо інша (MB) функціональність деактивована, тому шлях який включає цей вузол є потенційно корисним так як надає MB функціональність, якщо хоча б один з його вузлів активний. Також наявне подібне шаблонне обмеження для зв’язків, але даний варіант розглядатися не буде. Третє шаблонне обмеження, addPathDisableConstraint,

обмежує шлях для перенесення трафіку тільки якщо він є активним. Наприклад, розробник намагається реалізувати додаток з розділу 1.1.3, може змоделювати вимоги для вимкнення вузлів дата-центру додаванням addRequireAllNodesConstraint та addPathDisableConstraint шаблонів:

opt.addRequireAllNodesConstraint (trafficClasses)

opt.addPathDisableConstraint (trafficClasses)

Інші міркування з приводу ефективності можуть привносити(нав'язувати) бюджет на кількість активних вузлів, для моделювання обмежень на кінцеве споживання енергії комутаторами та МВ, вартість та бюджет встановлення/оновлення певного комутатору, абощо. Все це виконується через addBudgetConstraint шаблонну функцію.

Внутрішні особливості: В середині, ці шаблони модифікації топологій, реалізовані використовуючи бінарні змінні, які були представлені раніше. Конкретно, вищезгадані формулювання можуть бути формалізовані таким чином:

$$\forall p \in paths(c) : \quad (2.5)$$

$$\text{addRequireAllNodesConstraint} \quad \forall v \in p : b_p \leq b_v \quad (2.6)$$

$$\text{addRequireSomeNodesConstraint} \quad b_p \leq \sum_{v \in p} b_v \quad (2.7)$$

$$\text{addPathDisableConstraint} \quad x_{c,p} \leq b_p \quad (2.8)$$

Безумовно, схожі обмеження можна сконструювати і для зв'язків. Потрібно відмітити що, addPathDisableConstraint є критично важливим для

коректності оптимізації, щоб трафік не передавався через деактивовані шляхи. Формальні рівняння не будуть приведені для addBudgetConstraint..

2.2.5 Специфікація мережевих цілей

Ціль SDN додатків - оптимізація цілей в масштабі мережі, наприклад максимізація мережевої пропускної здатності, балансування завантаженості, або мінімізація сумарного сліду трафіку. Рисунок 2.6 містить спільні цільові функції, зображуючи додаток який був розглянутий в розділі 1. Наприклад, розробник TE додатку, може хотіти реалізувати таку ціль, як мінімізація максимального завантаження зв’язків а отже, додати наступний фрагмент коду

```
opt.setPredefinedObjective (minMaxLinkLoad,’bandwidth’)
```

Інші оптимізації (наприклад, розділ 1.1.4) можуть потребувати мінімізацію загальної вартості та включти minRoutingCost ціль. Ця ціль є параметризованою з routingCostFn(p); іншими словами, розробники можуть додати свої власні метрики витрат, такі як - кількість “стрибків” або ваги зв’язків. Як показано, також було надано діапазон натуральних шаблонів для балансування завантаженості. SOL також розкриває/надає низькорівневий API для специфікації інших складних функціональних цілей , які описані в Додатку Б.

2.3 Генерація та вибір шляху.

Враховуючи ці шаблонні обмеження, залишається питання, як заповнити шлях set paths(c) для кожного класу трафіку C для відповідності цим двом вимогам.

maxAllFlow	maximize $\sum_{c \in \text{classes}} a_c$
minMaxNodeLoad (r)	minimize $\max_{v \in \text{nodes}} load^r_v$
minMaxLinkLoad (r)	minimize $\max_{l \in \text{links}} load^r_l$
minRoutingCost	$\sum_{c,p} \text{routingCostFn}(p) \times x_{c,p}$

Рисунок 2.6: Функції спільних цілей

Перша, кожен повинен задовольняти бажану політику

$$p \in \text{paths}(c) \quad (2.9)$$

специфікації для класу C. Друга, paths(c) повинен містити шляхи для кожного класу C, що робить формулювання можливим для трактування та видає результати близькі до оптимальних. Далі буде детальніше описано, як адресувати кожне із питань.

Генерування: Перше, для заповнення шляхів, SOL виконує офлайн перерахунок всіх простих (тобто, не зациклених) шляхів для кожного класу. Маючи цей набір, потрібно відфільтрувати шляхи, які не задовольняють

предикати користувача predicate, тобто, де

$$predicate(p) = True \quad (2.10)$$

тільки якщо p валідний шлях (можна робити узагальнення, щоб дозволити використання різних предикатів для одного класу).

На практиці, предикати реалізується як гнучкі Python функції, а не з використанням понять валідності шляху що б обмежувало нас (наприклад, регулярні вирази як в попередній роботі [45]).

Використання цих предикатів надає користувачу гнучкість для захоплення діапазону можливих вимог. Приклади включають шляхове примусове виконання (примушуючи проходження трафіку через серію MB в певному порядку); Нав’язуючи надлишкову обробку (наприклад, через численні IDS, у випадку якщо один відмовив); та обмежує мережеву затримку вимагаючи коротших шляхів.

Вибірка: Використання всіх валідних шляхів для класу, може бути неефективним, так як кількість шляхів зростає експоненційно з розміром мережі, означаючи що LP/ILP які SOL генерує, досить швидко стануть занадто великими для вирішення у доцільний часовий проміжок. SOL отже надає алгоритми для вибору шляхів, які вибирають підмножину валідних шляхів (кількість шляхів позначено як selectNumber) яка більш за все, все ще на практиці видає результати близькі до оптимальних. Конкретно, два природних методи працюють добре для спектру додатків які були розглянуті: (1) найкоротший шлях для додатків чутливих до затримки (selectStrategy = shortest), або (2) випадкові шляхи для додатків які включають

балансування завантаженості (selectStrategy = random). SOL проявляє гнучкість в інкорпоруванні інші стратегії вибору, наприклад, вибір шляхів з мінімальним перевикористанням вузлів для відмовостійкості. Також було визначено, що random працює успішно для багатьох додатків які вимагають балансування завантаженості. Тому можна зробити припущення, що причиною для цього є те, що вибір випадкових шляхів в достатньо багатій топології виробляє високий рівень роззосередженості серед обраних шляхів, виробляючи достатній рівень свободи для балансування завантаженості.

API для розробників: Розробник може вказати предикат шляху або стратегію вибору, але він не вимушений використовувати низькорівневі деталі генерування та вибору. SOL також надає API для розробників для додавання їхньої власної логіки для генерування та вибору. Дані деталі не будуть розглядатися через обмеження місця.

Висновки до розділу

Основуючись на тому, що при розробці додатків для SDN мереж розробники фокусуються не на низькорівневих деталях реалізації API, а на інтерфейсах/контракту який цей же API надає, потрібно зазначити основні моменти які необхідно знати:

- Топологія мережі та специфікація класів є вхідними параметрами.
- Змінні рішення ідентифікують ключові оптимізаційні рішення контролю.

- Похідні змінні - функції визначені над змінними рішення що слугують як скорочення.
- Основні типів обмежень які застосовуються для спрощення моделювання: активаційні обмеження вузлів/зв’язків, обмеженн ресурсної ємності та маршрутизаційні вимоги.

3. ПРАКТИЧНА РЕАЛІЗАЦІЯ SOL РІШЕННЯ

3.1 Загальний приклад

Далі, буде показано приклад повного циклу (E2E) для підкреслення простоти використання SOL API для написання існуючих та нових мережевих оптимізацій SDN. Ці приклади - це фрагменти Python коду, який може бути виконаний, а не просто псевдокод. В порівняння, код є високорівневим та є більш читабельним в порівнянні з еквівалентом CPLEX коду, так як немає необхідності мати справу з великою кількістю змінних та обмежень, які лежать в основі.

