Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Штучного інтелекту
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Моделі штучного інтелекту для управління та оптимізації peer-to-peer мереж
(тема)

Виконав:
студент 2 курсу, групи СШІм-21-1
Кучук А.О.
(прізвище, ініціали)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)
Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)
Освітня програма Системи штучного інтелекту
(повна назва спеціалізації)

Керівник доц. Волощук О.Б.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ________________ В.О. Філатов
(підпис) (прізвище, ініціали)

2023 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Штучного інтелекту
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системи штучного інтелекту (СШІ)
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ________________
(підпис)
«______» ___________ 20 ___ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Кучук Аліна Олегівна
(прізвище, ім'я, по батькові)

1. Тема роботи «Моделі штучного інтелекту для управління та оптимізації peer-to-peer мереж»

затверджена наказом університету від 31 березня 20 23 р. № 306Ст
2. Термін подання студентом роботи до екзаменаційної комісії 16 травня 20 23 р.
3. Вихідні дані до роботи Книги з розробки нейронних мереж, статі на тему моделей штучного інтелекту для управління та оптимізації peer-to-peer мереж. Документація хмарного провайдера, документація сервісів хмарного провайдера.

4. Перелік питань, що потрібно опрацювати в роботі Аналіз предметної області та постановка задачі, Аналіз та опис peer-to-peer мереж та їх учасників, Огляд методів та алгоритмів штучного інтелекту для управління та оптимізації peer-to-peer мереж, Огляд Freenet, Огляд Gnutella, Проектування системи, Рішення фундаментальних проблем, Проектування нейронної мережі для оптимізації трафіку, Застосування алгоритму штучного інтелекту для оптимізації марштрутизації, Розробка додатка, Розробка інфраструктури додатка, Розробка інтелектуальної частини додатка, Розробка мурашиної колонії ACO.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) Рисунок 1.1 – Схема стільникової мережі; Рисунок 1.2 – Схема комунікації агентів через Wi-Fi; Рисунок 2.1 – Мурахи обирають найкоротший шлях; Рисунок 2.2 – Список однорангових партнерів у Freenet 0.7; Рисунок 3.1 – Схема DHT; Рисунок 3.2 – Схема кодеру; Рисунок 3.3 – Схема декодеру; Рисунок 3.4 – Схема предиктору ймовірностей;

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

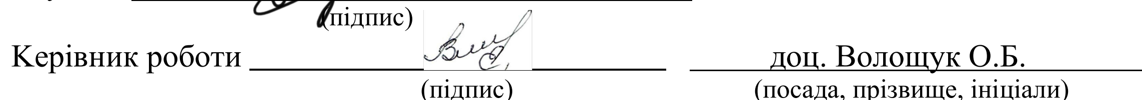
Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання	03.04.2023	Виконано
2	Аналіз завдання та пошук літератури	04.04.2023-07.04.2023	Виконано
3	Аналіз предметної області	08.04.2023-10.04.2023	Виконано
4	Огляд існуючих рішень	12.04.2023-13.04.2023	Виконано
5	Огляд методів та алгоритмів штучного інтелекту	14.04.2023-16.04.2023	Виконано
6	Проектування архітектури додатка	16.04.2023-18.04.2023	Виконано
7	Розробка інфраструктурної частини додатка	18.04.2023-20.04.2023	Виконано
8	Розробка інтелектуальної складової додатка	20.04.2023-25.04.2023	Виконано
9	Оформлення пояснювальної записки	26.04.2023-01.05.2023	Виконано
10	Захист роботи	16.05.2023	Виконано

Дата видачі завдання 3 квітня 20 23 р.

Студент
(підпис)

Керівник роботи  доц. Волощук О.Б.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 100 с., 8 рис., 1 табл., 3 дод., 39 джерел.

АЛГОРИТМИ ЗБЕРЕЖЕННЯ ДАНИХ, МАШИННЕ НАВЧАННЯ, МЕТОДИ ТА АЛГОРИТМИ ШТУЧНОГО ІНТЕЛЕКТУ, ОПТИМІЗАЦІЯ ТРАФІКУ РОЗПОДІЛЕНИХ СИСТЕМ, РОЗПОДІЛЕНІ PEER-TO-PEER СИСТЕМИ.

Об'єкт дослідження – розробка системи машинного навчання для оптимізації трафіку або алгоритмів збереження даних в peer-to-peer мережі.

Мета роботи – проектування та реалізація машинного навчання на великій кількості агентів з використанням хмарних технологій та поширення моделей навчання між агентами для оптимізації пропускної спроможності та ефективності мережі.

Області застосування – комунікації та хмарні технології.

Методи дослідження – аналіз теоретичного матеріалу, технічної літератури, ринку існуючих рішень та практична реалізація самостійно розробленого додатка.

Визначено вимоги до роботи мережі, проблеми пов’язані з розподіленими мережами, розроблено та запроваджено алгоритми машинного навчання для пошуку патернів в даних та зменшення розміру навантаження.

ABSTRACT

The explanatory note: 100 p., 8 fig., 1 tabl., 3 app., 39 sources.

ARTIFICIAL INTELLIGENCE METHODS AND ALGORITHMS, DATA STORAGE ALGORITHMS, DISTRIBUTED PEER-TO-PEER SYSTEMS, MACHINE LEARNING, TRAFFIC OPTIMISATION OF DISTRIBUTED SYSTEMS.

The object of research is the development of a machine learning system for traffic optimisation or data storage algorithms in a peer-to-peer network.

The purpose of the work is to design and implement machine learning on many agents using cloud technologies and distribute learning models between agents to optimise network capacity and efficiency.

Scopes – communications and cloud technologies.

Research methods – analysis of theoretical material, technical literature, the market of existing solutions and practical implementation of a self-developed application.

The requirements for network operation and problems associated with distributed networks were identified, and machine learning algorithms were developed and implemented to search for patterns in the data and reduce the size of the load.

ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень та термінів..............8

Вступ..9

1 Аналіз предметної області та теоретичні основи peer-to-peer мережі..............11

1.1 Аналіз предметної області 11

1.2 Вимоги до агентів розподіленої системи 12

1.3 Способи комунікації... 13

1.3.1 Стільникова мережа 13

1.3.2 Мережа інтернет 14

1.4 Вимоги до функціональності мережі.. 16

1.5 Архітектура розподіленої peer-to-peer мережі збереження даних......... 17

1.5.1 Концептуальна архітектура мережі 17

1.5.2 Процес масштабування 18

1.5.3 Процес реплікації... 19

1.5.4 Членство в мережі 21

1.5.5 Виявлення відмов 22

1.5.6 Локальні стани агентів.. 24

1.5.7 Приватність та безпека даних в мережі... 24

1.6 Штучний інтелект в розподілених системах 25

1.6.1 Алгоритми штучного інтелекту для оптимізації маршрутизації даних 27

1.7 Постановка задачі дослідження 28

1.7.1. Класифікація обраної задачі... 28

1.7.2 Визначення вимог до функціоналу 28

2 Методи та алгоритми штучного інтелекту застосовані в peer-to-peer мережах 29

2.1 Ройовий інтелект 29

2.1.1 Алгоритми оптимізації мурашиних колоній.. 30

2.1.2 Алгоритми оптимізації бджолосімей.. 32

2.1.3 Алгоритми оптимізації рою частинок 34
2.2 Арифметичне кодування на основі нейронних мереж............................... 36
2.3 Огляд існуючих рішень.. 39
2.3.1 Freenet 39
2.3.2 Gnutella 41
2.2.3 Інші імплементації.. 44
3 Проектування системи 45
3.1 Рішення фундаментальних проблем.. 45
3.2 Локальне сховище даних 46
3.3 Реплікація 50
3.4 Нейронна мережа для оптимізації даних 51
3.5 Застосування ШІ для оптимізації маршрутизації...................................... 57
3.6 Деталі імплементації 59
4 Розробка додатка 60
4.1 Розробка інфраструктури додатка 60
4.2 Розробка інтелектуальної частини додатка.. 66
Висновки..72
Перелік джерел посилання 74
Додаток А Імплементація ACO алгоритму 78
Додаток Б Імплементація DRAC Model 95
Додаток В Відомість кваліфікаційної роботи 100

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ ТА ТЕРМІНІВ

AC – Arithmetic Coding – арифметичне кодування;

ACO – Ant Colony Optimization – оптимізація мурашиних колоній;

ASP.NET Core – Active Server Pages Network Enabled Technologies – платформа розробки web-додатка;

DHT – Distributed Hash Table – розподілена хеш-таблиця;

DRAC – Delta Recurrent Arithmetic Coder – арифметичне кодер за використанням дельта-рекурентної нейронної мережі;

FL – Federated Learning – федеративне навчання;

GRU – Gated Recurrent Unit – вентильні рекурентні вузли;

LAN – Local Area Network – локальні мережі;

LSTM – Long Short-Term Memory – довга короткочасна пам'ять;

P2P – Peer-To-Peer – однорангова мережа;

RNN – Recurrent Neural Network – рекурентна нейронна мережа;

TCP – Transmission Control Protocol – транспортний протокол передачі даних;

UDP – User Datagram Protocol – найпростіший протокол транспортного рівня моделі OSI;

WAN – Wide Area Network – глобальні мережі.

ВСТУП

Темп, з яким змінюються комп'ютерні системи, був, є і продовжує бути приголомшливим. Починаючи з 1945 року, коли розпочалася сучасна ера комп'ютерів, і до приблизно 1985 року, комп'ютери були великими та дорогими. Крім того, через відсутність способу підключення, ці комп'ютери працювали незалежно один від одного. Починаючи з середини 1980-х років, дві прогресивні технологічні розробки почали змінювати цю ситуацію.

Перша розробка – це потужні мікропроцесори. Спочатку це були машини з 8-бітним процесором, але незабаром з'явилися 16-, 32- та 64-бітні ЦПУ. З мультиядерними ЦПУ з'явився виклик адаптувати та розробляти програми таким чином, щоб ефективно використовувати паралелізм. Теперішнє покоління машин має обчислювальну потужність головних комп'ютерів, що були використані 30 або 40 років тому, але за 1/1000 частки ціни або ще менше.

Другою розробкою стала розробка швидкісних комп'ютерних мереж. Локальні мережі, або LAN, дозволяють підключати тисячі комп'ютерів в будівлі таким чином, що невеликі об'єми інформації можуть бути передані за декілька мікросекунд. Більші обсяги даних можуть переміщуватися між комп'ютерами зі швидкістю від мільярдів бітів на секунду (біт/с). Глобальні мережі, або WAN, дозволяють підключати сотні мільйонів комп'ютерів по всьому світу з різною швидкістю від десяти до тисячі тисяч мільйонних бітів.

Сьогодні смартфони, які має кожен у світі, і які можуть вміститися в кишеню, є потужнішими, ніж обчислювальна машина, яка керувала Аполлоном під час польоту на місяць в 1969 році. Об'єднавши велику кількість таких пристроїв, можна створювати peer-to-peer мережу, яка може бути потужнішою, ніж будь-який мейнфрейм у світі, та яка здатна оперувати ресурсами пам'яті та обчислювальної потужності в кількості, яку неможливо зібрати в одному місці. Однак управління та оптимізація

розподілених систем стають все складнішими з розвитком технологій та збільшенням їх обсягів.

У peer-to-peer мережах, де кожен вузол може бути одночасно і клієнтом, і сервером, кількість збоїв може значно збільшуватися. Крім того, відмова одного вузла може мати вплив на роботу всієї мережі. Швидкість передачі даних є іншою важливою проблемою в peer-to-peer мережах, оскільки вона залежить від того, скільки вузлів зможуть поділитися файлами і як швидко вони зможуть це зробити. Це може призвести до повільної передачі даних у великих мережах з великою кількістю вузлів. Проте, застосувавши моделі штучного інтелекту для управління та оптимізації peer-to-peer мереж, можна досягти досить ефективного рішення. Пошуку варіанту такого оптимального та ефективного рішення і присвячена ця робота.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ТЕОРЕТИЧНІ ОСНОВИ PEER-TO-PEER МЕРЕЖІ

1.1 Аналіз предметної області

В індустрії було запропоновано різні визначення розподілених систем, проте жодне з них не є повним і не погоджується з жодним з сучасних визначень. Для цілей даної роботи достатньо визначити розподілену систему як сукупність обчислювальних елементів, які можуть поводитися незалежно один від одного. Кожен з обчислювальних елементів може бути апаратним пристроєм або процесом програмного забезпечення і мати назву вузла або агента. Другою характеристикою є те, що користувачі (люди чи програми) повинні вважати, що вони мають справу з єдиною системою, незважаючи на те, що розподілені вузли можуть бути автономними і мати різні способи зв'язку. Тому налагодження співпраці між вузлами є важливою особливістю розвитку розподілених систем [1].

На практиці вузли програмуються з метою досягнення спільних цілей, які реалізуються шляхом обміну повідомленнями між ними. Кожен вузол реагує на отримані повідомлення, які потім обробляються та викликають подальше спілкування через передачу повідомлень. Фундаментальною проблемою у роботі незалежних вузлів є те, що кожен з них має своє власне уявлення про час, і в системі не існує жодного зразка глобального годинника. Це призводить до фундаментальних питань щодо синхронізації та координації в розподіленій системі. Крім того, оскільки мережа складається з набору вузлів, то може знадобитися керування членством та організацією цієї колекції. Іншими словами, може бути необхідним зареєструвати, які вузли можуть або не можуть належати до системи, а також надати кожному учаснику список вузлів, з якими він може безпосередньо спілкуватися.

Як зазначалося, розподілена система повинна виглядати як єдина цілісна система. У деяких випадках дослідники навіть зайшли так далеко, щоб почали стверджувати, що повинно бути єдине системне уявлення, тобто кінцеві користувачі не повинні навіть помічати, що вони мають справу з розподіленою комп'ютерною мережею, де процеси, дані та контроль розподілені по вузлах мережі. Досягнення єдиної системи часто вимагає великих зусиль. З цієї причини у визначенні розподіленої системи зазначено, що розподілена система є узгодженою, якщо вона поводиться відповідно до очікувань своїх користувачів, або, іншими словами, в єдиній узгодженій системі набір вузлів працює однаково, незалежно від того, де, коли та як відбувається взаємодія між користувачем і системою.

1.2 Вимоги до агентів розподіленої системи

Для спрощення та доведення працездатності та стабільності системи можна описати потенціального агента мережі та припустити, що тільки такого виду вузли будуть членами мережі. Найпоширенішими і найпотужнішими портативними девайсами є смартфони та комп'ютери.

