НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

«На правах рукопису»
УДК 004.021

ДО ЗАХИСТУ ДОПУЩЕНО:
В.О. Завідувача кафедри
_____ Михайло НОВОТАРСЬКИЙ
«___»_____ 2024 р.

Магістерська дисертація
на здобуття ступеня магістра
за освітньою-професійною програмою «Комп'ютерні системи та мережі»
зі спеціальності 123 «Комп'ютерна інженерія»
на тему: «Спосіб багатошляхової маршрутизації в програмно-конфігурованих мережах»

Виконав:
Студент ІІ курсу, групи ІО-31мп
Гутов Віталій Вікторович _____

Керівник:
асистент, к.т.н.
Кулаков Олексій Юрійович _____

Консультант з нормоконтролю:
проф., д.т.н.
Кулаков Юрій Олексійович _____

Рецензент:
доцент кафедри ІСТ, к.т.н.
Шимкович Володимир Миколайович _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2024 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Факультет/ННІ інформатики та обчислювальної техніки
(повна назва)

Кафедра обчислювальної техніки
(повна назва)

Рівень вищої освіти – другий (магістерський)

Спеціальність 123 «Комп'ютерна інженерія»
(код і назва)

Освітньо-професійна програма «Комп'ютерні системи та мережі»

ЗАТВЕРДЖУЮ:
В.О. Завідувача кафедри
_____ Михайло НОВОТАРСЬКИЙ
«___»_____ 2024 р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Гутову Віталію Вікторовичу

1. Тема дисертації «Спосіб багатошляхової маршрутизації в програмно-конфігурованих мережах»
науковий керівник дисертації Кулаков Олексій Юрійович, асистент, кандидат технічних наук
затверджені наказом по університету від «08» листопада 2024 р. № 5016-с

2. Строк подання студентом дисертації 24.11.2024 р.

3. Об'єкт дослідження процеси маршрутизації в програмно-конфігурованих мережах

4. Предмет дослідження (Вихідні дані – для магістерської дисертації за ОПП/ОНП)

розробка та оптимізація алгоритму багатошляхової

5. Перелік завдань, які потрібно розробити:
   1. Огляд джерел з маршрутизації в sdn.
   2. Спосіб багатошляхової маршрутицзації в програмно-конфігурованих мережах.

3. Практична розробка модифікованого способу багатошляхової маршрутизації.
4. Тестування модифікованого способу багатошляхової маршрутизації.
5. Стартап-проєкт

6. Консультанти розділів проєкту:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	проф. Кулаков Ю.О.		

7. Дата видачі завдання 01.09.2024 року

Календаний план

№ з/п	Найменування етапів виконання магістерської дисертації	Строк виконання етапів магістерської дисертації	Примітка
	Затвердження теми проекту	01.09.24 - 07.09.24	
	Вивчення та аналіз завдання	22.09.24 - 28.09.24	
	Вивчення та аналіз існуючих	29.09.24 - 05.10.24	
	Опис та обґрунтування рішення	06.10.24 - 10.10.24	
	Розробка алгоритму маршрутизації	10.10.24 - 19.10.24	
	Програмна реалізація запропонованого способу	10.11.24 - 13.11.24	
	Розробка стартам-проекту	14.11.24 - 18.11.24	
	Оформлення магістерської	19.11.24 - 24.11.24	

Студент _____ (підпис) Віталій ГУТОВ

Науковий керівник _____ (підпис) Олексій КУЛАКОВ

Реферат

на магістерську дисертацію

виконану на тему: *Спосіб багатошляхової маршрутизації в програмно-конфігурованих мережах*

студентом: Гутовим Віталієм Вікторовичем

Робота складається зі вступу та п’яти розділів. Загальний обсяг роботи 108 аркушів основного тексту, 22 ілюстрацій, 27 таблиць. При підготовці використоно дітературу з 20 джерел.

Актуальність теми. Програмно-конфігурована мережа стала наріжним каменем сучасних мереж завдяки можливості програмування, централізованому контролю та адаптивності, що робить її придатною для систем реального часу, де надійність і ефективність є вирішальними. Однак існуючі проблеми маршрутизації, такі як забезпечення низької затримки та високого використання пропускної здатності в динамічних середовищах, залишаються критичними для цих систем.

У невеликих системах реального часу, де обмеження ресурсів і суворі вимоги до часу є помітними, багатошляхова маршрутизація забезпечує значні переваги завдяки дослідженню всіх можливих шляхів. Тим не менш, вибір єдиного оптимального шляху на основі кількох показників, таких як пропускна здатність, затримка та кількість переходів, є важливим для забезпечення ефективного та надійного зв’язку. Сучасні підходи часто не відповідають цим вимогам, особливо щодо ефективного балансування цих показників.

Мета і завдання дослідження. Основною метою цієї магістерської роботи є розробка та аналіз модифікованого алгоритму багатошляхової маршрутизації для програмно-визначених мереж (SDN) для оптимізації вибору шляху на основі затримки мережі та показників пропускної здатності. Дослідження спрямоване на розробку методу маршрутизації, адаптованого для невеликих систем реального часу, що забезпечує ефективне використання ресурсів, низьку затримку та високу пропускну здатність.

Завдання:

1. Огляд джерел маршрутизації в SDN.
2. Розробка методу багатошляхової маршрутизації для SDN.
3. Практична реалізація модифікованого методу багатошляхової маршрутизації.
4. Тестування та перевірка модифікованого алгоритму.
5. Розробка стартап-проекту.

Об’єкт дослідження. Процеси маршрутизації в програмно-конфігурованих мережах.

Предмет дослідження. Розробка та оптимізація алгоритму багатошляхової маршрутизації для програмно-конфігурованих мереж.

Наукова новизна. Наукова новизна даної роботи полягає в розробці модифікованого алгоритму багатошляхової маршрутизації для SDN з використанням підходу пошуку в глибину (DFS), оптимізованого для невеликих систем реального часу. Алгоритм унікально інтегрує показники пропускної здатності та затримки для вибору оптимального шляху, забезпечуючи покращену пропускну здатність і затримку порівняно з традиційними методами, такими як Дейкстра.

Практичне значення одержаних результатів. Практичне значення результатів полягає в реалізації модифікованого алгоритму багатошляхової маршрутизації для SDN, адаптованого для невеликих систем реального часу. Алгоритм покращує продуктивність мережі шляхом оптимізації вибору шляху на основі показників пропускної здатності та затримки, забезпечуючи низьку затримку та високу пропускну здатність. Його адаптивність робить його придатним для програм реального часу, тоді як перевірена реалізація забезпечує готове до використання рішення для розгортання SDN у різних сценаріях.

Ключові слова. ПРОГРАМНО-КОНФІГУРОВАНА МЕРЕЖА, SDN, БАГАТОШЛЯХОВА МАРШРУТИЗАЦІЯ, ПОШУК У ГЛИБИНУ, DFS, АЛГОРИТМ МАРШРУТИЗАЦІЇ, КЕРУВАННЯ ТРАФІКОМ.

ABSTRACT

The paper consists of an introduction and five chapters. The total volume of the work is 108 pages of the main text, 22 illustrations, 27 tables. In preparation, the literature from 20 sources was used.

Relevance of the topic. Software-defined networking has become a cornerstone of modern networks due to its programmability, centralized control, and adaptability, making it suitable for real-time systems where reliability and efficiency are critical. However, existing routing challenges, such as ensuring low latency and high bandwidth utilization in dynamic environments, remain critical for these systems.

In small real-time systems, where resource constraints and strict time requirements are prominent, multi-path routing provides significant benefits by exploring all possible paths. However, selecting a single optimal path based on several metrics such as throughput, delay, and number of hops is essential to ensure efficient and reliable communication. Current approaches often fall short of these requirements, especially in terms of efficiently balancing these metrics.

Purpose and objectives of the study. The main purpose of this master's thesis is to develop and analyze a modified multi-path routing algorithm for software-defined networks (SDN) to optimize path selection based on network delay and throughput metrics. The research aims to develop a routing method adapted for small real-time systems that provides efficient resource utilization, low latency, and high throughput.

Objectives:

1. Review of routing sources in SDN.
2. Development of a multi-path routing method for SDN.
3. Practical implementation of the modified multipath routing method.
4. Testing and verification of the modified algorithm.
5. Development of a startup project.

Object of research. Routing processes in software-configurable networks.

Subject of research. Development and optimization of a multi-path routing algorithm for software-configurable networks.

Scientific novelty. The scientific novelty of this work is the development of a modified multi-path routing algorithm for SDN using a depth-first search (DFS) approach optimized for small real-time systems. The algorithm uniquely integrates throughput and delay metrics to select the optimal path, providing improved throughput and delay compared to traditional methods such as Dijkstra.

Practical significance of the results. The practical significance of the results lies in the implementation of a modified multi-path routing algorithm for SDN, adapted for small real-time systems. The algorithm improves network performance by optimizing path selection based on throughput and delay, providing low latency and high throughput. Its adaptability makes it suitable for real-time applications, while its proven implementation provides a ready-to-use solution for deploying SDN in a variety of scenarios.

Keywords. SOFTWARE-DEFINED NETWORK, SDN, MULTIPATH ROUTING, DEPTH-FIRST SEARCH, DFS, ROUTING ALGORITHM, TRAFFIC MANAGEMENT.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

SDN – software-defined networking

DFS – depth-first search

GUI - graphical user interface

ВСТУП

За останні роки сфера комп’ютерних мереж зазнала значних трансформацій, головним чином зумовлених зростанням попиту на більш гнучкі, масштабовані та ефективні мережеві архітектури. Традиційні мережеві структури, хоч і є основоположними, але вони наразі є недостатніми для обробки складності та динамічності вимог у сучасних мережах. Програмно-конфігурована мережа (SDN) стала багатообіцяючим підходом до вирішення цих проблем, забезпечуючи більший контроль і програмованість мережевих ресурсів. Ця зміна парадигми від апаратно-орієнтованого керування мережею до програмно-орієнтованого забезпечує динамічну реконфігурацію та більш інтелектуальну обробку трафіку, адаптовану до конкретних потреб додатків і умов мережі в реальному часі.

Ключовою проблемою в середовищах SDN є розробка ефективних механізмів маршрутизації, які можуть повністю використовувати потенціал цих мереж. Традиційні методи одношляхової маршрутизації, незважаючи на надійність, часто недостатні для задоволення вимог високої доступності, низької затримки та балансування навантаження у великих мережах. Як наслідок, багатошляхова маршрутизація привертає все більше уваги через її потенціал для покращення використання ресурсів, підвищення відмовостійкості та оптимізації потоку даних у мережах. Багатошляхова маршрутизація на основі SDN пропонує надійну структуру для усунення обмежень пропускної здатності та зменшення вузьких місць. Проте, має власний набір проблем, від складності вибору шляху до керування потенційними перевантаженнями на кількох шляхах.

У цій дисертації розглядається модернізований алгоритм маршрутизації, розроблений спеціально для середовищ SDN. Метою дослідження є розробка методу багатошляхової маршрутизації, який не тільки покращує ефективність мережі, але й динамічно адаптується до змінних умов трафіку. Важливість цієї

роботи полягає в її потенціалі для підвищення надійності та продуктивності програмно-конфігурованих мереж, які стають все більш критичними для підтримки інфраструктури сучасних додатків, включаючи хмарні сервіси, IoT та сервіси з інтенсивним об’ємом даних.

Дисертація має на меті зробити внесок у поточні дослідження SDN, пропонуючи власний підхід, спрямований на мінімізацію затримки, підвищення пропускної здатності і загальної стійкості мережі. В дослідженні буде розглянуто як теоретичні, так і практичні аспекти багатошляхової маршрутизації на основі SDN, представлено детальний аналіз існуючих методологій та їх обмежень, проведено розробку та оцінку нового алгоритму маршрутизації, адаптованого до вимог сучасних архітектур SDN.

РОЗДІЛ 1
ОГЛЯД ДЖЕРЕЛ З МАРШРУТИЗАЦІЇ В SDN

1.1 Опис SDN мережі

Об’єми інформації в наш час стрімко зростають, зростає і потреба до її обміну і передачі. Безліч спеціалістів невпинно шукають нові рішення, щоб мати змогу впоратись з невпинно збільшуючимся потоком даних. В мережевому трафіку таким рішення стали програмно-конфігуровані мережі, що прийшли на заміну традиційним мережам, котрі дуже важко піддаються своєчасним змінам та оновленням задля вирішення стрімко набігаючих викликів. Задача програмно-конфігурованих мереж – впоратись з сучасними викликами в сфері мережевого трафіку.

Програмно-конфігурована мережа (SDN) є сучасною парадигмою, її суть полягає в відокремленні рівня даних від рівня контролю. Тобто має існувати певний центральний контролер, який забезпечує керування всією мережею – рівень контролю. Рівень даних в свою чергу містить віртуальні пристрої, котрі виконують обмін пакетами згідно встановлених віддалено правил, прописаних в програмі центрального контролера [1].

Розглянемо архутектуру програмно-конфігурованої мережі більш детально. Мережу прийнято поділяти на шари, їх 3: шар застосунків, шар контролю та шар даних. Ці шари зображено на рис. 1.1.

Шар застосунків складається з програм, що визначають логіку роботи мережі. Вони надсилають високорівневі політики до шару управління. Програми відповідають за балансування навантаження, безпеку, моніторинг трафіку та інше. Взаємодія з контролерами задля отримання інформації про мережу та управління правилами відбувається через Northbound API [1].

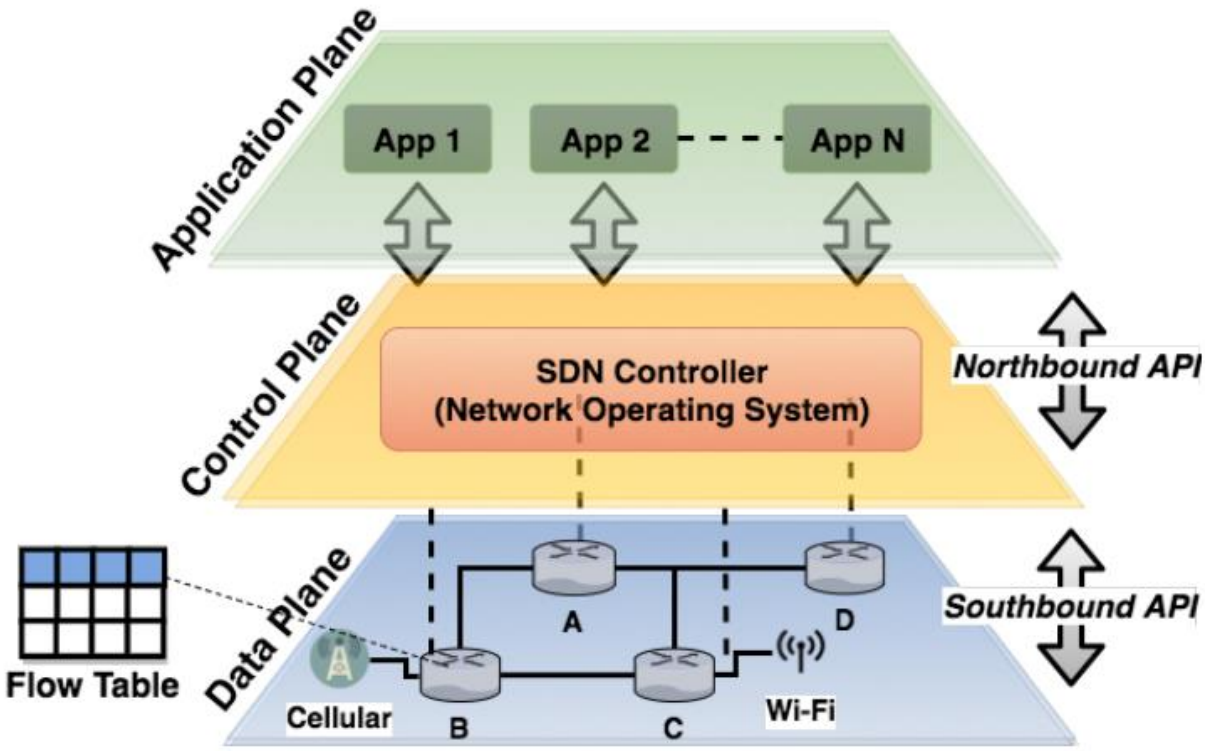

Рисунок 1.1 – Архітектура програмно-конфігурованої мережі [1]

Шар конотролю є «серцем» мережі, він відповідає за визначення політик мережі та передачу інструкцій до шару даних. За допомогою API забезпечують централізоване управління мережею і налагоджують взаємодію між верхнім та нижнім шарами. Розрізняють централізовані (наприклад, Floodlight) та розподілені (наприклад, Hyperflow) контролери. Перші чудово виконують свої функції, але є вразливою та єдиною точкою відмови і не можуть забезпечити великого масштабування мережі. Розподілені контролери вирішують дану проблему, обмін інформацією в таких контролерах також відбувається за допомогою East та Westbound API [1].

Шар даних складається з пристроїв мережі: комутаторів, маршрутизаторів тощо. Ці пристрої займаються обміном пакетів. Ключовою відмінністю від традиційних мереж є те, що у випадку програмно-конфігурованих мереж ці пристрої не мають власних інструкцій, а виконують правила пересилання, котрі

було встановлено віддалено за допомогою певного протоколу і таблиць пересилання, найбільш розповсюдженим є протокол OpenFlow. Ці мережеві пристрої не мають жодної вбудованої логіки та доступу до глобальної інформації мережі. Обмін інформації між шаром даних та шаром контролю відбувається через Southbound API [1].

Таблиця пересилання OpenFlow містить поля співставлення, котрі визначають під які правила підходить пакет, необхідні дії (пересилання на інший порт, зміна пакету, пересилання на котролер, скидання пакету) щодо пакету, лічильники статистики пакетів, що відповідають певним правилам [1].

Важливими характеристиками програмно-конфігурованої мережі, на які слід звертати увагу є:

- Затримка – важлива характеристика, оскільки вона впливає на швидкість реагування мережі, що напряму залежить від затримок в передачі трафіку між контролерами. Цій характеристиці слід приділяти особливу увагу і, де це можливо, мінімізувати затримку.
- Пропускна здатність – ця характеристика вимірює спроможність мережі до обробки трафіку. Чим вища пропускна здатність мережі – тим більшу кількість трафіку може обробити мережа. На цю характеристику впливають налаштування правил пересилки, доступні в мережі шляхи, вузькі місця тощо.
- Тремтіння – характеристика, котра відображає затримку в надходженні пакетів, що є важливим показником для систем реального часу.
- Втрата пакетів – характеристика, що відображає відсоток втрати пакетів при пересиланні.
- Надійність – показник, що відображає ймовірність ззбоїв у мережі і втраті даних.
- Масштабованість – характеристика, що відображає спроможність мережі до збільшення кількості пристроїв, потоків і так далі. На дану характеристику в

більшій мірі впливає початкове проектування мережі, можливості її контролера, або ж можливість горизонтального масштабування додатковими контролерами.

- Quality of Service (QoS) – сукупність характеристик затримки, пропускної здатності, тремтіння, втрати пакетів, надійності. Є характеристикою спроможності мережі до якісного виконання задач.

1.2 Протокол обміну даними в програмно-конфігурованих мережах

В програмно-визначених мережах керування мережевим трафіком відбувається за використання визначених протоколів обміну даними. Найбільш поширеним з них є протокол OpenFlow. Він забезпечує стандартизований інтерфейс для роботи з даними. Надає централізованому контролеру можливість відстежувати стан мережі за її метриками в реальному часі і коригувати маршрутизацію відповідним чином, в нашому випадку – на основі пказників пропускної здатності і затримки на каналах, забезпечуючи відбір оптимальних шляхів для кожного з потоків даних.

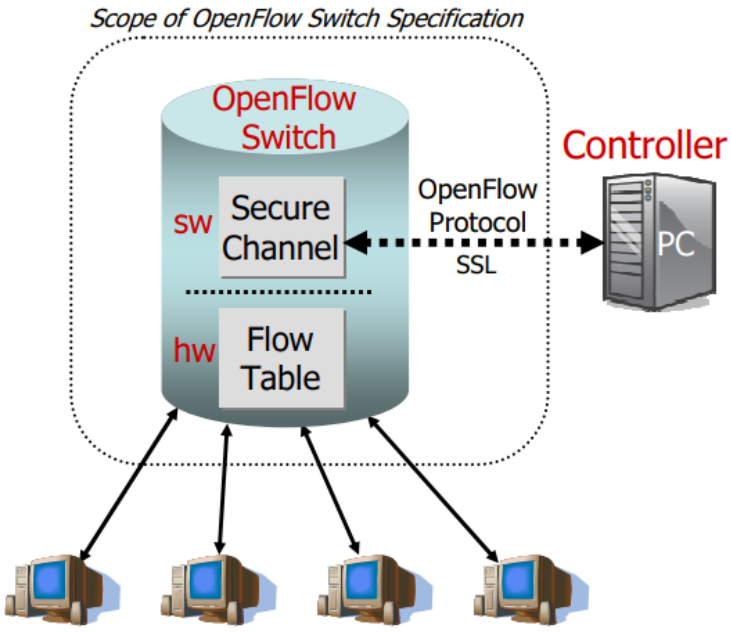

Рисунок 1.2 – Протокол OpenFlow [2]

З описаного в роботі [2] - протокол OpenFlow працює шляхом визначення правил потоку в мережевих пристроях, які визначають спосіб обробки та пересилання пакетів. Комутатори OpenFlow підтримують таблиці потоку, і коли пакет надходить, він порівнюється з записами таблиці, щоб визначити його обробку. Такі функції OpenFlow, як моніторинг метрик мережі у реальному часі, модифікація потоку та обмін повідомленнями про вхід/вихід пакетів, роблять його ідеальним протоколом для керування та оптимізації метрик програмно-конфігурованої мережі.

1.3 Огляд наукових статей з SDN тематики

Метою статті [3] є розробка алгоритму Path Selection with Low Complexity (PSLC) для оптимізації маршрутизації в мобільних мережах із обмеженою пропускною здатністю. Алгоритм забезпечує ефективне управління ресурсами, балансування навантаження та мінімізацію перешкод між вузлами, динамічно

адаптуючись до змін мережевої топології. Основні етапи алгоритму алгоритму включають визначення критичних з’єднань за поточним попитом і залишковою пропускною здатністю для уникнення перевантажень. Використання модифікованого алгоритму найкоротшого шляху з урахуванням мінімальних стрибків і максимальної пропускної здатності. Алгоритм перевірено на 15-вузловій топології та порівняно з MHA, WSP і MIRA за такими показниками: Коефіцієнт блокування викликів нижчий, що свідчить про кращу обробку трафіку. Середня довжина шляху більша, але з більшою кількістю прийнятих запитів. Максимальний потік близький до MIRA. Процесорний час кращий - нижча обчислювальна складність. PSLC виявився ефективним для динамічного трафіку, балансування навантаження та мінімізації затримок, що робить його придатним для мобільних мереж, включаючи 5G.

Стаття [4] пропонує алгоритм багатошляхового балансування навантаження на основі пропускної здатності, що розподіляє трафік між кількома шляхами для оптимізації використання ресурсів і мінімізації затримок, тремтіння та втрат пакетів. Алгоритм використовує централізоване керування SDN для адаптації до умов мережі та покращення QoS. Основні етапи алгоритму включають пошук усіх можливих шляхів за допомогою DFS. Визначення вартості шляхів за залишковою пропускною здатністю. Вибір оптимального шляху з найбільшою залишковою пропускною здатністю. Результати тестування на емуляторі Mininet, контролер Ryu SDN, двоканальна і триканальна топології, порівняння із із алгоритмами Дейкстра та ECMP за показниками затримки, тремтіння, втрата пакетів. Результати наступні: затримка зменшена до 52% порівняно з Дейкстра та до 20% з ECMP; тремтіння знижено до 54% і 43% відповідно. Втрата пакетів скорочена до 59% і 40% відповідно. Алгоритм покращує продуктивність мережі, ефективно балансує трафік і забезпечує високу якість обслуговування в SDN, що робить його придатним для мереж 5G.

У статті [5] представлено схему оптимізації таблиці потоків LFOD (Lightweight Flow Table Optimization Scheme) для підвищення продуктивності SDN шляхом ефективного керування потоками мишей (короткотривалі) та слонів (великі). LFOD вирішує проблему переповнення таблиці потоків, оптимізуючи використання ресурсів, зменшуючи навантаження на контролер і пріоритезуючи великі потоки. Особливостями LFOD є застосування принципу Парето для класифікації потоків, використання Count-Min Sketch для ідентифікації великих потоків із мінімальними витратами пам'яті, слони обробляються через таблиці потоків, миші перенаправляються контролером без створення записів. Результати тестування (контролер Ryu SDN, Open vSwitch) показали зменшення записів у таблиці потоків на 75% порівняно з OpenFlow і на 68% із DIEE, мінімальне зростання навантаження на контролер (на 3–4%), покращення пропускної здатності завдяки звільненню місця для великих потоків. LFOD є легким у реалізації рішенням, яке оптимізує використання таблиць потоків у SDN, забезпечуючи ефективність мережі без значного навантаження на обладнання.

У статті [6] запропоновано схему балансування навантаження в SDN за допомогою алгоритму міграції комутаторів на основі прогнозування трафіку. Дослідження стосується проблеми нерівномірного навантаження між контролерами, що виникає через статичне закріплення комутаторів та коливання трафіку. Ключовими елементи рішення є використання нейронної мережі LSTM для аналізу історичних даних і прогнозування майбутнього навантаження, врахування поточного та прогнозованого навантаження для визначення пріоритету міграції комутаторів, переведення комутаторів із перевантажених контролерів до тих, що мають резервну ємність. Архітектурними складовими рішення є збір і прогнозування трафіку, аналіз навантаження між контролерами та переміщення комутаторів для балансування.Результати експериментів демонструють зменшення частоти міграції, швидке вирівнювання навантаження між контролерами, зниження

часу відгуку та підвищення продуктивності. Ця схема пропонує практичне рішення для масштабованих мереж, зменшуючи вплив дисбалансу навантаження на мережевий трафік і покращуючи якість обслуговування.

У статті [7] запропоновано новий алгоритм маршрутизації для супутників LEO в SDN. Вмотивоване дане дослідження тим, що традиційні протоколи не підходять для супутників через часті зміни топології та обмежені ресурси, SDN надає централізоване керування для ефективного використання ресурсів і забезпечення QoS. Алгоритм fybrrLink інтегрує лінійний алгоритм Брезенхема та Дейкстру для швидкого й оптимального маршруту, використовує моніторинг топології й передачу правил потоку для безперервного зв'язку, враховує пропускну здатність, затримку та втрати пакетів для вибору ефективного маршруту. Результати оцінки продуктивності показали до 99,65% швидше обчислення маршруту порівняно з SDRA, відповідність вимогам QoS для понад 93% потоків (Dijkstra — лише 45%), зниження затримки до 34,86% і мінімізація втрат пакетів. Алгоритм fybrrLink забезпечує надійну маршрутизацію з низькою затримкою та високою QoS, що робить його придатним для додатків реального часу. Дослідження також відкриває перспективи інтеграції наземних і супутникових мереж.

Стаття [8] представляє детальний аналіз різних алгоритмів балансування навантаження, реалізованих за допомогою мови програмування P4 у програмно-визначеній мережі (SDN). Дослідження зосереджено на тому, як ці алгоритми допомагають керувати зростаючим мережевим трафіком, викликаним новими технологіями, такими як 5G, хмарні обчислення, Інтернет речей (IoT) і віртуалізація мережевих функцій (NFV). Зростання трафіку через 5G, IoT і хмарні обчислення викликає перевантаження серверів і знижує QoS. Традиційні підходи неефективні в умовах великих навантажень. У дослідженні розглядаються чотири алгоритми балансування навантаження із збереженням стану, реалізовані за допомогою мови P4, розробленої для програмування незалежних від протоколу пакетних процесорів

у середовищах SDN: хеш підключення - маршрутизація за хешем п’яти кортежів; випадковий вибір - простий, але непередбачуваний розподіл; Round Robin - рівномірний розподіл між серверами; Weighted Round Robin - облік потужності серверів. Використання об’єктів зі збереженням стану в P4 зменшує залежність від зовнішніх контролерів. P4 забезпечує програмованість комутаторів для ефективного розподілу трафіку. Рішення пропонує інтеграцію моніторингу працездатності серверів без перевантаження системи. Результати показали підвищення QoS, швидкості обробки пакетів, адаптацію до умов у реальному часі. P4 демонструє потенціал для створення гнучких і продуктивних механізмів балансування навантаження, особливо для додатків із високими вимогами, таких як 5G та IoT.