Ланцюгування сервісів (розділ 1.1.2): Як конкретний приклад ланцюгування сервісів, було реалізовано SIMPLE. SIMPLE включає наступні вимоги: перенаправляти весь трафік через мережу, таким чином вносячи ланцюг сервісів (наприклад, фаєрвол який викликається після IDS), політики для всього трафіку, балансування завантаженості поміж MB, та виконання даних дій враховуючи CPU, TCAM та вимоги пропускної здатності. Рисунок 5 (Додаток В), показує як SIMPLE оптимізація може бути написана в ~25 рядків коду. Цей перелік припускає, що топологія та класи трафіку були встановлені, в topo та trafficClasses об’єкти, відповідно. Перша частина рисунку показує визначення функції та кроки генерування шляху, які зазвичай були б виконані одноразово як крок попереднього обчислення. Почнемо з визначення предикати шляху (рядок 1) для базового примусового виконання через MB використовуючи надані SOL функції з порядком MB.

Наступні декілька рядків (2-4 рядки) показують спеціальну функцію ємності вузлів для нормалізації завантаження CPU поміж 0 та 1. Так виконується обрахунок вартості обробки для класу трафіку (кількість часових потоків вартості CPU) нормалізований базуючись на поточній ємності вузлів. Схоже, TCAM функція ємності захоплює що кожен шлях споживає одне правило для комутатора (рядок 7), Користувач отримує оптимізаційний об’єкти (рядок 10), та генерує шляхи (рядок 11), отримуючи об’єкт шляхів для кожного класу “paths per traffic class” (pptc). Алгоритм генерування шляхів параметризований спеціальним predicate_SIMPLE, лімітом на довжину шляху, в 10 вузлів, та обмеженням зв’язків для класу - в 100 одиниць. Також алгоритм виконує оцінку кожного можливого використання двох MB на шляху маршрутизації для включення роздільного шляху у вихідному результаті.

Рядки які залишились, показують що було б виконано коли нова алокація трафіку до шляху була б необхідною. Рядки 13 вибирають 5 випадкових шляхів на клас трафіку; рядки 14 - 20 додають маршрутизаційні та ємнісні обмеження. Було використано стандартні функції ємності зв’язків для обмежень пропускної здатності, та власна функція для ємності CPU та TCAM. Тому що функція ємності CPU нормалізує завантаження, ємність для кожного вузла тепер 1 (рядок 19). Програма вибирає визначені заздалегідь цілі для мінімізації завантаженості CPU (рядок 21) та викрикає “вирішувача” (рядок 22). Зрештою, програма отримує результат та взаємодіє з SDN контролером для автоматичного встановлення правил (рядок 26).

ElasticTree [19]: Для цього випадку покажемо найбільш значну та важливу різницю між Elastic-Tree та SIMPLE. Так як в цьому випадку ми не маємо

вимог які стосуються шляхів, тому nullPredicate використаний для генерування шляхів. Було використано бінарні змінні зв'язків (див. рядок 1 нижче) та активаційні обмеження вузлів/зв'язків (рядки 2-4). Зрештою, було використано низькорівневі API (див. додаток В) для визначення споживання енергії для комутаторів та зв'язків (рядки 5, 6 де "opt.bn(node)" та "opt.be(u,v)" отримані назви змінних Bnode and B(u,v) з рисунку 2.3, відповідно) та використані ці змінні для визначення спеціальних функцій (рядок 7). В додатку Б приведені інші приклади для більш складних додатків.

3.2. Детальний опис реалізації

Інтерфейс розробника: На даний момент надано Python API для SDN оптимізації, що є розширеною версією інтерфейсу описаного в розділі 2.
Вирішувачі які були викликані: Було використано CPLEX (через його існуючий Python API) як вирішувач який лежить в основі. Цей вибір в більшості відображає обізнаність з інструментом, та CPLEX може бути замінений іншими вирішувачами, такими як Gurobi. SOL пропонує API для використання можливості вирішувача для використання обчислювальних рішень які були реалізовані та інкрементально знаходити нові рішення. Цей підхід зазвичай швидший в порівнянні з розробкою з чистого листа, і тому корисний для швидкої реконфігурації. SOL також дозволяє жорстке обмеження часу виконання оптимізації, хоч і впливаючи на оптимальність рішення.

Генерування шляхів: Генерування шляхів по суті паралізує процес; для уникнення цього, було вирішено запускати окремий Python процесс для

різних класів трафіку. На даний момент підтримується два алгоритми вибору шляху: випадковий та найкоротший. Досить просто додати більше алгоритмів при появі нових додатків.

Інтерфейси генерування та контролю правил: Було реалізовано додатки для ONOS [5] та використано спеціалізоване REST API щоб дозволити віддалене групове встановлення релевантних правил. Було згенеровано правила що базуються на оптимізації вихідних даних, використовуючи поділ мережевого префіксу для реалізації частинних відповідальностей представлених в Xc,p змінних. Цей крок схожий на роботу що була виконана заздалегідь зі складання відповідності часткових обробок та направлення відповідальностей в мережевий потік (наприклад [47, 20]), і так далі. З ONOS, є можливість досягти "path intents"[5] наміри шляхів: в той же час дозволяючи полегшти інтеграції.

Мінімізація реконфігураційних змін: мережі знаходяться в постійному русі протягом реконфігурації з потенційними наслідками на виконання та консистентність(цілісність), а отже, бажано мінімізувати непотрібні конфігураційні зміни. SOL підтримує обмеження які стримують(або мінімізують) догічну відстань між попереднім та новим рішенням з ціллю зменшення кількості потоків яким буде назначена нова маршрутизація. Таким чином SOL підтримує вибір шляхів, який надає перевагу шляхам котрі були обрані раніше.

3.3 Оцінка та порівняння

В цій секції буде показано що SOL

- працює успішно з ONOS контролерами;
- обчислює оптимальні рішення для опублікованих порядків додатків в рази швидше в порівнянні з їх оригінальною оптимізацією; дозволяючи мінімізувати змішування трафіку.
- швидший або надає ширшу функціональність аніж схожі сучасні рішення;
- Значно зменшує зусилля які потрібно витрачати на розробку в порівнянні з ручною розробкою оптимізаційних додатків,
- добре піддається масштабуванню, тому що він обраховує близькі до оптимальних рішень використовуючи декілька шляхів на кожен клас трафіку.