Першою і найбільш важливою характеристикою, якою повинен бути наділений потенційний агент – це надійність, адже агент повинен працювати стабільно та надійно в умовах мережі, забезпечуючи безперебійну передачу даних.

Смартфони, зазвичай, мають три способи обміну інформацією: стільникова мережа, мережа інтернет та Bluetooth. Останній спосіб не варто розглядати як надійний та швидкий, оскільки до специфікації Bluetooth 5.3 пропускна здатність контроллерів – 1-2 Мбіт/с [2].

У таблиці 1.1 приведена максимальна пропускна здатність в залежності від способу комунікації.

Таблиця 1.1 – Способи комунікації

Назва	Пропускна здатність (Гбіт\с)
Bluetooth	0.001~
Стільникова мережа	10.0-1.0
Мережа Інтернет	10.0-2.0

З таблиці видно, що Bluetooth є найповільнішим, та не підходить для використання в peer-to-peer мережах. Іншим важливим фактором та характеристикою агентів – є наявність довгої пам’яті для збереження даних. Зважаючи на це можна розширити список пристроїв потенціальних агентів до комп’ютерів та смартфонів.

1.3 Способи комунікації

1.3.1 Стільникова мережа

У стільниковій мережі беруть участь мобільні пристрої, які здатні комунікувати з вежами стільникового зв’язку. Зазвичай, сигнал надходить до вежі, яка має зрозуміти, куди передати сигнал. Наступним може бути або інша вежа, або перенаправлення на інтернет інфраструктуру комунікації. Таким чином, якщо запит надходить до якогось веб-сайту або серверу, то вежа використовує інтернет інфраструктуру. У іншому випадку, якщо запит, наприклад VOIP, то вежа знаходить коротший шлях до іншого пристрою, що має бути в зоні покриття мережі стільникового зв’язку [2].

На рисунку 1.1. зображена типова стільникова мережа з декількома вежами та агентами, які комунікують через мережу.

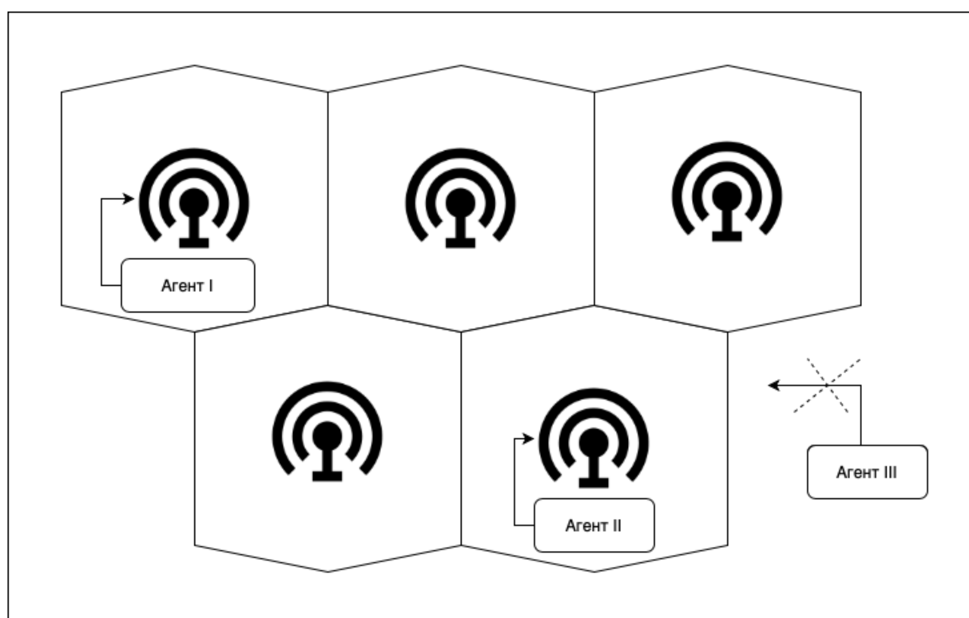


Рисунок 1.1 – Схема стільникової мережі

Таким чином, якщо агент 1 захоче відіслати дані агенту 2, то сигнал пройде через три стільникової вежі і агент 1 вже не буде мати можливості передати дані до агенту 3, або агент 3 не може передати дані агенту 1 і агенту 2. Така вершина не може бути частиною мережі або бути членом розподіленої мережі. Якщо до цього агент 3 був частиною розподіленої мережі, і зараз знаходиться поза мережею, то дані цього агенту тимчасово, або назавжди втрачено.

1.3.2 Мережа інтернет

Найрозповсюдженіший спосіб комунікації – це мережа інтернет. Інтернет має розповсюджену інфраструктуру зв’язку, починаючи від локальних провайдерів, які підключені до більших частин інфраструктури, до кабелів оптичного волокна на дні океану, для зв’язку між континентами [2].

Щоб мати можливість бездротового зв’язку було розроблено Wi-Fi технологію, яка покриває незначну площу.

На рисунку 1.2 зображена схема комунікації агентів через Wi-Fi та оптичне волокно.

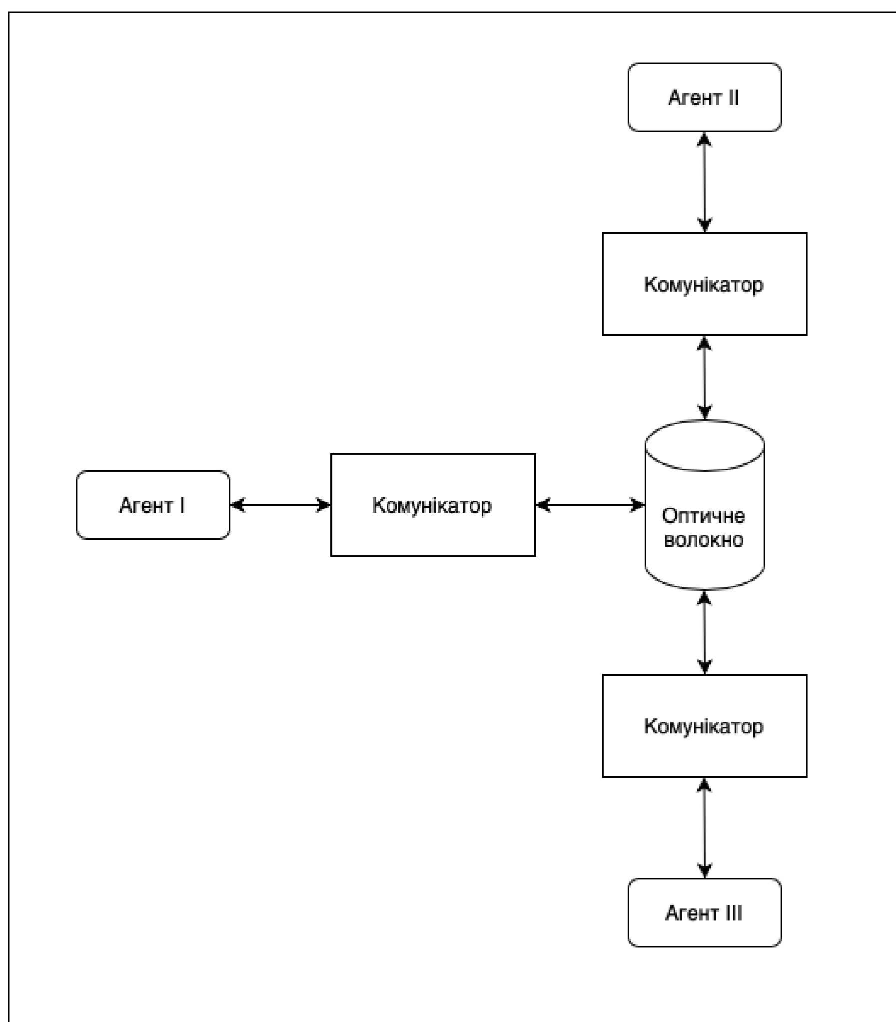


Рисунок 1.2 – Схема комунікації агентів через Wi-Fi

Комунікатор – це девайс, який підтримує специфікації Wi-Fi 802.11, в свою чергу агент також повинен підтримувати стандарт. Комунікатори використовують оптичне волокно (найшвидший фізичний спосіб передачі даних по кабелю) та інфраструктурну мережу інтернет провайдера для комунікації.

Таким чином, щоб відправити дані або зв’язатися агенту 1 з агентом 3, девайс надсилає запит до інфраструктури та локального провайдера, що

відповідає за перенаправлення і маршрутизацію запиту до конкретного агенту, в цьому випадку, агенту 3.

1.4 Вимоги до функціональності мережі

Найпростішою вимогою до розподіленої мережі є зберігання даних, що, в свою чергу, передбачає обмін та зчитування даних. Таким чином, першою та найголовнішою вимогою є здатність будь-якого агента записувати дані в мережу, зазначаючи при цьому рівень доступності, що впливає на надійність та транзакційність операцій. Крім того, будь-який агент повинен мати змогу зчитувати дані з мережі, до яких він має доступ, або дані, які він записав раніше.

Оскільки розроблювана мережа є розподіленою системою, то при записі в мережу, рівнем прийняття варто назвати чисельну міру, при якій вважається, що мережа прийняла дані, та дані будуть збережені. Мережа буде робити все можливе, щоб підтримувати дані доступними в мережі (реплікувати, переміщати) в будь-який момент часу.

Однією з проблем розподіленої peer-to-peer мережі, яка виступає в ролі розподіленої бази даних, є проблема втрати агентів з будь-яких причин. Фізичне переривання комунікації, закінчення трафіку, збої в мережних сервісах провайдерів – все це може бути причиною збоїв та втрати даних. Фактично не існує способу отримати дані з агента, який вийшов з мережі з будь-яких причин. Тому однією з вимог є реплікація даних.

Варто зазначити, що якщо всі агенти мережі втратять доступ або здатність комунікації, то це означає нездатність мережі функціонувати і вважається критичним станом, відновлення після якого можливе тільки за умови, що агенти почнуть знову приєднуватись. Повністю відновленою мережею варто вважати такий набір підключених агентів, при якому можливо читати будь-які дані, що були записані до цього, у найгіршому випадку – підключення одного агента з реплікаційної групи.

Існують строгі вимоги до надійності та ефективності системи, і для підтримки постійного зростання вона повинна бути високомасштабованою. Робота з відмовами в інфраструктурі, що складається з тисяч компонентів, є стандартним режимом роботи, оскільки завжди є невелика, але значна кількість серверних та мережевих компонентів, які можуть відмовляти у будь-який момент. Тому програмні системи повинні бути побудовані таким чином, щоб вони розглядали відмови як норму, а не виняток.

Також система повинна мати дуже високу пропускну здатність для запису, мільярди записів на день, і також масштабуватися з кількістю користувачів або агентів. Оскільки агенти можуть брати участь з різних географічних точок, здатність реплікувати дані між регіонами є ключовою для зниження часу очікування на результати пошуку.

1.5 Архітектура розподіленої peer-to-peer мережі збереження даних

1.5.1 Концептуальна архітектура мережі

Архітектура системи зберігання даних, яка повинна працювати в умовах реального часу, є складною. Крім компонента збереження даних, система повинна мати наступні характеристики: масштабованість та надійність рішення для балансування навантаження, визначення членства та виявлення відмов, відновлення відмов, синхронізація реплік, обробка перевантажень, передача стану, конкурентоспроможність та планування задач, маршрутизація запитів, моніторинг та сповіщення про аварії та управління конфігурацією. Описання деталей кожного рішення виходить за межі цієї роботи, тому варто сконцентруватися на основних техніках розподілених систем, які використовуються в світі:

– розбиття;

– реплікація;

– визначення членства;

– обробка відмов;

– масштабування.

Усі ці модулі повинні працюють у взаємозв'язку, щоб обробляти запити на читання/запис. Зазвичай запит на читання/запис ключа направляється до будь-якого вузла у кластері мережі. Далі вузол визначає репліки для цього певного ключа. Для записів система повинна направляти запити до реплік та чекати на кворум реплік для підтвердження завершення записів. Для читання, залежно від гарантій сталості, необхідних для клієнта, система направляє запити до найближчої репліки або до всіх реплік та чекає на кворум відповідей.

1.5.2 Процес масштабування

Одна з ключових особливостей дизайну мережі – можливість інкрементного масштабування. Це потребує можливості динамічного розподілу даних на набір вузлів (тобто, агентів зберігання) у кластері. Пропускна здатність повинна пропорційно зростати при збільшенні агентів у мережі. Це досягається при розподіленні даних, що зберігаються, або при балансуванні навантаження [3].

Мережа може розподіляти дані до інших агентів по кластеру за допомогою консистентного хешування, яке використовує функцію хешування зі збереженням порядку. Тобто, кожен сусід мережі буде мати свій хеш код, який використовується для визначення надійних сусідів, які можуть реплікувати дані [4].

Визначення хеш коду мережі використовують консистентне хешування, де діапазон виходу функції хешування розглядається як фіксований круговий простір або «кільце» (тобто, найбільше значення хешу обертається до найменшого значення хешу). Кожен вузол системи отримує випадкове значення в межах цього простору, що відображає його позицію на кільці. Кожен елемент даних, ідентифікований ключем, призначається

вузлу, хешуючи ключ елемента даних, щоб отримати його позицію на кільці, та обходячи кільце за годинниковою стрілкою, щоб знайти перший вузол з позицією, більшою за позицію елемента. Цей вузол вважається координатором для цього ключа. Додаток вказує цей ключ, а система використовує його для маршрутизації запитів. Таким чином, кожен вузол стає відповідальним за регіон на кільці між ним і його попередником на кільці. Основна перевага консистентного хешування полягає в тому, що вибуття або прибуття агенту впливає лише на його найближчих сусідів, і інші вузли залишаються незмінними.

Основний алгоритм консистентного хешування створює деякі складнощі. По-перше, випадкове призначення позицій кожному вузлу на колі призводить до нерівномірного розподілу даних та навантаження. По-друге, основний алгоритм не знає про різні рівні продуктивності вузлів. Зазвичай існує два способи вирішення цієї проблеми: один полягає в тому, щоб вузлам було призначено кілька позицій на колі, а другий – у аналізі інформації про навантаження на колі та переміщенні вузлів з легким навантаженням на колі, щоб зменшити навантаження на важко навантажених вузлах. Для цілей цієї роботи краще підходить другий спосіб, оскільки це робить дизайн та реалізацію дуже простими і допомагає приймати дуже детерміновані рішення щодо балансування навантаження.