У статті [9] досліджуються проблеми масштабованості рівня керування SDN, зокрема його логічної централізації, яка може обмежити продуктивність за високих вимог до мережі. дослідження класифікує рішення масштабованості на підходи «пов’язані з топологією» та «пов’язані з механізмами». Централізовані контролери SDN мають проблеми з масштабованістю через вузькі місця обробки запитів потоку та завдань моніторингу. Підходи, «пов’язані з топологією», включають централізовані, розподілені, ієрархічні та гібридні конструкції контролерів, кожен зі своїми перевагами та обмеженнями. Наприклад, гібридні конструкції включають пристрої площини даних у процес прийняття рішень, щоб зменшити навантаження на контролер. Підходи, «пов’язані з механізмами», оптимізують продуктивність контролера за допомогою паралелізму (наприклад, багатопоточності, пакетного введення/виведення) і розширених механізмів маршрутизації. Автори надають таксономію найсучасніших рішень масштабованості та оцінюють такі показники, як пропускна здатність і затримка, щоб порівняти продуктивність контролерів. Такі інструменти, як Cbench і Mininet, використовуються в багатьох дослідженнях для моделювання та оцінки контролерів. У статті висвітлено кілька контролерів, які

досягають різних рівнів масштабованості. NOX-MT і Beacon є багатопоточними контролерами, що забезпечують високу пропускну здатність (1,8 і 12,8 млн потоків/с відповідно). Ієрархічні структури, такі як Kandoo, ефективно делегують завдання, щоб зменшити навантаження на центральні контролери. Розподілені рішення, такі як Onix і ElastiCon, динамічно розподіляють навантаження між контролерами. Покращення масштабованості часто передбачає компроміси, такі як збільшення складності синхронізації або зменшення гнучкості реактивного керування. Відкритими завданнями є забезпечення надійності контролера під час збоїв, підтримка узгодженості стану в розподілених налаштуваннях і оптимізація обробки міждоменного трафіку. Дослідження забезпечує важливу основу для розуміння та вирішення проблем масштабованості в SDN, наголошуючи на необхідності індивідуальних рішень залежно від розміру мережі та вимог програми.

Дослідження [10] розглядає балансування навантаження в програмно-визначених мережах (SDN) для вирішення таких проблем, як перевантаження трафіку, масштабованість мережі та доступність послуг. У ньому підкреслюється можливість програмування SDN як вирішення обмежень традиційних балансувальників навантаження від виробника. В дослідженні наявний огляд архітектури SDN та її можливості програмування за допомогою OpenFlow, огляд інтелектуальних схем балансування навантаження, класифікованих за різними мережевими компонентами (наприклад, контролери, сервери та бездротові з’єднання), обговорення показників (пропускна здатність, затримка, використання ресурсів) та інструментів (Mininet, Iperf) для оцінки методів балансування навантаження. Інтелектуальні балансувальники навантаження в SDN використовують глобальне уявлення про мережу, надане контролером SDN, для оптимізації використання ресурсів і мінімізації часу відповіді. Такі методи, як генетичні алгоритми, оптимізація роїв частинок і штучні нейронні мережі, покращують маршрутизацію та розподіл навантаження. Архітектури розподілених

контролерів і схеми динамічного вибору шляху (RLMD і ElastiCon) є критичними для масштабування середовищ SDN. Енергоефективність і відмовостійкість залишаються недостатньо вивченими в балансуванні навантаження SDN. Інтеграція SDN із такими новими технологіями, як 5G та IoT, потребує передових інтелектуальних алгоритмів для обробки високого трафіку та щільності пристроїв. Дослідження адаптивних і стійких алгоритмів для програм реального часу є важливими. У документі підкреслюється потенціал SDN для революції в балансуванні мережевого навантаження, виступаючи за подальший розвиток інтелектуальних і адаптивних методів. Це підкреслює здатність SDN абстрагувати складність мережі, прокладаючи шлях до інноваційних рішень для управління трафіком.

Стаття [11] досліджує інженерію трафіку (TE) у програмно-визначених мережах (SDN), наголошуючи на тому, як гнучкість SDN і глобальний вигляд мережі можуть покращити традиційні методи TE. Автори надають комплексний огляд класичних методів і методів TE на основі SDN, підкреслюючи перехід від розподіленого до централізованого керування мережею. Аналіз механізмів організації трафіку, таких як Equal Cost Multi-Path (ECMP), MPLS-TE, та їх обмежень у традиційних мережах. Уявлення про специфічні для SDN підходи TE, такі як B4, Hedera, DevoFlow і MSDN-TE, які використовують централізовані контролери для динамічного керування трафіком. SDN дозволяє динамічно змінювати конфігурацію мережевих маршрутів на основі вимог трафіку, вирішуючи такі проблеми, як перевантаження та збої з’єднань, ефективніше, ніж традиційні методи. Такі методи, як MSDN-TE і B4, покращують використання ресурсів шляхом динамічного призначення пропускної здатності та регулювання розподілу трафіку між кількома шляхами. Розширене планування потоку та методи виявлення потоку слона (DevoFlow, Mahout) покращують масштабованість і пропускну здатність. Відмовостійкість, енергоефективність і планування оновлення

потоку є критичними проблемами в SDN TE. Можливість адаптації в режимі реального часу та застосування політики на вищому рівні залишаються недостатньо вивченими. У документі підкреслюється потенціал SDN для революції в управлінні трафіком шляхом поєднання можливості програмування мережі з централізованим контролем. Майбутні дослідження мають бути зосереджені на механізмах TE, що зважають на енергію, застосуванні політики на прикладному рівні та оптимізації трафіку на основі машинного навчання.

У дослідженні [12] представлено багатошляхову транспортну схему, спрямовану на покращення мультимедійних послуг у реальному часі за допомогою програмно-визначеної мережі (SDN) і сегментної маршрутизації (SR). Він вирішує проблеми, пов’язані з високими вимогами до пропускної здатності та жорсткими вимогами до затримки в сучасних інтернет-інфраструктурах. Автори пропонують централізований SDN-контролер, який динамічно розподіляє непересічні шляхи, підтримуваний Multipath Media Server (MMS) для адаптивного планування пакетів. Рішення використовує SR для пом’якшення проблем масштабованості в SDN і зменшення наскрізної затримки для покращення якості роботи (QoE). Алгоритм оптимізує використання мережевих ресурсів шляхом балансування трафіку та зменшення заторів. Він перевершив традиційні алгоритми SPF і WSP під час моделювання, знизивши рівень відхилень і підвищивши пропускну здатність. MMS-повідомлення з динамічним регулюванням розподіляє навантаження між декількома шляхами, мінімізуючи перевпорядкування пакетів і затримки буферизації. Завдяки вбудовуванню інформації про маршрутизацію в заголовки пакетів через SR цей підхід зменшив накладні витрати на комутатори та контролери SDN. Тести на платформі на основі Mininet продемонстрували чудову пропускну здатність мережі, зменшену затримку та збалансоване використання з’єднання порівняно з традиційними методами. Запропонована структура багатошляхової передачі на основі SDN ефективна в управлінні мультимедійними послугами з

високою пропускною здатністю, чутливими до затримки. Вона забезпечує масштабоване та гнучке рішення, поєднуючи централізоване керування з сегментною маршрутизацією. Результати моделювання підкреслюють його потенціал для підвищення як продуктивності мережі, так і якості користувача в реальних програмах.

Автори робти [13] досліджують роль програмно-визначених мереж (SDN) в оптимізації продуктивності мережі для систем 5G, зосереджуючись на зниженні операційних витрат (OPEX) і забезпеченні якості обслуговування (QoS). Вони пропонують підхід багатошляхової маршрутизації з використанням OpenFlow для розподілу ресурсів і керування трафіком. У документі оцінюються три основні рішення: повне повторне обчислення шляхів (FPR), повторне обчислення евристичних шляхів (HPR) і часткове повторне обчислення шляхів (PPR), кожне з яких спрямоване на ефективність і масштабованість у розподілі шляхів і управлінні мережею. FPR досягає оптимальних конфігурацій, але потребує більшого часу обчислення, що підходить для невеликих мереж. HPR на основі алгоритму Дейкстра збалансовує ефективність і швидкість, зосереджуючись на швидких налаштуваннях у динамічних мережах. PPR скорочує час обчислення, зосереджуючись на поступових оновленнях шляху для нових або мобільних користувачів. FPR використовує менше мережевих комутаторів, мінімізуючи витрати, але за рахунок обчислювального часу. HPR пропонує найшвидші результати, але активує більше перемикачів, що збільшує OPEX. PPR досягає балансу, демонструючи ефективність у динамічних середовищах із помірними обчислювальними вимогами. Усі рішення підтримують динамічне коригування шляху для вирішення проблеми мобільності та змінного навантаження на мережу, що є важливим для гарантії якості обслуговування 5G. Результати моделювання підтверджують стійкість запропонованих алгоритмів з PPR і HPR, придатними для мереж реального часу з високою щільністю. У статті показано, що багатошляхова маршрутизація на основі

SDN ефективно підвищує продуктивність мережі 5G, зменшуючи OPEX і забезпечуючи QoS за різних умов. У той час як FPR пропонує теоретичну оптимальність, PPR і HPR більш практичні для реальних застосувань, збалансовуючи ефективність і обчислювальні витрати.

Автори статті [14] досліджують розгортання та операційні стратегії гібридних програмно-визначених мереж (SDN), які поєднують традиційні та SDN технології. Гібридні мережі SDN розглядаються як поступовий підхід до переходу від застарілих мереж до повної реалізації SDN, вирішуючи такі проблеми, як обмеження вартості, побоювання простою та можливості поступового оновлення. В дослідженні наведено стратегії розгортання гібридних SDN, конструкції та можливості контролера, транспортні інженерні механізми, інструменти управління та перевірки, рамки безпеки. Такі стратегії, як Panopticon і моделі поступового розгортання, знижують витрати та дозволяють поетапні переходи, підвищуючи програмованість мережі, зберігаючи застарілі пристрої. Гібридні контролери інтегрують успадковані функції та функції SDN, забезпечуючи плавний зв’язок і взаємодію. Ефективні методи розподілу та оптимізації шляхів, такі як бюджетні моделі максимального покриття, покращують потік трафіку та зменшують перевантаження мережі. Такі інструменти, як модуль SDN Shim і моделі на основі послуг, покращують масштабованість, динамічну маршрутизацію та відновлення після збоїв. Виявлені прогалини включають інтеграцію розширеної техніки трафіку зі зниженою операційною складністю та забезпечення надійних механізмів обробки відмов. У дослідженні робиться висновок, що гібридні мережі SDN представляють життєздатний шлях до впровадження SDN завдяки балансуванню інновацій із сумісністю застарілих систем. Виділені підходи забезпечують економічно ефективні та масштабовані рішення для модернізації мережі, одночасно зменшуючи ризики, пов’язані з повним переходом.

Стаття [15] фокусується на покращенні вибору шляху в SDN шляхом використання генетичного алгоритму MOCell для багатоцільової оптимізації. У дослідженні розглядаються обмеження традиційних алгоритмів маршрутизації за найкоротшим шляхом у SDN, які часто пропускають критичні параметри, такі як пропускна здатність, затримка та втрата пакетів, що призводить до перевантаження та неоптимальної продуктивності. Автори пропонують багатоцільовий клітинний генетичний алгоритм (MOCell) для оптимізації вибору шляху. Цей метод оцінює кілька конфліктуючих параметрів QoS — смугу пропускання, затримку та втрату пакетів — для надання збалансованих рішень. Дослідження порівнює MOCell з алгоритмом Дейкстра та іншим еволюційним методом (MOEA/D) з використанням змодельованої топології SDN із кількома доменами. Використовувалися такі засоби моделювання, як Mininet і MATLAB, з оцінкою показників за умовами трафіку TCP і UDP. Метод MOCell значно перевершив традиційну маршрутизацію за найкоротшим шляхом, підвищивши пропускну здатність до 54% для трафіку TCP і зменшивши втрату пакетів для потоків UDP. Він також продемонстрував кращу ефективність порівняно з MOEA/D, досягнувши вищої середньої швидкості передачі даних за всіма критеріями QoS. Дослідження підкреслює потенціал генетичних алгоритмів для вирішення складних проблем оптимізації в SDN. Підхід MOCell забезпечує надійний механізм багатопараметричної оптимізації, що робить його придатним для великомасштабних програм реального часу.

Стаття [16] фокусується на проблемах і рішеннях для покращення масштабованості та продуктивності програмно-визначеної мережі (SDN) на основі OpenFlow. Автори досліджують, як OpenFlow-SDN може вирішувати проблеми масштабованості, що виникають через централізований контроль і динамічні умови мережі. Вони висвітлюють критичні області, такі як логічно централізована видимість, виявлення стану зв’язку, розміщення правил потоку та балансування навантаження між контролерами. Підтримка узгодженого глобального подання в

розподілених контролерах призводить до накладних витрат на синхронізацію та вузьких місць продуктивності. Динамічні зміни топології вимагають ефективних протоколів виявлення каналів, щоб уникнути помилок маршрутизації. Обмежена ємність TCAM у комутаторах створює обмеження для зберігання записів потоку, особливо у великих мережах. Такі підходи, як розподілені, ієрархічні та гібридні контролери, оптимізують розподіл обов'язків і зменшують затримку. Такі механізми, як стиснення, агрегація та проактивне встановлення правил, зменшують використання пам'яті та зв'язок рівня керування. Динамічний розподіл навантаження між контролерами покращує використання ресурсів і запобігає вузьким місцям. У дослідженні підкреслюється важливість адаптивних механізмів для забезпечення масштабованості SDN і оперативності в режимі реального часу.

Автори статті [17] досліджують інтеграцію механізмів QoS у середовищах SDN, наголошуючи на мережах із підтримкою OpenFlow. Вони класифікують методи QoS за такими областями, як маршрутизація потоку мультимедіа, маршрутизація між доменами, резервування ресурсів, керування чергами, механізми контролю якості (QoE) та моніторинг мережі. Наведено детальну таксономію QoS-орієнтованих механізмів для SDN, визначення проблем і пропозиція рішень для кожної категорії. Огляд можливостей QoS, наданих різними версіями протоколу OpenFlow і широко використовуваними контролерами SDN, такими як OpenDaylight і ONOS. SDN покращує QoS, забезпечуючи централізоване керування мережею та динамічне коригування політики. Розвиток OpenFlow поступово включав такі функції, як керування чергами, моніторинг потоку та динамічний розподіл пропускної здатності, які є важливими для QoS. Такі контролери, як OpenDaylight, надають API для керування політиками QoS та інтегруються з такими інструментами, як OVSDB, для налаштування черги. Викликами є масштабованість надання QoS у великомасштабних середовищах SDN; інтеграція метрик QoE із традиційними параметрами QoS для підвищення

задоволеності користувачів; вирішення проблем сумісності між різнорідними доменами SDN. У документі наголошується на необхідності передових структур QoS, які включають машинне навчання для прогнозованого управління трафіком. Майбутні дослідження мають бути зосереджені на багатодоменній маршрутизації QoS, енергоефективності та адаптивних механізмах у реальному часі для підтримки нових технологій, таких як 5G та IoT.

У дослідженні [18] представлено алгоритм оптимізації маршрутизації African Vulture (AVRO), метаевристичний підхід до оптимізації маршрутизації в середовищах програмно-конфігурованої мережі. Розглядає обмеження традиційних і керованих штучним інтелектом методів, такі як проблеми масштабованості, конвергенції до локальних оптимумів і проблеми з розгортанням. Автори прагнуть досягти балансування навантаження шляхом мінімізації максимального використання каналу в різних сценаріях мережі. Алгоритм AVRO покращує алгоритм оптимізації африканського стерв'ятника (AVOA), покращуючи ініціалізацію популяції та додаючи фазу оптимізації, що забезпечує кращу обізнаність про мережу та швидшу конвергенцію. Порівняно з традиційними методами (OSPF) і вдосконаленими методами штучного інтелекту (DDPG), AVRO продемонстрував чудове балансування навантаження та знизив максимальне використання каналу до 16,9% порівняно з методами штучного інтелекту та на 71,8% порівняно з традиційною маршрутизацією. AVRO досяг вищої стабільності та масштабованості для різних інтенсивностей трафіку та мережевих топологій (GBN, GEANT2, NSFNET). Дослідження підкреслило здатність AVRO перевершувати альтернативні алгоритми, такі як GTO та RSIR, у швидкості конвергенції та показниках придатності. Алгоритм AVRO пропонує надійне рішення для оптимізації маршрутизації SDN, ефективного балансування мережевого навантаження та перевершення існуючих підходів у сценаріях динамічної мережі. Результати підкреслюють потенціал евристичних алгоритмів

для доповнення або перевершення методів штучного інтелекту в практичному розгортанні.

У статті [19] досліджується ряд алгоритмів оптимізації для програмно-конфігурованої мережі, зосереджуючись на маршрутизації, балансуванні навантаження та оптимізації затримок. Вона містить теоретичний огляд традиційних, евристичних методів і методів, на основі штучного інтелекту, порівнює їх показники у покращенні ефективності та адаптивності SDN. Дослідження підкреслює, як ці алгоритми вирішують такі проблеми, як мінімізація затримки, динамічний розподіл ресурсів і забезпечення якості обслуговування в складних середовищах SDN. радиційні алгоритми (OSPF, BGP) ефективні для простої маршрутизації, але не мають масштабованості для складних проблем NP. Евристичні методи, як-от оптимізація роєм частинок (PSO) і оптимізація колонії мурах (ACO), пропонують практичні рішення для динамічних і великомасштабних SDN. Такі підходи ШІ, як ANN і SVM, покращують адаптивність і прогнозування трафіку, підвищуючи безпеку та ефективність. Евристичні алгоритми та алгоритми, на основі штучного інтелекту, перевершили традиційні в масштабованості та адаптивності, що є критично важливим для програм реального часу. Методи оптимізації затримки значно покращили реакцію мережі, що має вирішальне значення для таких програм, як VoIP. Алгоритми балансування навантаження зменшили вузькі місця та збільшили пропускну здатність мережі, особливо в сценаріях центрів обробки даних. Дослідження підкреслює важливість інтеграції традиційних, евристичних підходів і підходів, на основі ШІ, для комплексного вирішення проблем SDN.

Стаття [20] представляє комплексне дослідження, присвячене проблемам програмно-конфігурованих мереж з кількома клієнтами. Автори пропонують адаптивне рішення, яке інтегрує детальну мережеву віртуалізацію та динамічну маршрутизацію, щоб відповідати вимогам якості обслуговування (QoS).

Архітектура системи використовує централізоване керування SDN для покращення розподілу ресурсів і ефективності маршрутизації, усуваючи обмеження традиційної розробки трафіку. Основні внески включають мережевий гіпервізор, який ізолює та встановлює пріоритети для конкретних ресурсів орендаря, мінімізуючи спільні межі мережі, а також структуру динамічного розподілу потоків, що забезпечує наскрізне надання QoS. Запропонований алгоритм маршрутизації з підтримкою віртуалізації (QVR) з урахуванням QoS поєднує ці компоненти, динамічно адаптуючись до змінних у часі умов мережі та вимог орендарів. Результати моделювання підкреслюють ефективність QVR, досягаючи зменшення спільних країв, затримки перевантаження та затримок трафіку порівняно зі звичайними підходами. Ця робота підкреслює потенціал архітектур SDN для сценаріїв з декількома орендарями, пропонуючи масштабоване та ефективне управління ресурсами. Однак практичне розгортання може зіткнутися з проблемами, такими як інтеграція QVR в існуючу мережеву інфраструктуру та підтримка продуктивності в екстремальних умовах трафіку.

Висновок до розділу 1

Описано тематику програмно-конфігурованих мереж, наявних в ній протоколів обміну інформацією. Доведено актуальність даного напрямку в побудові мереж.

Досліджено проблеми та досягнення багатошляхової маршрутизації в програмно-конфігурованих мережах, з акцентом на покращенні якості обслуговування (QoS), оптимізації балансування навантаження та забезпеченні ефективного використання ресурсів.

Аналіз висвітлив існуючі алгоритми та механізми маршрутизації, призначені для QoS і балансування навантаження (джерела [3, 4, 5 і 7]), а також адаптивні підходи до оптимізації (джерела [13, 15, 18 і 19]).). Крім того, дослідження щодо масштабованості SDN, контролю потоку та проектування трафіку (джерела [9, 11, 16 і 17]) надали повне уявлення про поточні проблеми та рішення.

Незважаючи на те, що ці дослідження підкреслюють значний прогрес у маршрутизації та управлінні ресурсами на основі SDN, більшість методів або зосереджені на великомасштабних системах, або їм бракує адаптивності, необхідної для малих систем реального часу. Сучасні підходи часто ігнорують комбіновану оптимізацію затримки та пропускної здатності в обмежених середовищах, що є критичним для програм реального часу. Крім того, існуючим методам багатошляхової маршрутизації часто важко підтримувати ефективність у динамічних сценаріях із обмеженими ресурсами.

Ця прогалина в наявних рішеннях підкреслює необхідність розробки нового алгоритму для багатошляхової маршрутизації на основі SDN. Такий алгоритм повинен ефективно вирішувати проблеми затримки та оптимізацію пропускної здатності мережі, залишаючись адаптованим до невеликих систем реального часу. На основі інформації, отриманої в результаті даного аналізу, буде запропоновано

алгоритм спрямований на вирішення даної проблеми, забезпечуючи індивідуальне рішення для ефективного використання ресурсів, низької затримки та високої пропускної здатності в обмежених середовищах. Тому вважаю доцільним подальший розвиток та оцінку нового способу маршрутизації, представленого в цій дисертації.

РОЗДІЛ 2
СПОСІБ БАГАТОШЛЯХОВОЇ МАРШРУТИЗАЦІЇ В ПРОГРАМНО-КОНФІГУРОВАНИХ МЕРЕЖАХ

У цьому розділі буде розглянуто розробку нового алгоритму багатошляхової маршрутизації в програмно-конфігурованих мережах (SDN). Запропонований алгоритм використовує принципи пошуку в глибину (DFS) для виявлення потенційних шляхів між мережевими вузлами, і має на меті забезпечити відповідність високим вимогам щодо ефективності та стійкості мережі. Через постійне ускладнення і збільшення мереж з’являється необхідність визначити методи маршрутизації, які не тільки підтримують швидку передачу даних, але й зберігають мінімальну затримку між шляхами. Отже, спосіб буде заточений на оптимізацію двох критично важливих мережевих показників — пропускної здатності та затримки — щоб вибрати найоптимальніший на даний момент шлях зі знайдених.

Підхід запропонованого способу маршрутизації складається з етапів:

- вичерпне дослідження доступних маршрутів, де має забезпечуватись повне, і, що важливо, швидке виявлення шляхів у складних мережевих топологіях.
- прорахунок метрик знайдених маршрутів
- відбір найоптимальнішого шляху

Далі детальніше буде розглянуто теоретичні основи модифікованого підходу до маршрутизації, показано як інтеграція показників пропускної здатності та затримки може суттєво вплинути на вибір шляху в програмно-конфігурованих мережах.. Даний спосіб має на меті забезпечувтаи баланс між оптимальним використанням ресурсів і чудовою ефективністю мережі. У цьому розділі буде

закладено теоретичну основу для практичної реалізації модифікованого способу маршрутизації і подальшого дослідження його ефективності.

2.1. Огляд поточних проблем у багатошляховій маршрутизації програмно-конфігурованих мереж

Маршрутизація в програмно-конфігурованих мережах, а в особливості – багатошляхова маршрутизація, стикається з багатьма унікальними проблемами, що потребують нових рішень, покликаних забезпечувтаи ефективну передачу даних в мережі. Навідміну від традиційної моделі мережі з фіксованими апаратно визначеними шляхами, котра все гірше може справлятися з сучасними викликами через архітектурні обмеження – програмно-конфігурована мережа має для цієї задачі всі сучасні і потрібні інструменти. Програмно-визначена мережа забезпечує відокремлену площину керування від площини даних, що надає великі можливості для контролю маршрутизації. Але ця можливість несе за собою і певні проблеми, які потрібно вирішувати.

Найпершою проблемою є проблема зростання мережі, що робить її складною і створює значне навантаження на мережеві контролери, що відповідають за керування з’єднаннями у ній. В свою чергу багатошляхова маршрутизація, що передбачає під собою виявлення та підтримку кількох шляхів між джерелом і місцем призначення, ще сильніше ускладнює цю мережу та створює більше навантаження на контролери; адже управління цими шляхами є вимогливим до обчислень. Тож використання ефективних способів пошуку шляхів в масштабованих мережах є дуже важливим фактором в забезпеченні високої якості та ефективності мережі.

Важливим фактором еективності мережі є значення затримки в обміні даними. Багатошляхова маршрутизація в своїй основі націлена покрашувати

продуктивність мережі, шляхом розподілення навантаження за маршрутами, однак затримки в мережі можуть спричиняти до проблем, що погіршують якість обслуговування QoS в задачах, чутливих до затримок в передачі даних, таких як потокова передача даних в реальному часі, VoIP. Тому традиційні алгоритми, що не враховують даний показник, можуть не підходити для такого типу задач.

Неоптимальний вибір шляху є ще однією з проблем багатошляхової маршрутиації на яку критино важливо звертати увагу. Поганий маршрут, що не підходить для вимог передачі даних в контексті даної мережі, може спричинити погану якість обслуговування. Тому важливо розробляти оптимальні способи відбору маршрутів під кожну окрему задачу.

Зменщення обчислювальних витрат контролера в програмно-конфігурованих мережах є ще однією фундаментальною проблемою. Використання складних алгоритмів пошуку оптимальних маршрутів спричинятиме до перевантаження контролера, що керує трафіком в мережі і, відповідно, виникнення збоїв у ній.

Звідси виникає наступна проблема – менеджмент збоїв у мережі. Алгоритми багатошляхової маршрутизації повинні враховувати наявність мережевих збоїв і адаптовуватися до них для забезпечення належної відмовостійкості і якості обслуговування мережі.

Також варто звернути увагу на безпекові проблеми програмно-конфігурованої мережі, адже найвність програмно-конфігурованого контролера відкриває додаткові можливості для втручання в керування трафіком і відповідних ризиків безпеки. Ідеальний спосіб маршрутизації мав би інтегрувати заходи безпеки, що зводили б до мінімуму такі можливості.

В своїй дисертації я прагнутиму розробити модифікований спосіб маршрутизації, зосередившись на вирішенні певних проблем для задач передачі даних в реальному часі, VoIP де важливі показники великої пропускної здатності та малих затримок мережі.

2.2. Огляд метрик програмно-конфігурованої мережі

Відбір оптимального шляху в мережі має орієнтуватись на якісь вхідні дані, за якими можна було б оцінювати певні знайдені маршрути. Такими даними в програмно-конфігурованих мережах є її метрики. Серед них варто звернути увагу на ті, що є найбільш критичними у забезпечені якості обслуговування мережі: пропускна здатність (bandwidth), затримка (delay), кількість переходів. Ці метрики впливають як на продуктивність мережі, так і на надяйність обміну даними між відправником та отримувачем.

2.2.1 Пропускна здатність

Пропускна здатність є максимально можливою швидкістю передачі даних на шляху, що вимірюється в бітах на секунду (біт/с). В контексті програмно-визначених мереж пропускна здатність є важливою метрикою вибору шляху для досягнення ефективного потоку даних та уникнення перевантажень на шляхах.

Доступну пропускну здатність обчислюємо за формулою (2.1):

$$B = \min(B_I), \tag{2.1}$$

де B_i представляє пропускну здатність кожного каналу *i* на заданому шляху.

Оскільки загальна пропускна здатність шляху обмежена посиланням із найменшою пропускною здатністю, цей показник у формулі (2.1) визначається мінімальною пропускною здатністю каналу вздовж маршруту.

2.2.2 Затримка

Затримка в програмно-конфігурованих мережає є часом, що потрібний пакету даних для проходження від джерела до місця призначення. Певні задачі, що виникають в програмно-визначених мережах вимагають якнайменших затримок в мережі. Вимірюється в секундах.

Затримка обчислюється за формулою (2.2):

$$D = \sum_{i=1}^{n} D_I, \tag{2.2}$$

де D_i представляє затримку кожного каналу *i* на заданому шляху;

n – загальна кількість каналів на шляху.

Затримка є комплексною метрикою. Вона може бути розбита на затримку розповсюдження (час, необхідний для поширення сигналу вздовж каналу зв'язку), затримку передачі (час, необхідний для надсилання всіх бітів пакету на канал), затримку в черзі (час, витрачений на очікування в чергах уздовж шляху). і затримку обробки (час, витрачений на обробку заголовків пакетів).