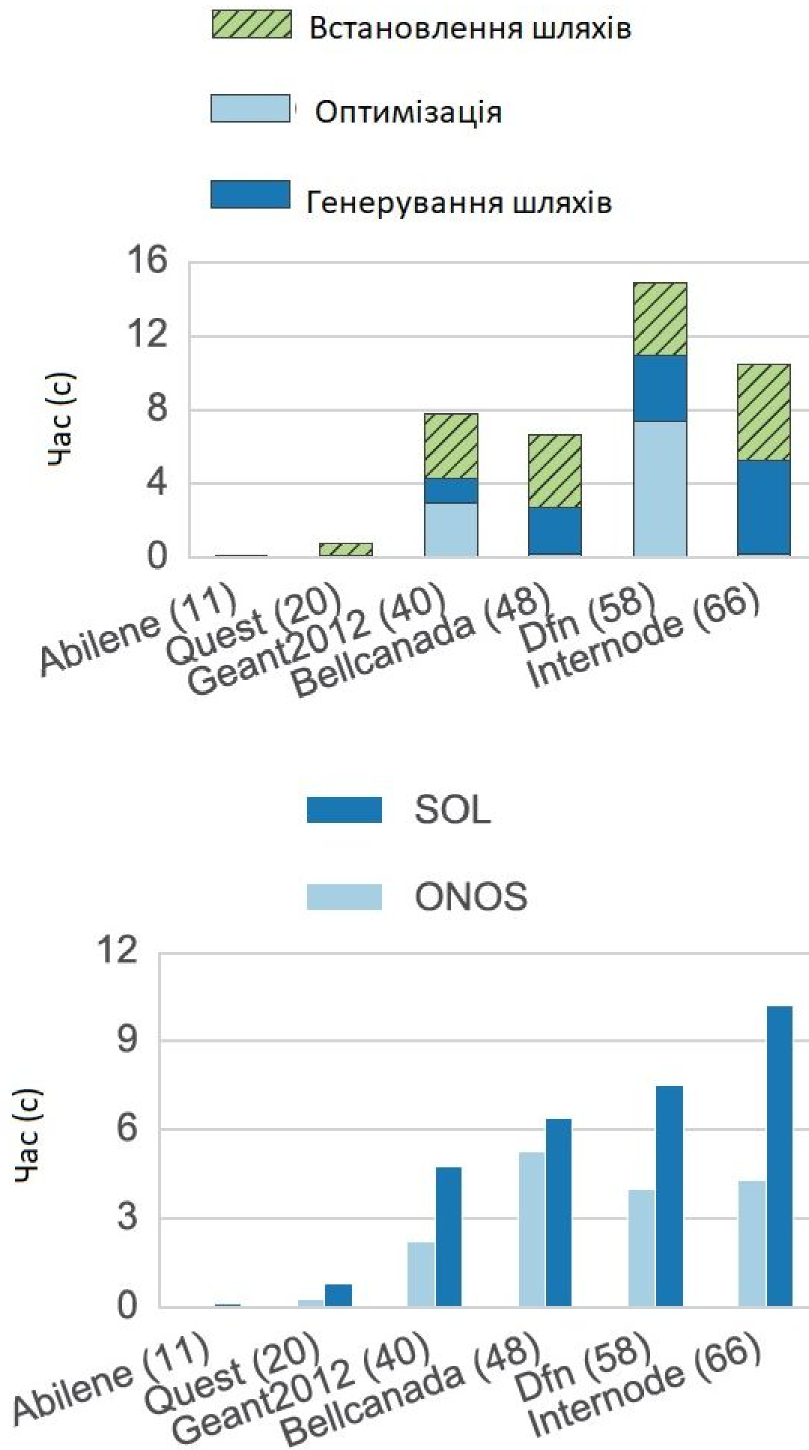


Рисунок 3.1: Оцінка Розгортання з використанняи ONOS контролеру

Налаштування: Було оцінено ефект від використання SOL для реалізації трьох існуючих SDN додатків: ElasticTree [19], SIMPLE, та Panopticon [29]. Кожен додаток, було реалізовано відповідно до оригінального формулювання яке представлене в попередній роботі, або отриманий оригінальний вихідний код. Посилання йдуть до цих, ніби “оригінальних” формулювань (або рішень). Було обрано топології різних розмірів з набору TopologyZoo; індикуючи топологію, в основному включався набір вузлів в топології в дужках, наприклад “Abilene (11)” для 11-вузлової топології Abilene. Для ElasticTree, також було побудовано FatTree топології різних розмірів [2]. Генерація матриць трафіку відбувалась систематично використовуючи уніфіковану матрицю трафіку для FatTree мереж та модель на основі гравітації (gravity-based[43]) для TopologyZoo топологій.

Використовувались випадкові вибіркові значення з логнормального розподілення як заповнення для гравітаційної моделі. Якщо не вказано протилежне, використовувались шляхів для кожного класу трафіку при запуску SOL. Всі приклади нижче посилаються до обрахунків на комп’ютерах із ядрами в 2.4 ГГц та 128 Гб оперативної пам’яті. Для орієнтовної оцінки розгортання, використовувався стандартний Mininet[31] віртуальна машина для емуляції топологій.

3.3.1 Орієнтовне оцінювання розгортання

Було налаштовано різноманітні емульовані мережі використовуючи Mininet та ONOS. Був виміряний час для SOL для виконання оптимізації для

цілей інжинірингу трафіку, обрахунку та встановлення мережевих маршрутів. Рисунок 3.1 a відображає час для виконання кожного кроку. SOL демонструє низький оптимізаційний час та час генерації маршрутів, роблячи встановлення маршрутів частиною яка займає найбільше часу при процесі конфігурації. Це вузьке місце викликане: кількістю правил які повинні бути встановлені, а також платформою контролеру. Рисунок 3.1 b демонструє час для генерації та встановлення маршрутів для додатків інжинірингу трафіку з використанням SOL, у контрасті до встановлення найкоротших шляхів з використанням існуючих методів в ONOS. Різниця є незначною, та існує через додатковий оптимізаційний час та через множину шляхів на кожну пару джерело-призначення у випадку SOL.

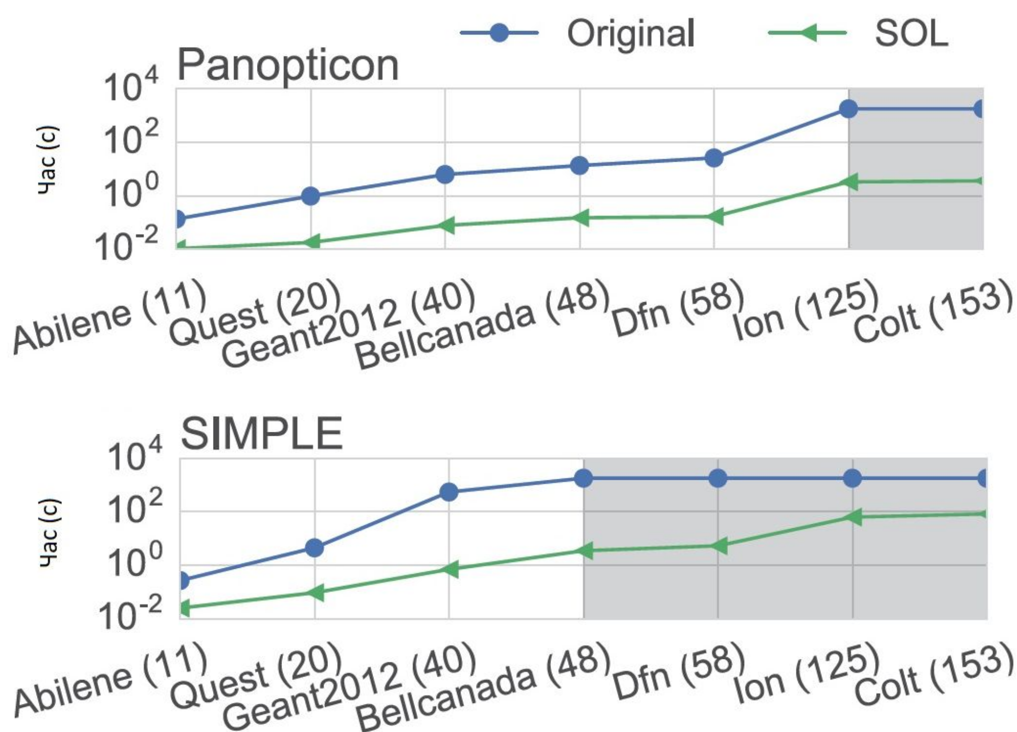


Рисунок 3.2: Оптимізація часу виконання в SOL та оригінальні формулювання; Сірі регіони показують де оригінальні формулювання не можуть бути вирішені в діапазоні 30 хвилин.

3.3.2 Оптимальність та Масштабованість.

Порівнюючи оптимальність: Далі, було розглянуто/оцінено наскільки добре результати SOL відповідають оригінальним рішенням, які оптимальні за визначенням. В усіх випадках крім ElasticTree, SOL знаходив оптимальне рішення. Через складність ElasticTree оптимізації, SOL мав 10%

відставання оптимальності: відносна помилка в дійсному значенні обчисленому SOL (тобто, відносно до дійсно оптимального значення).