Саме ця проблема є однією з фундаментальних проблем, які може вирішити штучний інтелект. Пошук інших вузлів для найкращого масштабування та перенаправлення навантаження може здійснюватися за допомогою алгоритмів штучного інтелекту, для оцінки різних аспектів вузлів, такі як його надійність, відмовостійкість та швидкість.

1.5.3 Процес реплікації

Реплікація – це практика збереження копій даних на інших агентах, що робить їх доступними в будь-який момент часу [5]. Мережі

використовують реплікацію як засіб досягнення високої доступності та стійкості даних. Кожен елемент даних реплікується на N хостах, де N – фактор реплікації, налаштований на кожному екземплярі. Кожному агенту призначається координаторний вузол, який відповідає за реплікацію елементів даних, що потрапляють в його діапазон. Крім локального зберігання кожного ключа у своєму діапазоні, координатор реплікує ці ключі на N-1 вузлів у кільці.

Існують різні політики реплікації, такі як «Rack Unaware», «Rack Aware» та «Neighbour Aware». Для стратегії реплікації «Rack Unaware» нерезидентні репліки вибираються шляхом вибору N-1 наступників координатора в кільці. Для стратегій «Rack Aware» та «Datacenter Aware» алгоритм є складнішим [6], [7].

Мережі можуть також використовувати імплементацію типу zookeeper, систему, яка обирає лідера серед своїх вузлів, щоб визначити, які вузли відповідають за які діапазони. Лідер забезпечує те, що жоден вузол не відповідає за більше, ніж N-1 діапазонів у кільці, а метадані про діапазони, за які відповідає кожен вузол, кешуються локально на кожному вузлі та в стійкому до помилок способі всередині Zookeeper. Вузли, які відповідають за певний діапазон, вважаються «списком переваг» для діапазону [8].

Кожен вузол у своїй системі знає про кожен інший вузол і діапазон, за який він відповідає. У такому випадку, мережа забезпечує гарантії стійкості в присутності відмов вузлів та розділень мережі, послаблюючи вимоги до кворуму.

Відмови в мережі можуть статися через відключення живлення, відмову мережі та природні катастрофи. Кластер повинен бути налаштований так, що кожна частина даних реплікується в кількох регіонах мережі.

1.5.4 Членство в мережі

У peer-to-peer мережі виявлення нових вузлів є важливою проблемою, яку необхідно вирішити, щоб вузли могли з'єднуватися і спілкуватися один з одним. Існує декілька підходів до вирішення цієї проблеми.

Підхід на основі завантажувального вузла. У цьому підході існує заздалегідь визначений вузол, відомий як завантажувальний вузол, який вже підключений до мережі. Коли новий вузол приєднується до мережі, він спочатку зв'язується з завантажувальним вузлом, щоб отримати список інших вузлів у мережі. Потім новий вузол підключається до цих вузлів і починає брати участь у мережі.

Цей метод виглядає централізованим і змушує мати псевдо-агентів, або згенеровані точки в мережі, які призначені для координації з'єднань, що є досить суперечливим рішенням. Згідно з вимогами до мережі, кожен агент повинен поводитися незалежно і покладатися на інших агентів лише в невеликій кількості випадків, що має зробити його надійним учасником мережі.

Підхід на основі широкомовлення. У цьому підході кожен вузол періодично транслює в мережу повідомлення про свою присутність. Коли новий вузол приєднується до мережі, він прослуховує ці повідомлення і використовує їх для виявлення інших вузлів у мережі.

Цей метод схожий на підхід Query Flooding, який змушує надсилати багато запитів до невідомих мереж через спеціальний DNS-маршрут або в обхід NAT, щоб знайти інших агентів. Цей метод не є продуктивним з точки зору затримок.

Розподілені хеш-таблиці також можна використовувати для пошуку і виявлення інших агентів, а не тільки для пошуку в сховищі даних. Цей метод є високопродуктивним, оскільки пошуковий запит виконується за одиницю складності.

Підхід на основі випадкового блукання. У цьому підході новий вузол вибирає випадковий вузол з мережі і запитує у нього список інших вузлів. Потім новий вузол зв'язується з цими вузлами і повторює процес, поки не отримає достатню кількість вузлів для з'єднання. У цьому методі важко передбачити, який агент буде обраний першим, оскільки вимога до реплікації і доступності в регіональному сенсі є важливою, цей алгоритм також важливо застосовувати в мережі, що розробляється [9].

Це лише кілька прикладів підходів, які можуть бути використані для виявлення нових вузлів в peer-to-peer мережі. Найкращий підхід залежить від конкретних вимог і обмежень мережі. Наприклад, якщо мережа велика і дуже динамічна, ефективнішим може бути підхід на основі DHT, тоді як якщо мережа невелика і стабільна, може бути достатньо завантажувального вузла.

1.5.5 Виявлення відмов

Виявлення відмов – це механізм, за допомогою якого вузол може локально визначити, чи працює будь-який інший вузол у системі. У кластері виявлення відмов також використовується для уникнення спроб зв'язатися з недосяжними вузлами під час різних операцій.

Phi Accrual Failure Detection – це адаптивний алгоритм виявлення збоїв, який забезпечує будівельний блок для реалізації детекторів збоїв у будь-якій розподіленій системі. Замість того, щоб надавати вихідні дані у вигляді булевої логіки (система в робочому чи неробочому стані), детектор відмов на основі нарахування виводить інформацію про підозру (рівень) на безперервній шкалі, так що чим вище значення підозри, тим більша ймовірність того, що система не в робочому стані [10].

Рівень підозри визначається як φ, що видається детектором збоїв, і оскільки алгоритм є адаптивним, значення буде динамічним і відображатиме поточні умови мережі та поведінку системи. Чим менша

ймовірність отримання «серцебиття» [11] від агенту, тим більша ймовірність того, що система вийшла з ладу, отже, тим більшим має бути значення φ; деталі щодо математичного вираження φ наведені нижче.

$$\varphi(t_{now}) = -log_{10}P(t_{now}), \quad (1.1)$$

де $P(t_{now})$ – ймовірність отримання «серцебиття» прямо зараз.

Значення φ, обчислене за допомогою $-log_{10}$, також вказує на те, що ймовірність помилки зменшується експоненціально зі збільшенням значення φ. Отже, якщо стверджувати, що вузол А є підозрілим, коли $\varphi = 1$, то ймовірність того, що буде зроблено помилку становить приблизно 10%, з $\varphi = 2$ ймовірність становить близько 1%, з $\varphi = 3$ – $0{,}1\%$ і тощо.

Ймовірність отримання наступного «серцебиття» від агенту пропорційна ймовірності того, що відповідь прийде більш ніж через t одиниць після попереднього, тобто чим довше чекати, тим менше шансів отримати «серцебиття».

Щоб реалізувати це, зберігається вибіркове ковзаюче вікно, що містить час приходу минулих «серцевих скорочень». Щоразу, коли надходить нове «серцебиття», час його надходження зберігається у вікні, а дані щодо найстарішого «серцебиття» видаляються.

Інтервали надходження відповідають нормальному розподілу, що вказує на те, що більшість «серцевих скорочень» надходять в межах певного діапазону, в той час як є декілька, які надходять із запізненням через різні умови мережі або системи. З інформації, що зберігається у вікні, можна легко обчислити інтервали надходження, середнє значення і дисперсію, які потрібні для оцінки ймовірності.

1.5.6 Локальні стани агентів

Мережа покладається на локальну файлову систему для збереження даних. Дані на диску повинні бути представлені у форматі, який підходить для ефективного отримання даних. Звичайна операція запису включає запис у файл реєстрації для забезпечення стійкості та відновлення, та оновлення у внутрішній структурі даних у пам'яті. Запис у внутрішню структуру даних у пам'яті виконується лише після успішного запису у файл реєстрації.

Кожен агент повинен мати відведений диск на кожному девайсі для файлу реєстрації, оскільки всі записи в файл реєстрації є послідовними, що допоможе максимізувати пропускну здатність диску. Коли внутрішня структура даних у пам'яті перетинає певний поріг, який обчислюється на основі розміру даних та кількості об'єктів, вона сама себе виводить на диск. Ця операція запису виконується на одному з багатьох звичайних дисків, якими обладнані машини. Усі записи є послідовними на диску та також генерують індекс для ефективного пошуку за ключем рядка. Ці індекси також зберігаються разом із файлом даних. З часом на диску може бути багато таких файлів, тому фоновий процес злиття запускається, щоб об'єднати різні файли у один. Цей процес дуже схожий на процес компактизації, який відбувається у системі Bigtable від Google [12].

1.5.7 Приватність та безпека даних в мережі

Для покращення безпекового фактору мережі можна використовувати федеративне навчання. Федеративне навчання (FL) спрямоване на навчання алгоритму машинного навчання, наприклад, глибокої нейронні мережі, на декількох локальних наборах даних, що містяться в локальних вузлах, без явного обміну зразками даних. Загальний принцип полягає у навчанні місцевих моделей на місцевих зразках даних та обміні ними параметри (наприклад, ваги та упередження глибокої нейронної мережі) між цими

локальними вузлами на деякій частоті, щоб сформувати глобальну модель, спільну для всіх вузлів.

Федеративне навчання допомагає у навчанні алгоритму машинного навчання та зберігає дані на рівні пристроїв. Це означає, що FL дозволяє кожному пристрою зберігати свої особисті та локальні дані. Ця технологія забезпечить широке поширення рішень машинного навчання, а також гнучкі та керовані дані в режимі реального часу [13].

Методику можна використовувати для вирішення численних завдань та у різних контекстах. Вона включає процедури навчання алгоритмів в режимі «офлайн» і «онлайн». В залежності від умов експлуатації та типу даних алгоритм підбере відповідну методику. Традиційний метод, наприклад, централізоване машинне навчання, не включає ці переваги і пов'язаний з високим ризиком для захисту даних і передачі великих файлів.

Горизонтальне федеративне навчання та однорідне федеративне навчання можуть вирішувати технічні та практичні завдання шляхом поділу даних на різні підрозділи. Процес працює за рахунок введення аналогічних наборів даних у порівнянний простір. Алгоритм порівнює характеристики та пов'язує їх відповідним чином.

У об'єднаному вертикальному навчанні різні набори даних мають однакові ідентифікатори вибірки, але різні функціональні простори.

Таким чином федеративне навчання допомагає покращити безпековий фактор мережі. Кожен агент навчає свою модель на своїх даних та розповсюджує її на інші агенти.

1.6 Штучний інтелект в розподілених системах

Існує кілька способів використання моделей штучного інтелекту в peer-to-peer мережах для оптимізації та покращення їх продуктивності.

Балансування навантаження – це процес розподілу мережевого трафіку між декількома вузлами в мережі peer-to-peer для оптимізації

продуктивності та забезпечення того, щоб жоден вузол не перевантажувався. Моделі ШІ можна використовувати для прогнозування навантаження на кожен вузол і динамічного коригування розподілу навантаження для оптимізації продуктивності та запобігання вузьким місцям [14].

У peer-to-peer мережі такі ресурси, як пропускна здатність та простір для зберігання, обмежені та мають розподілятися ефективно. Моделі штучного інтелекту можна використовувати для прогнозування попиту на ресурси та їх динамічного розподілу для оптимізації продуктивності та забезпечення ефективного використання ресурсів.

Peer-to-peer мережі можуть бути вразливими до збоїв, якщо значна кількість вузлів виходить з ладу або стає недоступною. Моделі штучного інтелекту можна використовувати, щоб передбачати, коли вузли ймовірно вийдуть з ладу, і вживати профілактичних заходів для запобігання збою, наприклад тиражування даних між кількома вузлами.

Peer-to-peer мережі зазвичай зберігають великі обсяги даних, і ефективне керування даними має важливе значення для забезпечення їх доступності та актуальності. Моделі штучного інтелекту можна використовувати для аналізу шаблонів використання даних і оптимізації розміщення та отримання даних для підвищення продуктивності та зменшення затримки мережі.

У peer-to-peer мережі вузли повинні спілкуватися один з одним, щоб обмінюватися даними та координувати свою діяльність. Моделі ШІ можна використовувати для оптимізації алгоритмів маршрутизації та забезпечення ефективної та надійної маршрутизації даних між вузлами [15].

Пропонується зосередити розробку алгоритму та моделі штучного інтелекту для оптимізації управління даними для підвищення відмовостійкості, так як відмовостійкість в мережі, де агенти непередбачувані в своїй поведінці є критично важливою.

1.6.1 Алгоритми штучного інтелекту для оптимізації маршрутизації даних

Існує декілька моделей і алгоритмів ШІ, які можна використовувати для керування даними в peer-to-peer мережах.

Для локалізації агентів використовують розподілені хеш-таблиці (DHT) – це децентралізована структура даних, яка використовується для розподіленого зберігання та пошуку пар ключ-значення. DHT можна використовувати в peer-to-peer мережах для керування зберіганням і пошуком даних, а моделі ШІ можна використовувати для оптимізації алгоритмів DHT для кращої продуктивності та масштабованості [16].

Один з розділів штучного інтелекту є ройовий інтелект який вивчає колективну поведінку децентралізованих самоорганізованих систем. У peer-to-peer мережах інтелект роїв може використовуватися для керування зберіганням і пошуком даних, а також для оптимізації продуктивності мережі в цілому [17].

Алгоритми машинного навчання можна використовувати для аналізу моделей використання даних у peer-to-peer мережах і прогнозування, до яких даних, ймовірно, часто звертатимуться. Це можна використовувати для оптимізації розміщення та пошуку даних і покращення загальної продуктивності мережі. Використовуючи навчання з підкріпленням, який зосереджується на оптимізації прийняття рішень у складних середовищах.

Загалом, вибір моделей штучного інтелекту та алгоритмів для керування даними в peer-to-peer мережах залежить від конкретних вимог і обмежень мережі, а також від типу даних, які зберігаються та витягуються.

Отже, знаючи вимоги до мережі, маючи її архітектуру та просту імплементацію потрібно використати машинне навчання для пошуку патернів та ройовий інтелект для підвищення продуктивності.