2.2.3 Кількість переходів

Кількість переходів є метрикою, що відображає кількість мережевих пристроїв (комутатори, маршрутизатори), котрі долає пакет даних на шляху від джерела до місця призначення. Потенційно, шлях з меншою кількістю переходів може означати меншу кількість проміжних етапів обробки, відповідно – меншу затримку. Також, потенційно, меншу кількість збоїв, що можуть виникнути на шляху.

Кількість переходів обчислюється за формулою (2.3):

$$H = \sum_{i=1}^{n} 1, \tag{2.3}$$

де 1 представляє перехід між каналом *i* та каналом (*i+1)* на заданому шляху;

n – загальна кількість каналів на шляху.

Зменшення кількості переходів може призвести до більш прямих маршрутів, потенційно покращуючи затримку та використання пропускної здатності мережі.

2.2.4 Втрата пакетів

Втрата пакетів є критичною метрикою, що відображає показник кількості пакетів, які не доставляються до місця призначення. Причини втрати пакетів можуть бути різними, як перевантаження мережі, та к і різні інші помилки передачі; що критично відображається на покзнику якості обслуговування мережі. Втрата пакетів є дуже шкідливою для задач систем реального часу, VoIP.

Втрата пакетів обчислюється за формулою (2.4):

$$P_{loss} = \frac{P_{sent} - P_{received}}{P_{sent}} \times 100\%, \quad (2.4)$$

де P_{sent} представляє кількість відправлених пакетів;

$P_{received}$ представляє кількість успішно отриманих пакетів в місці призначення.

Показник втрати пакетів має бути якнайнижчим, оскільки втрата пакетів призводить до їх повторної передачі, що збільшує затримки мережі та знижує пропускну здатність. В програмно-конфігурованих мережах можна пом’якшити цей показник шляхом вибору менш перевантажених шляхів.

2.3 Алгоритм пошуку маршрутів

Алгоритм багатошляхової маршрутизації, розроблений у цьому дослідженні, забезпечує надійне рішення пітдримки високої якості обслуговування мережі

методом ідентифікації кількох можливих шляхів між початковим і кінцевим вузлом, дотримуючись обмежень пропускної здатності та затримки мережі. Цей алгоритм інтегрує пошук у глибину (DFS) для виявлення шляху з налаштованим методом оцінки для визначення оптимального шляху, збалансовуючи ключові показники, такі як пропускна здатність, затримка. Підхід призначений для балансування між значеннями мінімально можливої затримки та максимально можливої пропускної здатності, для застосування в системах реального часу. Далі описано етапи алгоритму.

Алгоритм поділяється на два основних етапи:

1. Визначення кількох непересічних шляхів від джерела до пункту призначення, використовуючи пошук у глибину для дослідження всіх можливих шляхів між вершинами.
2. Вибір оптимального шляху на основі показників, де серед усіх можливих шляхів алгоритм застосовує систему оцінки, щоб визначити найефективніший шлях на основі нормалізованих показників. Ці показники включають пропускну здатність, затримку мережі, співвідношення яких об’єднується в остаточну оцінку для вибору оптимального маршруту.

На вхід алгоритм приймає граф мережі, ідентифікатор вузла джерела та ідентифікатор вузла призначення. Граф мережі складається з вузлів, з’єднаних ребрами, кожен з яких містить інформацію про пропускну здатність, затримку та вагу переходу. Вага переходу приймається за одиницю і еквівалентна хопу.

Вхідні параметри формально визначаються таким чином:

- Граф - представлення мережі, де вузли пов’язані ребрами з атрибутами пропускної здатності, затримки та ваги переходу.
- Ідентифікатори вузлів джерела та призначення.

2.3.1 Перший етап алгоритму

Виявлення шляху за допомогою пошуку в глибину (DFS). Алгоритм досліджує всі можливі шляхи від джерела до призначення, виконуючи рекурсивний обхід сусідніх вузлів, доки не буде відшукано дійсний шлях або поки не буде вичерпано всі вершини.

Формально описати можна так:

1. Починається з вихідного вузла та досліджує кожен суміжний вузол, який не було відвідано в межах поточного шляху.
2. Додає знайдені шляхи до набору можливих для маршрутизації шляхів.
3. Зворотній шлях після досягнення кінцевого вузла або вичерпання всіх суміжних вузлів, гарантуючи відсутність повторного перегляду в межах одного шляху для збереження роз'єднаності шляху.

Псевдокод:

1. Start: ініціалізація процесу маршрутизації за допомогою ідентифікаторів вузлів джерела та призначення, наданих контролером SDN.

2. ** Ініціалізація даних графа**:

- Отримання вузлів `джерело` і `призначення` з мережевого графа.

- Створення порожнього списку `шляхів` для зберігання всіх виявлених шляхів між `джерелом` і `призначенням`.

3. **Перевірка вузлів**:

- ЯКЩО вузли `джерело` або `призначення` не знайдено на графіку ТО

- Припинити пошук і ПОВЕРНУТИ порожній список шляхів, оскільки шляхи не можуть бути встановлені.

4. ** Виявлення шляху DFS**:

- Визначити порожній набір `відвіданих`, щоб відстежувати вузли на поточному шляху, з метою уникнення циклів.

- ВИКЛИК `findPathsDFS(джерело, призначення, [], шляхи, відвідані)`:

- Ця функція рекурсивно досліджуватиме шляхи з використанням DFS, щоб знайти всі непересічні маршрути між `джерелом` і `призначенням`.

5. ** Рекурсивне виявлення шляху DFS**:

ФУНКЦІЯ `findPathsDFS(поточний, призначення, шлях, шляхи, відвідані)`:

- Додати `поточний` вузол до `шляху` та позначте його як відвіданий.
- ЯКЩО `поточний` вузол дорівнює `призначенню`, ТОДІ:
  - Знайдено повний шлях від `джерела` до `призначення`.
  - Збереження шляху до `шляхи`.
- ІНАКШЕ:
  - ДЛЯ кожного «ребра», підключеного до «поточного» вузла:
    - ЯКЩО кінцевий вузол `ребра` не знаходиться в `відвіданому`:
      - РЕКУРСИВНО ВИКЛИКАТИ `findPathsDFS` з наступним вузлом і оновленими `шлях` і `відвідані`.
  - видалити `поточний` вузол із `шлях` і позначити його як невідвіданий.

6. **Кінець DFS**:

- Після завершення DFS усі можливі непересічні шляхи від `джерела` до `призначення` зберігаються в `шляхи`.

2.3.2 Другий етап алгоритму

На цьому етапі алгоритм аналізує виявлені шляхи та вибирає найбільш оптимальний методом обчислення оцінки для кожного шляху на основі показників пропускної здатності та затримки.

Формально описати можна так:

1. Нормалізація показників, де проводиться обчислення значення максимальної затримки та максимальної пропускної здатності по всіх шляхах, на основі яких проводиться нормалізація значень у всіх наявних маршрутах.

2. Для кожного шляху розраховується загальна затримка та мінімальна пропускна здатність. Потім ці значення нормалізуються та об’єднуються в оцінку, де шляхи з вищою пропускною здатністю та меншою затримкою дають вищі оцінки.
3. Шлях з найвищим значенням оцінки вибирається як оптимальний маршрут.

Псевдокод:

7. ** Оцінка шляху та вибір оптимального шляху**:

ФУНКЦІЯ `selectOptimalPath(шляхи, граф)`:

- Ініціалізація `оптимальний шлях` як NULL і `краща оцінка` як -∞.

- Визначення `максимальна затримка` і `максимальна пропускна здатність` як -∞, щоб зберегти максимальні значення для нормалізації.

- **Крок 1**: обчислити максимальну затримку та пропускну здатність:

- ДЛЯ кожного `шлях` в `шляхи`:

- Обчислити `загальна затримка` і `мінімальна пропускна здатність` для кожного шляху:

- ДЛЯ кожного ребра в `шлях` додати `затримка` ребра до `загальна затримка`.

- Відстежити мінімальну пропускну здатність між ребрами.

- Оновити `максимальна затримка` і `максимальна пропускна затність` на максимальні знайдені значення.

- **Крок 2**: оцінка та вибір оптимального шляху:

- ДЛЯ кожного `шлях` в `шляхи`:

- Обчислити `загальна затримка` і `мінімальна пропускна здатність` для шляху.

- Нормалізація `загальна затримка` і `мінімальна пропускна здатність` за допомогою `максимальна затримка` і `максимальна пропускна здатність`.

- Обчислити оцінку на основі нормалізованих значень, обираючи низьку затримку та високій пропускну здатності.

- ЯКЩО оцінка поточного шляху перевищує `краща оцінка `, оновити `краща оцінка` і встановіть `оптимальний шлях` на поточний шлях.

8. **Повернути результат**:

- ПОВЕРНУТИ `оптимальний шлях` як найкращий маршрут для передачі даних.

Визначення оптимального шляху є ключовим моментом в розробленому способі багатошляхової маршрутизації, адже цей етап є відповідальним за ефективий обмін даними в мережі.

Нормалізація значень є важливим етапом задля коректного порівняння значень з різними одиниці виміру. В розробленому методі нормалізація застосовується до значень пропускної здатності та затримки кожного шляху, щоб гарантувати, що ці показники, незважаючи на різні одиниці вимірювання та масштаби, можна безпосередньо порівнювати та зважувати у формулі оцінки.

На формулі (2.5) наведено процес нормалізації мінімальної пропускної здатності, що доступна на шляху, що оцінюється.

$$normalizedBandwidth(P) = \frac{\min(Bandwidth(P))}{\max(Bandwidth)}, \quad (2.5)$$

де P – шлях, що розглядається;

normalizedBandwidth(P) — нормалізована пропускна здатність для шляху P;

min(Bandwidth(P)) — мінімальна пропускна здатність на шляху P;

max(Bandwidth) — максимальна пропускна здатність серед усіх знайдених шляхів у графі мережі.

На формулі (2.6) наведено процес нормалізації затримки на шляху, що оцінюється.

$$normalizedDelay(P) = \frac{totalDelay(P)}{\max(Delay)}, \quad (2.6)$$

де P – шлях, що розглядається;

normalizedDelay(P) — нормалізована затримка для шляху P;

totalDelay(P) — сумарна затримка на шляху P;

max(Delay) — максимальне значення затримки серед усіх знайдених шляхів у графі мережі.

На формулі (2.7) наведено процес оцінювання кожного знайденого шляху.

$$Score(P) = \frac{normalizedBandwidth(P) \times \alpha}{normalizedDelay(P) \times \beta}, \quad (2.7)$$

де P – шлях, що розглядається;

Score(P) — показник оптимальності маршруту P;

normalizedBandwidth(P) — нормалізована пропускна здатність для шляху P;

normalizedDelay(P) — нормалізована затримка для шляху P;

α - ваговий коефіцієнт для пропускної здатності;

β - ваговий коефіцієнт для затримки.

Вагові коефіцієнти у формулі (2.7) дозволяють точно налаштувати вплив кожної метрики на загальне значення оцінки кожного шляху для індивідуальних потреб кожної мережі. Тобто можна зробити значення затримки пріоритетнішим за пропускну здатність і навпаки.

Далі, на для графа мережі, зображеного на рис. 2.1, проведемо маршрутизацію від точки С1 до точки С23 з використанням розробленого способу багатошляхової маршрутиації.

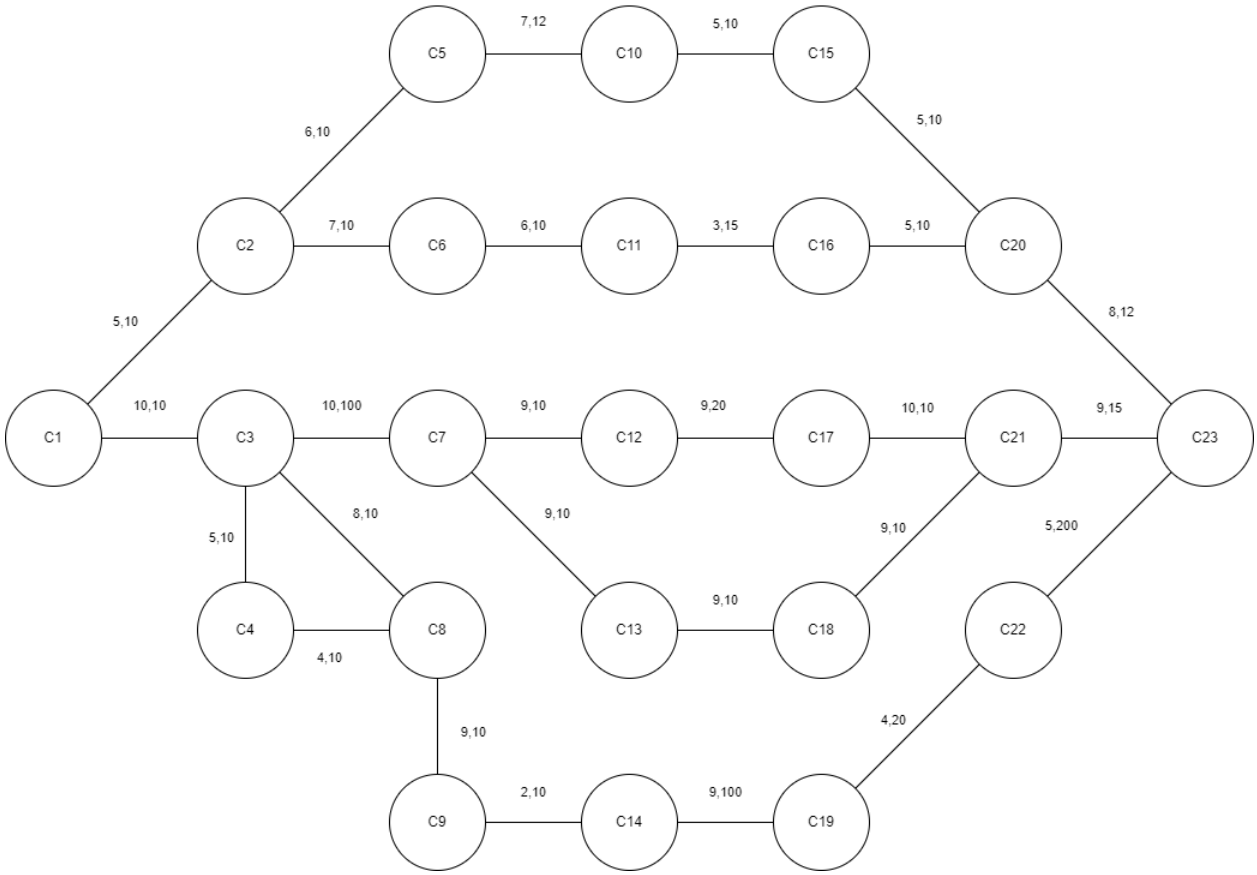

Рисунок 2.1 – Граф мережі з метриками

В таблиці 2.1 показано всі, знайдені алгоритмом, маршрути з підсумованими метриками між вершинами С1, С23.

Таблиця 2.1

Знайдені маршрути з метриками

№	Маршрут	Пропускна здатність	Затримка
1	С1→С2→С5→С10→С15→С20→С23	5	64

Таблиця 2.1 (продовження)

Знайдені маршрути з метриками

2	C1→C2→C6→C11→C16→C20→C23	3	67
3	C1→C3→C7→C12→C17→C21→C23	9	165
4	C1→C3→C7→C13→C18→C21→C23	9	155
5	C1→C3→C8→C9→C14→C19→C22→C23	2	360
6	C1→C3→C4→C8→C9→C14→C19→C22→C23	2	370

Проведено оцінку кожного маршруту для визначення оптимального. Вагові коефіцієнти α та β приймемо за 1. В таблиці 2.2 наведено результат.

Таблиця 2.2

Оцінка маршрутів

№ маршруту	Нормалізована пропускна здатність	Нормалізована затримка	Оцінка
1	0.56	0.17	3.29
2	0.33	0.18	1.83
3	1	0.45	2.22
4	1	0.42	2.38
5	0.22	0.97	0.23
6	0.22	1	0.22

Як можна побачити з результатів в таблиці 2.2 – найоптимальнішим виявився маршрут під номером 1 (C1→C2→C5→C10→C15→C20→C23), котрий з поміж інших має найкраще співвідношення значень пропускної здатності і затримки, що робить його найоптимальнішим вибором. В разі його недоступності, використовувати можна маршрут під номером 4 (C1→C3→C7→C13→C18→C21→C23) і так далі.

Висновок до розділу 2

Розроблено та детально описано комплексний метод багатошляхової маршрутизації в програмно-конфігурованих мережах. Систематично розглянуто фундаментальні аспекти та закладено основу для впровадження алгоритму та його подальшого дослідження.

Розглянуто поточні проблеми, пов’язані з багатошляховою маршрутизацією в програмно-визначених мережах. Описано обмеження традиційних методів маршрутизації – їх неефективність у динамічній обробці сучасного трафіку. Нездатність наявних рішень адаптовуватись під певні метрики мережі. Аналіз підкреслив необхідність розробки гнучкого та адаптивного алгоритму для вирішення цих проблем.

Розглянуто основні метрики програмно-конфігурованої мережі, їх важливість у забезпеченні якості обслуговування. Виділено дві з них: пропускну здатність,як критично важливий фактор в здатності мережі обробляти трафік, та затримку, як фактор, що є критичним для чутливих до швидкості відклику задач; як такі, під які розробляється рішення, що запропоноване в цій дисертації. Наведено формули розрахунку даних метрик.

Представлено модифікований спосіб багатошляхової маршрутизації, що складається з двох етапів.

Першим етапом є пошук у глибину (DFS) для визначення всіх можливих шляхів між джерелом і місцем призначення в мережі. Що гарантує ретельне вивчення потенційних варіантів маршрутизації.

Другим етапом є уточнення необхідного шляху, шляхом оцінки ефективності кожного з них для конкретно визначеної ситуації за допомогою нормалізованих показників, балансуючи пропускну здатність і затримку для вибору найбільш оптимального маршруту.

Представлено псевдокод алгоритму, пояснено техніку нормалізації показників, наведено формулу визначення оцінки маршрутів.

Перевагами запропонованого методу є використання мережевих показників в реальному часі, що дозволяє йому адаптовуватись до змін в трафіку та топології мережі. Використання DFS гарантує, що алгоритм може обробляти великомасштабні мережі з різними параметрами маршрутизації. Використання пропускної здатності і затримки в механізмі оцінки забезпечує збалансовану оптимізацію, придатну для систем реального часу. Дизайн алгоритму дозволяє подальше вдосконалення шляхом інтеграції додаткових метрик в механізм оцінки маршруту.

Наступним кроком буде реалізація розробленого способу багатошляхової маршрутизації у програмному забезпеченні для подальших перевірок, з метою оцінки ефективності такого рішення.

РОЗДІЛ 3
ПРАКТИЧНА РОЗРОБКА МОДИФІКОВАНОГО СПОСОБУ БАГАТОШЛЯХОВОЇ МАРШРУТИЗАЦІЇ

Практична реалізація здійснюватиметься за допомогою моделювання мережі, призначеного для оцінки продуктивності алгоритму в керованому програмно-визначеному мережевому середовищі.

Вирішено розробляти симуляцію програмно-конфігурованого середовища, з метою дослідження ефективності алгоритму.

Далі детально буде описано вибір інструментів розробки, описано програмні компоненти застосунку.

3.1 Вибір інструментів розробки

Для реалізації модифікованого алгоритму багатошляхової маршрутизації у формі симуляції мережі було обрано мову програмування Java, асемблер Gradle та специфічні бібліотеки — jfreechart, javafx-base, javafx-controls, javafx-fxml та javafx-graphics.

3.1.1 Мова програмування Java

Мову програмування Java обрано в якості основної і єдиної мови для розробки через її незалежність від платформи виконання, надійність, широкий вибір потрібних бібліотек та інших інструментів. Також через особисті навички володіння даною мовою програмування.

ООП дизайн даної мови полегшує модульну реалізацію алгоритму, забезпечуючи чітке представлення елементів мережі у симуляції (наприклад, вузлів, ребер і шляхів).

Вона надає вбудовані бібліотеки для керування мережевим зв’язком і обчисленнями, пов’язаними з графіками, що скорочує час розробки.

Ефективне збірники сміття тамеханізми керування пам’яттю нададуть можливість створення симуляції великої мережі без жорстких вимог до апаратного забезпечення.

Популярність даної мови зумовлює наявність величезної кількості документації та варіантів рішень проблем, з якими можна стикнутись під час розробки продукту.

Сукупність озвучених факторів робить цілком логічним вибір даної мови програмування для розробки програмного продукту.

3.1.2 Збірник Gradle

Gradle обрано в якості гнучкого і ефективного інструменту для збирання проекту. Він спрощує конфігурацію проекту і керування залежностями в ньому. Надає можливість поступового підключення необхідних сторонніх бібліотек, що значним чином полегшує розробку і надає необхідну гнучкість під час створення програмного продукту.

3.1.3 Бібліотека jfreechart

Важливою складовою в оцінці розроблего способу багатошляхової маршрутизації в мережах SDN є візуалізація результатів. Саме для цього завдання обрано бібліотеку jfreechart, яка надає потужні можливості для створення якісних діаграм.

ЇЇ можливості буде використано для відображення метрик мережі в симуляції за різниї обставин, порівняння обраних шляхів і відображення статистики. Обрана бібліотека сприятиме кращій інтерпретації та представленню результатів.

3.1.4 Бібліотеки JavaFX

З метою створення інтерактивного інтерфейсу користувача для симуляції обрано набір бібліотек JavaFX, включаючи javafx-base, javafx-controls, javafx-fxml і javafx-graphics

Сукупність цих бібліотек надає різноманітні готові елементи керування інтерфейсом користувача, такі як кнопки, повзунки та діаграми, що дозволяє створювати інтуїтивно зрозуміле та зручне середовище моделювання.

Дозволяє детальну та динамічну візуалізацію топології мережі з такими функціями, як оновлення в реальному часі шляхів маршрутизації та станів моделювання.

Інтеграція FXML спрощує дизайн і розробку макета інтерфейсу користувача, полегшуючи відокремлення презентації від логіки.

Забезпечує узгоджену поведінку інтерфейсу користувача в різних операційних системах.

3.2 Компоненти програми

Програма моделювання для реалізації модифікованого алгоритму багатошляхової маршрутизації структурована на кілька основних компонентів, кожен з яких виконує певну роль у процесі розробки та аналізу. Ці компоненти створені для бездоганної взаємодії, забезпечуючи комплексне середовище для тестування та оцінки алгоритмів маршрутизації.

Розроблену програму умовно можна поділити на складові

- Конструювання графа мережі
- Відображення графа мережі
- Генерація топології мережі
- Маршрутизація розробленим алгоритмом

- Маршрутизація алгоритмом Дейкстра
- Симуляція трафіку
- Генерація і відображення діаграм

Підсумовуючи, створений застосунок симулює роботу програмно-конфігурованої мережі, в межах, достатніх для розуміння і відображення його роботи і ефективності. Надає можливість порівняння з наявним рішенням алгоритму Дейкстра.

3.2.1 Компонент репрезентації графа мережі

Мета цього компоненту - забезпечити надійну та розширювану структуру для представлення мережі у вигляді графа. Вузли на графі представляють мережеві пристрої (наприклад, комутатори, маршрутизатори), тоді як ребра представляють канали зв'язку з різними метриками. Цей граф служить основою для реалізації та тестування алгоритмів маршрутизації в симуляції. На рис. 3.1 зображено бізнес-логіку даного компоненту.

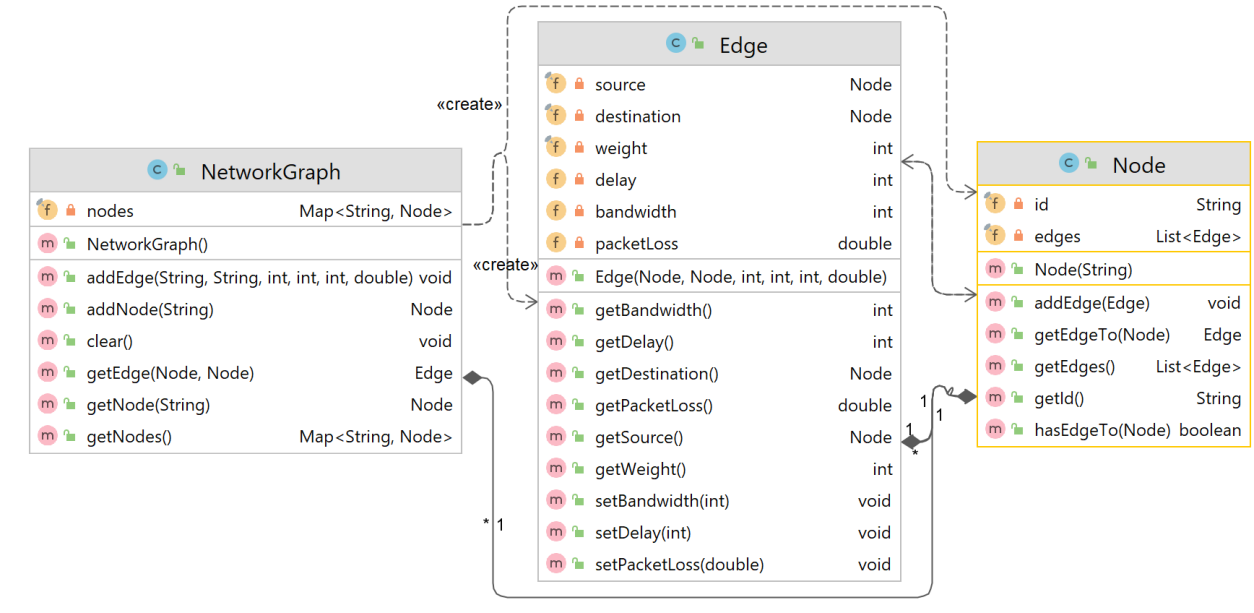

Рисунок 3.1 – Компонент репрезентації графа мережі

Клас *NetworkGraph* представляє загальну мережу як набір взаємопов'язаних вузлів.

Атрибути:

- nodes: словникова структура для зберігання всіх вузлів у графі з ідентифікаторами вузлів як ключами та об'єктами вузлів як значеннями.

Ключові методи:

- addNode(String id): додає новий вузол до графа, якщо він ще не існує.
- addEdge(String sourceId, String destId, int weight, int delay, int bandwidth, double packetLoss): додає двонаправлене ребро між двома вузлами з певними метриками:
  - Weight: представляє вартість або кількість стрибків.
  - Delay: затримка передавання в мілісекундах.
  - Bandwidth: пропускна здатність зв'язку в Мбіт/с.
  - Packet Loss: ймовірність втрати пакетів на каналі.
- getNode(String id): отримує вузол за його ідентифікатором.
- getEdge(Node source, Node destination): отримує ребро, що з'єднує два вузли.
- clear(): очищає всі вузли та ребра, скидаючи граф.

Клас *Node* представляє один вузол (або вершину) на графі.

Атрибути:

- id: унікальний ідентифікатор для вузла.
- edges: список, що містить усі ребра, підключені до вузла.

Ключові методи:

- addEdge(Edge edge): додає ребро до списку ребер вузла.

- hasEdgeTo(Node destination): перевіряє, чи існує ребро між поточним вузлом і заданим вузлом призначення.
- getEdgeTo(Node destination): отримує ребро, що з’єднує поточний вузол із вказаним вузлом призначення.

Клас *Edge* представляє спрямоване з’єднання (або посилання) між двома вузлами.

Атрибути:

- source: вихідний вузол краю.
- destination: кінцевий вузол ребра.
- weight: вартість, пов'язана з ребром.
- delay: затримка передачі.
- bandwidth: пропускна здатність каналу.
- packetLoss: ймовірність втрати пакета.

Ключові методи:

- Методи отримання та встановлення атрибутів.

Методи addNode і addEdge дозволяють створювати мережеві топології шляхом визначення вузлів і їх взаємозв’язків.

Повторюваних з’єднань можна уникнути за допомогою методу hasEdgeTo у класі Node.

Методи getNode, getEdge і getEdgeTo спрощують запити до графу для конкретних вузлів або посилань, що є важливим для маршрутизації та алгоритмів пошуку шляху.

Метод addEdge гарантує, що кожне посилання є двонаправленим, створюючи два ребра (від джерела до пункту призначення та від пункту призначення до джерела), імітуючи двонаправлені мережеві посилання в реальному світі.

Атрибути ребра, такі як затримка, пропускна здатність і втрата пакетів, можна змінювати під час виконання, що дозволяє симулювати динамічні умови мережі.