Час рішень SOL принаймні на один порядок швидші ніж рішення оригінальних формулювань, та часто на два і навіть три порядки швидші. Рисунок 3.2 демонструє час виконання для знаходження оригінального рішення. Час виконання обмежувався 30 хвилинами (1800 секунд), після чого виконання скасовувалось/обривалося. Декілька оригінальних формулювань не виконались у відведений час, такі як SIMPLE для топологій Bellcanada та більших, та Panopticon для Ion та більших. Топології для яких оригінальні рішення не могли бути знайдені позначені в “сірій” частині на рисунку 3.2.

Порівняння до спеціалізованих евристик: Було знайдено що SOL працює досить добре, навіть в порівнянні до спеціалізованих евристик. Особливо, при порівнянні продуктивності SOL з спеціальною евристикою для SIMPLE, отриманою від її авторів. Час виконання SOL є порівнюваним до часу виконання алгоритму евристики SIMPLE, з проміжком в продуктивності максимум в 3 секунди на найбільших топологіях які розглядалися (для 58 вузлів, яка називалася Dfn топологія). Також передбачається, що перевага від того що не потрібно моделювати спеціалізовану евристику переважує це відставання по продуктивності.

Реагування на зміни трафіку: було досліджено переваги при реконфігурації мінімізації можливостей SOL, для простоти скорочено назвали “mindiff”. Спершу було обчислено оптимальне рішення додатку з інжинірингу трафіку; потім, випадкова перестановка матриці трафіку викликала перерахування з увімкненим mindiff. При обрахунку нового рішення, було призначено в 4 рази більший пріоритет для TE цілей, аніж

mindiff цілям. Рисунок 3.3а показує, що з увімкненим mindiff, до додаткових 35%.

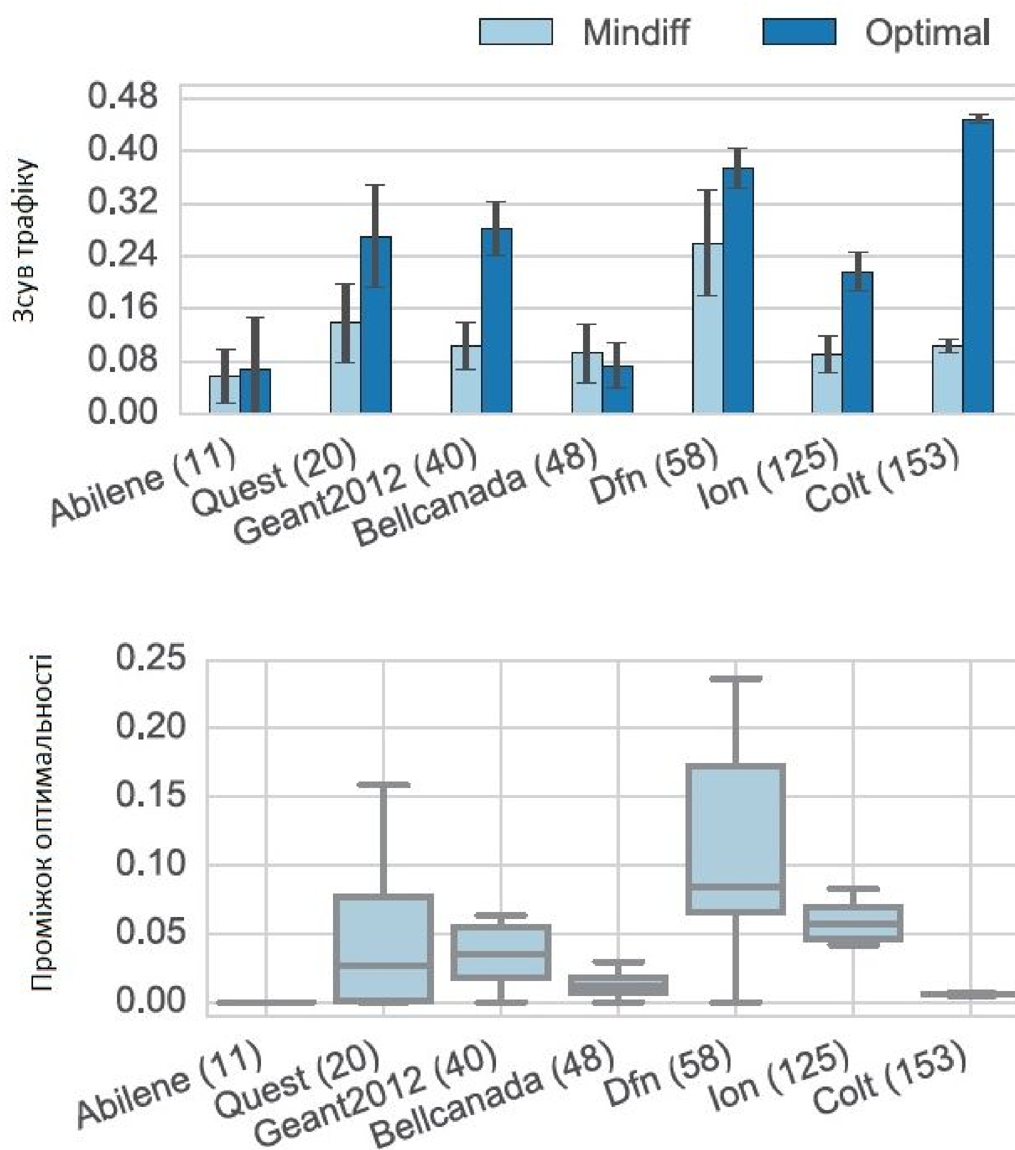
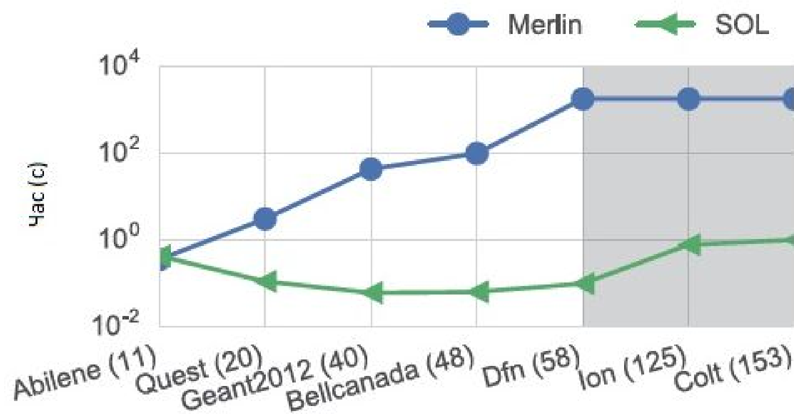


Рисунок 3.3: Зсув трафіку та проміжок оптимальності при використанні можливостей реконфігураційної мінімізації SOL



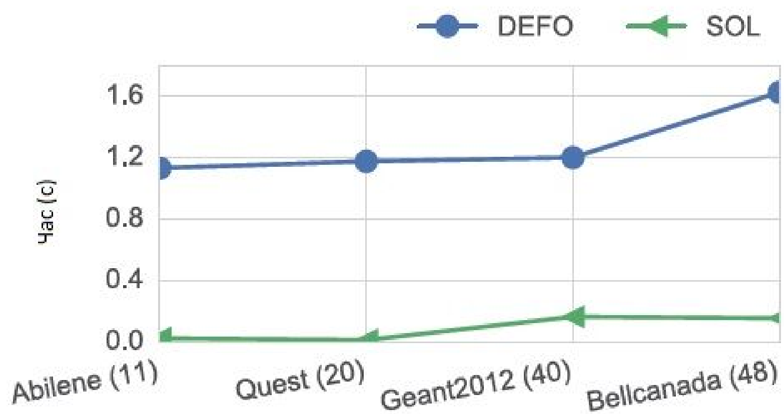


Рисунок 3.4: Період виконання SOL у порівнянні з state-of-the-art оптимізаційними фреймворками frameworks

Від всього трафіку залишається на своїх оригінальних шляхах через переконфігурацію, в порівнянні з перепризначенням до нових шляхів оптимальним рішенням. Природно, SOL жертвує деякою оптимальністю в оригінальних TE цілях (показано на рисунку 3.3 b).