1.7 Постановка задачі дослідження

1.7.1. Класифікація обраної задачі

Вирішувана задача відносяться до числа важливих у цій сфері, а саме полягає в розробці алгоритмів машинного навчання на великій кількості агентів для покращення працездатності мережі. А саме необхідно реалізувати:

– модель пошуку шаблонів даних;

– модель пошуку та забезпечення найкращого шляху передачі даних в мережі;

– розробити алгоритм для поширення збереженої та навченої моделі на агенті через мережу інтернет;

– застосувати алгоритм для шифрування моделі;

1.7.2 Визначення вимог до функціоналу

Ця робота визначає мету як: отримання працездатної системи розподіленої мережі, яка здатна самовдосконалюватись за допомогою машинного навчання на великій кількості агентів підключених до неї.

Ідеальним результатом було би розробити програмне забезпечення, яке з’єднується з мережею та бере участь в кластері, який має змогу зберігати, читати дані та виступати як єдиний механізм розподілених обчислювань, що в потенціалі може замінити централізовані сервери.

Пристрій повинен працювати через інтернет або інші засоби комунікації, проводити оновлення навченої моделі та відправлення того, чому локальна модель була навчена.

2 МЕТОДИ ТА АЛГОРИТМИ ШТУЧНОГО ІНТЕЛЕКТУ ЗАСТОСОВАНІ В PEER-TO-PEER МЕРЕЖАХ

2.1 Ройовий інтелект

Peer-to-peer мережі при певному рівні абстрагування можуть представляти зручне середовище для застосування алгоритмів ройового інтелекту.

Ройовий інтелект можна описати як колективну поведінку, що виникла у соціальних комах, які працюють за дуже обмеженою кількістю правил. Самоорганізація є основною темою з обмеженими обмеженнями від взаємодії між агентами. Багато відомих прикладів ройового інтелекту походять зі світу тварин, таких як пташині зграї, косяки риб і рої жуків. Соціальна взаємодія між окремими агентами допомагає їм більш ефективно адаптуватися до навколишнього середовища, оскільки більше інформації збирається від усього рою [18], [19].

Для моделювання широкої поведінки рою, виведено та адаптовано під задачі цієї роботи кілька загальних принципів інтелекту рою.

Принцип близькості: базові одиниці мережі повинні бути здатні до простих обчислень, пов'язаних з навколишнім середовищем. Тут обчислення розглядаються як пряма поведінкова реакція на відхилення в навколишньому середовищі, наприклад, викликані взаємодією між агентами. Залежно від складності задіяних агентів, реакції можуть сильно відрізнятися. Однак, деякі фундаментальні моделі поведінки є спільними, наприклад, реплікація даних на певні агенти або пошук коротшого та оптимальнішого, з точки зору витраченого часу, маршрут.

Принцип якості: окрім базової обчислювальної здатності, мережа повинена мати можливість реагувати на фактори пропускної здатності, доступності агентів, на які кожен окремий агент може покластися.

Принцип диверсифікованого реагування: ресурси не повинні бути сконцентровані у вузькому регіоні. Розподіл повинен бути спланований таким чином, щоб кожен агент був максимально захищений від коливань навколишнього середовища.

Принцип стабільності та адаптивності: очікується, що мережа адаптуються до коливань навколишнього середовища без швидкої зміни режимів, оскільки зміна режимів вимагає витрат енергії.

2.1.1 Алгоритми оптимізації мурашиних колоній

Найбільш визнаним прикладом ройового інтелекту в реальному світі є мурахи. У пошуках їжі мурахи вирушають зі своєї колонії і рухаються в довільному порядку в усіх напрямках. Як тільки мураха знаходить їжу, вона повертається до колонії і залишає на своєму шляху слід хімічних речовин, які називаються феромонами. Інші мурахи можуть виявити феромон і йти тим самим шляхом. Цікаво те, що частота відвідування мурахами стежки визначається концентрацією феромону вздовж неї. Оскільки феромон природно випаровується з часом, довжина шляху також є фактором. Отже, з огляду на всі ці міркування, коротший шлях буде кращим, оскільки мурахи, які йдуть цим шляхом, продовжують додавати феромон, що робить його концентрацію достатньо сильною, щоб протистояти випаровуванню.

Якщо привести аналогії з розподіленими мережами, то можна сказати, що шлях, який знаходить одиниця в системі може і потрібен бути доступний всій системі. Таким чином, якщо поєднавши декілька агентів в регіональний кластер і реплікувати дані, то щоб підвищити доступність та час відклику варто забезпечити механізм запам’ятовування коротших і найкращих шляхів, на які інші агенти можуть покластися. Якщо між двома агентами, протягом якогось проміжку часу встановився стабільний зв’язок, це може відбутися з різних причин, агенти можуть буть супер девайсами, або мати надшвидкий зв’язок, то кожний агент в регіональному кластері повинен

знати про цей зв’язок та надавати перевагу, якщо таке можливо в спілкуванні через цих двох агентів.

У результаті виникає найкоротший шлях від колонії до їжі (рисунок 2.1).

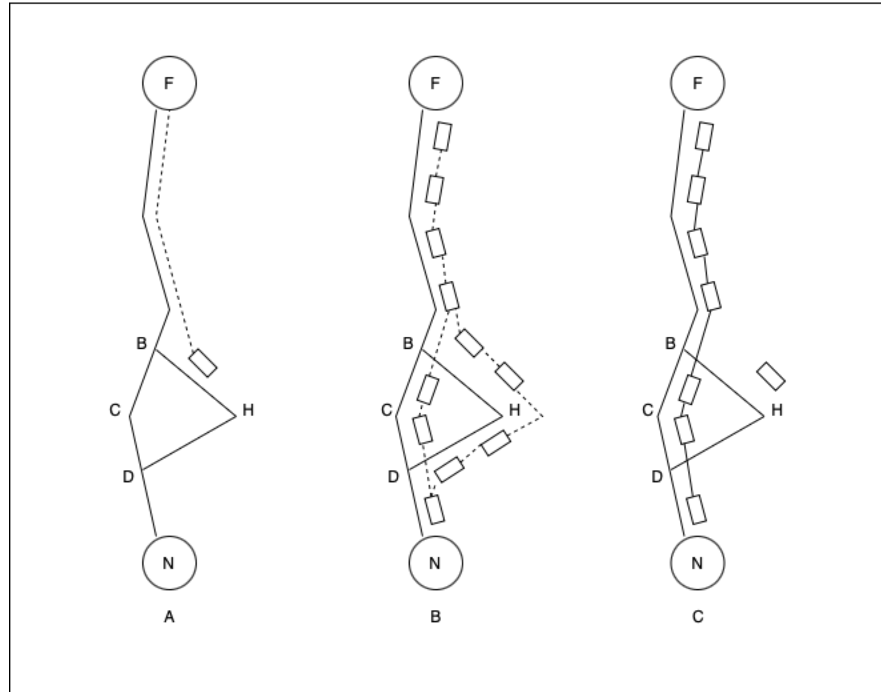


Рисунок 2.1 – Мурахи обирають найкоротший шлях

На основі вивчення поведінки реальних мурах було сформульовано метаевристику для розв'язування комбінаторних задач, таких як задача комівояжера [20].

Визначимо комбінаторну задачу P як триплет (S, Ω, f), де S – простір пошуку над дискретними змінними рішення $X_i \in D_i = \{v_i^1, \dots, v_i^{|D_i|}\}$, Ω – множина обмежень, а цільова функція $f: S \rightarrow R$, яку потрібно максимізувати або мінімізувати.

Розв'язок $s \in S$ присвоює значення змінним, які задовольняють Ω, і шукає розв'язок $s^* \in S$ такий, що $f(s^*)$ є глобальним мінімумом або максимумом.

Алгоритми оптимізації мурашиних колоній вирішують проблеми цієї категорії за допомогою концепції феромонів. Метаевристика ACO поділяється на три етапи (лістинг 2.1) [21].

Лістинг 2.1 – Метаевристика ACO

```
Initialization;
while not terminated do
     Construct solution using artifical ants;
     Local search (optional);
     Update pheromones;
end
```

Рішення з використанням штучних псевдо агентів: використовуючи m штучних псевдо агентів, розв'язки $C = \{c_{ij}\}, i = 1 \ldots n, j = 1 \ldots D_i$, які задовольняють усі обмеження Ω, де c_{ij} задає змінну рішення $X_i = v_i^j$. Це також можна розглядати як випадкове маршрут на маршрутному графі $G_c(V, E)$.

Локальний пошук: за допомогою спеціального дизайну для конкретної проблеми локальний пошук може покращити знайдене рішення. Однак, оскільки він сильно варіюється залежно від проблеми, локальний пошук є необов'язковим процесом.

Оновлення оптимальних шляхів і слідів: значення маршрутних агентів для перспективних рішень буде збільшено, а значення для небажаних рішень буде зменшено за рахунок старіння. Таким чином, найкращі рішення будуть винагородженні найвищою концентрацією запитів.

2.1.2 Алгоритми оптимізації бджолосімей

Як і мурахи, бджоли мають схожу поведінку збирання їжі. Замість феромонів алгоритм оптимізації бджолиних колоній покладається на

поведінку медоносних бджіл під час пошуку їжі. На першому етапі деякі бджоли відправляються на пошуки перспективних джерел їжі. Після того, як хороше джерело їжі знайдено, бджоли повертаються назад до колонії і виконують танець виляння, щоб поширити інформацію про це джерело. Три частини інформації включають: відстань, напрямок, якість джерела їжі. Чим краща якість джерела їжі, тим більше бджіл буде приваблено. Таким чином, з'являється найкраще джерело їжі.

Провести паралелі з розподіленими системами досить легко. Якщо кожен з агентів виконавши запис буде мати зворотній зв’язок та розуміти, що агенти або агенти до якого він записав мають швидкий доступ та багато постійної пам’яті, то цей агент повинен мати змогу розширити знання про надійність агенту з іншим регіональним кластером.

Метаевристика, виведена з харчової поведінки бджіл, також може бути застосована для розв'язання комбінаторних задач, особливо задач на глобальний мінімум або максимум. Аналогічно, метаевристика BCO проходить кілька етапів (лістинг 2.2) [22].

Лістинг 2.2 – Метаевристика BCO

```
Initialization;
while not terminated do
     Employed Bees Phase;
     Onlooker Bees Phase;
     Scout Bees Phase;
     Memorize the best solution;
end
```

Ініціалізація: всі джерела агентів $\vec{F}_m, m = 1, \ldots, N$ ініціалізовано. $\vec{F}_m$ є розв'язками оптимізаційних задач і будуть налаштовані алгоритмом BCO на мінімізацію або максимізацію цільової функції f, визначеної вище.

Агенти в мережі будуть шукати в регіональному кластері $\vec{F}_m$ з пам'яті випадковий вектор $\vec{R}_m$. Для визначення того, чи веде $\vec{R}_m$ до кращого агенту,

обчислюється підходяща-функція. Звичайний вибір для підходящої-функції Т такий:

$$T(\vec{x}_m) = \begin{cases} \frac{1}{1 + f(\vec{x}_m)}, & \text{якщо } f(\vec{x}_m) \geq 0 \\ 1 + |f(\vec{x}_m)|, & \text{якщо } f(\vec{x}_m) < 0 \end{cases} \quad (2.1)$$

Агенти в кластер після того, як один з агентів поділилися інформацією про надійне джерело, виберуть місце призначення відповідно до неї. Зазвичай це розраховується залежно від значень підходящої-функції, наданих робочими агентами. Наприклад, при визначеному вище значенні $T(\vec{x}_m)$, значення ймовірності p_m може бути розраховане:

$$p_m = \frac{T(\vec{x}_m)}{\sum_{m=1}^{N} T(\vec{x}_m)} \quad (2.2)$$

Чим більше агентів залучається до багатших ресурсів, тим більше позитивних відгуків виникає і про багатші ресурси.

2.1.3 Алгоритми оптимізації рою частинок

Пташині зграї та косяки риб є натхненням від природи для алгоритмів оптимізації рою частинок. Вперше його запропонували Еберхарт і Кеннеді. Імітуючи фізичні величини, такі як швидкість і положення в пташиних зграях, штучні частинки створюються для "польоту" в просторі пошуку оптимізаційних задач [23].

Однак, на відміну від попередніх двох алгоритмів, які використовують феромон або зворотний зв'язок як інструменти для позбавлення від небажаних розв’язань, алгоритми оптимізації рою частинок оновлюють поточний розв'язок безпосередньо. Як можна бачити з наступного опису

роботи алгоритмів рою частинок, з меншою кількістю параметрів алгоритми рою частинок прості в реалізації і досягають глобальних оптимальних рішень з високою ймовірністю.

Спочатку популяція частинок рівномірно розподіляється в багатовимірному просторі пошуку цільової функції задачі оптимізації. З частинками асоціюються дві величини: вектор положення $\vec{x}_i$. та швидкість $\vec{v}_i$. На кожному часовому кроці швидкості частинок будуть оновлюватися згідно з наступною формулою:

$$\vec{v}_i^{t+1} = v_i^t + r_1 * \alpha\left(\vec{b} - \vec{x}_i^t\right) + r_2 * \beta(\vec{n} - \vec{x}_i^t) \tag{2.3}$$

де $\vec{b}$ – глобальне найкраще розташування;

$\vec{n}$ – найкраще розташування в околі частинки p_i.

Обидва α, β, є параметрами навчання, а r_1, r_2 є випадковими параметрами, що змінюються від 0 до 1.

Позиції будуть оновлюватися просто за допомогою:

$$\vec{x}_i^{t+1} = \vec{x}_i^t + \vec{v}_i^{t+1} \tag{2.4}$$

Важливість врахування найкращого сусідства $\vec{n}$ для кожної частинки полягає в тому, щоб уникнути потрапляння рою в пастку локального мінімуму. Саме тут в оптимізації рою частинок вступають у гру соціальні зв'язки. Топологія соціальних зв'язків називається топологією популяції рою. Існують різні типи топології, наприклад, топологія gbest. При такій топології всі частинки притягуються до глобального найкращого рішення, тому вона являє собою повністю пов'язану соціальну мережу. Топологія lbest, з іншого боку, з'єднує кожну частинку лише з C сусідами. Така топологія сповільнює процес збіжності порівняно з gbest, однак вона також робить алгоритми PSO більш здатними уникати локальних мінімумів.

Топологія колеса з'єднує сусідні частинки лише з однією частинкою – локальною. Цей тип має найменшу кількість зв'язків.