3.2.2 Компонент імплементації розробленого способу маршрутизації

Мета цього компоненту – імплементувати розроблений спосіб багатошляхової маршрутизації в симуляції програмно-конфігурованої мережі. На рис. 3.2 зображено бізнес-логіку даного компоненту.

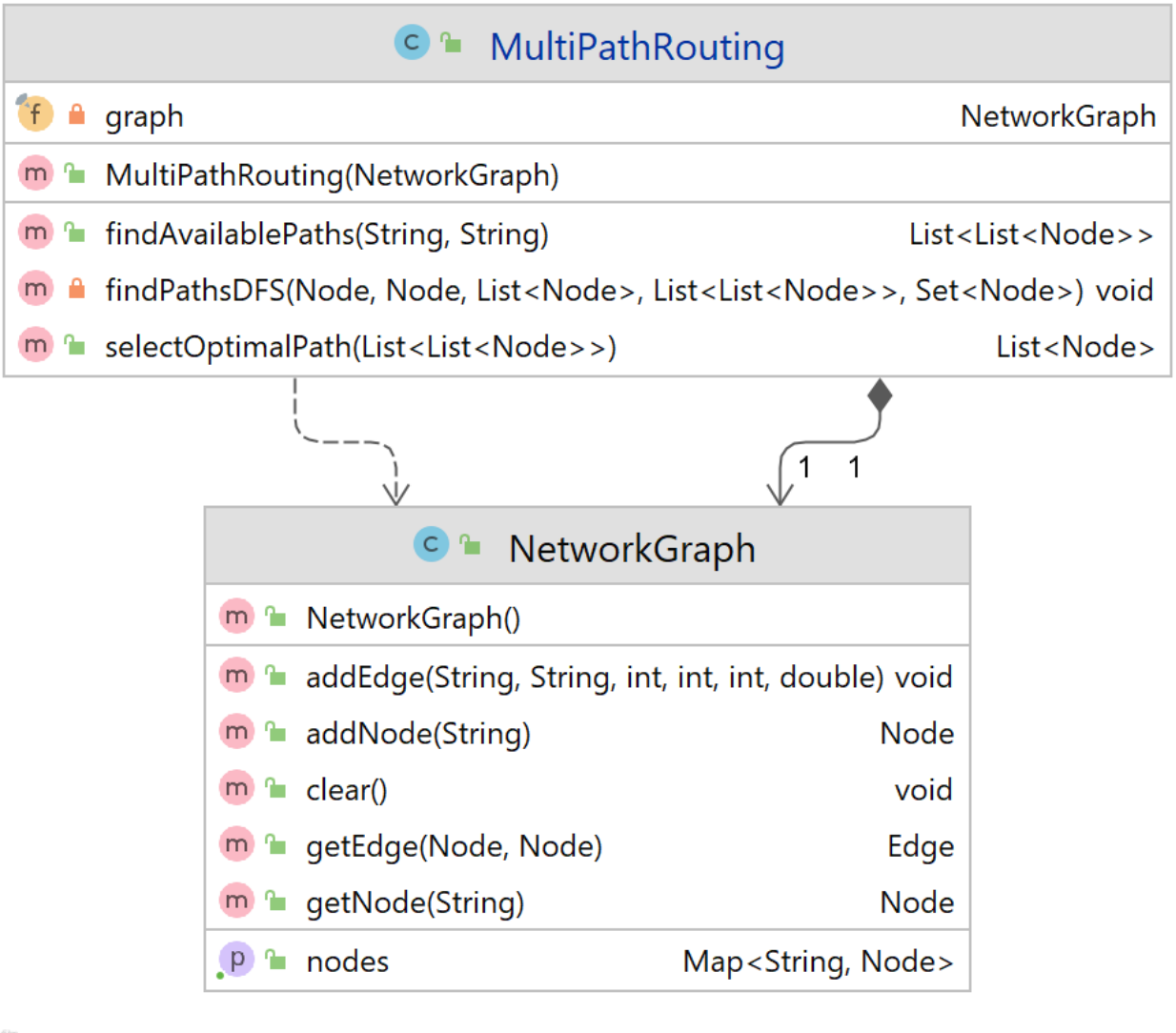

Рисунок 3.2 – Компонент імплементації розробленого способу маршрутизації

Створений для:

1. Визначення всіх доступних шляхів між джерелом і пунктом призначення на графі мережі за допомогою алгоритму пошуку в глибину (DFS).
2. Оцінки виявлених шляхи на основі показників затримки та пропускної здатності, щоб вибрати найбільш підходящий шлях для маршрутизації в системах реального часу.

Атрибути:

- graph: екземпляр класу NetworkGraph, що представляє топологію мережі. Він надає вузли та ребра, необхідні для пошуку шляху та оцінки метрик.

Ключові методи:

- findAvailablePaths(String sourceId, String destinationId) - знаходить усі шляхи між вихідним і кінцевим вузлом за допомогою DFS.
    - Параметри:
        - sourceId: ідентифікатор вихідного вузла.
        - destinationId: ідентифікатор вузла призначення.
    - Результат: список усіх шляхів, де кожен шлях представлено як список об’єктів Node.
    - Процес: використовує допоміжний метод findPathsDFS для виконання рекурсивного обходу DFS.
- findPathsDFS(Node current, Node destination, List<Node> path, List<List<Node>> paths, Set<Node> visited) - рекурсивно досліджує всі шляхи між поточним вузлом і пунктом призначення.
    - Параметри:
        - current: відвідуваний вузол.

        - destination: цільовий вузол.
        - path: шлях, який будується.
        - paths: колекція всіх знайдених шляхів.
        - visited: набір відвіданих вузлів для запобігання циклів.
- selectOptimalPath(List<List<Node>> paths) - вибір найкращого шляху серед виявлених на основі нормалізованих показників затримки та пропускної здатності.
    - Параметри:
        - paths: список шляхів для оцінки.
    - Результат: найоптимальніший шлях у вигляді списку об’єктів Node.
    - Процес: Обчислює показники затримки та пропускної здатності для кожного шляху. Нормалізує показники, щоб забезпечити справедливе порівняння між шляхами. Використовує розроблену формулу (2.7) для оцінки. Вибирає шлях із найвищим балом.

Підхід на основі DFS гарантує, що досліджуються всі можливі шляхи між джерелом і одержувачем. Відвідана множина шляхів гарантує, що шляхи не мають петель

Алгоритм обчислює загальну затримку як суму затримок уздовж усіх ребер шляху; мінімальну пропускна здатність як пропускну здатність найвужчого місця на шляху. Нормалізація застосовується для обробки розбіжностей у масштабі та одиницях вимірювання, забезпечуючи порівнюваність показників.

Формула оцінки визначає пріоритетність шляхів із вищою пропускною здатністю та меншою затримкою. Поєднуючи ці метрики, алгоритм врівноважує суперечливі цілі в вимогах до пропускної здатності та затримки, що робить його придатним для систем реального часу.

3.2.3 Компонент імплементації алгоритму Дейкстра

Мета цього компоненту – імплементувати маршрутизацію за допомогою алгоритма Дейкстра в симуляції програмно-конфігурованої мережі задля порівняння з розробленим способом маршрутизації. На рис. 3.3 зображено бізнес-логіку даного компоненту.

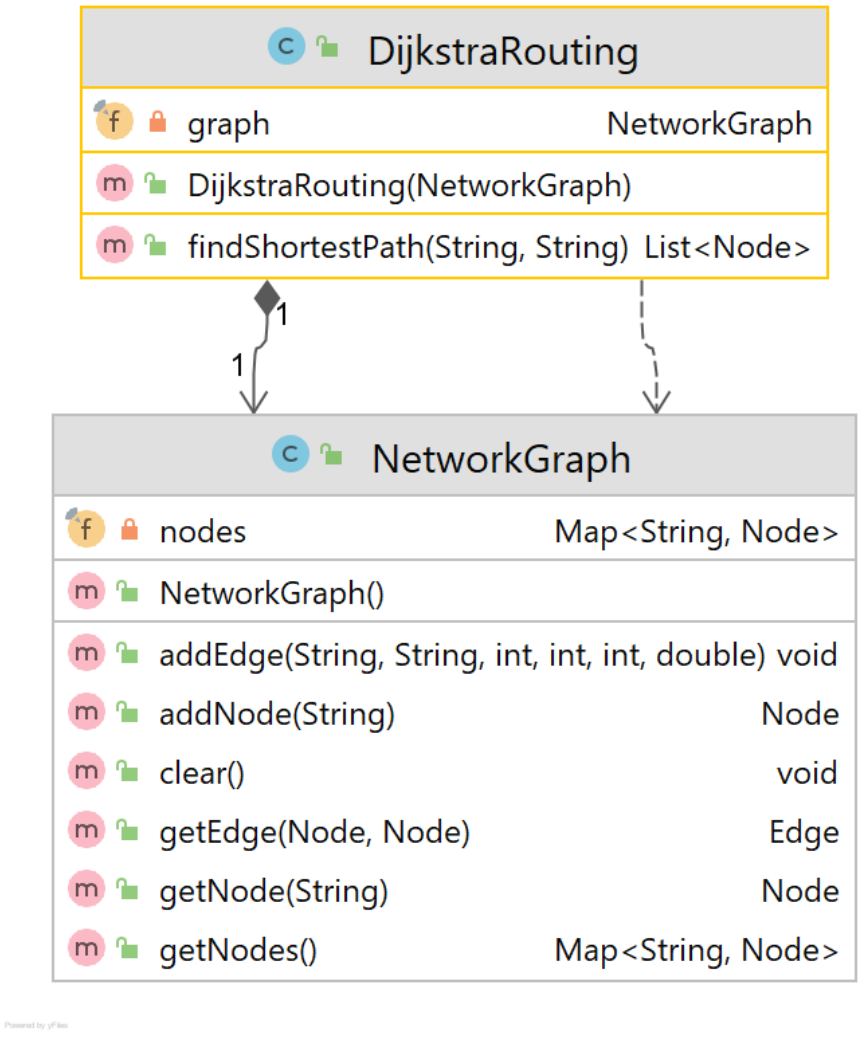

Рисунок 3.3 – Компонент імплементації алгоритму Дейкстра

Створений для обчислення найкоротшого шляху між вузлом джерела та вузлом призначення на основі ваги ребер. Забезпечує вибір шляху з найменшою загальною вартістю з точки зору вагової метрики, що робить його придатним для традиційних сценаріїв маршрутизації.

Атрибути:

- graph: екземпляр класу NetworkGraph, що представляє вузли та ребра мережі.

Ключовий метод:

- findShortestPath(String sourceId, String destinationId) - обчислює найкоротший шлях від вузла джерела до вузла призначення за допомогою алгоритму Дейкстри.
    - Параметри:
        - sourceId: ідентифікатор вихідного вузла.
        - destinationId: ідентифікатор вузла призначення.
    - Результат: список об’єктів Node, що представляють найкоротший шлях. Якщо шлях не існує - повертає порожній список.

Алгоритм реалізовано чотирма етапами. Ініціалізація, обробка вершин, відтворення шляху, вивід результату.

Під час ініціалізації створюється словник відстаней для збереження найкоротшої відомої відстані до кожного вузла, ініціалізується 0 для вихідного вузла та ∞ (представлено як Integer.MAX_VALUE) для інших. Створюється словник попередніх вузлів, щоб відстежувати попередника кожного вузла на шляху. Використовється PriorityQueue, щоб визначити пріоритетність вузлів із найменшою відомою відстанню для обробки.

Під час обробки вершин витягується вузол із найменшою відстанню з черги пріоритету. Для кожного сусіда поточного вузла обчислюється орієнтовна відстань через поточний вузол. Якщо орієнтовна відстань менша за попередньо записану

відстань, оновлюється словник відстаней і словник попередніх вузлів, а потім додається сусід до черги пріоритетів.

Під час відтворення шляху, починаючи з вузла призначення, простежується шлях назад через словник попередніх вузлів. Змінюється зібраний шлях, щоб переконатися, що він упорядкований від джерела до місця призначення.

Далі повертається побудований шлях, якщо він містить принаймні два вузли (джерело та призначення); інакше повертається порожній список.

Клас DijkstraRouting є прикладом традиційного алгоритму пошуку найкоротшого шляху, який широко використовується в мережах. Його включення в структуру симуляції дозволить проводити прямі порівняння з розробленим алгоритмом багатошляхової маршрутизації.

3.2.4 Компонент графічного інтерфейсу

Мета цього компоненту – надати графічний інтерфейс для створення графу мережі, пошуку шляхів між вузлами за допомогою кількох алгоритмів і візуалізації цих шляхів. На рис. 3.4 зображено бізнес-логіку даного компоненту.

Клас *NetworkTesterUI* - графічний інтерфейс користувача (GUI) для тестування генерації графа мережі та алгоритмів пошуку шляхів у програмно-визначених мережах (SDN).

Поля:

- BorderPane root: кореневий макет інтерфейсу користувача.
- GraphVisualizer graphVisualizer: візуальне представлення мережевого графа.
- NetworkGraph graph: логічне представлення мережевого графа.
- TextArea logArea: текстова область для реєстрації подій.
- Logger logger: обробляє журнал шляхів і подій до logArea.

Конструктор NetworkTesterUI() ініціалізує елементи інтерфейсу користувача та налаштовує макет.

Методи:

- BorderPane getRoot(): повертає кореневий макет для вбудовування в програму JavaFX.
- private void setupUI(): налаштовує основні компоненти інтерфейсу користувача та макет, включаючи створення граа, елементи керування пошуком шляху та рядок меню.
- private void generateGraph(int nodeCount, int edgeCount): генерує випадковий мережевий граф із вказаною кількістю вузлів і ребер.
- private void findPath(String sourceId, String destinationId, String algorithm): знаходить шлях між вказаними вузлами за допомогою вибраного алгоритму (MultiPathRouting або Dijkstra).
- private int calculateDelay(List<Node> path): обчислює загальну затримку для заданого шляху.
- private int calculateBandwidth(List<Node> path): обчислює мінімальну пропускну здатність уздовж заданого шляху.
- private void openTestWindow(): відкриває нове вікно для додаткових функцій тестування.
- private void executeTests(): виконує задану кількість тестів на мережевому графі та візуалізує результати за допомогою ChartPlotter.

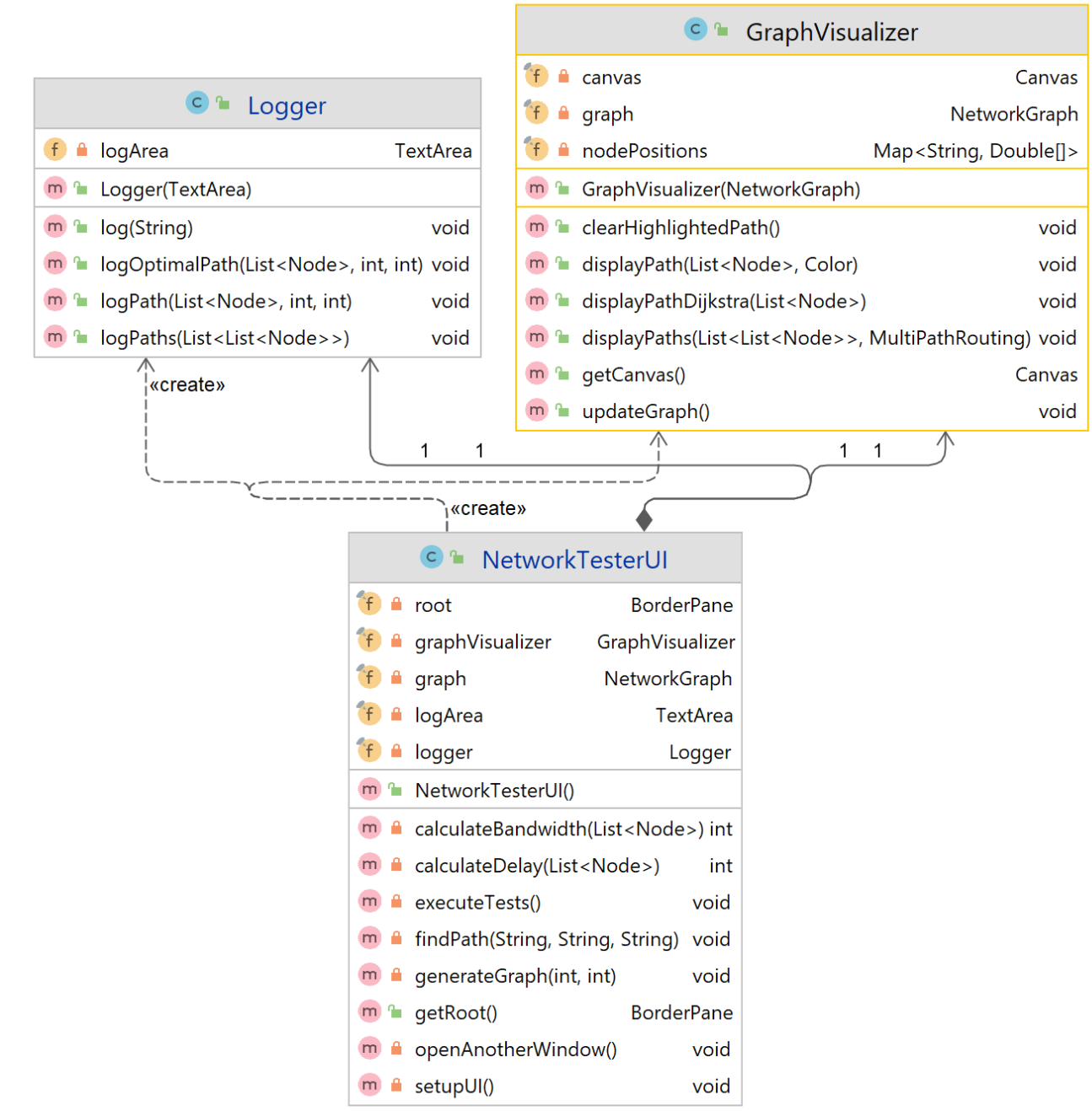

Рисунок 3.4 – Компонент графічного інтерфейсу

Клас *GraphVisualizer* - клас візуалізації мережевого графа.

Поля:

- Canvas canvas: полотно, на якому намальовано граф.
- NetworkGraph graph: логічне представлення мережевого графа.

- Map<String, Double[]> nodePositions: зберігає положення вузлів на полотні.

Конструктор GraphVisualizer(NetworkGraph graph) ініціалізує візуалізатор заданим мережевим графом.

Методи:

- Canvas getCanvas(): повертає полотно для вбудовування в програму JavaFX..
- void updateGraph(): малює поточний стан графа на полотні.
- void displayPaths(List<List<Node>> paths, MultiPathRouting routingAlgorithm): виділяє всі шляхи, знайдені алгоритмом MultiPathRouting, і підкреслює оптимальний шлях.
- void displayPathDijkstra(List<Node> path): підсвічує шлях, знайдений алгоритмом Дейкстри.
- void displayPath(List<Node> path, Color color): виділяє один шлях на полотні вказаним кольором.
- void clearHighlightedPath(): очищає полотно та перемальовує графік без виділення.

Клас *Logger* – клас-утиліта для реєстрації інформації про шляхи та події в інтерфейсі тестера мережі.

Поля:

- TextArea logArea: текстова область, де відображаються логи.

Конструктор Logger(TextArea logArea) ініціалізує логер заданою текстовою областю.

Методи:

- void log(String message): додає повідомлення до області логів.

- void logPath(List<Node> path, int delay, int bandwidth): логує деталі одного шляху, включаючи його затримку та пропускну здатність.
- void logPaths(List<List<Node>> paths): логує декілька шляхів із відповідними послідовностями.
- void logOptimalPath(List<Node> optimalPath, int delay, int bandwidth): логує детальну інформацію про оптимальний шлях, включаючи його затримку та пропускну здатність.

3.2.5 Компонент тестування алгоритмів

Мета цього компоненту – надати необхідні механіми для порівняння розробленого способу багатошляхової маршрутизації і алгоритму Дейкстра. На рис. 3.5 зображено бізнес-логіку даного компоненту.

Клас *ChartPlotter* - відповідає за створення та відображення діаграм порівняння між двома алгоритмами маршрутизації: MultiPath Routing і Dijkstra Routing.

Методи:

- plotComparisonCharts(List<TestResult> multiPathResults, List<TestResult> dijkstraResults) - перевіряє, чи списки вхідних даних для результатів мають однаковий розмір, а потім створює вікно JFrame для відображення діаграм порівняння затримки, пропускної здатності, втрати пакетів і кількості переходів. Використовує допоміжний метод createChartPanel для створення окремих діаграм для кожного показника.
- createChartPanel(String title, String categoryAxisLabel, String valueAxisLabel, List<TestResult> multiPathResults, List<TestResult> dijkstraResults, Function<TestResult, Integer> metricFunction) - створює панель діаграм, що відображає лінійну діаграму для кожного показника, використовуючи дані з multiPathResults і dijkstraResults. Кожна діаграма

базується на відповідних значеннях результатів тестування для затримки, пропускної здатності, втрати пакетів і кількості переходів. Налаштовує набір даних і тип діаграми (лінійна діаграма).

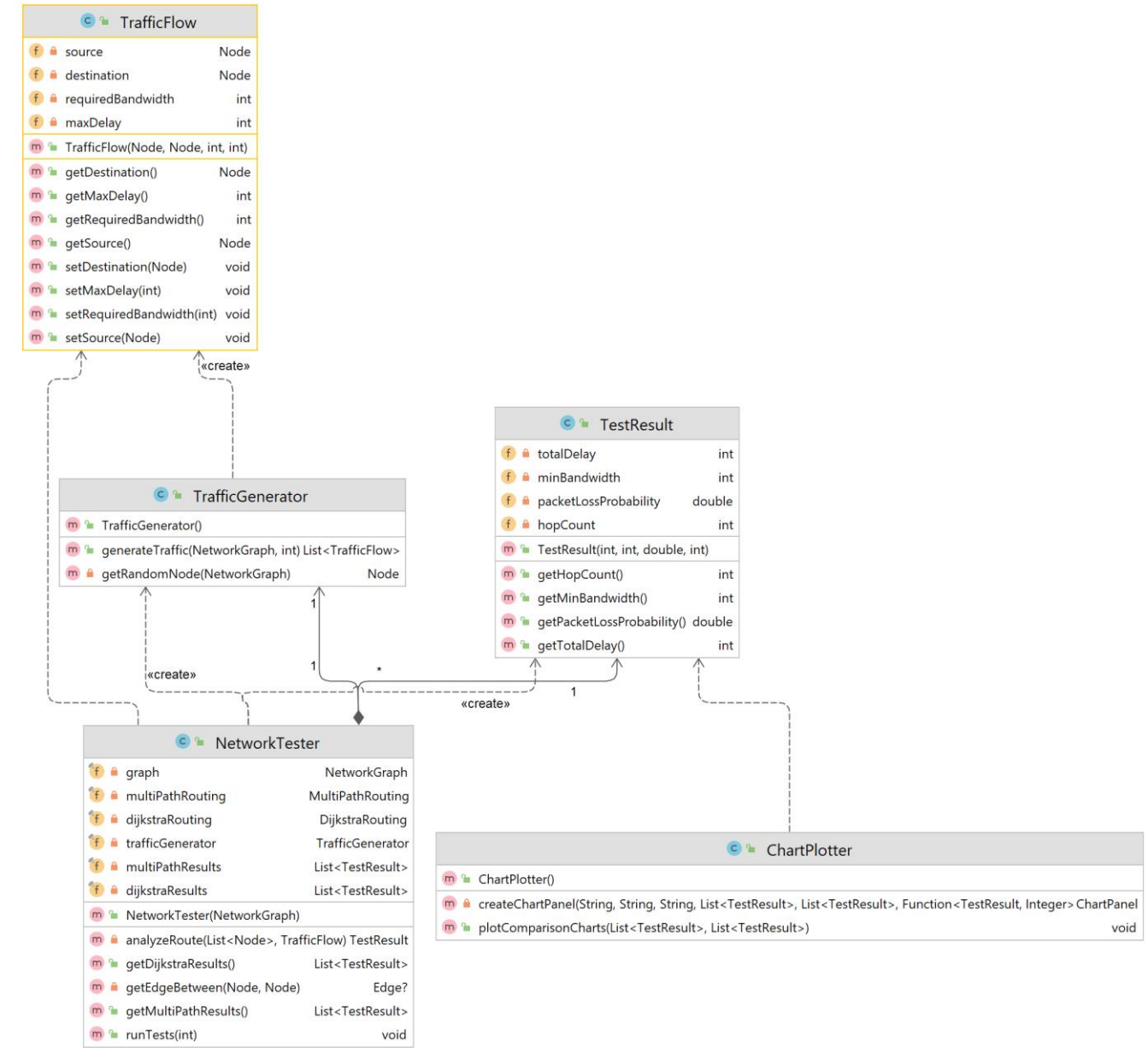

Рисунок 3.5 – Компонент тестування алгоритмів

Клас *NetworkTester* - проводить мережеві тести для обох алгоритмів маршрутизації (MultiPath Routing і Dijkstra) і збирає результати продуктивності для різних потоків трафіку.

Методи:

- runTests(int numberOfFlows) - Генерує задану кількість випадкових потоків трафіку за допомогою TrafficGenerator. Для кожного потоку трафіку маршрути розраховуються за допомогою алгоритмів MultiPath Routing і Dejkstra Routing. Шляхи аналізуються та обчислюються відповідні тестові показники (затримка, пропускна здатність, втрата пакетів, кількість переходів).
- analyzeRoute(List<Node> route, TrafficFlow flow) - Аналізує вибраний маршрут, обчислюючи загальну затримку, мінімальну пропускну здатність, втрату пакетів і кількість переходів. Повертає об’єкт TestResult, який містить ці значення.
- getMultiPathResults() & getDijkstraResults() - Повертає списки результатів для обох алгоритмів, які пізніше використовуються в ChartPlotter для візуалізації даних.

Клас *TestResult* - інкапсулює результати одного тесту для алгоритмів MultiPath Routing або Dejkstra Routing.

Функції:

- Зберігає метрики: totalDelay, minBandwidth, packetLossProbability та hopCount.
- Надає методи отримання доступу до цих значень для створення та аналізу діаграм.

Клас *TrafficFlow* - представляє потік трафіку між вузлом-джерелом і вузлом-приймачем із додатковими параметрами, такими як requiredBandwidth і maxDelay.

Функції:

- Зберігає вузли джерела та призначення потоку трафіку.
- Зберігає вимоги до пропускної здатності та затримки для кожного потоку трафіку.

Клас *TrafficGenerator* - генерує випадкові потоки трафіку на основі мережевого графіка.

Методи:

- generateTraffic(NetworkGraph graph, int numberOfFlows) - випадково генерує numberOfFlows потоків трафіку, кожен із випадковим джерелом, пунктом призначення, необхідною пропускною здатністю та максимальною затримкою.
- getRandomNode(NetworkGraph graph) - довільно вибирає вузол із мережевого графіка.

Висновок до розділу 3

Описано практичну реалізацію модифікованого способу багатошляхової маршрутизації в програмно-конфігурованих мережах. Розроблено комплексну структуру для моделювання та аналізу алгоритмів маршрутизації в програмно-визначених мережах (SDN). Нижче наведено стислий опис ключових кроків, які було здійснено.

Обрано мову програмування Java через її надійність, незалежність від платформи та розгалужену екосистему, що дозволяє ефективно впроваджувати графові структури мережі, алгоритми та графічні інтерфейси.

Обрано Gradle як інструмент для збірки через його гнучкість і бездоганне керування залежностями.

Інтегровано бібліотека jfreechart для забезпечення детального візуального представлення показників продуктивності алгоритму.

Використано набір JavaFX для розробки інтуїтивно зрозумілого графічного інтерфейсу користувача (GUI), що підтримує інтерактивну візуалізацію графів і виконання алгоритмів.

Для моделювання топології SDN створено компонент представлення мережевого графа, що включає вузли, ребра та пов’язані показники (затримка, пропускна здатність і втрата пакетів). Цей компонент забезпечує основу для алгоритмів маршрутизації.

Реалізовано модифікований метод багатошляхової маршрутизації з використанням пошуку в глибину (DFS) для визначення всіх можливих шляхів і вибору найбільш оптимального шляху на основі нормалізованих показників затримки та пропускної здатності.

Реалізовано алгоритм Дейкстра з метою його порівняння з запропонованим у цій дисертації алгоритмом, адже він пропонує надійну точку відліку для маршрутизації за найкоротшим шляхом.

Розроблено компонент графічного інтерфейсу, який дозволяє користувачам інтерактивно створювати графи мережі, виконувати алгоритми маршрутизації та візуалізувати шляхи та показники в реальному часі.