3.3.3 Порівняння з Merlin та DEFO

Merlin [45] призначений/вирішує проблеми менеджменту мережевих ресурсів схоже до SOL. Коли цілі та формулювання Merlin та SOL досить різні, використовуючи це порівняння для підкреслення узагальненість SOL та силу його абстракцій шляху. Особливо, Merlin використовує більш трудомістку оптимізацію яка завжди ILP та оперує на графах які значно більші ніж фізичні мережі. Було реалізовано приклад додатку взятого зі статті Merlin, використовуючи як SOL так і Merlin. Рисунок 3.4 а показує що SOL випереджає Merlin на порядок або два.

DEFO [18] це оптимізаційний фреймворк який націлений на спрощення інжинірингу трафіку [18]. Було отримано реалізацію авторів DEFO та порівняли оптимізаційний час DEFO та SOL на простому додатку інжинірингу трафіку. DEFO та SOL показують порівнювані часи виконання (див. рисунок 3.4 b). Однак, DEFO не вистачає можливості вираження більш складних додатків та цілей, та фільтрації шляхів по довільним предикатам.

3.4 Переваги для розробників

SOL є набагато простішим фреймворком для кодування SDN оптимізаційних завдань, в порівнянні з розробкою спеціалізованих рішень вручну. Прагнучи продемонструвати цю простоту, якось чисельно, Таблиця 1 показує кількість рядків коду (LOC) в рішенні SOL імплементації різних додатків (SOL LOC), та відношення LOC оригінального формулювання до LOC SOL реалізації (Передбачуване покращення)(“Estimated improvement”). Визнаючи що порівняння рядків коду не є точним, але важко вигадати інші шляхи порівняння зусиль при розробці без проведення досліджень користувачів.

Таблиця 3.1: Переваги відносно зусиль при розробці отримані при використанні SOL

Name	SOL lines of code	Estimated improvement
ElasticTree	16	21.8×
Panopticon	13	25.7×
SIMPLE	21	18.6×

Покращення наведені в Таблиці 3.1 є консервативними. Перше, виробляти оригінальні формулювання є набагато складнішим та витонченішим процесом ніж написання SOL коду. Насамперед цю різницю можна приписати для необхідності обліку для CPLEX деталей (або інших

вирішувачів, наприклад [17, 33]); з SOL, ці деталі повністю сховані від розробника. Друге, SOL переводить свої оптимізаційні результати до конфігурації пристрою, тоді як ця функціональність навіть не включена в скриптах для отримання оригінальних формулювань. Виробляючи конфігурацію для пристроїв з оригінальних рішень потребувало б моделювання додаткових алгоритмів для співставлення змінних в кожному формулюванні до релевантних конфігурацій для пристроїв.

3.5 Чутливість

SOL рішення вимагає специфікації як кількості (selectNumber) так і типу (shortest чи random) шляхів для вибору для кожного класу трафіку. В цій секції, буде проведена кількісна оцінка, наскільки SOL чутливий до цих параметрів.

Кількість шляхів: Рисунки 3.5 та 3.6 показують час виконання SOL та розрив оптимальності як функцію числа шляхів на кожен клас для двох додатків SIMPLE та Panopticon. Не дивно, з більшою кількістю шляхів, час виконання SOL зростає. Однак, це не значна причина для занепокоєння, так як було знайдено оптимальне рішення selectNumber на рівні 5. Ці числа репрезентативні для всіх додатків та топологій, які були взяті до уваги/розглянуті.

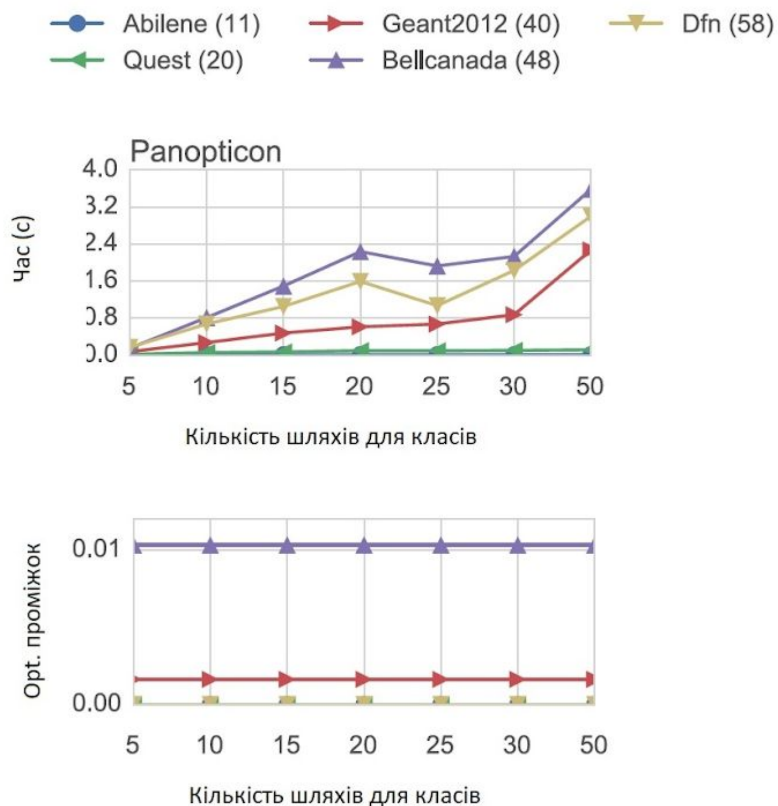


Рисунок 3.5 : Час виконання та проміжок оптимальності як функція від шляху; оптимальність досягнута у більшості випадків з такою малою кількістю шляхів як 5 на кожен клас

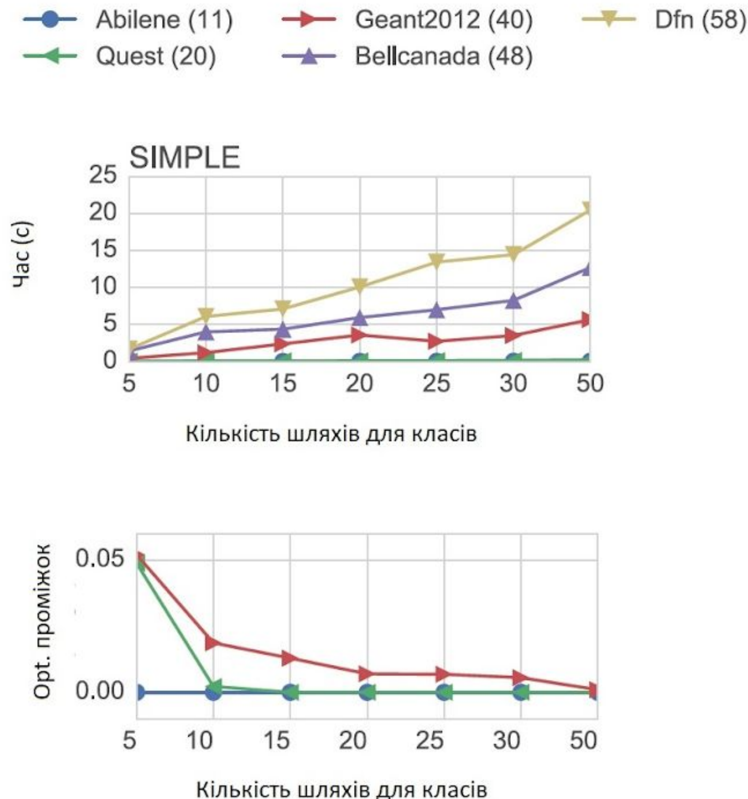


Рисунок 3.6 : Час виконання та проміжок оптимальності як функція від шляху; оптимальність досягнута у більшості випадків з такою малою кількістю шляхів як 5 на кожен клас