2.2 Арифметичне кодування на основі нейронних мереж

Арифметичне кодування (AC) – це метод, який використовується в багатьох традиційних алгоритмах стиснення без втрат, де дані кодуються як дробове число в певному діапазоні, а діапазон оновлюється на основі ймовірностей значень даних. Нейронні мережі можуть бути використані для оцінки цих ймовірностей, що може бути використано для покращення продуктивності стиснення [24].

Нижче приведений високорівневий огляд кроків, пов'язаних з підходом до арифметичного кодування на основі нейронних мереж.

Першим кроком є навчання. Нейронна мережа навчається на вхідних даних, щоб вивчити розподіл ймовірностей значень даних. Нейронна мережа приймає вхідні дані і виводить оцінені ймовірності для кожного значення даних. Навчальні дані зазвичай складаються з великого набору репрезентативних вибірок даних, які покривають діапазон можливих значень даних.

Наступним кроком є стиснення. Під час стиснення вхідні дані подаються в навчену нейронну мережу для оцінки ймовірностей значень даних. Ці ймовірності потім використовуються в поєднанні з арифметичним кодуванням для кодування даних. Процес арифметичного кодування враховує ймовірність присвоєння коротших кодів більш ймовірним значенням даних і довших кодів менш ймовірним значенням даних. Стислі дані разом з моделлю нейронної мережі та необхідною інформацією для декодування даних зберігаються або передаються.

Останнім кроком є декомпресія. Під час декомпресії стиснуті дані декодуються за допомогою арифметичного кодування, а оцінені ймовірності значень даних отримуються з навченої нейронної мережі. Ці

ймовірності використовуються для декодування стиснутих даних і відновлення вихідних даних. Модель нейронної мережі, що використовується під час стиснення, також необхідна під час декомпресії для точної оцінки ймовірностей.

Однією з переваг цього підходу є те, що нейронна мережа може вивчати розподіл ймовірностей на основі даних, вловлюючи складні закономірності та залежності, які не можуть бути легко використані традиційними алгоритмами стиснення. Однак цей підхід також може вимагати більше обчислювальних ресурсів порівняно з традиційними алгоритмами через залучення нейронних мереж. Ефективність підходу залежить від розміру та якості навчальних даних, архітектури та навчання нейронної мережі, а також специфічних характеристик даних, що стискаються.

Рекурентні нейронні мережі (RNN) можуть бути використані для покращення арифметичного кодування за допомогою збереження контексту попередніх символів в повідомленні [25].

Ідея полягає у створенні RNN, яка буде навчатися на вхідних повідомленнях та зберігати контекст попередніх символів. На кожному кроці RNN приймає на вхід символ повідомлення та контекст, а на виході генерує ймовірності для наступного символу у повідомленні. Ці ймовірності можуть бути використані для арифметичного кодування повідомлення.

Конкретний спосіб використання RNN в арифметичному кодуванні можна описати такими формулами:

Визначити вектор внутрішнього стану RNN h та початковий контекст c.

Приймати на вхід символ x_t повідомлення та попередній контекст c_{t-1}:

$$\text{input}_t = [x_t, c_{t-1}] \quad (2.5)$$

Обчислити вектор внутрішнього стану h_t та вектор вихідних значень y_t на кроці t:

$$h_t, y_t = \text{RNN}(\text{input}_t, h_{t-1}) \quad (2.6)$$

Застосувати softmax до вектору вихідних значень y_t, щоб отримати розподіл ймовірностей для наступного символу:

$$P(x_{t+1}|x_{\frac{1}{t}}, h_t) = softmax(y_t) \quad (2.7)$$

Використовувати арифметичне кодування з отриманими ймовірностями для кодування повідомлення.

Таким чином, RNN можна використовувати для навчання та генерації розподілу ймовірностей для наступного символу у повідомленні на кожному кроці. Цей розподіл може бути використаний для ефективного арифметичного кодування повідомлення.

Для покращення арифметичного кодування можна використовувати різні архітектури рекурентних нейронних мереж (RNN), такі як прості RNN, LSTM (Long Short-Term Memory) та GRU (Gated Recurrent Unit) [26].

Прості RNN мають проблему зі зникненням градієнту при тренуванні на довгих послідовностях, що може призвести до недостатньої передачі контексту на віддалених від початку послідовності кроках. Тому, для покращення арифметичного кодування краще використовувати більш складні архітектури, такі як LSTM та GRU.

LSTM та GRU здатні зберігати та передавати довгострокову залежність, що робить їх особливо корисними для задач обробки послідовностей. Вони досягають цього за рахунок використання механізмів, які дозволяють видаляти або додавати інформацію в залежності від поточного стану.

Зокрема, LSTM має структуру, яка дозволяє контролювати потік інформації в мережі та видаляти незначиму інформацію зі стану, що дозволяє уникнути проблеми з використанням попереднього контексту при тренуванні та генерації на віддалених кроках послідовності.

GRU має меншу кількість параметрів та меншу складність, ніж LSTM, тому він може бути ефективнішим використовувати в більш швидких системах арифметичного кодування, якщо достатньо здатний до передачі контексту на віддалених кроках послідовності.

Крім вибору конкретної архітектури RNN, важливо також визначити розмір внутрішнього стану мережі, що може впливати на її ефективність та точність.

Крім того, можна використовувати такі методи як dropout, batch normalization та інші методи регуляризації для покращення стійкості та зниження перенавчання мережі.

У цілому, використання RNN у арифметичному кодуванні може покращити ефективність та точність генерації стислого коду, зменшивши кількість бітів, що передаються, та зберігаючи якомога більше інформації про оригінальний текст.

2.3 Огляд існуючих рішень

2.3.1 Freenet

Freenet – це peer-to-peer платформа для анонімного спілкування, стійкого до цензури. Вона використовує децентралізоване розподілене сховище даних для зберігання та передачі інформації, а також має набір вільного програмного забезпечення для публікації та спілкування в Інтернеті без страху цензури. Як Freenet, так і деякі пов'язані з ним інструменти були спочатку розроблені Яном Кларком, який визначив мету

Freenet як забезпечення свободи слова в Інтернеті з надійним захистом анонімності [27].

Розподілене сховище даних Freenet використовується багатьма сторонніми програмами та плагінами для забезпечення мікроблогів та обміну медіа, анонімного та децентралізованого відстеження версій, ведення блогів, загальної мережі довіри для децентралізованого захисту від спаму.

Freenet може забезпечити анонімність в Інтернеті, зберігаючи невеликі зашифровані фрагменти контенту, що розповсюджуються на комп'ютерах користувачів, і підключаючись лише через проміжні комп'ютери, які передають запити на контент і відправляють їх назад, не знаючи вмісту повного файлу. Це схоже на те, як маршрутизатори в Інтернеті маршрутизують пакети, не знаючи нічого про файли – за винятком того, що Freenet має кешування, шар надійного шифрування і не покладається на централізовані структури. Це дозволяє користувачам публікувати анонімно або отримувати різні види інформації (рисунок 2.2).

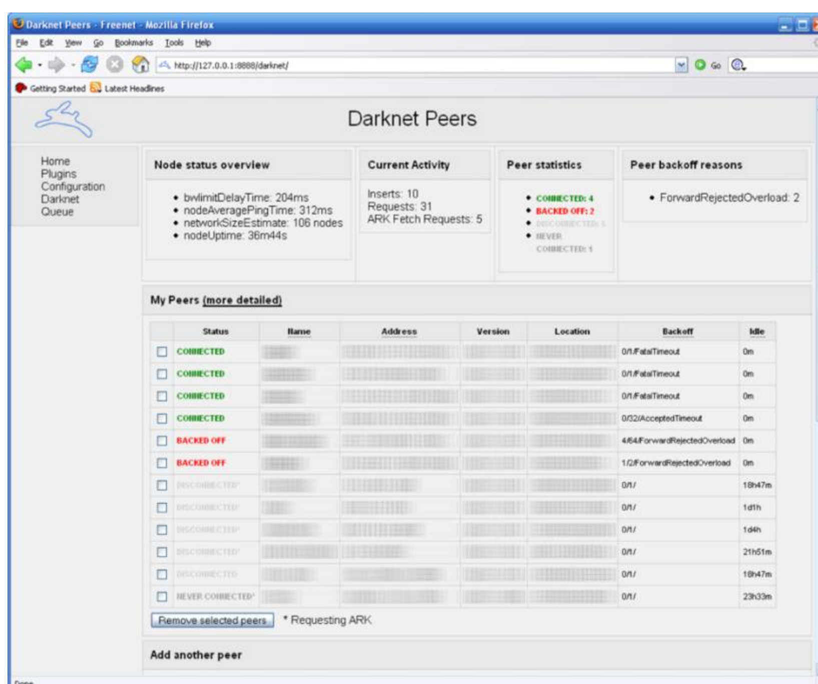


Рисунок 2.2 – Список однорангових партнерів у Freenet 0.7.

Файлообмінна мережа Freenet зберігає документи і дозволяє отримати їх пізніше за допомогою пов'язаного з ними ключа, як це зараз можливо за допомогою таких протоколів, як HTTP. Мережа розроблена таким чином, щоб бути дуже живучою. Система не має центральних серверів і не підконтрольна жодній особі чи організації, включаючи розробників Freenet. Розмір кодової бази становить понад 192 000 рядків коду. Інформація, що зберігається у Freenet, розподілена по мережі і зберігається на декількох різних вузлах. Шифрування даних і ретрансляція запитів ускладнює визначення того, хто вставив контент у Freenet, хто запитував цей контент або де він зберігається. Це захищає анонімність учасників, а також дуже ускладнює цензуру певного контенту. Контент зберігається в зашифрованому вигляді, тому навіть оператору вузла важко визначити, що саме зберігається на цьому вузлі. Це забезпечує правдоподібну можливість заперечення, що в поєднанні з ретрансляцією запитів означає, що закони «безпечної гавані», які захищають провайдерів послуг, можуть також захищати операторів вузлів Freenet. Коли їх запитують про це, розробники Freenet посилаються на дискусію EFF, в якій йдеться про те, що нездатність нічого фільтрувати є безпечним вибором.

2.3.2 Gnutella

Gnutella – це мережевий протокол peer-to-peer мережі. Заснована у 2000 році, вона стала першою децентралізованою peer-to-peer мережею у своєму роді, що призвело до того, що інші, пізніші мережі перейняли цю модель [28].

Перший клієнт, від якого мережа отримала свою назву, був розроблений Джастіном Френкелем і Томом Пеппером з Nullsoft на початку 2000 року, невдовзі після того, як компанія була придбана AOL.

Наступного дня після того як програма стала доступною для завантаження на серверах Nullsoft AOL припинила доступ до програми з

юридичних міркувань і заборонила Nullsoft виконувати будь-яку подальшу роботу над проектом. Це не зупинило Gnutella і через кілька днів протокол було перероблено, і почали з'являтися сумісні клони з відкритим вихідним кодом. Така паралельна розробка різних клієнтів різними групами залишається способом розробки Gnutella і сьогодні.

Одними з перших незалежних піонерів Gnutella були Джин Кан і Спенсер Кімбалл, які запустили перший портал, що мав на меті зібрати спільноту з відкритим вихідним кодом для роботи над Gnutella, а також розробили «GNUbile», одну з перших програм з відкритим вихідним кодом (GNU-GPL) для реалізації протоколу Gnutella.

Мережа Gnutella є повністю розподіленою альтернативою таким напівцентралізованим системам, як FastTrack (KaZaA) та оригінальному Napster. Цей зростаючий сплеск популярності виявив межі масштабованості початкового протоколу. На початку 2001 року варіації протоколу дозволили покращити масштабованість. Замість того, щоб розглядати кожного користувача як клієнта і сервер, деякі користувачі стали розглядатися як ультрапірингові, маршрутизуючи пошукові запити і відповіді для підключених до них користувачів. Це дозволило мережі зростати в популярності.

Слово Gnutella сьогодні відноситься не до якогось одного проекту чи програмного забезпечення, а до відкритого протоколу, що використовується різними клієнтами.

Щоб уявити, як Gnutella працювала спочатку, можна уявити собі велике коло користувачів, які називаються вузлами, кожен з яких має клієнтське програмне забезпечення Gnutella. Під час першого запуску клієнтське програмне забезпечення має завантажитися і знайти принаймні один інший вузол. Для цього використовуються різні методи, включаючи попередній список адрес можливих робочих вузлів, що постачається з програмним забезпеченням, використання оновлених веб-кешів відомих вузлів (так званих Gnutella Web Caches), кешів хостів UDP і, зрідка, навіть

IRC. Після підключення клієнт запитує список робочих адрес. Клієнт намагається з'єднатися з вузлами, з якими він був доставлений, а також з вузлами, які він отримує від інших клієнтів, поки не досягне певної квоти. Він з'єднується лише з цією кількістю вузлів, локально кешуючи адреси, які він ще не спробував, і відкидаючи адреси, які він спробував і визнав недійсними.

Коли користувач хоче здійснити пошук, клієнт надсилає запит до кожного активно підключеного вузла. У версії 0.4 протоколу кількість активно підключених вузлів для клієнта була досить малою (близько 5). У цій версії протоколу кожен вузол пересилає запит всім своїм активно підключеним вузлам, які, в свою чергу, пересилають запит. Це триває до тих пір, поки пакет не досягне заздалегідь визначеної кількості хопів від відправника (максимум 7).

Починаючи з версії 0.6, Gnutella – це складена мережа, що складається з листових вузлів та ультра-вузлів (також званих ультрапірами). Листяні вузли з'єднані з невеликою кількістю ультравузлів (зазвичай 3), в той час як кожен ультравузол з'єднаний з більш ніж 32 іншими ультравузлами. З цим вищим степенем максимальна кількість стрибків, яку може пройти запит, була знижена до 4.

Листя і надвузли використовують протокол маршрутизації запитів для обміну таблицею маршрутизації запитів (Query Routing Table, QRT), таблицею з 64 Ki-слотів і до 2 Mi-слотів, що складається з хешованих ключових слів. Вузол листа надсилає свою QRT кожному з ультрамауерів, до яких він підключений, а ультрамауери об'єднують QRT всіх своїх листів (зменшені до 128 Ki-слотів) плюс свою власну QRT (якщо вони спільно використовують файли) і обмінюються нею зі своїми сусідами. Маршрутизація запитів виконується шляхом хешування слів запиту і перевірки, чи всі вони збігаються у QRT. Надвузли виконують цю перевірку перед пересиланням запиту до листового вузла, а також перед пересиланням

запиту до однорангового надвузла за умови, що це останній крок, який може бути зроблений для запиту.