Реалізовано компонент тестування алгоритму для полегшення систематичної оцінки розробленого методу багатошляхової маршрутизації та алгоритму Дейкстра за різних сценаріїв трафіку та топології.

Перевагами розробленого рішення є: модульна конструкція, котра дозволяє легко розширювати програмне забезпечення; підтримка великомасштабних топологій SDN компонентом мережевого графа, що робить рішення максимально подібним до реальних сценаріїв; графічний інтерфейс на основі JavaFX спрощує створення графів, виконання алгоритму та інтерпретацію результатів; інтеграція бібліотеки jfreechart дозволяє детально порівнювати алгоритми; рішення підтримує динамічні оновлення графів і візуалізацію рішень щодо маршрутизації в режимі реального часу.

Розроблене програмне забезпечення забезпечує міцну базу для моделювання та аналізу продуктивності алгоритмів маршрутизації в програмно-конфігурованих мережах. Далі потрібно провести моделювання мережі за допомогою розробленого ПЗ, провести порівняльний аналіз модифікованого алгоритму багатошляхової маршрутизації та алгоритму Дейкстра, оцінюючи їх продуктивність за визначеними метриками.

РОЗДІЛ 4
ТЕСТУВАННЯ МОДИФІКОВАНОГО СПОСОБУ БАГАТОШЛЯХОВОЇ МАРШРУТИЗАЦІЇ

Буде проведено огляд розробленого в попередньому розділі програмного рішення для нового способу багатошляхової маршрутизацій в програмно-визначених мережах, моделювання роботи алгоритмів на мережевому графі, тестування за різних розмірностей мережі та для різних умов, відображено результати тестувань у вигляді графів, оцінку розробленого рішення.

4.1 Огляд розробленого програмного забезпечення

Розроблений додаток служить комплексною платформою для моделювання та аналізу алгоритмів маршрутизації в програмно-визначених мережах (SDN). Основною метою цієї програми є оцінка продуктивності модифікованого алгоритму багатошляхової маршрутизації в порівнянні з алгоритмом Дейкстра. У цьому підрозділі розглядається структура програми, основні функції та її можливості.

На рисунку 4.1 зображено головне вікно застосунку, на якому користувач має можливість задати вхідні параметри кількості вершин (комутаторів, маршрутизаторів) і кількості зв’язків між ними для випадкової генерації графа мережі. Також користувач може обрати вхідну і кінцеву вершини для пошуку шляху між ними. Можна обрати алгоритм, який буде виконувати маршрутизацію. Надано вікно, в якому відображатимуться логи виконання програми.

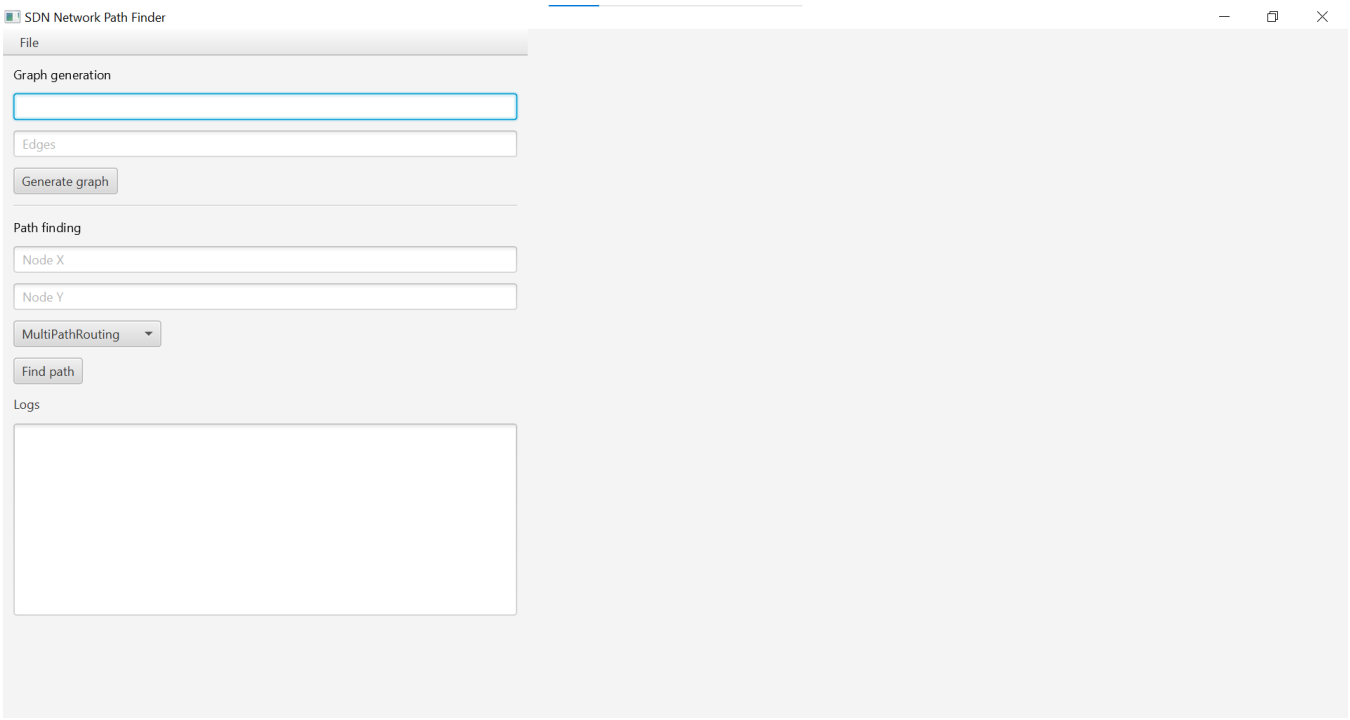

Рисунок 4.1 – Головне вікно застосунку

На рисунку 4.2 зображено додаткове меню, за допомогою якого користувач може відкрити нове вікно з функціями тестування алгоритмів. Це вікно зображено на рисунку 4.3.

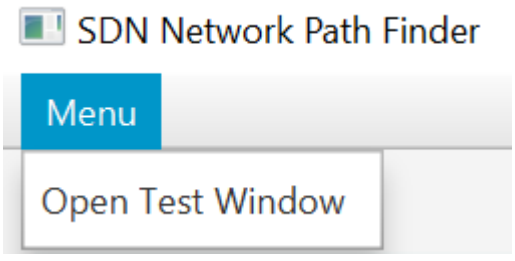

Рисунок 4.2 – Додаткове меню застосунку

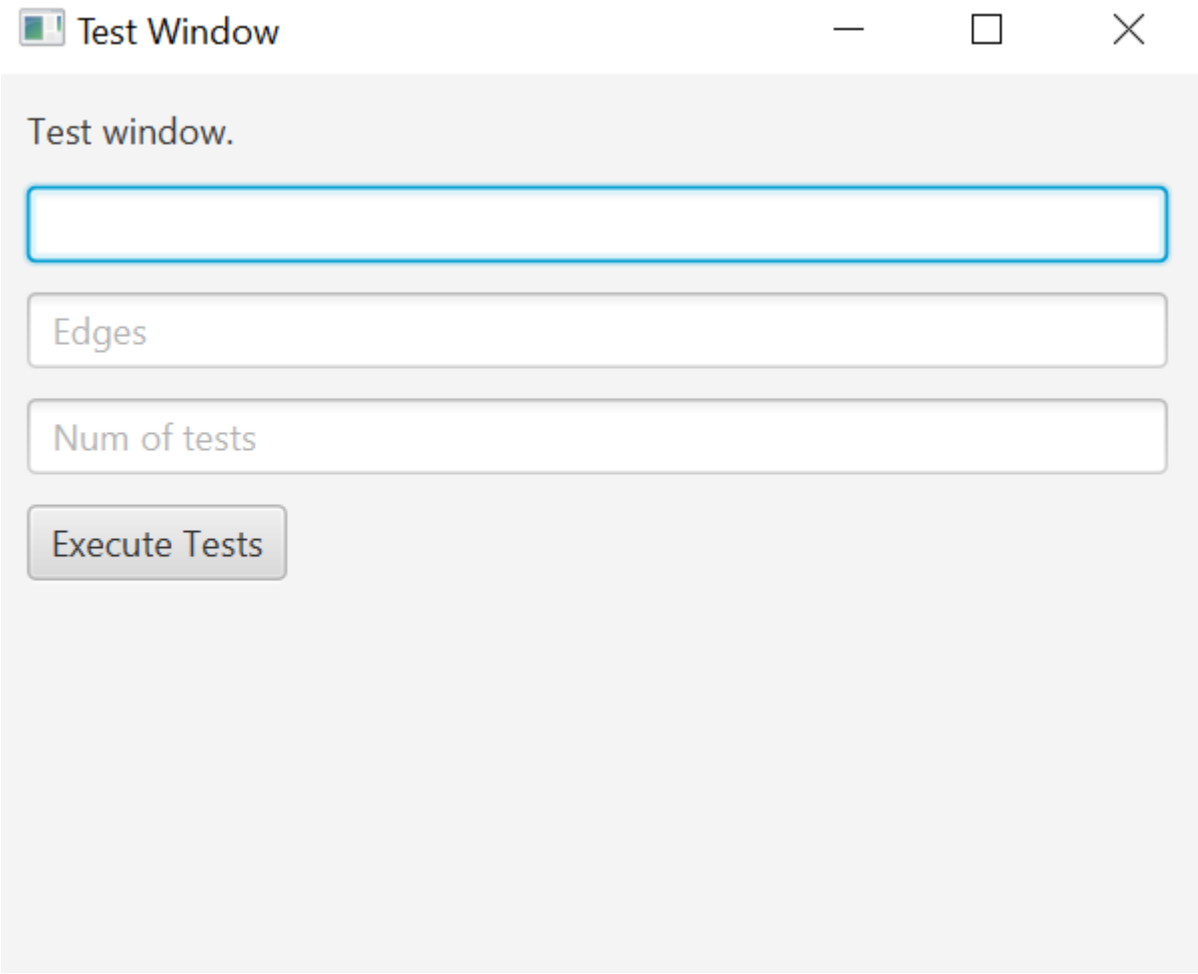

Рисунок 4.3 – Вікно запуску тестів

Вікно запуску тестів, зображене на рисунку 4.3 дає користувачу можливість задати параметри графа (кількість вершин та з’єднань), а також – кількість тестів, що буде проводитись для формування результатів.

Тестом є пошук маршуруту між двома випадковими вершинами на заданому графі за допомогою розробленого багатошляхового алгоритму та алгоритму Дейкстра, запис метрик відповідного шляху і виведення результатів у вигляді діаграми. На рисунку 4.4 наведено приклад такого тесту, де синій графік відображає метрики маршрутів, що знайдені алгоритмом Дейкстра, червоний графік – запропонованим в даній дисертації алгоритмом.

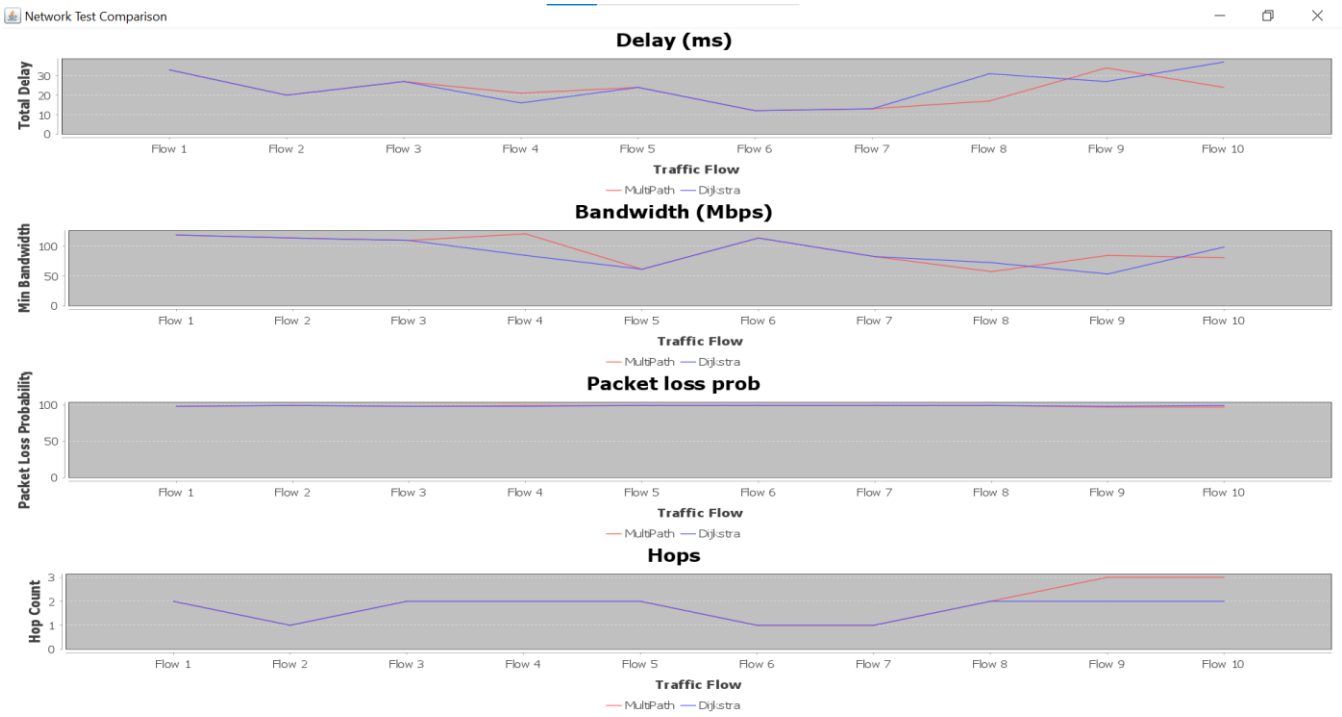

Рисунок 4.4 – Приклад тесту алгоритмів

Зегенруємо граф мережі з 10 вершинами і максимальною кількістю зв’язків 25. На рисунку 4.5 показано реультат, де зображені проіменовані вершини графа, а також ребра із зазначеними метриками D – delay та B – bandwidth для цього ребра.

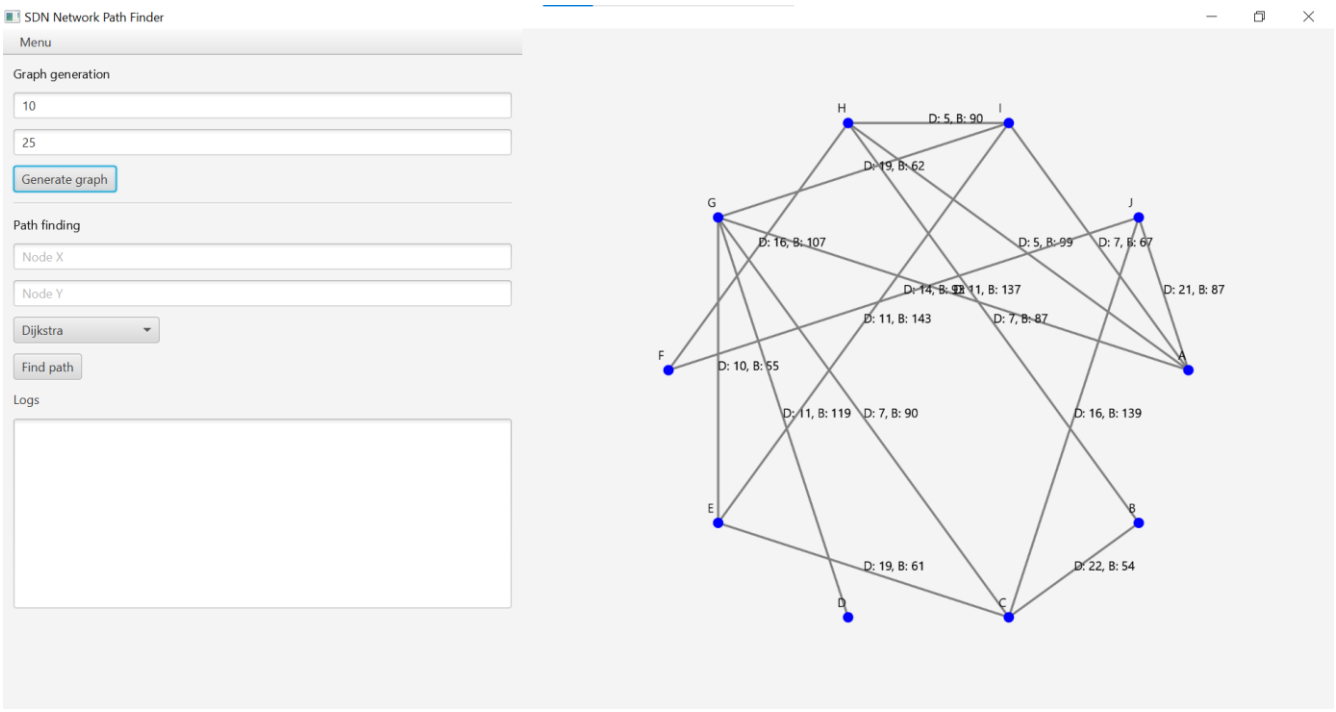

Рисунок 4.5 – Згенерований граф розмірністю 10 вершин

Виконаємо пошук шляху між вершинами F та E за допомогою розробленого алгоритму. На рисунку 4.6 можна побачити результат. Зеленим кольором виділені всі можливі шляхи, червоним – оптимальний шлях, який обрано за допомогою наведеного способу маршрутизації. В логах можна побачити всі шляхи, які було знайдено, оптимальний шлях і його метрики заримки та пропускної здатності.

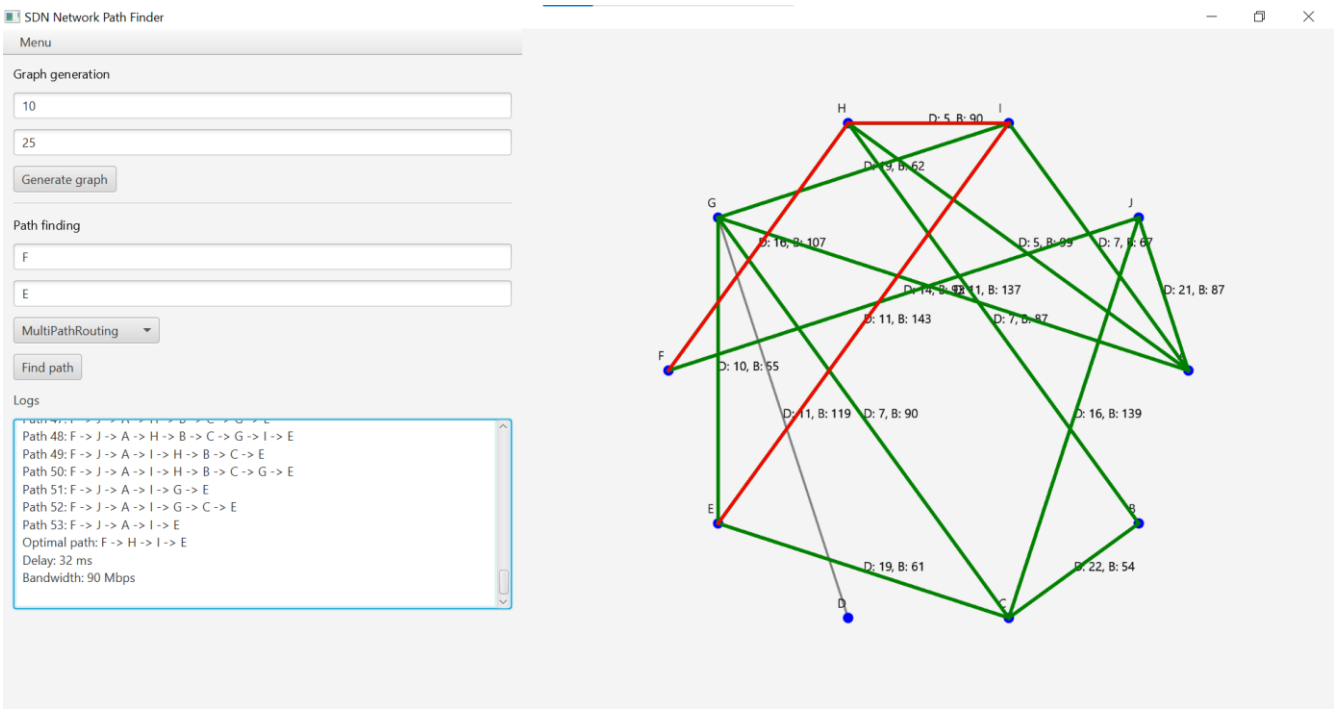

Рисунок 4.6 – Пошук шляху між вершинами F та E за допомогою розробленого алгоритму багатошляхової маршрутизації.

Виконаємо пошук шляху між вершинами F та E за допомогою алгоритму Дейкстра. На рисунку 4.7 можна побачити результат. Знайдений маршрут відрізняється від того, що знайшов алгоритм багатошляхової маршрутизації. Метрики також відрізняються – алгоритм Дейкстра знайшов єдиний маршрут з гіршими показниками затримки і пропускної здатності, адже не орієнтується на них.

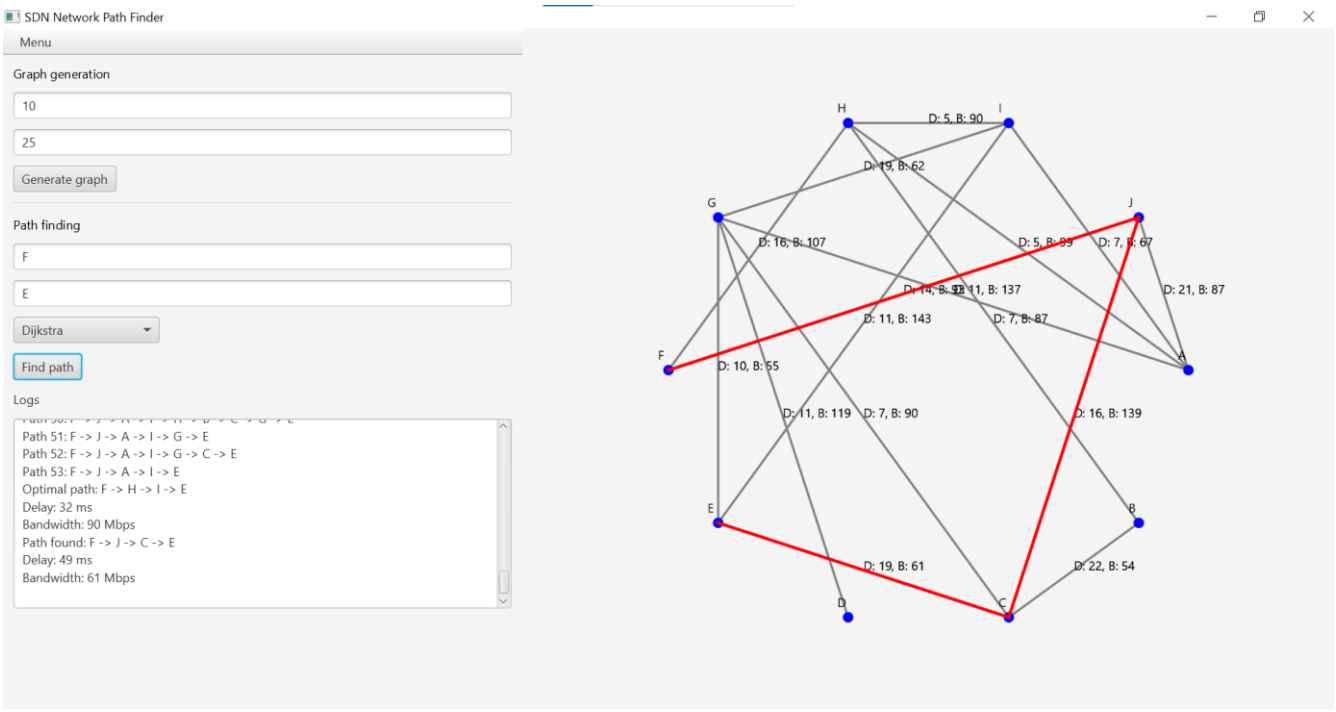

Рисунок 4.7 – Пошук шляху між вершинами F та E за допомогою алгоритму Дейкстра.

4.2 Тестування ефективності алгоритмів

Для оцінки продуктивності розробленого алгоритму багатошляхової маршрутизації в порівнянні з алгоритмом Дейкстра проведемо моделювання на мережевих графах різної розмірності. Мета полягє в тому, щоб проаналізувати, як алгоритми працюють у різних умовах мережі, враховуючи такі показники, як затримка мережі, пропускна здатність, ймовірність втрати пакетів, кількість переходів на маршруті. Експерименти включатимуть графи мережі в діапазоні від малих (5 вузлів) до великих (50 вузлів) топологій, з призначеними випадковим чином метриками затримки, пропускної здатності, ймовірності втрати пакетів.

Отже, проведімо по 10 експериментів для графів різної розмірності, на рисунках 4.8 – 4.14 наведено результати.