Стратегія вибору шляху: було оцінено різні стратегії вибору серед топологій та додатків (упущено для стислості). Результати були консистентними з досвідом більш загально ніж більшість проблем визначають себе до очевидної стратегії вибору шлях: проблеми з потребою в розподіленні/балансуванні завантаження здебільшого отримують користь від random та проблеми з чутливістю до затримки - від shortest. Якщо є сумніви - обидві стратегії можуть бути виконані водночас для порівняння.

Вартість вибору та генерації шляху: Так як вибір шляху є частиною оптимізаційного циклу, потрібно було переконатися що час вибору шляху є невеликим, в діапазоні від 0.1 до 3 секунд для різних топологій.

Вибір шляху передується фазою генерування шляху який перераховує прості шляхи для класів трафіку. Генерація шляху помірно затратна для великих топологій, наприклад < 300 секунд для найбільших топологій які були представлені, при паралелюванні до 60 потоків. Однак, потрібно наголосити/підкреслити, що генерування шляху може відноситись до офлайн фази попереднього обчислення та обраховуватись тільки разово.

3.6 Виразність SOL

Виразність SOL: Але як не крути. SOL не є панацеєю, якою можна виразити будь-яку оптимізацію, і також не існує формального шляху щоб вирішити чи проблема підходить під SOL парадигму. Можна надати вказівки для яких проблем SOL підходить краще, а для яких ні. Оптимізації зі складними предикатами шляху дійсно отримують користь з SOL, так як генерування шляху та валідація виконуються офлайн , а не під час самої оптимізації. Так само і проблеми з обмеженням ресурсів залежні від шляхів (наприклад SIMPLE зі своїми TCAM обмеженнями). Проблеми в яких не беруть участь предикати шляхів та дуже великі взаємозв’язані топології (наприклад, мережі великих дата центрів) більш за все не отримають великої переваги з використанням SOL. Однак, планується дослідити альтернативні підходи (наприклад, ієрархічна оптимізація), підходи для надання переваг і в тій області також.

Аналітичні гарантії: Поки емпіричні результати пропонують що random або shortest шляхи виробляють близькі до оптимальних рішень з відносно низьким показником selectNumber, вони також піднімають цікаві теоретичні питання: чи можемо ми довести що ці стратегії вибору надають близькі до оптимальних рішення для специфічних класів проблем?

Дуже великі/динамічні мережі: Для дуже великих мереж (>100 вузлів) SOL може не працювати так само як евристичні рішення, особливо для додатків які вимагають ILP, так як число шляхів зростає дуже стрімко. в таких випадках, ми можемо використати загальне приближення евристик, таке як випадкове округлення, для підтримки здатності реагувати на зміни в мережі швидко.

Композиція: Маючи унфікований оптимізаційний шар поверх SDN контролеру, відкриває можливості для композиції додатків. Це планується зробити в майбутніх роботах.

3.7 Зв’язані праці/роботи

Вже було обговорено оптимізаційні додатки які змотивували появу SOL. Тут, зфокусуємося на іншій пов’язаній роботі.

Абстракція високих шарів для SDN: Ця робота включає нові мови програмування (наприклад, [41, 12]), тестування та верифікаційні інструменти (наприклад, [42], компілятори для правил генерування (наприклад [26], абстракції для вирішення конфліктів контролю (наприклад, [4]), та API для вираження користувальницьких вимог (наприклад, [10]). Ці роботи не адресовані оптимізувати компоненти, що є фокусом для SOL.

Мови для оптимізації: Є декілька фреймворків для моделювання, такі як AMPL [13], Mosek [33], PyOpt [37], та PuLP [32] для вираження оптимізаційних завдань. Однак, вони не спрощують оптимізацію мереж. SOL це доменно-спеціалізована бібліотека що оперує на вищому рівні семантики в порівнянні з цими “обгортками”. SOL пропонує абстракцію що основується на шляху для написання мережевих оптимізацій, експлуатує цю структуру для вирішення цих оптимізацій швидко, та генерувати конфігурації мережевих пристроїв для реалізації своїх рішень.

Менеджмент мережевих ресурсів: Merlin це мова для менеджменту мережевих ресурсів [45]. В термінах додатків, які вона може підтримувати, Merlin має обмеження до використання предикатів шлях, виражених як регулярні вирази. Наш експеримент підрозуміває що SOL на три порядки швидший за Merlin, використовуючи ті ж самі вирішувачі що лежать в основі. Тому, підхід на основі мови Merlin надає інші можливості (наприклад, верифіковані делегування) які SOL не (та не намагається) пропонує. DEFO інший оптимізаційний фреймворк який зосереджується на інжинірингу трафіку та додатках ланцюгування сервісів. Їхня ціль - не розробити узагальнений фреймворк, а підтримати простий менеджмент мереж операторів, що вони здійснюють використовуючи двух-рівневу архітектуру та підтримку для мереж, що не сумісні з OpenFlow через сегменту маршрутизацію. Інші роботи фокусуються на оптимізації направленні трафіку(наприклад, [39, 7]). SOL пропонує уніфіковані абстракції що покривають багато додатків управління мережею.

Висновки до розділу

В цій частині було показано що SOL

- може працювати успішно з ONOS контролерами;
- обчислювати оптимальні рішення для опублікованих порядків додатків в рази швидше в порівнянні з їх оригінальною оптимізацією; дозволяючи мінімізувати змішування трафіку.
- швидший і/або надає ширшу функціональність аніж схожі сучасні рішення;
- Значно зменшує зусилля які потрібно витрачати на розробку в порівнянні з ручною розробкою оптимізаційних додатків,
- добре піддається масштабуванню, тому що він обраховує близькі до оптимальних рішень використовуючи декілька шляхів на кожен клас трафіку.

ЗАГАЛЬНІ ВИСНОВКИ ПО РОБОТІ

Поки оптимізація мереж є центральною для багатьох SDN додатків, пару зусиль/спроб для досягнення їх доступності. Головне бачення - це узагальнений, ефективний фреймворк для вираження та вирішення мережевих оптимізацій. SOL фреймворк, досягає обох цілей, узагальненості та ефективності шляхом абстракцій на основі шляху.

Було показано, що SOL може постійно виражати додатки для різних цілей (модифікація топологій, ланцюгування сервісів, інжиніринг трафіку, розвантаження і т.п.) та видає оптимальні або близькі до оптимальних рішення зачасту з кращими показниками по продуктивності в порівнянні з спеціалізованими формулюваннями. Отже, SOL може знизити бар'єр для входження нових SDN додатків для оптимізації мереж.