Якщо пошуковий запит дає результат, то вузол, який його отримав, зв'язується з шукачем. У класичному протоколі Gnutella повідомлення-відповіді надсилалися назад за маршрутом, пройденим запитом, оскільки сам запит не містив ідентифікаційної інформації для вузла. Пізніше ця схема була переглянута, щоб доставляти результати пошуку по UDP, безпосередньо вузлу, який ініціював пошук, як правило, ультраранньому вузлу. Таким чином, у поточному протоколі запити містять IP-адресу і номер порту будь-якого з вузлів. Це зменшує обсяг трафіку, що проходить через мережу Gnutella, роблячи її значно більш масштабованою.

2.2.3 Інші імплементації

Варто відмітити, що існують й інші peer-to-peer імплементації, такі як Skype та Bittorrent [29]. Проте ці імплементації не реалізують тих функцій та вимог, які висунуті до мереж в цій роботі.

Крім того є мережі для peer-to-peer стримингу гіпермедіа файлів і тому подібне. Цього роду імплементації упущені та не розглядаються так як концептуально peer-to-peer відношення може бути реалізовано на будь-якому функціоналі, перечислити всі неможливо. Розроблювана мережа повинна бути стабільним та надійним рішенням, яке повинно виглядати як централізований сервер, але мати децентралізовану архітектуру та поведінку не помітну для користувачів.

3 ПРОЕКТУВАННЯ СИСТЕМИ

3.1 Рішення фундаментальних проблем

Розробляючи архітектурне рішення потрібно вирішити багато фундаментальних проблем, які було описано в аналізі предметної області.

Проблема, яку необхідно вирішити, відома як побудова децентралізованої мережі, де кілька вузлів працюють разом, утворюючи розподілену систему. Щоб досягти цього, потрібно визначити алгоритм, який дозволяє вузлам знаходити і з'єднуватися один з одним, надійно зберігати дані і гарантувати, що дані будуть доступні, навіть якщо деякі вузли вийдуть з ладу. Для досягнення поставленої мети потрібно вирішити декілька послідовних завдань, а саме: виявлення вузлів, зберігання даних, реплікація, узгодженість, відмовостійкість, безпека.

Кожен вузол повинен мати можливість виявляти інші вузли в мережі. Кожен вузол зберігає список інших вузлів, про які він знає, і періодично розсилає запити на виявлення нових вузлів.

Кожен вузол повинен мати можливість зберігати дані безпечно і надійно. Цього можна досягти за допомогою розподіленої хеш-таблиці (DHT), такої Chord DHT [30].

Щоб забезпечити доступність даних, навіть якщо деякі вузли виходять з ладу, кожен фрагмент даних повинен бути реплікований на декілька вузлів. Цього можна досягти за допомогою техніки, яка називається «коефіцієнт реплікації», коли кожен фрагмент даних реплікується на певну кількість вузлів, наприклад, 3 або 5.

Щоб гарантувати, що всі вузли мають узгоджене уявлення про дані, потрібно використовувати протокол консенсусу, наприклад, алгоритм Paxos [31] або алгоритм Raft [32]. Ці алгоритми гарантують, що всі вузли погоджуються щодо стану системи, навіть якщо деякі вузли тимчасово відключені.

Щоб забезпечити відмовостійкість системи, потрібно спроектувати її так, щоб вона м'яко справлялася з відмовами вузлів. Цього можна досягти за допомогою таких методів, як резервування, коли кілька вузлів виконують одну і ту ж задачу, або шардинг, коли дані розподіляються між кількома вузлами.

Для забезпечення безпеки системи необхідно використовувати механізми шифрування, такі як криптографія з відкритим ключем та цифрові підписи. Це гарантує, що тільки вузли мережі можуть отримати доступ до даних, і що дані не будуть підроблені або викрадені [33], [34].

Загалом, побудова децентралізованої мережі, яка може працювати з тисячами вузлів і забезпечувати надійне зберігання та доступ до даних, є складною задачею. Однак, використовуючи комбінацію протоколів peer-to-peer, DHT, алгоритмів консенсусу, методів відмовостійкості та механізмів безпеки, можна створити надійну, масштабовану та безпечну систему.

3.2 Локальне сховище даних

Розподілена хеш-таблиця (DHT) може використовуватися для знаходження даних на агенті. Основна перевага DHT полягає в тому, що вузли можна додавати або видаляти з мінімальними зусиллями, пов'язаними з перерозподілом ключів. Ключі – це унікальні ідентифікатори, які зіставляються з певними значеннями, які, в свою чергу, можуть бути чим завгодно – від адрес документів до довільних даних. Відповідальність за підтримання відповідності між ключами та значеннями розподіляється між вузлами таким чином, що зміна набору учасників спричиняє мінімальну кількість збоїв у роботі. Це дозволяє DHT масштабуватися до надзвичайно великої кількості вузлів і обробляти безперервні прибуття, вибуття та збої вузлів.

На рисунку 3.1 зображена схема DHT.

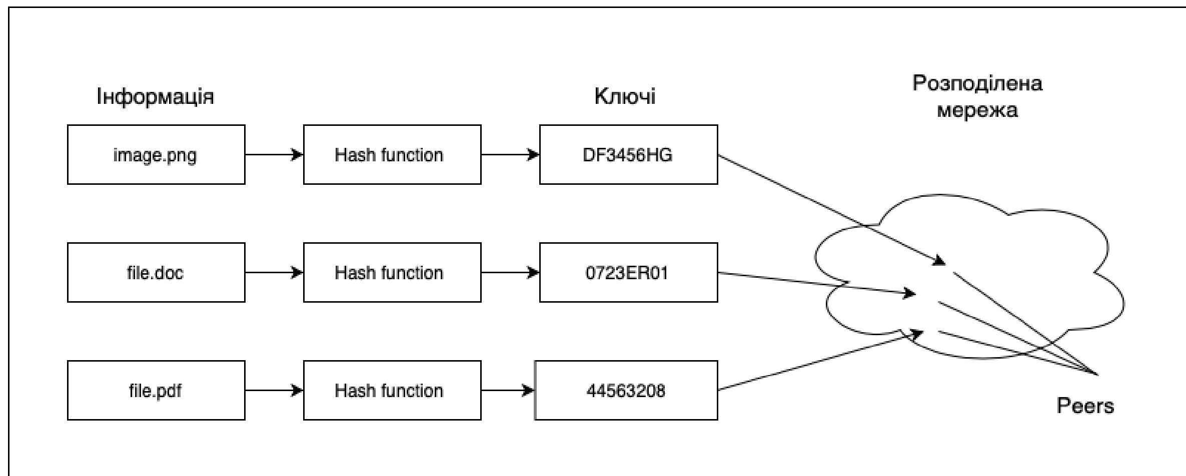


Рисунок 3.1 – Схема DHT

Для DHT характерно підкреслювати такі властивості:

– автономність і децентралізація: вузли колективно формують систему без будь-якої центральної координації;

– відмовостійкість: система надійна навіть за умови, що вузли постійно приєднуються, від'єднуються та виходять з ладу;

– масштабованість: система ефективно функціонує навіть з тисячами або мільйонами вузлів.

Ключовий метод, який використовується для досягнення цих цілей, полягає в тому, що будь-який один вузол повинен координувати роботу лише з кількома іншими вузлами в системі – найчастіше з O(log n) з n учасників – таким чином, що при кожній зміні членства потрібно виконувати лише обмежений обсяг роботи.

Структуру DHT можна розкласти на кілька основних компонентів. Фундаментом є абстрактний ключовий простір, наприклад, набір 160-бітових рядків. Схема розбиття ключового простору розподіляє право власності на цей простір між вузлами-учасниками. Потім мережа накладання з'єднує вузли, дозволяючи їм знайти власника будь-якого ключа в ключовому просторі [35].

Після того, як ці компоненти встановлено, типове використання DHT для зберігання та пошуку може відбуватися наступним чином. Припустимо, що простір ключів – це набір 160-бітових рядків. Щоб проіндексувати файл із заданим іменем і даними у DHT, генерується SHA-1 хеш імені файлу, який створює 160-бітний ключ k, і надсилається повідомлення put(k, дані) до будь-якого вузла, що бере участь у DHT. Повідомлення пересилається від вузла до вузла через оверлейну мережу, поки не досягне єдиного вузла, відповідального за ключ k, як визначено розбиттям простору ключів. Цей вузол зберігає ключ і дані. Будь-який інший клієнт може отримати вміст файлу, знову хешуючи ім'я файлу для отримання k і запитуючи будь-який вузол DHT знайти дані, пов'язані з k, за допомогою повідомлення get(k). Повідомлення знову буде перенаправлено через оверлей до вузла, відповідального за k, який відповість збереженими даними.

Компоненти мережі розбиття простору ключів та оверлейної мережі описані нижче з метою висвітлення основних ідей, спільних для більшості DHT.

Більшість DHT використовують той чи інший варіант консистентного хешування або рандеву-хешуванні для зіставлення ключів з вузлами. Ці два алгоритми, схоже, були розроблені незалежно і одночасно для вирішення проблеми розподілених хеш-таблиць.

І консистентне, і рандеву-хешуванні мають важливу властивість: видалення або додавання одного вузла змінює лише набір ключів, якими володіють вузли з сусідніми ідентифікаторами, і не впливає на всі інші вузли. На відміну від традиційної хеш-таблиці, в якій додавання або видалення одного відра призводить до перерозподілу майже всього ключового простору. Оскільки будь-яка зміна власника зазвичай відповідає інтенсивному переміщенню об'єктів, що зберігаються в DHT, від одного вузла до іншого, мінімізація такої реорганізації необхідна для ефективної підтримки високих темпів плинності (прибуття і вибуття вузлів).

Консистентне хешування використовує функцію $\delta(k_1, k_2)$, яка визначає абстрактне поняття відстані між ключами k_1 і k_2, яка не пов'язана з географічною відстанню або затримкою мережі. Кожному вузлу призначається єдиний ключ, який називається його ідентифікатором (ID). Вузол з ідентифікатором i_x володіє всіма ключами k_m, для яких i_x є найближчим ідентифікатором, виміряним відповідно до $\delta(k_m, i_x)$.

Наприклад, Chord DHT використовує консистентне хешування, яке розглядає вузли як точки на колі, а $\delta(k_1, k_2)$ – це відстань за годинниковою стрілкою по колу від k_1 до k_2. Таким чином, круговий ключовий простір розбивається на суміжні відрізки, кінцеві точки яких є ідентифікаторами вузлів. Якщо i_1 та i_2 – два сусідні ідентифікатори, відстань між якими за годинниковою стрілкою менша від i_1 та i_2, то вузол з ідентифікатором i_2 володіє всіма ключами, що знаходяться між i_1 та i_2.

У рандеву-хешуванні, яке також називають хешуванням з найбільшою випадковою вагою (HRW), всі клієнти використовують одну і ту ж хеш-функцію $h()$, щоб зв'язати ключ з одним з n доступних вузлів. Кожен клієнт має однаковий список ідентифікаторів $\{S_1, S_2, \dots, S_n\}$, по одному для кожного вузлу. За деяким ключем k клієнт обчислює n хеш-ваг $w_1 = h(S_1, k)$, $w_2 = h(S_2, k), \dots, w_n = h(S_n, k)$. Клієнт асоціює цей ключ з вузлом, який має найбільшу хеш-вагу для цього ключа. Вузол з ідентифікатором S_x володіє всіма ключами k_m, для яких хеш-вага k_m більша за хеш-вагу будь-якого іншого вузла для цього ключа.

Локально-зберігаюче хешування гарантує, що подібні ключі призначаються подібним об'єктам. Це може уможливити більш ефективне виконання запитів до діапазону, однак, на відміну від консистентного хешування, немає більшої впевненості в тому, що ключі (а отже, і навантаження) рівномірно розподілені у просторі ключів і серед однорангових об'єктів, що беруть участь у запиті.

Протокол DHT такий як Self-Chord [36] вирішує ці проблеми. Self-Chord відокремлює ключі об'єктів від ідентифікаторів вузлів і сортує ключі

вздовж кільця за допомогою статистичного підходу, заснованого на парадигмі ройового інтелекту. Сортування гарантує, що схожі ключі зберігаються на сусідніх вузлах і що процедури виявлення, включаючи запити діапазону, можуть бути виконані за логарифмічний час.

Кожен вузол підтримує набір зв'язків з іншими вузлами (своїми сусідами або таблицею маршрутизації). Разом ці зв'язки утворюють мережу накладання. Вузол обирає своїх сусідів відповідно до певної структури, яка називається топологією мережі.

Всі топології DHT поділяють певний варіант найбільш важливої властивості: для будь-якого ключа k кожен вузол або має ідентифікатор вузла, якому належить k, або має зв'язок з вузлом, ідентифікатор якого ближчий до k, з точки зору визначеної вище відстані між вузлами ключового простору. Тоді легко направити повідомлення до власника будь-якого ключа k, використовуючи наступний жадібний алгоритм (який не обов'язково є глобально оптимальним): на кожному кроці пересилати повідомлення до сусіда, ідентифікатор якого найближчий до k. Якщо такого сусіда немає, то ми, мабуть, дійшли до найближчого вузла, який і є власником k, як визначено вище. Цей стиль маршрутизації іноді називають маршрутизацією на основі ключів.

3.3 Реплікація

Реплікація є важливою проблемою, яку необхідно вирішувати в розподілених системах, в тому числі в peer-to-peer мережах. Мета реплікації – забезпечити доступність даних, навіть якщо деякі вузли виходять з ладу. Коефіцієнт реплікації та динамічна реплікація є поширеними методами вирішення проблеми.

Найпростішим методом реплікації є використання фіксованого коефіцієнта реплікації, який визначає кількість вузлів, на які реплікується кожен фрагмент даних. Наприклад, коефіцієнт реплікації 3 означає, що

кожен фрагмент даних реплікується на 3 вузли в мережі. Це гарантує, що навіть якщо один або два вузли вийдуть з ладу, дані будуть доступні на інших вузлах [36].

У деяких випадках може бути корисно – динамічно регулювати коефіцієнт реплікації в залежності від стану мережі. Наприклад, якщо в мережі багато вузлів, коефіцієнт реплікації можна зменшити, щоб зменшити накладні витрати на реплікацію. Аналогічно, якщо вузлів мало, коефіцієнт реплікації можна збільшити, щоб забезпечити високу доступність даних.