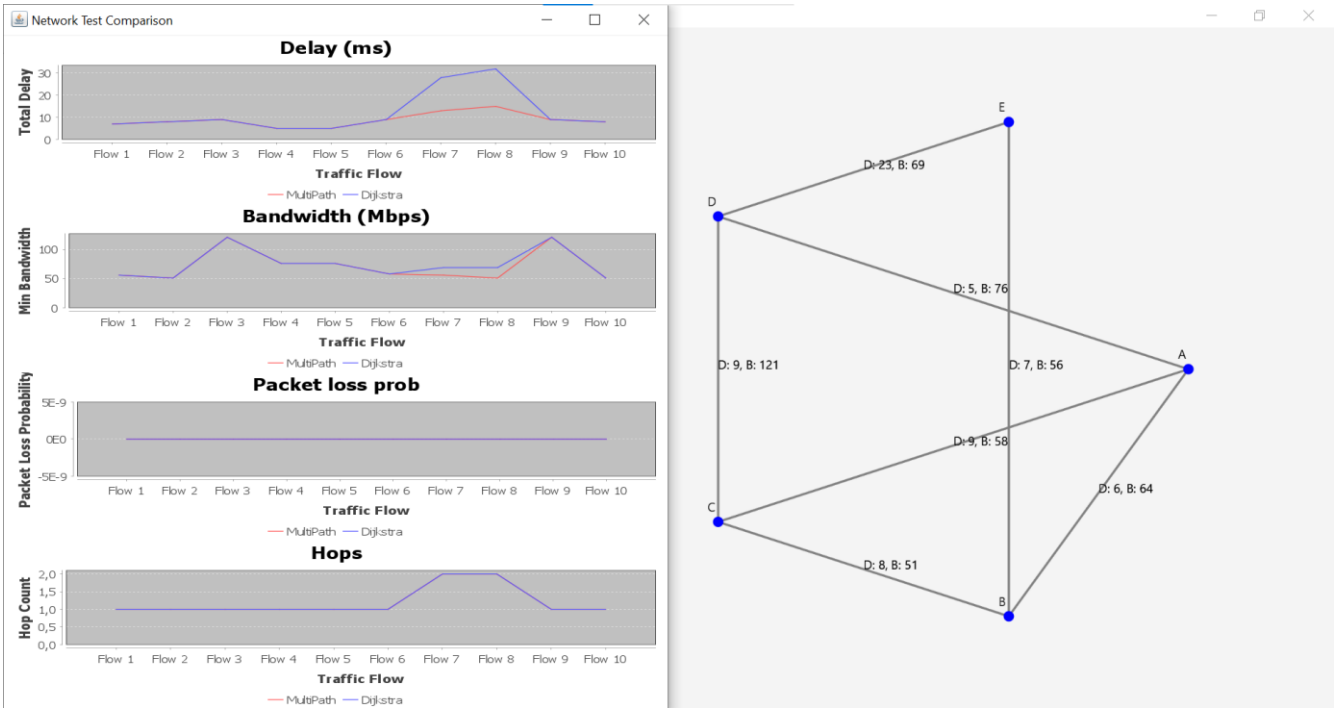

Рисунок 4.8 – Тест для графа розмірності 5 вершин

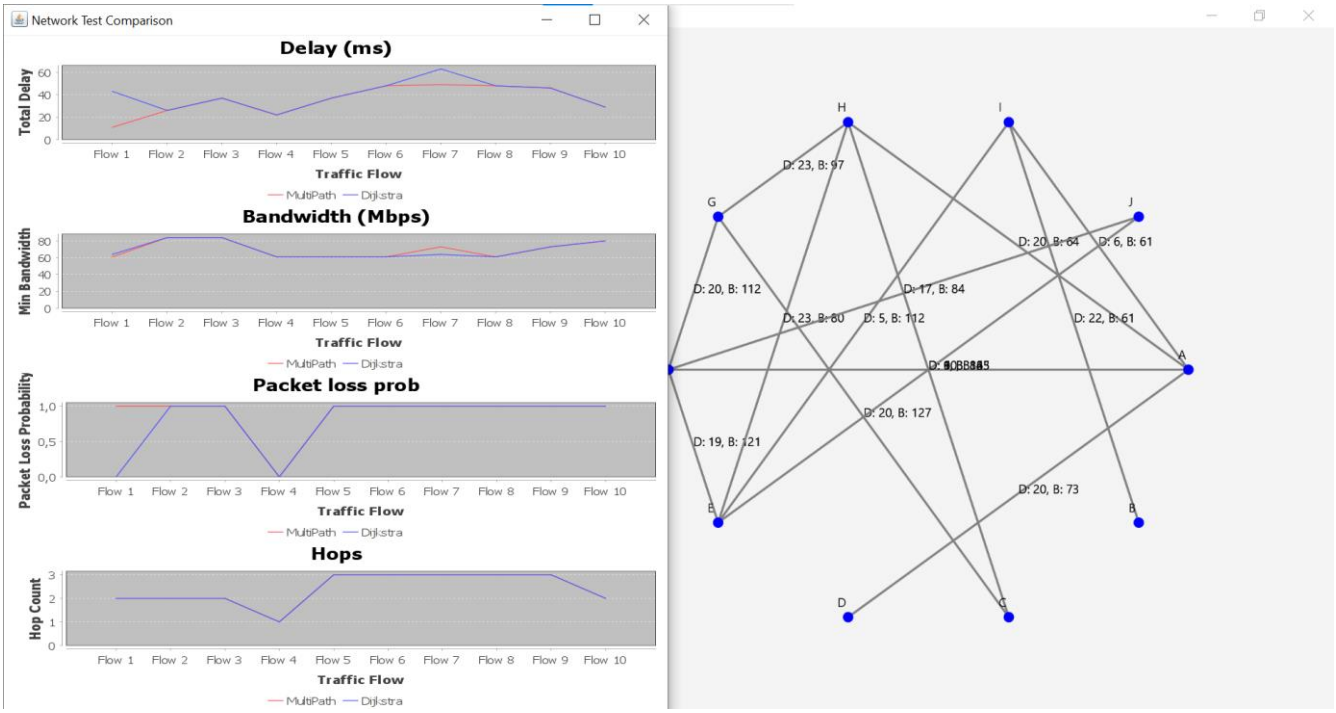

Рисунок 4.9 – Тест для графа розмірності 10 вершин

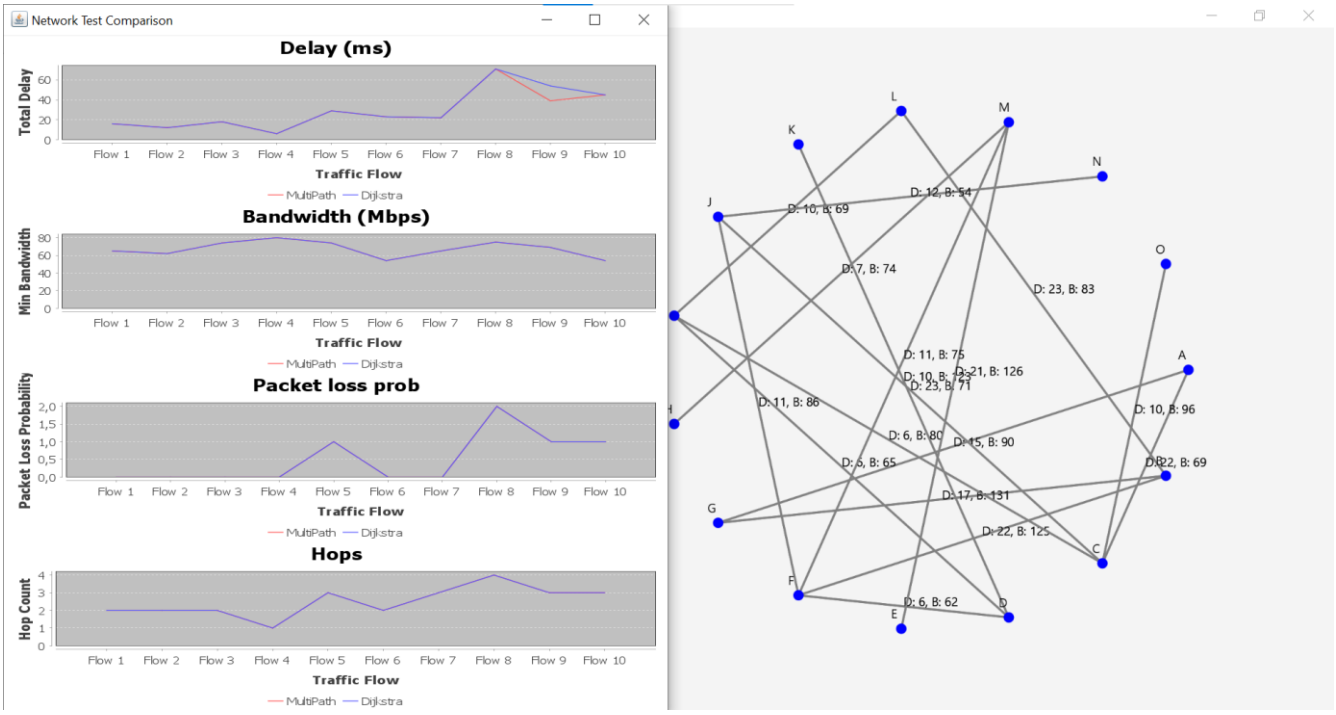

Рисунок 4.10 – Тест для графа розмірності 15 вершин

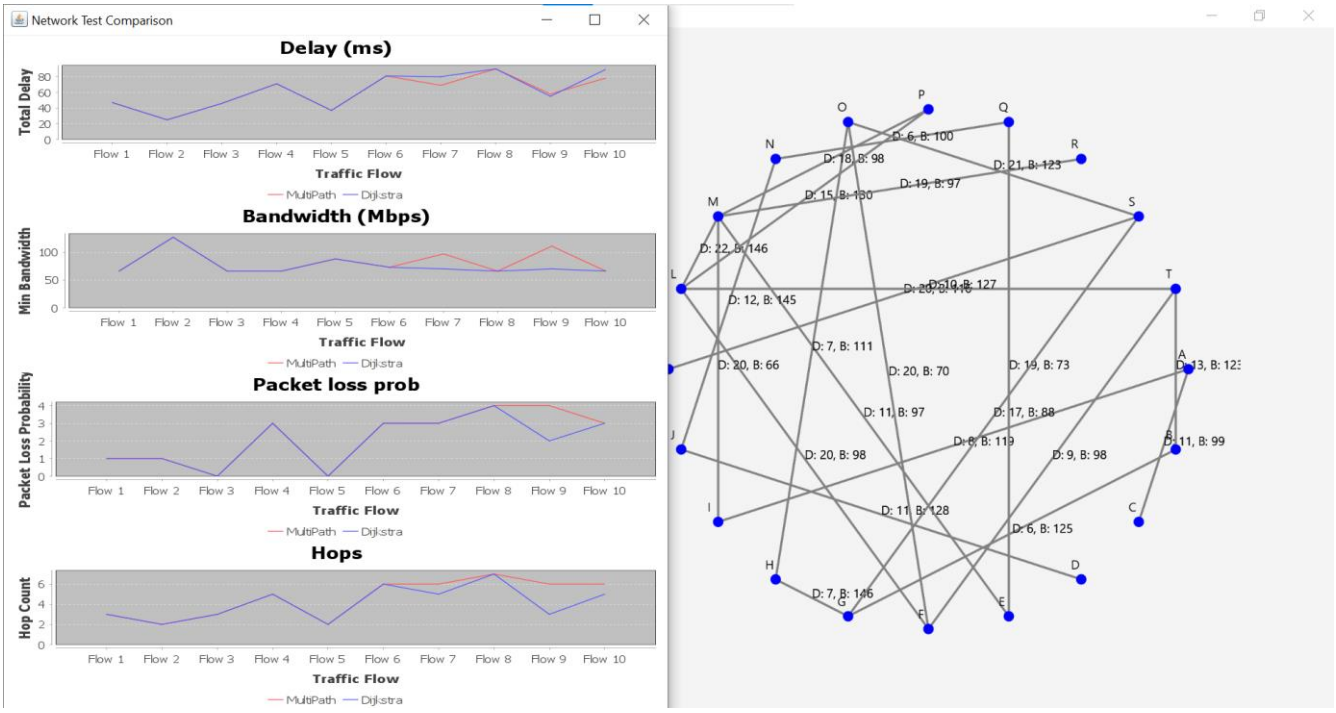

Рисунок 4.11 – Тест для графа розмірності 20 вершин

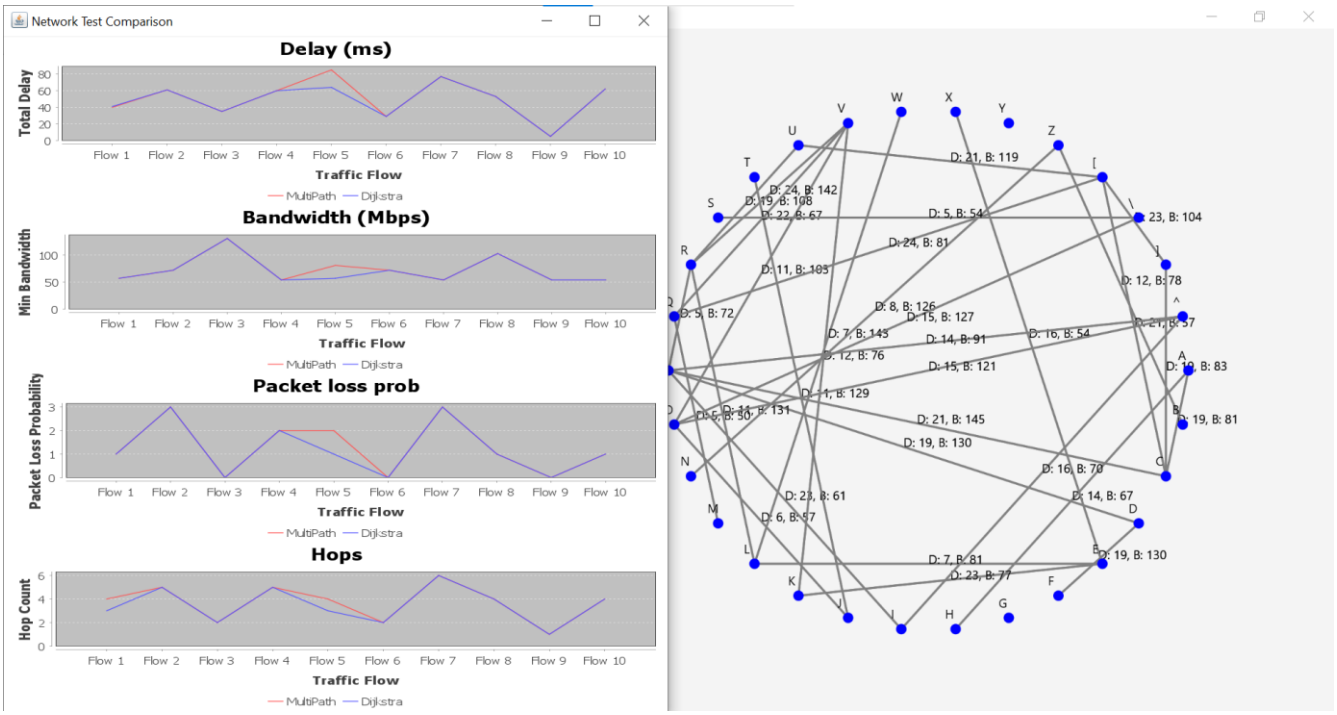

Рисунок 4.12 – Тест для графа розмірності 30 вершин

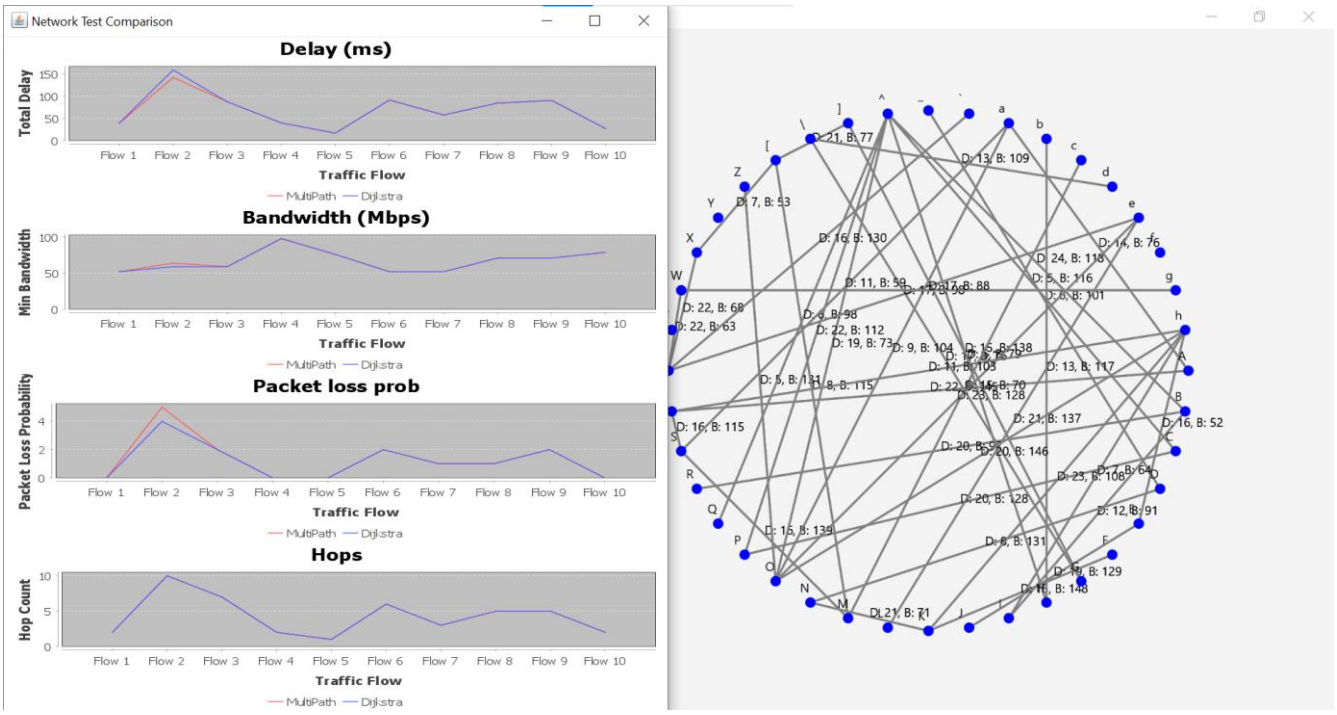

Рисунок 4.13 – Тест для графа розмірності 40 вершин

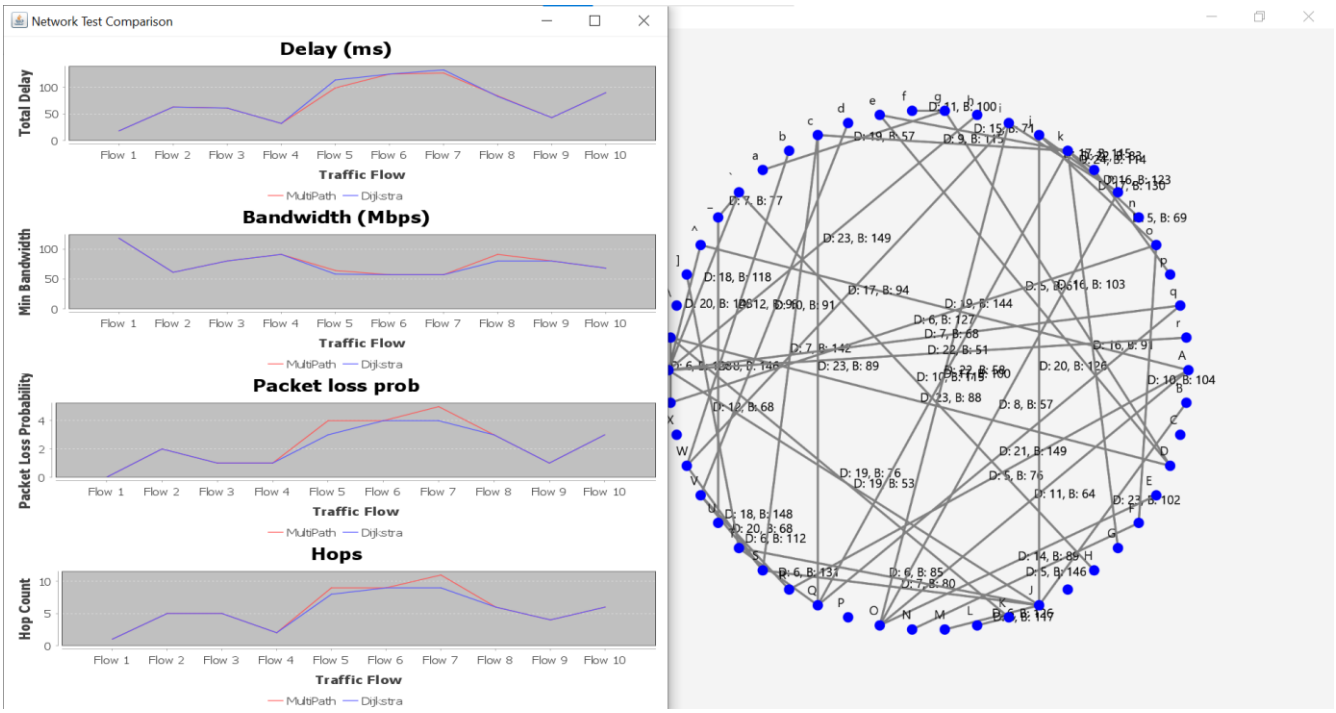

Рисунок 4.14 – Тест для графа розмірності 50 вершин

Згрупуємо отримані в експериментах результати у таблиці. Отже, в таблиці 4.1 наведено середньоарифметичне значення затримки передачі даних для двох досліджуваних алгоритмів на графах різної розмірності. В таблиці 4.2 - середньоарифметичне значення пропускної здатності. В таблиці 4.3 - середньоарифметичне значення ймовірності втрати пакетів. В таблиці 4.4 - середньоарифметичне значення кількості переходів.

Таблиця 4.1

Середньоарифметичні значення затримки

Розмірність графа	5	10	15	20	30	40	50
Модифікований алгоритм (Затримка мс)	8.8	35.3	28.1	60.2	50.7	68.0	74.2

Алгоритм Дейкстра (Затримка мс)	12	39.9	29.6	62.1	48.7	69.7	76.2

Таблиця 4.2

Середньоарифметичні значення пропускної здатності

Розмірність графа	5	10	15	20	30	40	50
Модифікований алгоритм (Пропускна здатність Мбіт/с)	71.7	69.9	67.2	82.6	73.2	67.4	76.7
Алгоритм Дейкстра (Пропускна здатність Мбіт/с)	74.8	69.3	67.2	75.8	70.8	66.9	75.0

Таблиця 4.3

Середньоарифметичні значення ймовірності втрати пакетів

Розмірність графа	5	10	15	20	30	40	50
Модифікований алгоритм (Ймовірність втрати пакетів %)	0.316	1.285	1.137	2.786	1.831	1.917	2.884
Алгоритм Дейкстра (Ймовірність втрати пакетів %)	0.303	1.20	1.104	2.637	1.735	1.842	2.674

Таблиця 4.4

Середньоарифметичні значення кількості переходів

Розмірність графа	5	10	15	20	30	40	50
Модифікований алгоритм (Кількість переходів)	1.2	2.4	2.5	4.6	3.7	4.3	5.8
Алгоритм Дейкстра (Кількість переходів)	1.2	2.4	2.5	4.1	3.5	4.3	5.5

З даних таблиці 4.1 порахуємо середньоарифметичне значення затримки для двох алгоритмів:

- Запропонований алгоритм – 46.47 мс
- Дейкстра – 48.31 мс

З даних таблиці 4.2 порахуємо середньоарифметичне значення пропускної здатності для двох алгоритмів:

- Запропонований алгоритм – 72.67 Мбіт/с
- Дейкстра – 71.4 Мбіт/с

З даних таблиці 4.3 порахуємо середньоарифметичне значення ймовірності втрати пакетів для двох алгоритмів:

- Запропонований алгоритм – 1.74 %
- Дейкстра – 1.64 %

З даних таблиці 4.4 порахуємо середньоарифметичне кількості переходів для двох алгоритмів:

- Запропонований алгоритм – 3.5

- Дейкстра – 3.36

На основі порахованих даних можна сказати, що запропонований алгоритм забезпечує нижчу затримку, трохи краще використовує доступну пропускну здатність, що може покращити ефективність передачі даних. Алгоритм Дейкстра показує дещо кращу надійність з точки зору втрати пакетів. Це може свідчити про його стабільніший вибір маршрутів, але різниця є несуттєвою. Алгоритм Дейкстра обирає маршрути з меншою кількістю переходів, що може позитивно впливати на надійність системи. Варто зазначити, що час виконання запропонованого алгоритму сильно зростає на великомасштабних мережах, що може вимагати потужніших обчислювальних можливостей.

З проведеного дослідження можна сказати, що запропонований спосіб маршрутизації цілком виправдовує свою спроможність до розв'язання задач знаходження маршруту для систем реального часу, де важливі показники пропускної здатності та мінімально можливих затримок.

Висновок до розділу 4

Проведено огляд розробленого програмного застосунку, що надає можливості для симуляції програмно-конфігурованої мережі з метою тестування алгоритмів маршрутизації.

Проведено експерименти для графів мережі різної розмірності (від 5 вершин до 50 вершин), відображено результати у вигляді діаграм та таблиць, зроблено аналіз результатів експериментів та порівняно запропонований алгоритм багатошляхової маршрутизації із алгоритмом Дейкстра.

Результати експериментів підтвердили перевагу використання розробленого рішення для систем реаьного часу, де важливі показники, за якими запропонований спосіб і обирає маршрут – метрики затримки та пропускної здатності. Однак, алгоритм Дейкстра обирає маршрути з меншою кількістю переходів, що, потенційно, може позитивніше складуватись на показниках надійності мережі і втрат пакетів.

Алгоритм Дейкстра також виявився швидшим в пошуку маршруту на мережевих графах великої розмірності, адже він знаходить лише один шлях, тоді як запропонований алгоритм відшукує всі можливі шляхи. Тож запропоноване рішення в такому вигляді як є – не зовсім підходить для мереж великої розмірності, адже може вимагати значних обчислювальних потужностей для прорахунку маршруту. Тож для мереж великої розмірності може знадобитись додаткова адаптація даного рішення, наприклад, у вигляді кластеризації мережі і проведення маршрутизації окремо в кластерах і між ними з використанням запропонованого способу маршрутизації.

РОЗДІЛ 5
РОЗРОБКА СТАРТАП ПРОЄКТУ

5.1 Загальна характеристика стартап-проекту

Основна ідея стартапу полягає в розробці інтелектуального алгоритму багатошляхової маршрутизації, який забезпечує оптимальний вибір маршрутів у програмно-конфігурованих мережах (SDN) на основі багатометричного аналізу. Рішення спрямоване на підвищення пропускної здатності та зменшення затримок у передачі даних, що є критично важливим для систем реального часу. Стартап пропонує універсальне програмне забезпечення, яке інтегрується в існуючі мережеві інфраструктури, адаптуючись до специфічних потреб бізнесу, та гарантує стабільність і надійність мережі навіть у складних умовах. Більш детальна інформація по особливостям та характеристиці проекту наведена у таблицях 5.1 – 5.3.

Таблиця 5.1

Опис ідеї стартап-проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Спосіб багатошляхової маршрутизації в програмно-конфігурованих мережах	1.Автономні транспортні системи та розумні пристрої IoT для моніторингу стану здоров’я, систем безпеки чи домашньої автоматизації.	Мінімізація затримок у передачі даних, що критично для роботи систем реального часу. Оптимальне використання мережевих ресурсів із гарантією стабільності з’єднання.

Таблиця 5.1 (продовження)

Опис ідеї стартап-проекту

	2.Інфраструктура для відеоконференцій, таких як Zoom чи Microsoft Teams та стрімінгові платформ для передачі відео/аудіо контенту (Netflix, YouTube).	Зниження затримок і покращення якості потоків за рахунок вибору оптимальних шляхів. Стабільне з'єднання навіть за умов перевантаження мережі.
	3. Хмарні платформи для обробки великих обсягів даних, наприклад AWS, Azure чи Google Cloud. SaaS-рішення для бізнесу, такі як CRM, ERP або аналітичні системи.	Зменшення витрат на мережеву інфраструктуру завдяки ефективному використанню пропускної здатності. Підвищення доступності сервісів і швидкості обробки клієнтських запитів.
	4. Термінові передачі даних у медичних мережах (телемедицина, віддалені операції). Системи моніторингу пацієнтів у реальному часі.	Гарантована передача життєво важливих даних без затримок. Стійкість до збоїв або перевантажень у мережі.
	5.Застосування в мережах, де важливо забезпечити резервні канали для відмовостійкості, розподіл безпечних з'єднань у корпоративних мережах.	Стійкість до атак або відмов завдяки резервуванню маршрутів. Ефективне балансування навантаження, що мінімізує можливі точки відмови.

Таблиця 5.1 (продовження)

Опис ідеї стартап-проекту

	6.Розподіл навантаження між серверами та підвищення ефективності роботи хмарних сервісів.	Більш висока пропускна здатність і швидкість обробки запитів. Економія ресурсів завдяки зменшенню кількості неефективних маршрутів.
	7.Системи управління транспортом, енергоспоживанням, моніторинг стану навколишнього середовища.	Оптимізація роботи інфраструктури за рахунок зменшення затримок у передачі інформації. Забезпечення стійкої та надійної роботи мережі для мешканців.

Таблиця 5.2

Опис унікальних властивостей продукту

№ п/п	Назва властивості	Пояснення
1	Багатометричний аналіз маршрутів	Алгоритм враховує кілька ключових показників, таких як затримка, пропускна здатність і надійність. Це дозволяє забезпечити не лише найкоротший шлях, але й оптимальний з урахуванням реальних умов мережі.

Таблиця 5.2 (продовження)

Опис унікальних властивостей продукту

2	Модифікований алгоритм на основі DFS	Замість класичних підходів, таких як Дейкстра, використовується адаптована версія DFS, що забезпечує швидшу обробку великих графів і можливість пошуку кількох оптимальних шляхів одночасно. Це знижує час на маршрутизацію.
3	Підтримка реального часу	Продукт орієнтований на системи, які потребують мінімальних затримок і високої швидкості передачі даних. Це робить його ідеальним вибором для IoT, телемедицини, стрімінгу чи критичних систем.
4	Гнучкість та адаптивність	Алгоритм автоматично адаптується до змін у мережі, таких як перевантаження чи відмова вузлів, забезпечуючи стабільну передачу даних без втрати якості.
5	Інтуїтивний програмний інтерфейс	Розроблене програмне забезпечення включає зручний інтерфейс (JavaFX), який дозволяє візуалізувати мережу, аналізувати маршрути та отримувати статистику в реальному часі, що спрощує управління мережею.

Таблиця 5.2 (продовження)

Опис унікальних властивостей продукту

6	Кросплатформеність та інтеграція	Продукт побудований з використанням сучасних інструментів (Java, Gradle), що робить його легко інтегрованим у різні типи SDN-архітектур та сумісним із більшістю операційних систем.
7	Підвищена ефективність порівняно з класичними підходами	Порівняльні експерименти з алгоритмом Дейкстра продемонстрували, що запропонований підхід забезпечує кращі результати за пропускною здатністю та затримкою, що є ключовими показниками для високопродуктивних мереж.

Таблиця 5.3.