Також як уже було зазначено що SOL

- може працювати успішно з ONOS контролерами;
- обчислювати оптимальні рішення для опублікованих порядків додатків в рази швидше в порівнянні з їх оригінальною оптимізацією; дозволяючи мінімізувати змішування трафіку.
- швидший і/або надає ширшу функціональність аніж схожі сучасні рішення;
- Значно зменшує зусилля які потрібно витрачати на розробку в порівнянні з ручною розробкою оптимізаційних додатків,

- добре піддається масштабуванню, тому що він обраховує близькі до оптимальних рішень використовуючи декілька шляхів на кожен клас трафіку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. Network flows: theory, algorithms, and applications. Prentice hall, 1993.
2. M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In ACM SIGCOMM Computer Communication Review, volume 38, pages 63–74, 2008.
3. M. Allalouf and Y. Shavitt. Centralized and distributed algorithms for routing and weighted maxmin fair bandwidth allocation. ACM/IEEE Transactions on Networking, 16(5):1015–1024, 2008.
4. A. AuYoung, S. Banerjee, J. Lee, J. C. Mogul, J. Mudigonda, L. Popa, P. Sharma, and Y. Turner. Corybantic: Towards the modular composition of SDN control programs. In ACM HotNets, 2013.
5. P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, et al. ONOS: towards an open, distributed SDN OS. In Proceedings of the third workshop on Hot topics in software defined networking, pages 1–6. ACM, 2014.
6. A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral. Deep packet inspection as a service. In ACM CoNEXT, pages 271–282, 2014.
7. Z. Cao, M. Kodialam, and T. Lakshman. Traffic steering in software defined networks: planning and online routing. In ACM SIGCOMM Workshop on Distributed Cloud Computing, pages 65–70, 2014.

8. M. Charikar, Y. Naamad, J. Rexford, and K. Zou. Multi-Commodity Flow with In-Network Processing. Manuscript, www.cs.princeton.edu/~jrex/papers/mopt14.pdf.

9. E. Danna, S. Mandal, and A. Singh. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In IEEE Conference on Computer Communications, pages 846–854, 2012.

10. A. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory networking: An API for application control of SDNs. In ACM SIGCOMM, August 2013.

11. B. Fortz and M. Thorup. Internet traffic engineering by optimizing OSPF weights. In IEEE Conference on Computer Communications, volume 2, 2000.

12. N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In ACM SIGPLAN Notices, volume 46, pages 279–291, 2011.

13. R. Fourer, David. M. Gay, and Brian.W. Kernighan. AMPL: A mathematical programming language. A Modeling Language for Mathematical Programming AT&T Bell Laboratories Murray Hill, 1987.

14. A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, V. Sekar, and A. Akella. Stratos: A network-aware orchestration layer for virtual middleboxes in clouds. arXiv preprint arXiv:1305.0209, 2013.

15. A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling innovation in network function control. In ACM SIGCOMM, 2014.

16. G. Gibb, H. Zeng, and N. McKeown. Outsourcing network functionality. In ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, 2012.

17. Gurobi. http://www.gurobi.com/.

18. R. Hartert, S. Vissicchio, P. Schaus, O. Bonaventure, C. Filsfils, T. Telkamp, and P. Francois. A declarative and expressive approach to control forwarding paths in carrier-grade networks. In ACM SIGCOMM, 2015.

19. B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: Saving energy in data center networks. In USENIX Symposium on Networked Systems Design and Implementation, pages 19–21, 2010.

20. V. Heorhiadi, S. K. Fayaz, M. K. Reiter, and V. Sekar. SNIPS: A software-defined approach for scaling intrusion prevention systems via offloading. In 10th International Conference on Information Systems Security, Dec. 2014.

21. V. Heorhiadi, M. K. Reiter, and V. Sekar. New opportunities for load balancing in network-wide intrusion detection systems. In ACM CoNEXT, 2012.

22. V. Heorhiadi, M. K. Reiter, and V. Sekar. SOL bitbucket repository. https://bitbucket.org/progwriter/sol/, 2015. 13

23. C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In ACM SIGCOMM, pages 15–26, 2013.

24. S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J.Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software definedWAN. In ACM SIGCOMM, pages 3–14, 2013.

25. X. Jin, L. E. Li, L. Vanbever, and J. Rexford. Softcell: Scalable and flexible cellular core network architecture. In ACM CoNEXT, 2013.

26. N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the one big switch abstraction in software defined networks. In ACM CoNEXT, pages 13–24, 2013.

27. P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In USENIX Symposium on Networked Systems Design and Implementation, 2012.

28. S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. IEEE Journal on Selected Areas in Communications, 29(9):1765 –1775, October 2011.

29. D. Levin, M. Canini, S. Schmid, F. Schaffert, A. Feldmann, et al. Panopticon: Reaping the benefits of incremental sdn deployment in enterprise networks. In USENIX Annual Technical Conference, 2014.

30. X. Liu, S. Mohanraj, M. Pioro, and D. Medhi. Multipath routing from a traffic engineering perspective: How beneficial is it? In IEEE International Conference on Network Protocols, pages 143–154, 2014.

31. Mininet. http://mininet.org/.

32. S. Mitchell, M. O'Sullivan, and I. Dunning. Pulp: a linear programming toolkit for python, 2011.

33. Mosek. https://mosek.com/.

34. Network functions virtualisation – introductory white paper. http://portal.etsi.org/NFV/NFV_White_Paper.pdf.

35. Opendaylight SDN controller. http://www.opendaylight.org/.

36. S. Palkar, C. Lan, S. Han, K. Jang, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A runtime framework for network functions. In ACM Symposium on Operating Systems Principles, 2015.
37. R. E. Perez, P. W. Jansen, and J. R. R. A. Martins. pyOpt: A Python-based object-oriented framework for nonlinear constrained optimization. Structures and Multidisciplinary Optimization, 45(1):101–118, 2012.
38. M. Pióro, P. Nilsson, E. Kubilinskas, and G. Fodor. On efficient max-min fair routing algorithms. In Computers and Communication, pages 365–372. IEEE, 2003.
39. Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying middlebox policy enforcement using SDN. In ACM SIGCOMM, 2013.
40. S. Raza, G. Huang, C.-N. Chuah, S. Seetharaman, and J. P. Singh. Measurouting: a framework for routing assisted traffic monitoring. ACM/IEEE Transactions on Networking, 20(1):45–56, 2012.
41. J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker. Modular SDN programming with pyretic. ;login: Magazine, 38(5):128–134, 2013.
42. M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D.Walker. Abstractions for network update. In ACM SIGCOMM, 2012.
43. M. Roughan. Simplifying the synthesis of internet traffic matrices. ACM SIGCOMM Computer Communication Review, 35, 2005.
44. J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else’s problem: Network processing as a cloud service. In ACM SIGCOMM, 2012.

45. R. Soule, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for provisioning network resources. In ACM CoNEXT, 2014.

46. N. Vasi´c, P. Bhurat, D. Novakovi´c, M. Canini, S. Shekhar, and D. Kosti´c. Identifying and using energy-critical paths. In ACM CoNEXT, 2011.