3.4 Нейронна мережа для оптимізації даних

Арифметичне кодування (AC) є найсучаснішим алгоритмом ентропійного кодування з точки зору швидкості стиснення.

AC кодек складається з двох частин: оцінювача ймовірності, який оцінює ймовірність кожного символу; моделі AC, в якій кодер багаторазово відображає вхідну послідовність символів у підінтервал між одиничним інтервалом [0,1) на основі кумулятивної ймовірності символів; і декодер використовує код (дійсне число між інтервалом [0,1)) для виведення реконструйованої послідовності символів на основі кумулятивної ймовірності символів. Точна оцінка є ключовим моментом для кодека AC.

Кодек арифметичного кодування (AC) на основі дельта-рекурентної нейронної мережі (Delta-RNN) – це метод стиснення даних, який поєднує в собі можливості Delta-RNN для моделювання послідовних даних і AC для ефективного стиснення даних [37].

Основна ідея кодека AC на основі Delta-RNN полягає в тому, що спочатку використовується Delta-RNN для моделювання послідовних даних і генерування розподілу ймовірностей символів у даних. Розподіл ймовірностей можна використовувати як вхідний сигнал для кодера, який кодує дані за допомогою коротшого коду для символів, що зустрічаються частіше, і довшого коду для символів, що зустрічаються рідше.

При декодуванні стиснутих даних декодер використовує розподіл ймовірностей, згенерований Delta-RNN, для декодування стиснутих символів.

Така комбінація Delta-RNN та AC може призвести до значного покращення продуктивності стиснення даних, особливо для послідовних даних, де певні символи або шаблони зустрічаються частіше, ніж інші.

DRAC складається з двох блоків: алгоритму AC та предиктору ймовірностей. Нехай $Y = \{y_0, y_1, \dots, y_{n-1}\}$ – послідовність даних. Предиктор ймовірностей включає модель Delta-RNN, яка навчена на послідовності Y протягом кількох епох. Після завершення навчання, ваги навченої моделі відомі як кодеру, так і декодеру. При попередньому перегляді попередніх J символів, де значення J встановлено на 1 в моделі, ймовірність $P(y_i | y_{i-1}, \dots, y_{i-J})$ оцінюється моделлю Delta RNN, а потім передається в кодер Arithmetic Coder для виконання процесу кодування до виведення остаточного коду. Для перших J символів обирається рівномірний розподіл як початковий. Цей процес ілюструється на рисунку 3.2.

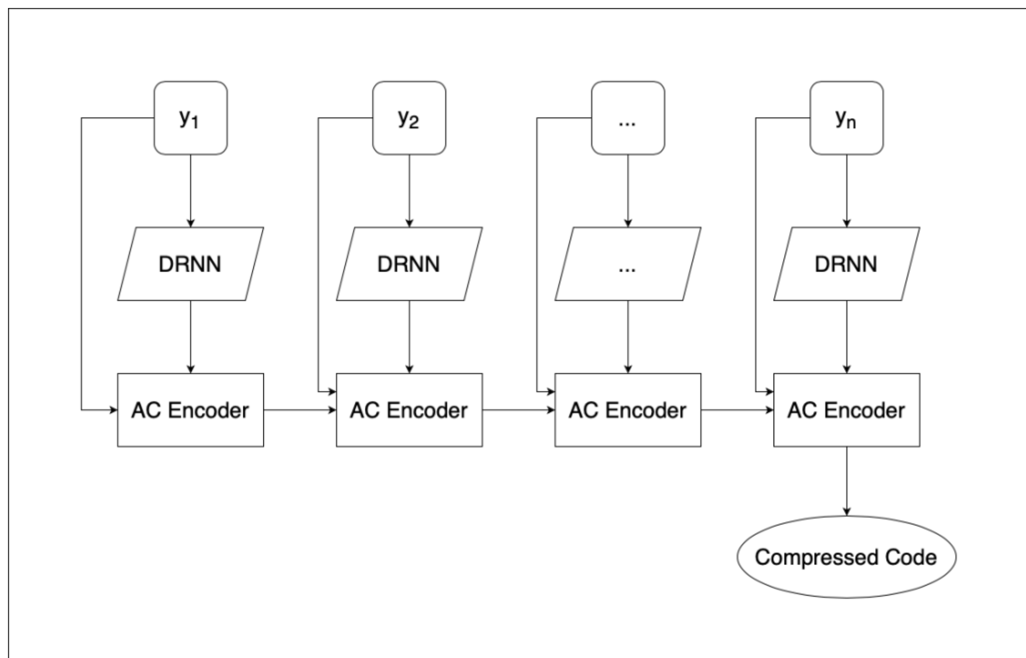


Рисунок 3.2 – Схема кодеру

Процес декодування просто симетричний до процесу кодування. Декодер бере код на вхід і повторно виводить реконструйовані символи, використовуючи ймовірність від предиктору ймовірностей, що проілюстровано на рисунку 3.3.

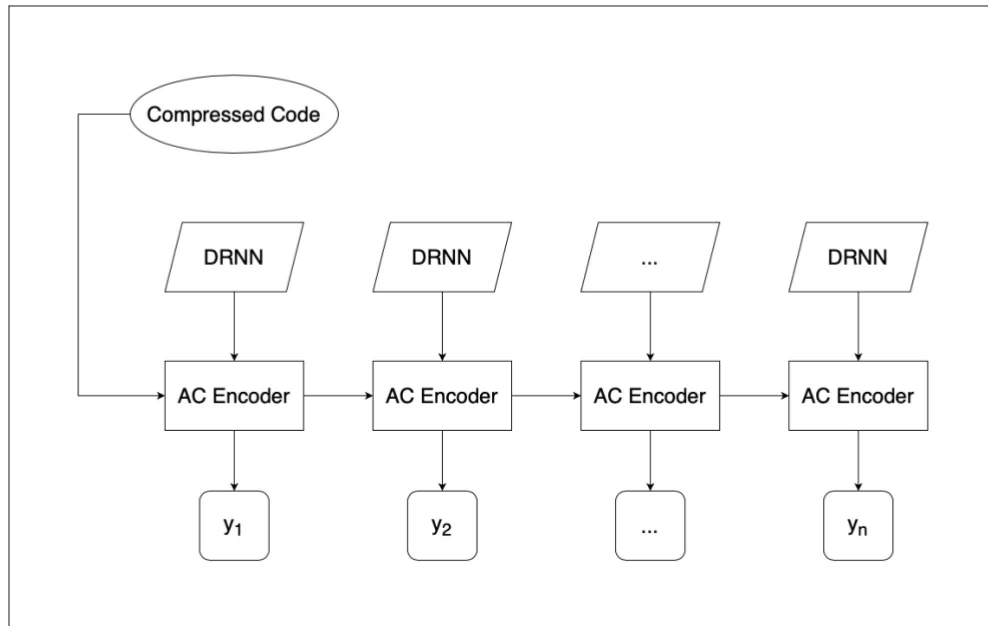


Рисунок 3.3 – Схема декодеру

Процес кодування за допомогою AC описано наступним чином. Нехай $X = \{x_0, x_1, \dots, x_{m-1}\}$ – алфавіт дискретного джерела, а $P = \{p(x_0), p(x_1), \dots, p(x_{m-1})\}$ – ймовірність кожного символу, яка буде передбачена предиктором ймовірностей. Кумулятивна ймовірність $C = \{c(x_0), c(x_1), \dots, c(x_{m-1})\}$ визначається як:

$$c(x_1) = \sum_{k=0}^{i} P(x_k), \qquad (3.1)$$

де $c(x_0) = 0$ та $c(M) = 1$.

Кодер АС рекурсивно відображає кожен символ вхідної послідовності у підінтервал, поділений на поточний інтервал. Нехай $Y = \{y_0, y_1, \dots, y_{n-1}\}$ – вхідна послідовність, а $I_0 = [L_0, H_0)$ – початковий інтервал. Тоді новий $I_j = [L_j, H_j)$ обчислюється за формулою.

$$H_j = L_{j-1} + \left(H_{j-1} - L_{j-1}\right) \times c\left(y_j + 1\right), \tag{3.2}$$

$$L_j = L_{j-1} + \left(H_{j-1} - L_{j-1}\right) \times c\left(y_j\right), \tag{3.3}$$

де $c(y_j + 1)$ – кумулятивна ймовірність підпослідовності символу y_j в алфавіті.

Після завершення процесу кодування будь-яке дійсне число D з останнього інтервалу може бути вибрано як код.

У процесі декодування нехай $Y = \{y_0, y_1, \dots, y_{n-1}\}$ – вихідна послідовність декодера АС. Декодер рекурсивно виконує наступні кроки:

1. Визначити символ a_i з C, виходячи з поточного коду D_j та вихідних даних $y_j = x_i$, тобто:

$$y_j = \left\{x_i : c(x_i) \le D_j \le c(x_i + 1)\right\}, \qquad j \in \{0, 1, \dots, N-1\} \tag{3.4}$$

2. Оновити D_j на:

$$D_{j+1} = \frac{D_j - c(x_i)}{p(x_i)} \tag{3.5}$$

Після успішного завершення процесу декодування можна отримати вихідну послідовність.

Предиктор ймовірностей – це дельта RNN мережа. Її архітектуру показано на рисунку 3.4.

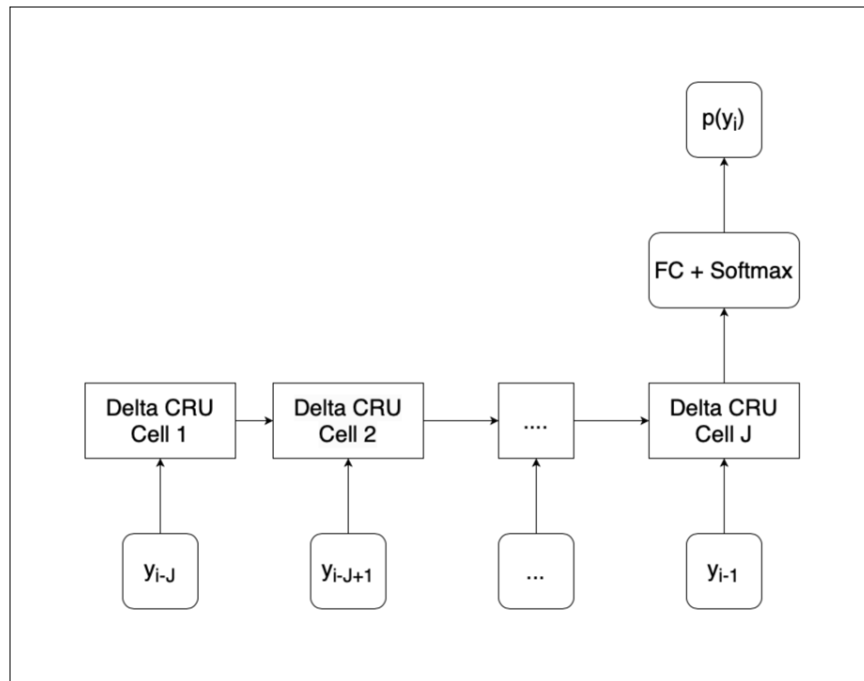


Рисунок 3.4 – Схема предиктору ймовірностей

Для символу y_j попередні J символів y_{i-J}, y_{i-J+1}, y_{i-1} подаються на вхід двонаправленої комірки Delta Gated Recurrent Unit (Delta GRU), яка буде описана нижче. В кінці комірки Delta GRU на приховані стани, отримані з комірки Delta GRU, накладається шар Softmax та шар Fully Connected (FC). Нарешті, отримується ймовірність $p(y_i)$. Шар Softmax визначається як:

$$softmax(z)_i = p_i = \frac{e^{z_i}}{\sum_{j=1}^{|X|} e^{z_i}} \quad (3.6)$$

де X – алфавіт символів;

$|\bullet|$ – кардіональність.

Для вхідних даних X, шар FC визначається наступним чином:

$$F = \sigma(XW^T + B), \quad (3.7)$$

де W – вагова матриця;

B – зсув;

$\sigma(x)$ – функція активації.

Закриті рекурентні одиниці (Gated Recurrent Units, GRU) – це клас закритих RNN, який є дуже популярним у галузі обробки природної мови.

Нехай R – множина дійсних чисел, а $X = \{x_0, x_1, \dots, x_i, x_{N-1}\}$ – вхідна послідовність довжиною N. Для комірки GRU з H нейронів, одновимірним входом x_i та виходом h_i, рівняння оновлення визначається як:

$$r_i = \sigma(W_{xr}x_i + W_{hr}h_{i-1} + b_r), \quad (3.8)$$

$$u_i = \sigma(W_{xu}x_i + W_{hu}h_{i-1} + b_u), \quad (3.9)$$

$$c_i = tanh(W_{xc}x_i + r_i \odot (W_{hc}h_{i-1} + b_c)), \quad (3.10)$$

$$h_i = (1 - u_i) \odot c_i + u_i \odot h_{i-1}, \quad (3.11)$$

де $r, u, c \in R^H$ – вентиль скидання, вентиль оновлення та стан комірки відповідно;

$W_x \in R^H, W_h \in R^{H\times H}$ – вагові матриці;

а $b \in R^H$ – вектор зсуву;

$\sigma(x)$ – функція активації;

а $\odot$ – поелементне множення векторів.

Для зменшення обчислювальної складності GRU було запропоновано дельта-рекурентні мережі.

Рівняння оновлення дельта GRU визначаються як:

$$M_{r,i} = W_{xr}\delta x_i + W_{hr}\delta h_{i-1} + M_{r,i-1}), \quad (3.12)$$

$$M_{xc,i} = W_{xc}\delta x_i + M_{xc,i-1}, \tag{3.13}$$

$$M_{hc,i} = W_{hc}\delta h_i + M_{hc,i-1}, \tag{3.14}$$

$$r_i = \sigma(M_{r,i}), \tag{3.15}$$

$$u_i = \sigma\left(M_{u,i}\right), \tag{3.16}$$

$$c_i = tanh(M_{xc,i} + r_i \odot M_{hc,i}), \tag{3.17}$$

$$h_i = (1 - u_i) \odot c_i + u_i \odot h_{i-1}, \tag{3.18}$$

де $M_{i=0} = b_r$, $M_{u,i=0} = b_u$, $M_{xc,i=0} = b_c$, $M_{hc,i=0} = 0$ – вектори дельта-пам'яті та $M \in R^H$.