Визначення сильних, слабких та нейтральних характеристик ідеї проекту

№ п/п	Техніко-економічні характеристики ідеї	(потенційні) товари/концепції конкурентів			W (слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
		Мій проект	Конкурент 1	Конкурент 2			

Таблиця 5.3 (продовження)

Визначення сильних, слабких та нейтральних характеристик ідеї проекту

1	Продуктивність маршрутизації	Оптимізований вибір маршрутів	Стандартне рішення, обмежена адаптивність	Статичний вибір шляхів, обмежена гнучкість	Низька пропускна здатність при великих навантаженнях	Не оптимізує ресурси на всіх етапах	Оптимізує пропускну здатність
2	Затримка	Знижена завдяки DFS-алгоритму	Стандартна затримка маршруту	Не адаптується до зміни навантаження	Затримка при високих навантаженнях	Залежить від параметрі в мережі	Знижує затримки при великих навантаженнях
3	Надійність	Автоматичний вибір резервних маршрутів	Не враховує змінні параметри мережі	Обмежена кількість маршрутів	Низька надійність при збої вузлів	Немає резервування маршрутів	Стабільність навіть при збоях

Таблиця 5.3 (продовження)

Визначення сильних, слабких та нейтральних характеристик ідеї проекту

4	Адаптивність до змін в мережі	Швидка адаптація до змін	Статичний вибір маршрутів	Зниження продуктивності при зміні умов	Не адаптується до змін	Реагує лише на великі зміни	Швидка адаптація до будь-яких змін
5	Економічна ефективність	Оптимізація використання ресурсів	Не максимізує ефективність	Дорогі обладнання і ліцензії	Висока вартість при великих мережах	Витрати без значних заощаджень	Знижує витрати на інфраструктуру
6	Інтеграція з іншими системами	Легка інтеграція з SDN	Потребує багато налаштувань	Потребує специфічного обладнання	Важка інтеграція з іншими мережами	Можливі труднощі при інтеграції	Сумісність з багатьма платформами
7	Простота налаштування та управління	Простота налаштування та управління	Потребує додаткових налаштувань	Складне адміністрування	Потребує висококваліфікованих адміністраторів	Стандартний рівень складності	Легке управління і налаштування

5.2 Технологічний аудит проекту

Таблиця 5.4

Технологічна здійсненність ідеї проекту

№ п/п	Ідея проекту	Технології її реалізації	Наявність технології	Доступність технологій
1	Спосіб багатошляхової маршрутизації в програмно - конфігурованих мережах	Gradle	Наявна	У відкритому доступі
2		IntelliJ IDEA Ultimate / Community	Наявна	US $499.00 на рік / Безкоштовна версія, на базі відкритого вихідного коду
3		Бібліотеки java	Наявні	У відкритому доступі

Застосування багатошляхової маршрутизації в програмно-конфігурованих мережах (SDN) відкриває нові можливості для підвищення ефективності та надійності мережевих з’єднань. У межах SDN використовується централізоване керування мережею, що дозволяє динамічно оптимізувати маршрути передачі даних. Одним з основних аспектів є багатошляхова маршрутизація, де враховуються різні метрики для вибору оптимальних шляхів. Завдяки використанню алгоритмів, таких як модифікований DFS, можливе забезпечення високої пропускної здатності та мінімізації затримки, що критично для систем реального часу, таких як телемедицина або автономні транспортні системи. Технології маршрутизації в SDN, зокрема багатошляхові алгоритми, дозволяють значно покращити надійність мережі, забезпечуючи резервування маршрутів та

автоматичне перенаправлення трафіку в разі відмови. Це забезпечує більш ефективну передачу даних в умовах високих навантажень та змінних умов мережі, що є важливим для таких застосунків, як мобільні мережі, Інтернет речей (IoT) та інші критичні інфраструктури. Завдяки використанню адаптивних і багатометричних алгоритмів, що включають параметри, як-от пропускна здатність, затримка та надійність, досягається значно краща продуктивність порівняно з традиційними методами маршрутизації, такими як алгоритм Дейкстри. Така технологія стає особливо корисною для мереж, де важлива швидка реакція на зміни в умовах трафіку та збоїв у мережі, забезпечуючи оптимізацію використання мережевих ресурсів і зниження витрат.

5.3 Аналіз ринкових можливостей запуску стартап-проекту

Потенційний ринок стартапу є привабливим для входження. Він демонструє стабільне зростання, значний обсяг продажів і конкурентну рентабельність. Хоча існують технічні та фінансові бар’єри для входу, використання нішевих переваг, таких як інноваційна багатошляхова маршрутизація, може забезпечити успішну конкуренцію з великими гравцями.

Таблиця 5.5

Попередня характеристика потенційного ринку стартап-проекту

№ п/п	Показники стану ринку (найменування)	Характеристика
1	Кількість головних гравців, од	4
2	Загальний обсяг продаж, usd/ум.од	225000
3	Динаміка ринку (якісна оцінка)	Зростає

Таблиця 5.5 (продовження)

Попередня характеристика потенційного ринку стартап-проекту

4	Наявність обмежень для входу (вказати характер обмежень)	Дослідження випереджають реальне впровадження
5	Специфічні вимоги до стандартизації та сертифікації	Відсутні
6	Середня норма рентабельності в галузі (або по ринку), %	20%

Таблиця 5.6

Характеристика потенційних клієнтів стартап-проекту

№ п/п	Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності поведінки потенційних цільових груп клієнтів	Вимоги споживачів до товару
1	Оптимізація мережевого трафіку	Інтернет-провайдери, корпоративні дата-центри, постачальники хмарних сервісів.	Провайдери: акцент на стабільності та масштабованості. Дата-центри: високі вимоги до інтеграції з існуючими рішеннями. Хмарні сервіси: потребують гнучкості та автоматизації.	Надійність, швидкодія, підтримка багатопротоколь ної маршрутизації

Таблиця 5.6 (продовження)

Характеристика потенційних клієнтів стартап-проекту

2	Забезпечення продуктив-ності в системах реального часу	Фінансові установи, розробники IoT-рішень, постачальники потокових сервісів	Банки: жорсткі вимоги до затримок і безпеки. IoT: висока масштабованість, можливість роботи з нестабільними підключеннями. Стримінг: акцент на пропускній здатності та мінімізації буферизації.	Гарантія низької затримки, висока пропускна здатність.
3	Скорочення операційних витрат	Малий і середній бізнес, мережеві оператори.	SMB: чутливість до ціни, потреба у простих рішеннях. Оператори: зацікавленість у довгостроковому скороченні витрат.	Доступність, зручний інтерфейс, автоматизація задач.
4	Гнучкість та адаптивність мереж	IT-аутсорсингові компанії платформи електронної комерції.	IT-компанії: потребують інтеграції з різними середовищами. E-комерція: важливі масштабованість і безперервна робота.	Інтеграція з іншими мережевими системами, швидке розгортання.

Таблиця 5.7

Аналіз загроз запуску та роботи стартап-проекту

№ п/п	Фактор загрози	Зміст загрози	Можлива реакція компанії
1	Висока конкуренція	Наявність великих гравців із розвиненою клієнтською базою та ресурсами.	Вихід у нішеві сегменти ринку, унікалізація продукту за рахунок інноваційності.
2	Технічні бар’єри	Необхідність сумісності з існуючими мережевими інфраструктурами клієнтів.	Ретельне тестування продукту на інтеграцію з популярними SDN-платформами.
3	Низька довіра до нових постачальників	Клієнти віддають перевагу перевіреним брендам із сильною репутацією.	Розробка кейсів успішного впровадження, активна робота зі зворотним зв’язком.
4	Інвестування в таргетовану рекламу та партнерства з галузевими медіа.	Складнощі з впізнаваністю нового бренду без значних витрат.	Інвестування в таргетовану рекламу та партнерства з галузевими медіа.
5	Стандартизація та сертифікація	Відповідність галузевим стандартам потребує часу та ресурсів.	Включення етапів стандартизації та сертифікації у план розробки.

Таблиця 5.8

Аналіз можливостей по реалізації стартап-проекту

№ п/п	Фактор загрози	Зміст можливості	Можлива реакція компанії
1	Зростаючий попит на SDN-рішення	Швидке зростання ринку створює сприятливе середовище для впровадження нових рішень.	Активна маркетингова кампанія, спрямована на демонстрацію переваг продукту.
2	Інноваційність продукту	Конкурентна перевага завдяки використанню багатометричного підходу та DFS.	Розробка кейсів для демонстрації ефективності алгоритму в різних мережах.
3	Підтримка відкритих стандартів	Легкість інтеграції з існуючими рішеннями завдяки підтримці стандартних протоколів.	Включення в опис продукту інформації про сумісність з популярними SDN-платформами.
4	Глобалізація бізнесів	Потреба в адаптивних рішеннях для компаній із розподіленими мережами.	Орієнтація на міжнародний ринок, створення документації англійською мовою.
5	Потреба в оптимізації мережевого трафіку	Зростаючий попит на продуктивні системи реального часу.	Акцент на показниках зниження затримок і підвищення пропускної здатності.

Таблиця 5.9

Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
Тип конкуренції – монополістична	Багато гравців ринку SDN пропонують схожі, але диференційовані рішення.	Диференціація продукту через інноваційність алгоритму та надання додаткових сервісів.
За рівнем конкурентної боротьби – міжнародний	Гравці ринку працюють на глобальному рівні, надаючи рішення для компаній різних країн.	Орієнтація на міжнародний ринок, створення локалізованої документації та підтримки.
За галузевою ознакою – внутрішньогалузева	Конкуренція відбувається між компаніями, що розробляють SDN-алгоритми та програмне забезпечення.	Впровадження унікальних функцій та надання конкурентних цінових умов для залучення клієнтів.
Конкуренція за видами товарів – товарно-видова	Змагання між рішеннями, що пропонують різні підходи до маршрутизації в SDN.	Акцент на технічній перевазі, таких як зменшення затримок та підвищення пропускної здатності.

Таблиця 5.9 (продовження)

Ступеневий аналіз конкуренції на ринку

За характером конкурентних переваг – нецінова	Конкуренція базується на технічних характеристиках, інноваційності та підтримці клієнтів.	Розробка кейсів успішного застосування, навчання клієнтів роботі з продуктом.
За інтенсивністю – марочна	Компанії на ринку активно просувають свої бренди як надійні та перевірені рішення.	Інвестування в маркетинг, створення впізнаваного бренду через якісну рекламу та репутацію.

Таблиця 5.10

Аналіз конкуренції в галузі за М. Портером

Складові аналізу	Прямі конкуренти в галузі	Потенційні конкуренти	Постачальники	Клієнти	Товари замінники
	Cisco (ACI), VMware (NSX), Nokia (Nuage Networks) Juniper (Contrail SDN).	Висока вартість розробки SDN-рішень і сертифікації, необхідність сильної експертизи	Постачальники спеціалізованого обладнання та ліцензованого ПЗ можуть диктувати ціни	Вимоги до якості, швидкості впровадження, інновацій і сумісності з існуючими мережами	Традиційні маршрутизатори та MPLS-рішення залишаються дешевшими

Таблиця 5.10 (продовження)

Аналіз конкуренції в галузі за М. Портером

Висновки	Головні конкуренти - великі корпорації з усталеними позиціями. Для успіху необхідна унікальність та інновації.	Високі бар'єри входу стримують нових гравців, але технологічні прориви можуть змінити ситуацію.	Постачальники мають сильний вплив через обмежену кількість альтернатив. Важливо диверсифікувати залежність.	Клієнти вимогливі до якості, інновацій та сумісності. Потрібно забезпечити високу цінність продукту.	Традиційні рішення залишаються дешевшими, але програють за продуктивністю та масштабованістю.

Таблиця 5.11

Обґрунтування факторів конкурентоспроможності

№ п/п	Фактор конкурентоспроможності	Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим)
1	Інноваційність рішення	Впровадження багатошляхової маршрутизації з багатометричною оцінкою підвищує продуктивність мережі.
2	Вартість впровадження	Клієнти орієнтовані на оптимальне співвідношення витрат на інтеграцію та довгострокову економію.

Таблиця 5.11 (продовження)

Обґрунтування факторів конкурентоспроможності

3	Продуктивність і надійність	Забезпечення низької затримки та високої пропускної здатності є критично важливим для реального часу.
4	Сумісність із існуючими рішеннями	Більшість клієнтів використовують гібридні мережі, що потребує інтеграції з традиційними технологіями.
5	Підтримка та оновлення	Наявність регулярних оновлень і технічної підтримки є ключовим для збереження довіри клієнтів.
6	Гнучкість конфігурації	Можливість адаптації продукту під специфічні вимоги клієнтів підвищує його привабливість.
7	Маркетингова стратегія	Активна промоція, орієнтація на ключові сегменти клієнтів сприяє швидкому впровадженню на ринок.

Таблиця 5.12

Порівняльний аналіз сильних та слабких сторін стартап проекту

№ п/п	Фактор конкуренто-спроможності	Бали 1-20	Порівняння рейтингу товарів-конкурентів						
			-3	-2	-1	0	+1	+2	+3
1	Лідерські позиції на ринку	20					+		
2	Цінова політика компанії	18						+	
3	Торговий маркетинг	18				+			

Таблиця 5.12 (продовження)

Порівняльний аналіз сильних та слабких сторін стартап проекту

4	Репутація компанії	19					+		
5	Лояльність до бренду	16				+			
6	Інвестиційний бюджет	19			+				

На основі попереднього аналізу різних факторів загроз та можливостей стартап-проекту по застосуванню способу багатошляхової маршрутизації в програмно-конфігурованих мережах, у таблиці 5.13 наведено матрицю SWOT. Схема відображає загальну картину про розвитку проекту на ринку.

Таблиця 5.13

SWOT- аналіз стартап-проекту

Сильні сторони: 1.Застосування сучасних технологій та підходів 2.Використання відтестованих рішень 3.Правильно обрана стратегія продажу 4.Забезпечення сервісної підтримки	**Слабкі сторони**: 1.Потреба у постійному фінансування 2.Відсутність спеціалізованого маркетингового досвіду 3.Складна політична та економічна ситуація
Можливості: 1.Використання запропонованого методу на ринку 2.Ріст обсягів продажу продукту	**Загрози**: 1.Відсутність попиту на продукт 2.Неконкурентоспроможність

Таблиця 5.14

Альтернативи ринкового впровадження стартап-проекту

№ п/п	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Активне опанування ринку та укладення нових договорів із підприємцями	Висока	Короткі
2	Слідування стратегії по забезпеченню конкурентоспроможності	Середня	Короткі

5.4. Розробка ринкової стратегії проекту

Таблиця 5.15

Вибір цільових груп потенційних споживачів

№ п/п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
1	Великі корпорації з розподільними мережами	Висока	Високий	Висока	Середня
2	Провайдери хмарних послуг	Висока	Середній	Середня	Середня
3	Малий і середній бізнес	Середня	Низький	Низька	Висока
4	Державні установи та організації	Середня	Середній	Низька	Низька

Таблиця 5.15 (продовження)

Вибір цільових груп потенційних споживачів

5	Постачальники послуг Інтернету	Висока	Високий	Висока	Середня
Обрані цільові групи: стратегія диференційованого маркетингу					

На основі аналізу потенційних груп споживачів для запропонованого продукту обрано стратегію диференційованого маркетингу, оскільки компанія орієнтується на кілька сегментів ринку та розробляє для кожного з них індивідуальні програми ринкового впливу.

Таблиця 5.16

Визначення базової стратегії розвитку

№ п/п	Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку
1	Стратегія зростання	Прагнення компаній до активних темпів економічного зростання	Стратегія концентрованого зростання, внутрішнє розширення продукту, а також виробництво нових продуктів	Стратегія диференці-йованого маркетингу

Таблиця 5.17

Визначення базової стратегії конкурентної поведінки

№ п/п	Чи є проект «першо-прохідцем» на ринку?	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, і які?	Стратегія конкурентної поведінки*
1	Так	Стратегія поєднання нових та старих споживачів продукту	Відсутність потреби	Стратегія наслідування лідера

Таблиця 5.18

Визначення стратегії позиціонування

№ п/п	Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартап проекту	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту (три ключових)
1	Забезпечення високої якості продукту	Стратегія диференційо-ваного маркетингу	Професіональний підхід до розробки продукту згідно стандартів	Якість, надійність, безпека
2	Вигідна ціна на продукт	Стратегія наслідування лідера	Широкий спектр можливостей по застосуванню додаткового пз	Доступність, унікальність, новизна

5.5 Розроблення маркетингової програми стартап-проекту

Таблиця 5.19

Визначення ключових переваг концепції потенційного товару

№ п/п	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити
1	Забезпечення надійного та швидкого маршрутування даних у реальному часі	Зниження затримки та підвищення пропускної здатності мережі	Модифікований алгоритм, що враховує багатометричну оцінку шляху
2	Оптимізація мережевого трафіку	Підвищення ефективності використання ресурсів мережі	Інтеграція з існуючими SDN-рішеннями та легкість масштабування
3	Скорочення часу простою мережі	Підвищення стійкості мережі до перевантажень	Використання гнучкого управління маршрутами на основі DFS-алгоритму
4	Зниження витрат на підтримку мережі	Автоматизація процесів управління трафіком	Легкість інтеграції та налаштування для різних типів мереж
5	Підвищення якості обслуговування (QoS)	Забезпечення стабільності для критично важливих додатків	Орієнтація на системи реального часу з адаптивною маршрутизацією

Таблиця 5.20

Формування системи збуту

№№ п/п	Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
1	Орієнтованість на високоякісний продукт	Пропозиція на надійність функціональності продукту	Значна	Влаштування тендерних торгів, сильна рекламна програма

Таблиця 5.21

Концепція маркетингових комунікацій стартап-проекту

№	Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
1	Орієнтація на ефективні рішення для SDN з низькою затримкою	Галузеві форуми, технічні конференції, LinkedIn, спеціалізовані блоги	Інноваційне рішення для багатошляхової маршрутизації	Підкреслити ефективність і адаптивність алгоритму	"Швидше, розумніше, ефективніше-революційний підхід до маршрутизації в SDN."

Таблиця 5.21 (продовження)

Концепція маркетингових комунікацій стартап-проекту

2	Пріоритет точних технічних характеристик та зручності інтеграції	Прямі продажі, вебінари, онлайн-демонстрації, технічна документація	Простота інтеграції та підвищення пропускної здатності мережі	Донести технічну перевагу і простоту впровадження	"Ваші мережі - ваші правила: універсальна інтеграція для оптимального результату."
3	Потреба в підвищенні конкурентоспроможності через сучасні мережеві рішення	Спеціалізовані журнали, кейс-стаді, партнерські семінари	Підвищення продуктивності та конкурентоспроможності мереж	Створити імідж технологічного лідера	"Лідируйте з мережею, яка адаптується: більше можливостей для вашого бізнесу."
4	Очікування вигідного співвідношення ціни і якості	Соціальні мережі, електронна розсилка, веб-сайт	Оптимальне співвідношення ціна/якість	Виділити економічну вигоду при використанні продукту	"Інновації, доступні для кожного: новий стандарт SDN за розумну ціну."

Висновок до розділу 5

Згідно з проведеним аналізом ринкових умов, потенційних клієнтів та конкурентного середовища, проект розробки багатошляхового алгоритму маршрутизації для програмно-конфігурованих мереж (SDN) має значний потенціал для комерціалізації. Пропоноване рішення забезпечує оптимізацію використання мережевих ресурсів, зниження затримок у передачі даних та підвищення пропускної здатності, що створює суттєву конкурентну перевагу на ринку.

Попит на продукт обумовлений його здатністю знижувати витрати на впровадження та обслуговування мережевої інфраструктури, забезпечуючи гнучкість та адаптивність до змінних умов. Враховуючи аналіз цільових груп споживачів, було обрано стратегію диференційованого маркетингу, що дозволяє ефективно охопити кілька ключових сегментів ринку.

Аналіз конкурентного середовища за моделлю М. Портера показав, що ринок є перспективним для входження, а загрози з боку замінників та бар’єрів входження є мінімальними. Разом із цим, враховані чинники впливу постачальників і клієнтів підтверджують доцільність виходу на ринок з цим продуктом.

Таким чином, стартап-проект з розробки алгоритму багатошляхової маршрутизації має всі підстави для успішного запуску та подальшого масштабування в сегменті SDN-мереж. Його впровадження сприятиме зростанню ефективності та інноваційності інтелектуальних мереж, відповідаючи актуальним потребам ринку.

ВИСНОВКИ

Ця магістерська робота була присвячена розробці та аналізу модифікованого алгоритму багатошляхової маршрутизації для програмно-конфігурованих мережах (SDN) з метою оптимізації вибору шляху на основі метрик затримки і пропускної здатності мережі. У роботі застосовано теоретичні знання та практичні розробки для досягнення поставлених завдань дослідження.

Основна мета дисертації — розробка та реалізація ефективного алгоритму багатошляхової маршрутизації для SDN, здатного збалансувати показники пропускної здатності та затримки — була успішно досягнута.

Комплексний аналіз існуючих алгоритмів маршрутизації та їх обмежень підкреслив потребу в реалізації запропонованого в дисертації рішення маршрутизації в SDN.

Розроблено модифікований алгоритм багатошляхової маршрутизації на основі техніки пошуку в глибину (DFS), котрий включає ключові метрики, що є критичними для систем реального часу, для вибору оптимального шляху.

Розроблено програмне забезпечення з використанням сучасних інструментів (Java, Gradle, JavaFX, jfreechart) для реалізації та тестування запропонованого алгоритму.

Проведено експерименти для перевірки та порівняння запропонованого рішення з алгоритмом Дейкстра на графах різної розмірності, що дало базис для висновків щодо ефективності алгоритму.

Продемонстровано застосовність багатометричної оцінки шляху для підвищення продуктивності маршрутизації. Надано алгоритм, який можна застосовувати до різних типів мереж SDN. Підтверджено переваги модифікованого алгоритму для задач систем реального часу. Запропоноване рішення показало кращі результати пропускної здатності та затримки мережі.

Достовірність отриманих результатів забезпечено за рахунок тестування алгоритму за різних розмірів топології і проведення великої кількості повторних експериментів для доведення відтворюваності результатів та їх порівняння з базовим алгоритмом Дейкстра.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Shaghaghi A., Kaafar M.A., Buyya R., Jha S. Software-Defined Network (SDN) Data Plane Security: Issues, Solutions, and Future Directions // *Handbook of Computer Networks and Cyber Security* / Gupta B., Perez G., Agrawal D., Gupta D. (eds). Cham: Springer, 2020. DOI: https://doi.org/10.1007/978-3-030-22277-2_14.
2. McKeown N., Anderson T., Balakrishnan H., Parulkar G., Peterson L., Rexford J., Shenker S., Turner J. OpenFlow: enabling innovation in campus networks // *ACM SIGCOMM Computer Communication Review*. 2008. Vol. 38, No. 2. P. 69–74. DOI: https://doi.org/10.1145/1355734.1355746.
3. Alidadi A., Arab S., Askari T. A novel optimized routing algorithm for QoS traffic engineering in SDN-based mobile networks // *ICT Express*. 2022. Vol. 8, Issue 1. P. 130–134. DOI: https://doi.org/10.1016/j.icte.2021.12.010.
4. Chahlaoui F., Dahmouni H., El Alami H. Multipath-routing based load-balancing in SDN networks // *Proceedings of the 2022 5th Conference on Cloud and Internet of Things (CIoT)*. Marrakech, Morocco, 2022. P. 180–185. DOI: https://doi.org/10.1109/CIoT53061.2022.9766801.
5. He H., Peng Z., Zhou X., Wang J. LFOD: A Lightweight Flow Table Optimization Scheme in SDN Based on Flow Length Distribution in the Internet // *Proceedings of the 2022 23rd Asia-Pacific Network Operations and Management Symposium (APNOMS)*. Takamatsu, Japan, 2022. P. 1–6. DOI: 10.23919/APNOMS56106.2022.9919927.
6. Zhong H., Xu J., Cui J., Sun X., Gu C., Liu L. Prediction-based dual-weight switch migration scheme for SDN load balancing // *Computer Networks*. 2022. Vol. 205. Article ID 108749. ISSN 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2021.108749.

7. Kumar P., Bhushan S., Halder D., Baswade A.M. fybrrLink: Efficient QoS-Aware Routing in SDN Enabled Future Satellite Networks // *IEEE Transactions on Network and Service Management*. 2022. Vol. 19, No. 3. P. 2107–2118. DOI: 10.1109/TNSM.2021.3129876.
8. Rodriguez Ruelas A.M., Quiñones Ccorimanya J., Quispe Barra M.A. An Overview of P4-Based Load Balancing Mechanism in SDN // *Proceedings of the 8th Brazilian Technology Symposium (BTSym'22): Emerging Trends and Challenges in Technology*. Springer Nature, 2023. P. 174.
9. Karakus M., Durresi A. A survey: Control plane scalability issues and approaches in Software-Defined Networking (SDN) // *Computer Networks*. 2017. Vol. 112. P. 279–293. ISSN 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2016.11.017.
10. Semong T., Maupong T., Anokye S., Kehulakae K., Dimakatso S., Boipelo G., Sarefo S. Intelligent Load Balancing Techniques in Software Defined Networks: A Survey // *Electronics*. 2020. Vol. 9, No. 7. Article 1091. DOI: https://doi.org/10.3390/electronics9071091.
11. Abbasi M.R., Guleria A., Devi M.S. Traffic engineering in software defined networks: a survey // *Journal of Telecommunications and Information Technology*. 2016. No. 4. P. 3–14.
12. Zhang W., Lei W., Zhang S. A multipath transport scheme for real-time multimedia services based on software-defined networking and segment routing // *IEEE Access*. 2020. Vol. 8. P. 93962–93977.
13. Bagaa M., Dutra D.L.C., Taleb T., Samdanis K. On SDN-Driven Network Optimization and QoS Aware Routing Using Multiple Paths // *IEEE Transactions on Wireless Communications*. 2020. Vol. 19, No. 7. P. 4700–4714. DOI: 10.1109/TWC.2020.2986408.

14. Amin R., Reisslein M., Shah N. Hybrid SDN Networks: A Survey of Existing Approaches // *IEEE Communications Surveys & Tutorials*. 2018. Vol. 20, No. 4. P. 3259–3306. DOI: 10.1109/COMST.2018.2837161.
15. Gonzalez-Trejo J.E., Rivera-Rodriguez R., Tchernykh A., Lozano-Rizk J.E., Villarreal-Reyes S., Galaviz-Mosqueda A., Gonzalez Compean J.L. A Novel Strategy for Computing Routing Paths for Software-Defined Networks Based on MOCell Optimization // *Applied Sciences*. 2022. Vol. 12, No. 21. Article 11590. DOI: https://doi.org/10.3390/app122211590.
16. Alsaeedi M., Mohamad M.M., Al-Roubaiey A.A. Toward Adaptive and Scalable OpenFlow-SDN Flow Control: A Survey // *IEEE Access*. 2019. Vol. 7. P. 107346–107379. DOI: 10.1109/ACCESS.2019.2932422.
17. Karakus M., Durresi A. Quality of Service (QoS) in Software Defined Networking (SDN): A survey // *Journal of Network and Computer Applications*. 2017. Vol. 80. P. 200–218. ISSN 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2016.12.019.
18. Chen J., Xiao W., Zhang H., et al. Dynamic routing optimization in software-defined networking based on a metaheuristic algorithm // *Journal of Cloud Computing*. 2024. Vol. 13. Article 41. DOI: https://doi.org/10.1186/s13677-024-00603-1.
19. Tache M.D., Păscuțoiu O., Borcoci E. Optimization Algorithms in SDN: Routing, Load Balancing, and Delay Optimization // *Applied Sciences*. 2024. Vol. 14, No. 14. Article 5967. DOI: https://doi.org/10.3390/app14145967.
20. Lin S.-C., Wang P., Luo M. Jointly optimized QoS-aware virtualization and routing in software defined networks // *Computer Networks*. 2016. Vol. 96. P. 69–78. ISSN 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2015.08.003.