47. R. Wang, D. Butnariu, and J. Rexford. Openflow based server load balancing gone wild. In Hot-ICE, 2011.

ДОДАТОК А

Нові гнучкі можливості масштабування: Тут, показується що SOL може бути використаний для нових SDN додатків. Особливо, розглядалися/враховувалися гнучкі NFV налаштування [14], які розміщують MB в мережі та виділяють ємності у відповідь на попит що був досліджений. Також можуть бути присутніми додаткові обмеження, такі як загальна кількість таких VM локацій. Як проста ціль, була розглянута/взята до уваги верхню числову границю вузлів що використовуються, в той же час проводячи балансування серед віртуальних екземплярів MB. Досить просто можна додати інші цілі, такі як мінімізація кількості VM. Для стислості, були підкреслені тільки ключові частини побудови таких нових додатків.

```
1 predicate = hasMboxPredicate
2 opt.addBinaryVariables(pptc, topo, ['path', 'node'])
3 opt.addNodeCapacityConstraint(pptc, 'cpu', {node: 'TBA'
      for node in topo.nodes()}, lambda node, tc, path,
      resource: tc.volFlows * tc.cpuCost)
4 opt.addRequireSomeNodesConstraint(pptc)
5 opt.addPathDisableConstraint(pptc)
6 opt.addBudgetConstraint(topo, lambda node: 1, topo.
      getNumNodes()/2)
7 opt.setPredefinedObjective('minmaxnodeload', resource='
      cpu')
```

Перше, було визначено валідні шляхи, що проходять через MB; SOL надає предикати(твердження) для цього (рядок 1). Головна різниця тут, це визначення ємностей з TBA значенням в рядку 3; вона позначає що наша

оптимізація повинна виділити ємності для вузлів. (SOL забезпечує/переконується що деактивовані вузли мають виділену ємність рівну 0). Отже вимога на те щоб принаймні один активний вузол на кожен шлях (рядки 4 та 5), обмежується кількість активних вузлів (рядок 6), та встановлюються цілі (рядок 7).

```
1  def _MaxMinFairness_MCF(topology, pptc, unstaturated,
        saturated, allocation, linkCaps):
2      opt = getOptimization()
3      opt.addDecisionVariables(pptc)
4      # setup flow constraints
5      opt.addAllocateFlowConstraint({tc: pptc[tc] for tc in
            unstaturated})
6      for i in saturated:
7          opt.addAllocateFlowConstraint({tc: pptc[tc] for
                tc in pptc[i]}, allocation[i])
8      # setup link capacities:
9      def linkcapfunc(link, tc, path, resource):
10         return tc.volBytes
11     opt.addLinkCapacityConstraint(pptc, 'bandwidth',
            linkCaps, linkcapfunc)
12     opt.setPredefinedObjective("maxallflow")
13     opt.solve()
14     return opt
```

```
16 def iterateMaxMinFairness(topology, pptc, linkCaps):
17     # Setup saturated and unsaturated commodities
18     saturated = defaultdict(lambda: [])
19     unsaturated = set(pptc.keys())
20     paths = defaultdict(lambda: [])

22     t = []  # allocation values per each iteration
23     i = 0  # iteration index
24     while unsaturated:
25         # Run slightly modified multi-commodity flow
26         opt = _MaxMinFairness_MCF(topology, pptc,
               unsaturated, saturated, t, linkCaps)
27         if not opt.isSolved():
28             raise FormulationException('No solution')
29         alloc = opt.getSolvedObjective()
30         t.append(alloc)
31         # Check if commodity is saturated, if so move it
               to saturated list
32         for tc in list(unsaturated):
33             # NOTE: this is an inefficient non-blocking
                   test, based on dual variables
34             # More efficient methods are available
35             dual = opt.getDualValue(opt.al(tc))
36             if dual > 0:
37                 unsaturated.remove(tc)
38                 saturated[i].append(tc)
39                 paths[tc] = opt.getPathFractions()[tc]
40         i += 1
41     return paths
```

Рисунок 3: Python код для Max-min оптимізації чесності

Складні багатоскладові оптимізації: Також було показано, як можна змоделювати складну оптимізацію, використовуючи SOL настільки примітивно, як вираження певних блоків оптимізацій. Рисунок 3 надає, код для вирішення проблеми min-max чесності. Воно базується на вираженні потокових проблем проміжної мультикомпонентності за допомогою

використання SOL (див. функцію _MaxMinFairness_MCF)) та написанням невеликого ітеративного алгоритму (див. функцію iterateMaxMinFairness) для отримання оптимального рішення. Моделювання коду відбувалося після того як алгоритм Pióro et al. [38] був представлений, однак вже наявні новіті та більш ефективні пропозиції, які також можуть бути виражені в SOL (наприклад, [3, 9]).

ДОДАТОК Б

Прогресивні користувачі та низькорівневі інтерфейси

В той час коли SOL API описано в розділі 2.2 узагальнено та досить виразно для збору різноманітних вимог широкого спектру додатків, низькорівневий API також відкритий, і він надає більше контролю користувачам відкриваючи доступ до внутрішніх змінних SOL. Прогресивні користувачі можуть використовувати це API для подальшої кастомізації.

	Var.	Description
Decision	$x_{c,p}$	Fraction of class-c flows allocated to path $p \in \mathtt{paths}(c)$; non-integer
	b_p	Is path p used; binary
	b_v	Is node v used; binary
	b_l	Is link l used; binary
	$capvar_v^r$	Capacity allocated for resource r at node v; non-integer
Derived	a_c	Fraction of c's "demand" routed; non-integer
	$load_l^r$	Amount of resource r consumed by flows routed over link l; non-integer
	$load_v^r$	Amount of resource r consumed by flows routed via node v; non-integer

Рисунок 4: Змінний інтервал для оптимізації

Наприклад, API виклики роблять придатними імена внутрішніх змінних на рисунку 4, для отримання та присвоєння їх значень. Схоже, використовуючи defineVar (name, coeffs, lb, ub) функцію, користувач може створити нові змінні з іменем name, задати нижні та верхні числові границі (lb та ub), та прирівняти її до лінійної комбінації багатьох інших існуючих

змінних, як вказано в coeffs, список співставлених імен змінних та їх числових коефіцієнтів.

Цей підхід примітивно корисний, для визначення складних цілей. SOL також дозволяє встановлення спеціалізованих цільових функцій, які представляють собою лінійну комбінацію будь-яких існуючих змінних, роблячи можливою оптимізацію за декількома цілями. Дана функціональність виконана через виклик setObjective (coeffs, dir) функції, яка приймає список (coeffs) співставлених імен змінних до їх коефіцієнтів. Бінарний вхідний параметр вказує на те, чи ціль повинна бути максимізована, чи мінімізована.

ДОДАТОК В

```
1 predicate_SIMPLE = functools.partial
    (waypointMboxPredicate, order=('fw','ids'))
2 def Func_SIMPLE_NodeCap(node,tc,path,resource,nodeCapaticies):
3 if resource=='cpu' and node in nodeCapaticies['cpu']:
4 return tc.volFlows*tc.cpuCost/nodeCapaticies[resource][node]
5 capacityFunc = functools.partial(Func_SIMPLE_NodeCap,
    nodeCapaticies=nodeCapaticies)
6
7 def Func_SIMPLE_TCAM(node, tc, path, resource): return 1
8
9 # Generation of path/s, assumption can be made that it should
    be executed once in a pre-computation phase
10 opt = getOptimization()
11 pptc = generatePathsPerTrafficClass
    (topo, trafficClasses, predicate_SIMPLE, 10, 1000,
    functools.partial(useMboxModifier, chainLength=2))
12 # Traffic to paths allocation
13 pptc = chooserand(pptc, 5)
14 opt.addDecisionVariables(pptc)
15 opt.addBinaryVariables(pptc, topo, ['path','node'])
16 opt.addAllocateFlowConstraint(pptc)
17 opt.addAllRouteConstraint(pptc)
18 opt.addLinkCapacityConstraint
    (pptc, 'bandwidth', linkCaps, defaultLinkFuncNoNormalize)
19 opt.addNodeCapacityConstraint
    (pptc, 'cpu', {node: 1 for node in topo.nodes() if 'fw'
    or 'ids' in topo.getServiceTypes(node)}, capacityFunc)
20 opt.addNodeCapacityPerPathConstraint
    (pptc, 'tcam', nodeCapaticies['tcam'], Func_SIMPLE_TCAM)
21 opt.setPredefinedObjective('minmaxnodeload','cpu')
22 opt.solve()
23 obj = opt.getSolvedObjective()
24 pathFractions = opt.getPathFractions(pptc)
25 c = controller()
26 c.pushRoutes(c.getRoutes(pathFractions))
```

Рисунок 5: Код для вираження SIMPLE [39] в SOL