ДОДАТОК А

Спосіб багатошляхової маршрутизації в програмно-конфігурованих мережах

Лістинг програми

Аркушів 19

Київ – 2024

```
package sdn.logical;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class MultiPathRouting {
    private final NetworkGraph graph;

    public MultiPathRouting(NetworkGraph graph) {
        this.graph = graph;
    }


    public List<List<Node>> findAvailablePaths(String sourceId, String
destinationId) {
        Node source = graph.getNode(sourceId);
        Node destination = graph.getNode(destinationId);
        List<List<Node>> disjointPaths = new ArrayList<>();

        if (source == null || destination == null) return disjointPaths;

        Set<Node> visited = new HashSet<>();
        findPathsDFS(source, destination, new ArrayList<>(), disjointPaths,
visited);

        return disjointPaths;
    }

    private void findPathsDFS(Node current, Node destination, List<Node> path,
List<List<Node>> paths,
                              Set<Node> visited) {
        path.add(current);
        visited.add(current);

        if (current.equals(destination)) {
            paths.add(new ArrayList<>(path));
        } else {
            for (Edge edge : current.getEdges()) {
                if (!visited.contains(edge.getDestination())) {

                    findPathsDFS(edge.getDestination(), destination, path, paths,
visited);
                }
            }
        }

        path.remove(path.size() - 1);
        visited.remove(current);
    }

    public List<Node> selectOptimalPath(List<List<Node>> paths) {
        List<Node> optimalPath = null;
        double bestScore = Double.NEGATIVE_INFINITY;

        // Знайти максимальні значення для нормалізації
        int maxDelay = Integer.MIN_VALUE;
```

```
        int maxBandwidth = Integer.MIN_VALUE;

        for (List<Node> path : paths) {
            for (int i = 0; i < path.size() - 1; i++) {
                Edge edge = graph.getEdge(path.get(i), path.get(i + 1));
                if (edge != null) {
                    maxDelay = Math.max(maxDelay, edge.getDelay());
                    maxBandwidth = Math.max(maxBandwidth, edge.getBandwidth());
                }
            }
        }

        for (List<Node> path : paths) {
            int totalDelay = 0;
            int minBandwidth = Integer.MAX_VALUE;

            for (int i = 0; i < path.size() - 1; i++) {
                Edge edge = graph.getEdge(path.get(i), path.get(i + 1));
                if (edge != null) {
                    totalDelay += edge.getDelay();
                    minBandwidth = Math.min(minBandwidth, edge.getBandwidth());
                }
            }

            // Нормалізація значень
            double normalizedDelay = (double) totalDelay / maxDelay; //
Нормалізація затримки
            double normalizedBandwidth = (double) minBandwidth / maxBandwidth; //
Нормалізація пропускної здатності

            // Обчислити бал з нормалізованими значеннями
            double score = (normalizedBandwidth * 0.1) / normalizedDelay;



            if (score > bestScore) {
                bestScore = score;
                optimalPath = path;
            }
        }

        return optimalPath;
    }
}

package sdn.logical;

import java.util.*;

public class DijkstraRouting {
    private final NetworkGraph graph;

    public DijkstraRouting(NetworkGraph graph) {
        this.graph = graph;
    }

    public List<Node> findShortestPath(String sourceId, String destinationId) {
        Map<Node, Integer> distances = new HashMap<>();
```

```
        Map<Node, Node> previousNodes = new HashMap<>();
        PriorityQueue<Node> nodes = new
PriorityQueue<>(Comparator.comparingInt(distances::get));

        Node source = graph.getNode(sourceId);
        Node destination = graph.getNode(destinationId);
        distances.put(source, 0);
        nodes.add(source);

        while (!nodes.isEmpty()) {
            Node current = nodes.poll();
            if (current.equals(destination)) break;

            for (Edge edge : current.getEdges()) {
                Node neighbor = edge.getDestination();
                int newDist = distances.get(current) + edge.getWeight();

                if (newDist < distances.getOrDefault(neighbor, Integer.MAX_VALUE))
{
                    distances.put(neighbor, newDist);
                    previousNodes.put(neighbor, current);
                    nodes.add(neighbor);
                }
            }
        }

        List<Node> path = new ArrayList<>();
        for (Node at = destination; at != null; at = previousNodes.get(at)) {
            path.add(at);
        }
        Collections.reverse(path);
        return path.size() > 1 ? path : Collections.emptyList();
    }
}
package sdn.logical;

import java.util.HashMap;
import java.util.Map;

public class NetworkGraph {
    private final Map<String, Node> nodes;

    public NetworkGraph() {
        nodes = new HashMap<>();
    }

    public Node getNode(String id) {
        return nodes.get(id);
    }

    public Node addNode(String id) {
        Node node = new Node(id);
        nodes.put(id, node);
        return node;
    }

    public void addEdge(String sourceId, String destId, int weight, int delay, int
bandwidth, double packetLoss) {
```

```
        Node source = nodes.computeIfAbsent(sourceId, Node::new);
        Node destination = nodes.computeIfAbsent(destId, Node::new);

        // Перевірка на існування зв'язку
        if (source.hasEdgeTo(destination)) {
            return; // Якщо зв'язок вже існує, не додаємо його повторно
        }

        // Додаємо з'єднання з source до destination
        Edge edge = new Edge(source, destination, weight, delay, bandwidth,
packetLoss);
        source.addEdge(edge);

        // Додаємо зворотнє з'єднання з destination до source
        Edge reverseEdge = new Edge(destination, source, weight, delay, bandwidth,
packetLoss);
        destination.addEdge(reverseEdge);
    }

    public Map<String, Node> getNodes() {
        return nodes;
    }

    public Edge getEdge(Node source, Node destination) {
        return source.getEdges().stream()
                .filter(edge -> edge.getDestination().equals(destination))
                .findFirst()
                .orElse(null);
    }

    public void clear() {
        nodes.clear();
    }
}
package sdn.logical;

import java.util.ArrayList;
import java.util.List;

public class Node {
    private final String id;
    private final List<Edge> edges;

    public Node(String id) {
        this.id = id;
        this.edges = new ArrayList<>();
    }

    public String getId() {
        return id;
    }

    public List<Edge> getEdges() {
        return edges;
    }

    public void addEdge(Edge edge) {
        edges.add(edge);
```

```
    }

    // Метод для перевірки, чи існує з'єднання до певного вузла
    public boolean hasEdgeTo(Node destination) {
        for (Edge edge : edges) {
            if (edge.getDestination().equals(destination)) {
                return true;
            }
        }
        return false;
    }
    // Метод для отримання з'єднання до певного вузла
    public Edge getEdgeTo(Node destination) {
        for (Edge edge : edges) {
            if (edge.getDestination().equals(destination)) {
                return edge;
            }
        }
        return null;
    }
}
package sdn.logical;

public class Edge {
    private final Node source;
    private final Node destination;
    private final int weight;
    private int delay;          // Затримка в мс
    private int bandwidth;      // Пропускна здатність у Мбіт/с
    private double packetLoss; // Ймовірність втрати пакету (0.0 - 1.0)

    public Edge(Node source, Node destination, int weight, int delay, int
bandwidth, double packetLoss) {
        this.source = source;
        this.destination = destination;
        this.weight = weight;
        this.delay = delay;
        this.bandwidth = bandwidth;
        this.packetLoss = packetLoss;
    }

    public Node getSource() {
        return source;
    }

    public Node getDestination() {
        return destination;
    }

    public int getWeight() {
        return weight;
    }

    public int getDelay() {
        return delay;
    }

    public void setDelay(int delay) {
```

```
        this.delay = delay;
    }

    public int getBandwidth() {
        return bandwidth;
    }

    public void setBandwidth(int bandwidth) {
        this.bandwidth = bandwidth;
    }

    public double getPacketLoss() {
        return packetLoss;
    }

    public void setPacketLoss(double packetLoss) {
        this.packetLoss = packetLoss;
    }
}
package sdn.ui;

import javafx.scene.Scene;
import javafx.stage.Stage;
import sdn.logical.*;

import javafx.geometry.Insets;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.scene.text.Text;
import sdn.tests.ChartPlotter;
import sdn.tests.NetworkTester;


import java.util.List;

public class NetworkTesterUI {
    private final BorderPane root;
    private final GraphVisualizer graphVisualizer;
    private final NetworkGraph graph;
    private final TextArea logArea;
    private final Logger logger;

    public NetworkTesterUI() {
        this.logArea = new TextArea();
        this.logger = new Logger(logArea);
        this.root = new BorderPane();
        this.graph = new NetworkGraph();
        this.graphVisualizer = new GraphVisualizer(graph);

        setupUI();
    }

    public BorderPane getRoot() {
        return root;
    }

    private void setupUI() {
        VBox controls = new VBox(10);
```

```
        controls.setPadding(new Insets(10));

        // Create the menu bar
        MenuBar menuBar = new MenuBar();
        Menu fileMenu = new Menu("Menu");
        MenuItem openWindowItem = new MenuItem("Open Test Window");
        openWindowItem.setOnAction(e -> openTestWindow());
        fileMenu.getItems().add(openWindowItem);
        menuBar.getMenus().add(fileMenu);

        // Existing controls setup
        TextField nodeCountInput = new TextField();
        nodeCountInput.setPromptText("Nodes");

        TextField edgeCountInput = new TextField();
        edgeCountInput.setPromptText("Edges");

        Button generateGraphButton = new Button("Generate graph");
        generateGraphButton.setOnAction(e -> {
            int nodeCount = Integer.parseInt(nodeCountInput.getText());
            int edgeCount = Integer.parseInt(edgeCountInput.getText());
            generateGraph(nodeCount, edgeCount);
            logArea.clear(); // Очищуємо логи при генерації нового графа
        });

        // Rest of the controls
        TextField sourceInput = new TextField();
        sourceInput.setPromptText("Node X");

        TextField destinationInput = new TextField();
        destinationInput.setPromptText("Node Y");

        ComboBox<String> algorithmChoice = new ComboBox<>();
        algorithmChoice.getItems().addAll("MultiPathRouting", "Dijkstra");
        algorithmChoice.setValue("MultiPathRouting");

        Button findPathButton = new Button("Find path");
        findPathButton.setOnAction(e -> {
            String source = sourceInput.getText();
            String destination = destinationInput.getText();
            String algorithm = algorithmChoice.getValue();
            findPath(source, destination, algorithm);
        });

        controls.getChildren().addAll(
                new Text("Graph generation"),
                nodeCountInput, edgeCountInput, generateGraphButton,
                new Separator(),
                new Text("Path finding"),
                sourceInput, destinationInput, algorithmChoice, findPathButton,
                new Label("Logs"), logArea
        );

        VBox mainLayout = new VBox(menuBar, controls);
        root.setLeft(mainLayout);
        root.setCenter(graphVisualizer.getCanvas());
    }
```

```
    private void generateGraph(int nodeCount, int edgeCount) {
        graph.clear();  // Очищаємо граф перед новою генерацією
        // Логіка для генерації графа з випадковими з'єднаннями
        for (int i = 0; i < nodeCount; i++) {
            graph.addNode(String.valueOf((char) ('A' + i))); // Додаємо вузли A, B,
C тощо
        }
        for (int i = 0; i < edgeCount; i++) {
            String source = String.valueOf((char) ('A' + (int)(Math.random() *
nodeCount)));
            String destination = String.valueOf((char) ('A' + (int)(Math.random() *
nodeCount)));
            if (!source.equals(destination)) {
                int delay = 5 + (int)(Math.random() * 20);
                int bandwidth = 50 + (int)(Math.random() * 100);
                graph.addEdge(source, destination, 1, delay, bandwidth, 0.01 *
Math.random());
            }
        }
        graphVisualizer.updateGraph();
    }

    private void findPath(String sourceId, String destinationId, String algorithm)
{
        if (algorithm.equals("MultiPathRouting")) {
            MultiPathRouting multiPathRouting = new MultiPathRouting(graph);
            List<List<Node>> paths = multiPathRouting.findAvailablePaths(sourceId,
destinationId);
            logger.logPaths(paths);
            List<Node> path = multiPathRouting.selectOptimalPath(paths);
            int delay = calculateDelay(path);
            int bandwidth = calculateBandwidth(path);
            logger.logOptimalPath(path, delay, bandwidth);
            graphVisualizer.displayPaths(paths, multiPathRouting);

        } else if (algorithm.equals("Dijkstra")) {
            DijkstraRouting dijkstraRouting = new DijkstraRouting(graph);
            List<Node> path = dijkstraRouting.findShortestPath(sourceId,
destinationId);
            int delay = calculateDelay(path);
            int bandwidth = calculateBandwidth(path);
            logger.logPath(path, delay, bandwidth);
            graphVisualizer.displayPathDijkstra(path);
        }
    }

    private int calculateDelay(List<Node> path) {
        int delay = 0;
        for (int i = 0; i < path.size() - 1; i++) {
            Node from = path.get(i);
            Node to = path.get(i + 1);
            Edge edge = from.getEdgeTo(to);
            delay += edge.getDelay();
        }
        return delay;
    }
```

```
    private int calculateBandwidth(List<Node> path) {
        int bandwidth = Integer.MAX_VALUE;
        for (int i = 0; i < path.size() - 1; i++) {
            Node from = path.get(i);
            Node to = path.get(i + 1);
            Edge edge = from.getEdgeTo(to);
            bandwidth = Math.min(bandwidth, edge.getBandwidth());
        }
        return bandwidth;
    }

    private void openTestWindow() {
        Stage newWindow = new Stage();
        newWindow.setTitle("Test Window");

        VBox layout = new VBox(10);
        layout.setPadding(new Insets(10));
        Label label = new Label("Test window.");

        // Existing controls setup
        TextField nodeCountInput = new TextField();
        nodeCountInput.setPromptText("Nodes");

        TextField edgeCountInput = new TextField();
        edgeCountInput.setPromptText("Edges");

        TextField testCountInput = new TextField();
        testCountInput.setPromptText("Num of tests");

        Button executeTests = new Button("Execute Tests");
        executeTests.setOnAction(e -> {
            int nodeCount = Integer.parseInt(nodeCountInput.getText());
            int edgeCount = Integer.parseInt(edgeCountInput.getText());
            int testCount = Integer.parseInt(testCountInput.getText());
            generateGraph(nodeCount, edgeCount);
            executeTests(testCount);
        });

        layout.getChildren().addAll(label, nodeCountInput, edgeCountInput,
testCountInput, executeTests);

        Scene scene = new Scene(layout, 400, 300);
        newWindow.setScene(scene);
        newWindow.show();
    }

    private void executeTests(int testCount) {

        NetworkTester tester = new NetworkTester(graph);

        // Запуск тестів
        tester.runTests(testCount);

        ChartPlotter plotter = new ChartPlotter();
        plotter.plotComparisonCharts(tester.getMultiPathResults(),
tester.getDijkstraResults());
    }
```

```
}
package sdn.ui;

import sdn.logical.Edge;
import sdn.logical.MultiPathRouting;
import sdn.logical.NetworkGraph;
import sdn.logical.Node;

import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class GraphVisualizer {
    private final Canvas canvas;
    private final NetworkGraph graph;
    private final Map<String, Double[]> nodePositions;

    public GraphVisualizer(NetworkGraph graph) {
        this.canvas = new Canvas(600, 600);
        this.graph = graph;
        this.nodePositions = new HashMap<>();
    }

    public Canvas getCanvas() {
        return canvas;
    }

    public void clearHighlightedPath() {
        GraphicsContext gc = canvas.getGraphicsContext2D();
        gc.clearRect(0, 0, canvas.getWidth(), canvas.getHeight()); // Очищення
канви
        updateGraph(); // Перемальовуємо граф без виділення шляху
    }

    public void updateGraph() {
        GraphicsContext gc = canvas.getGraphicsContext2D();
        gc.clearRect(0, 0, canvas.getWidth(), canvas.getHeight());

        // Розташування вузлів по колу для кращої структури
        int nodeCount = graph.getNodes().size();
        double radius = Math.min(canvas.getWidth(), canvas.getHeight()) / 2 - 50;
        double centerX = canvas.getWidth() / 2;
        double centerY = canvas.getHeight() / 2;
        int i = 0;

        for (Node node : graph.getNodes().values()) {
            double angle = 2 * Math.PI * i / nodeCount;
            double x = centerX + radius * Math.cos(angle);
            double y = centerY + radius * Math.sin(angle);

            nodePositions.put(node.getId(), new Double[]{x, y});
            i++;
        }
```

```
        // Відображення зв'язків
        for (Node node : graph.getNodes().values()) {
            Double[] startPos = nodePositions.get(node.getId());
            for (Edge edge : node.getEdges()) {
                Double[] endPos = nodePositions.get(edge.getDestination().getId());
                gc.setStroke(Color.GRAY);
                gc.setLineWidth(2);
                gc.strokeLine(startPos[0], startPos[1], endPos[0], endPos[1]);

                // Відображення метрик (затримка та пропускна здатність)
                double midX = (startPos[0] + endPos[0]) / 2;
                double midY = (startPos[1] + endPos[1]) / 2;
                gc.setFill(Color.BLACK);
                gc.fillText("D: " + edge.getDelay() + ", B: " +
edge.getBandwidth(), midX, midY);
            }
        }

        // Відображення вузлів
        for (Map.Entry<String, Double[]> entry : nodePositions.entrySet()) {
            String nodeId = entry.getKey();
            Double[] pos = entry.getValue();
            gc.setFill(Color.BLUE);
            gc.fillOval(pos[0] - 5, pos[1] - 5, 10, 10);
            gc.setFill(Color.BLACK);
            gc.fillText(nodeId, pos[0] - 10, pos[1] - 10);
        }
    }

    public void displayPaths(List<List<Node>> paths, MultiPathRouting
routingAlgorithm) {
        clearHighlightedPath(); // Очищення попередніх шляхів

        GraphicsContext gc = canvas.getGraphicsContext2D();
        gc.setStroke(Color.GREEN);
        gc.setLineWidth(3);

        // Відображення всіх шляхів
        for (List<Node> path : paths) {
            displayPath(path, Color.GREEN);
        }

        // Вибір та відображення оптимального шляху
        List<Node> optimalPath = routingAlgorithm.selectOptimalPath(paths);
        if (optimalPath != null) {
            displayPath(optimalPath, Color.RED); // Виділяємо оптимальний шлях
червоним кольором
        }
    }
    public void displayPathDijkstra(List<Node> path) {
        clearHighlightedPath(); // Очищення попередніх шляхів

        displayPath(path, Color.RED);
    }

    public void displayPath(List<Node> path, Color color) {
```

```
        if (path == null || path.size() < 2) return;

        GraphicsContext gc = canvas.getGraphicsContext2D();
        gc.setStroke(color);
        gc.setLineWidth(3);

        for (int i = 0; i < path.size() - 1; i++) {
            Node from = path.get(i);
            Node to = path.get(i + 1);

            Double[] startPos = nodePositions.get(from.getId());
            Double[] endPos = nodePositions.get(to.getId());

            gc.strokeLine(startPos[0], startPos[1], endPos[0], endPos[1]);
        }
    }
}
package sdn.ui;

import javafx.scene.control.TextArea;
import sdn.logical.Node;

import java.util.List;

public class Logger {
    private TextArea logArea;

    public Logger(TextArea logArea) {
        this.logArea = logArea;
    }

    public void log(String message) {
        logArea.appendText(message + "\n");
    }

    public void logPath(List<Node> path, int delay, int bandwidth) {
        StringBuilder sb = new StringBuilder();
        sb.append("Path found: ");

        for (Node node : path) {
            sb.append(node.getId()).append(" -> ");
        }

        // Видаляємо останню стрілку
        sb.setLength(sb.length() - 4);

        sb.append("\nDelay: ").append(delay).append(" ms");
        sb.append("\nBandwidth: ").append(bandwidth).append(" Mbps");

        log(sb.toString());
    }

    public void logPaths(List<List<Node>> paths) {
        log("Multiple paths found:");
        int index = 1;

        for (List<Node> path : paths) {
            StringBuilder sb = new StringBuilder();
```

```
            sb.append("Path ").append(index).append(": ");

            for (Node node : path) {
                sb.append(node.getId()).append(" -> ");
            }

            sb.setLength(sb.length() - 4);
            log(sb.toString());
            index++;
        }
    }

    public void logOptimalPath(List<Node> optimalPath, int delay, int bandwidth) {
        StringBuilder sb = new StringBuilder();
        sb.append("Optimal path: ");

        for (Node node : optimalPath) {
            sb.append(node.getId()).append(" -> ");
        }

        sb.setLength(sb.length() - 4);
        sb.append("\nDelay: ").append(delay).append(" ms");
        sb.append("\nBandwidth: ").append(bandwidth).append(" Mbps");

        log(sb.toString());
    }
}
package sdn.tests;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.category.DefaultCategoryDataset;
import javax.swing.*;
import java.util.List;

public class ChartPlotter {
    public void plotComparisonCharts(List<TestResult> multiPathResults,
List<TestResult> dijkstraResults) {
        if (multiPathResults.size() != dijkstraResults.size()) {
            throw new IllegalArgumentException("Results lists must be of the same
size.");
        }

        JFrame frame = new JFrame("Network Test Comparison");

        frame.add(createChartPanel("Delay (ms)", "Traffic Flow", "Total Delay",
                multiPathResults, dijkstraResults, TestResult::getTotalDelay));

        frame.add(createChartPanel("Bandwidth (Mbps)", "Traffic Flow", "Min
Bandwidth",
                multiPathResults, dijkstraResults, TestResult::getMinBandwidth));

        frame.add(createChartPanel("Packet loss prob", "Traffic Flow", "Packet Loss
Probability",
                multiPathResults, dijkstraResults, result -> (int)
(result.getPacketLossProbability() * 100)));
```

```
        frame.add(createChartPanel("Hops", "Traffic Flow", "Hop Count",
                multiPathResults, dijkstraResults, TestResult::getHopCount));

        frame.setLayout(new java.awt.GridLayout(4, 1)); // Змінено на 4, щоб
відповідати кількості графіків
        frame.setSize(800, 600);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }


    private ChartPanel createChartPanel(String title, String categoryAxisLabel,
String valueAxisLabel,
                                        List<TestResult> multiPathResults,
List<TestResult> dijkstraResults,
                                        java.util.function.Function<TestResult,
Integer> metricFunction) {
        DefaultCategoryDataset dataset = new DefaultCategoryDataset();
        int size = Math.min(multiPathResults.size(), dijkstraResults.size());

        for (int i = 0; i < size; i++) {
            dataset.addValue(metricFunction.apply(multiPathResults.get(i)),
"MultiPath", "Flow " + (i + 1));
            dataset.addValue(metricFunction.apply(dijkstraResults.get(i)),
"Dijkstra", "Flow " + (i + 1));
        }

        JFreeChart chart = ChartFactory.createLineChart(title, categoryAxisLabel,
valueAxisLabel,
                dataset, PlotOrientation.VERTICAL, true, true, false);

        return new ChartPanel(chart);
    }

}
package sdn.tests;

import sdn.logical.*;

import java.util.ArrayList;
import java.util.List;

public class NetworkTester {
    private final NetworkGraph graph;
    private final MultiPathRouting multiPathRouting;
    private final DijkstraRouting dijkstraRouting;
    private final TrafficGenerator trafficGenerator;

    private final List<TestResult> multiPathResults;
    private final List<TestResult> dijkstraResults;

    public NetworkTester(NetworkGraph graph) {
        this.graph = graph;
        this.multiPathRouting = new MultiPathRouting(graph);
        this.dijkstraRouting = new DijkstraRouting(graph);
        this.trafficGenerator = new TrafficGenerator();
        this.multiPathResults = new ArrayList<>();
```

```
        this.dijkstraResults = new ArrayList<>();
    }

    public void runTests(int numberOfFlows) {
        List<TrafficFlow> trafficFlows = trafficGenerator.generateTraffic(graph,
numberOfFlows);

        for (TrafficFlow flow : trafficFlows) {

            System.out.println("Source " + flow.getSource().getId() + " Dest " +
flow.getDestination().getId());
            // Багатошляховий алгоритм

            long startTime = System.nanoTime();
            List<List<Node>> multiPathRoutes = multiPathRouting.findAvailablePaths(
                    flow.getSource().getId(), flow.getDestination().getId()
            );
            System.out.println("Finding disjoint paths:");
            for (List<Node> path : multiPathRoutes) {
                path.forEach(node -> System.out.print(node.getId() + " "));
                System.out.println();
            }
            long endTime = System.nanoTime();

            long duration = (endTime - startTime);
            System.out.println("Multipath time:" + duration);
            startTime = System.nanoTime();
            if (!multiPathRoutes.isEmpty()) {
                System.out.println("Finding one paths:");

                multiPathRouting.selectOptimalPath(multiPathRoutes).forEach(node ->
System.out.print(node.getId() + " "));
                System.out.println();
                TestResult multiPathResult =
analyzeRoute(multiPathRouting.selectOptimalPath(multiPathRoutes), flow);
                multiPathResults.add(multiPathResult);
            }

            endTime = System.nanoTime();

            duration = (endTime - startTime);
            System.out.println("Optimal path time:" + duration);

            // Алгоритм Дейкстри
            startTime = System.nanoTime();
            List<Node> dijkstraRoute =
dijkstraRouting.findShortestPath(flow.getSource().getId(),
flow.getDestination().getId());
            if (!dijkstraRoute.isEmpty()) {
                System.out.println("Finding dijkstraRoute:");

                dijkstraRoute.forEach(node -> System.out.print(node.getId() + "
"));
                System.out.println();
                TestResult dijkstraResult = analyzeRoute(dijkstraRoute, flow);
                dijkstraResults.add(dijkstraResult);
            }
            endTime = System.nanoTime();
```

```
            duration = (endTime - startTime);
            System.out.println("Dijkstra time:" + duration);
        }
    }

    private TestResult analyzeRoute(List<Node> route, TrafficFlow flow) {
        int totalDelay = 0;
        int minBandwidth = Integer.MAX_VALUE;
        double packetLoss = 1.0;
        int hopCount = route.size() - 1; // Calculate hop count as the number of
edges

        for (int i = 0; i < route.size() - 1; i++) {
            Node source = route.get(i);
            Node destination = route.get(i + 1);
            Edge edge = getEdgeBetween(source, destination);

            if (edge != null) {
                totalDelay += edge.getDelay();
                minBandwidth = Math.min(minBandwidth, edge.getBandwidth());
                packetLoss *= (1 - edge.getPacketLoss());
            }
        }


        return new TestResult(totalDelay, minBandwidth, (1 - packetLoss),
hopCount);
    }


    private Edge getEdgeBetween(Node source, Node destination) {
        for (Edge edge : source.getEdges()) {
            if (edge.getDestination().equals(destination)) {
                return edge;
            }
        }
        return null;
    }

    public List<TestResult> getMultiPathResults() { return multiPathResults; }
    public List<TestResult> getDijkstraResults() { return dijkstraResults; }
}
package sdn.tests;

public class TestResult {
    private int totalDelay;
    private int minBandwidth;
    private double packetLossProbability;
    private int hopCount;


    public TestResult(int totalDelay, int minBandwidth, double
packetLossProbability, int hopCount) {
        this.totalDelay = totalDelay;
        this.minBandwidth = minBandwidth;
        this.packetLossProbability = packetLossProbability;
        this.hopCount = hopCount;
```

```
    }

    public int getTotalDelay() { return totalDelay; }
    public int getMinBandwidth() { return minBandwidth; }
    public double getPacketLossProbability() { return packetLossProbability; }

    public int getHopCount() {
        return hopCount;
    }
}
package sdn.tests;

import sdn.logical.Node;

public class TrafficFlow {
    private Node source;
    private Node destination;
    private int requiredBandwidth;
    private int maxDelay;

    public TrafficFlow(Node source, Node destination, int requiredBandwidth, int
maxDelay) {
        this.source = source;
        this.destination = destination;
        this.requiredBandwidth = requiredBandwidth;
        this.maxDelay = maxDelay;
    }

    public Node getSource() {
        return source;
    }

    public void setSource(Node source) {
        this.source = source;
    }

    public Node getDestination() {
        return destination;
    }

    public void setDestination(Node destination) {
        this.destination = destination;
    }

    public int getRequiredBandwidth() {
        return requiredBandwidth;
    }

    public void setRequiredBandwidth(int requiredBandwidth) {
        this.requiredBandwidth = requiredBandwidth;
    }

    public int getMaxDelay() {
        return maxDelay;
    }

    public void setMaxDelay(int maxDelay) {
        this.maxDelay = maxDelay;
```

```
    }
}
package sdn.tests;

import sdn.logical.NetworkGraph;
import sdn.logical.Node;

import java.util.ArrayList;
import java.util.List;

public class TrafficGenerator {
    public List<TrafficFlow> generateTraffic(NetworkGraph graph, int numberOfFlows)
{
        List<TrafficFlow> trafficFlows = new ArrayList<>();

        for (int i = 0; i < numberOfFlows; i++) {
            Node source = getRandomNode(graph);
            Node destination;

            // Генерація призначення, яке відрізняється від джерела
            do {
                destination = getRandomNode(graph);
            } while (destination.equals(source));

            // Створення трафіку з випадковими параметрами
            int requiredBandwidth = 10 + (int) (Math.random() * 50); // 10-50
Мбіт/с
            int maxDelay = 10 + (int) (Math.random() * 100);          // 10-100 мс

            TrafficFlow flow = new TrafficFlow(source, destination,
requiredBandwidth, maxDelay);
            trafficFlows.add(flow);
        }
        return trafficFlows;
    }

    private Node getRandomNode(NetworkGraph graph) {
        List<Node> nodes = new ArrayList<>(graph.getNodes().values());
        return nodes.get((int) (Math.random() * nodes.size()));
    }
}
package sdn;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import sdn.ui.NetworkTesterUI;

public class Main extends Application {
    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("SDN Network Path Finder");

        NetworkTesterUI networkTesterUI = new NetworkTesterUI();
        Scene scene = new Scene(networkTesterUI.getRoot(), 800, 600);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
```

```
    public static void main(String[] args) {
        launch(args);
    }
}
```