На відміну від роботи GRU, Delta GRU обрізає ваги і видаляє неважливі зв'язки між нейронами, що призводить до розріджених вагових матриць. Розріджені матриці можуть бути закодовані у формат розріджених матриць. Множення розріджених матриць-векторів можна прискорити, виконуючи операції множення та накопичення (MAC) лише над ненульовими вагами. Пропускаючи нульові елементи в дельта-векторах, можна пропустити цілі стовпці матрично-векторних операцій MAC. Після належного навчання моделі експерименти показують, що в Delta мережах кількість операцій можна зменшити десятки разів.

3.5 Застосування ШІ для оптимізації маршрутизації

Оптимізація мурашиних колоній (ACO) – це метаевристичний алгоритм, натхненний поведінкою мурах у пошуках їжі. ACO успішно

застосовується для різних оптимізаційних задач, включаючи маршрутизацію в децентралізованих peer-to-peer мережах.

В контексті peer-to-peer мереж ACO можна використовувати для оптимізації маршрутизації повідомлень між вузлами. Основна ідея полягає в тому, щоб розглядати вузли мережі як міста, а зв'язки між вузлами – як шляхи між містами. Кожна мураха в алгоритмі ACO представляє собою повідомлення, яке потрібно направити від джерела до місця призначення. Феромонний слід на шляху відображає якість цього шляху, причому вищі рівні феромонів вказують на кращу якість шляху.

Нижче описано більше детально кроки ACO алгоритму.

Першим кроком є крок ініціалізації, під час якого необхідно ініціалізувати рівень феромону на кожній ланці невеликим позитивним значенням.

Наступним кроком є крок переміщення мурашок, під час якого кожна мураха рухається від поточного вузла до сусіднього, обираючи наступний вузол на основі рівнів феромонів на лініях зв'язку. Ймовірність вибору ланки пропорційна рівню феромону на цій ланці.

Далі йде крок оновлення феромонів. Після того, як всі мурахи перемістилися, рівні феромонів на ланках оновлюються на основі якості шляхів, які пройшли мурахи. Шляхи, якими пройшли мурахи і які успішно досягли місця призначення, винагороджуються вищими рівнями феромонів, тоді як шляхи, якими не пройшли мурахи, караються нижчими рівнями феромонів.

І останнім кроком є крок завершення. алгоритм завершується після фіксованої кількості ітерацій або коли знайдено задовільний розв'язок.

Використовуючи ACO для оптимізації маршрутизації в децентралізованих peer-to-peer мережах, мережа може адаптуватися до змін в топології мережі і оптимізувати маршрутизацію повідомлень між вузлами. Алгоритм може бути налаштований з урахуванням різних факторів, таких як пропускна здатність і затримка каналів зв'язку, кількість переходів і

навантаження на вузли. Використовуючи ACO, мережа може знаходити ефективні шляхи маршрутизації навіть у великих і складних мережах.

3.6 Деталі імплементації

Програмне забезпечення складається з наступних абстракцій: модуль розподілу, модуль членства в кластері, модуль виявлення відмов та модуль сховища. Кожен з цих модулів заснований на подійо-орієнтованій основі, де конвеєр обробки повідомлень та конвеєр завдань розбиваються на кілька етапів. Кожен з цих модулів повинен бути реалізований з нуля за допомогою C++. Так як ця мова програмування має критичну роль в низькорівневому програмуванні.

Модуль членства в кластері та виявлення відмов побудований на мережевому рівні, який використовує неблокуючий ввід/вивід процес. Усі системні контрольні повідомлення засновані на мережевому протоколі UDP, тоді як повідомлення, пов’язані з реплікацією та маршрутизацією запитів, використовують мережевий протокол TCP. Модулі маршрутизації запитів реалізовані за допомогою певної машини станів.

Коли запит на читання/запис приходить до будь-якого вузла в кластері, станова машина переходить через такі стани:

1. Визначення вузлів, що володіють даними для ключа.
2. Маршрутизація запитів до вузлів і очікування на відповіді.
3. Якщо відповіді не надходять протягом налаштованого часу очікування, запит відхиляється і повертається клієнту.
4. Визначення останньої відповіді на основі мітки часу.
5. Планування відновлення даних на будь-якій репліці, якщо вона не має останнього шматка даних.

4 РОЗРОБКА ДОДАТКА

4.1 Розробка інфраструктури додатка

Додаток займає місце в категорії кричних та ті, які вимагають низькорівневого програмування, наприклад використання мови програмування C або C++. Варто відмітити, що менш критичні модулі можуть бути написані будь-якою мовою програмування, в управляючому середовищі, наприклад C# або Python.

Концептуально розробку можна розділити на розробку декількох модулів. Перший з яких модуль локального сховища. Вимоги до модулю прості, користувач коду повинен мати змогу використовувати методи модулю на будь-якій платформі (Mac OS, Windows, Linux) і всі деталі імплементації повинні бути приховані. Модуль надає методи запису по ключу маршрутизації деякого набору байтів, що представляють частину даних які потрібно записати та метод читання даних (лістинг 4.1 та лістинг 4.2).

Лістинг 4.1 – Сигнатура методу читання

```
byte * read(const byte * routerKey, const iooptions & options) const
```

Лістинг 4.2 – Сигнатура методу запису

```
long write (const byte * routerKey, const byte * data, const iooptions& options)
```

Оскільки ми працюємо з файловою системою, тому важливо, щоб після запису в реєстр файлів, методи модулю також мати змогу специфікувати спосіб блокування. Загалом блокування відіграє критичну роль в цій функціональності, оскільки заблокувавши файл даних жоден інший процес не зможе отримати доступ. У більшості операційних систем

існує поняття блокування запису, яке і гарантує блокування в цьому модулю.

Метод читання виглядає схожим на метод запису, різниця в тому, що метод читання повертає масив байтів, що представляють дані, які користувач модулю запитує. Говорячи в контексті файлових систем операційних систем, варто відмітити, що стриминг даних – це складний процес, який реалізується в більш вискорівневому модулі (лістинг 4.3).

Лістинг 4.3 – Методи класу Stream

```
public virtual IAsyncResult BeginRead (byte[] buffer, int offset, int count, AsyncCallback? callback, object? state);
public virtual IAsyncResult BeginWrite (byte[] buffer, int offset, int count, AsyncCallback? callback, object? state);
public virtual int Read (Span<byte> buffer);
public virtual int ReadByte ();
public System.Threading.Tasks.Task<int> ReadAsync (byte[] buffer, int offset, int count);
public virtual void Write (ReadOnlySpan<byte> buffer);
public virtual void WriteByte (byte value);
public virtual System.Threading.Tasks.ValueTask WriteAsync (ReadOnlyMemory<byte> buffer, System.Threading.CancellationToken cancellationToken = default);
```

Іншим, не менш критичним, модулем є модуль комунікації. Розробка модулю вимагає розуміння інтернет протоколів та комунікації. Типовим способом надіслати дані з комп’ютера на комп’ютер без посередників є реалізація віртуальної приватної мережі. У такий спосіб, програмне забезпечення може спілкуватися з різними агентам одночасно або в дуже малі проміжки часу, саме тому реалізація модулю повинна бути не блокуючою та приймати конфігурацію або параметри з’єднання, які кожен

раз специфікують та налаштовують приватну віртуальну мережу для зв’язку.

Для прямої реалізації віртуальної приватної мережі використовуються С++ бібліотеки, проте бібліотека обгортається в обгортку, що дозволяє модулю мати правильно виважені інтерфейси методів. Метод конфігурації приймає параметри віртуальної мережі та повертає об’єкт зв’язку, який може використовувати для запитів до іншого агенту (лістинг 4.4).

Лістинг 4.4 – Сигнатура методу коду конфігурації VPN

```
p2p:vpn_configuration& configure(const byte * ipAddres,
const byte * macAddress) const;
```

Метод відправки маючи IP адресу та MAC адресу іншого агенту відправляє дані через мережу інтернету.

Модуль безпеки відповідає за шифрування трафіку, що передається. Оскільки використовуються алгоритми, що розуміються на даних, то метод шифрування повинен повертати такий набір байтів для абсолютно однакових наборів, що дає змогу мати консистентне шифрування та дешифрування. Очевидно що потрібно використовувати асиметричні алгоритми шифрування.

Отже, вищеописані модулі є коренем всього додатку. Інші модулі реалізовані мовою С#.

Як було вище зазначено, потрібно реалізувати методи та класи стримингу даних. Методи та класи повинні бути конфігуруванні для зміни розміру шматка даних або часу тайм-ауту. . NET Core фреймворк має інтерфейс та вже реалізовані класи стримингу, тому імплементація стримингу покладається на ці класи.

Наслідування та реалізація методів з використанням COM об’єктів дозволяє зв’язати C# та C++ бібліотеки. .NET Core – це крос платформне рішення, для побудови прогресивного програмного забезпечення. COM

Interoperability – це функція Microsoft .NET, яка дозволяє керованому коду .NET взаємодіяти з некерованим кодом за допомогою семантики компонентної об’єктної моделі Microsoft [38].

Лістинг 4.5 – Конфігурація і використання модулю VPN

```
Edge_init_conf_defaults(&conf);
conf.allow_p2p = 1;
// Whether to allow peer-to-peer communication
conf.allow_routing = 1;
// Whether to allow the edge to route packets to other edges
snprintf((char *)conf.community_name, sizeof(conf.community_name), «%s», «mycommunity»);
// Community to connect to
conf.disable_pmtu_discovery = 1;
// Whether to disable the path MTU discovery
conf.drop_multicast = 0;
// Whether to disable multicast
conf.tuntap_ip_mode = TUNTAP_IP_MODE_SN_ASSIGN;
// How to set the IP address
conf.encrypt_key = «mysecret»;
// Secret to decrypt & encrypt with
conf.local_port = 0;
// What port to use (0 = any port)
conf.mgmt_port = N2N_EDGE_MGMT_PORT;
// Edge management port (5644 by default)
conf.register_interval = 1;
// Interval for both UDP NAT hole punching and supernode registration
conf.register_ttl = 1;
// Interval for UDP NAT hole punching through supernode
edge_conf_add_supernode(&conf, «localhost:1234»);
// Supernode to connect to
conf.tos = 16;
// Type of service for sent packets
```

Продовження лістингу 4.5

```
    conf.transop_id        =        N2N_TRANSFORM_ID_TWOFISH;
// Use the twofish encryption
    if(edge_verify_conf(&conf) != 0) {return -1;}

    if(tuntap_open(&tuntap,
    «edge0»,          // Name of the device to create
    «static»,            // IP mode; static|dhcp
    «10.0.0.1»,          // Set ip address
    «255.255.255.0»,     // Netmask to use
    «DE:AD:BE:EF:01:10», // Set mac address
    DEFAULT_MTU         // MTU to use
    #ifdef WIN32, 0
    #endif) < 0) {return -1;}

    eee = edge_init(&conf, &rc);
    if(eee == NULL) {exit(1);}

    keep_running = 1;
    eee->keep_running = &keep_running;
    rc = run_edge_loop(eee);

    edge_term(eee);
    tuntap_close(&tuntap);
```

Розробивши низкорівневі та вискорорівневі модулі та маючи робочий прототип, наступним питанням постає спосіб запуску та дистрибуції додатку. Для цілей цієї задачі, спосіб запуску та дистрибуції обрано – Docker контейнер.

Docker – це інструментарій для управління ізольованими Linux-контейнерами. Docker доповнює інструментарій LXC більш високорівневим API, що дозволяє керувати контейнерами на рівні ізоляції окремих процесів. Зокрема, Docker дозволяє не переймаючись вмістом контейнера запускати

довільні процеси в режимі ізоляції і потім переносити і клонувати сформовані для даних процесів контейнери на інші сервери, беручи на себе всю роботу зі створення, обслуговування і підтримки контейнерів [39]. Windows підтримує WSL, що дає змогу Docker працювати на цій операційній системі як на Linux та Macos. У свою чергу підтримує контейнер простору за замовчуванням так як це Linux подібна система.

Загорнувши додаток в Docker контейнер можна розповсюджувати його безкоштовно через Docker Hub.

Лістинг 4.6 – Імплементація Dockerfile

```
FROM mcr.microsoft.com/dotnet/sdk:6.0 as build
WORKDIR /app
COPY cpp-src/ ./
RUN g++ -o mycppapp main.cpp
COPY csharp-src/ ./
RUN dotnet build peer2peernet.csproj -c Release
application to access C++ library methods
COPY wrapper/ ./wrapper
RUN g++ -c -o wrapper.o -I./wrapper -fPIC wrapper/wrapper.cpp
RUN ar rcs libwrapper.a wrapper.o
FROM mcr.microsoft.com/dotnet/runtime:6.0
WORKDIR /app
COPY –from=build /app/ peer2peernet /bin/Release/net6.0/publish/ ./
COPY –from=build /app/libwrapper.a /usr/lib/
ENV LD_LIBRARY_PATH=/usr/lib/
ENTRYPOINT [«dotnet», « peer2peernet.dll»]
```

Кожен учасник мережі, який бажає приєднатися, може встановити Docker локально та виконати наступну команду (лістинг 4.7).

Лістинг 4.7 – Команда для приєднання до мережі

```
Docker pull [OPTIONS] IMAGE[:TAG|@DIGEST]
```

Де:

– OPTIONS: додаткові параметри, які можна передати команді;

– IMAGE: назва образу, який необхідно доєднати до мережі;

– TAG: конкретна версія образу, яку необхідно доєднати до мережі (якщо не вказано, за замовчуванням буде використано останню версію тегу);

– DIGEST: унікальний ідентифікатор образу, який можна використовувати замість тегу.

4.2 Розробка інтелектуальної частини додатка

Delta Recurrent Arithmetic Coder (DRAC) – це архітектура глибокої нейронної мережі, яка використовується для ефективного стиснення даних, що базується на арифметичному кодуванні. DRAC дозволяє отримати максимально можливе стискання даних, використовуючи відносно невелику кількість нейронів, що дозволяє зберігати обсяг пам'яті і ресурси при обробці великих обсягів даних [37].

DRAC був запропонований у статті «DRAC: a delta recurrent neural network-based arithmetic coding algorithm for edge computing» в 2022 році. В статті автори пропонують використовувати DRAC для периферійні обчислення та порівнюють його з іншими методами стиснення даних. В результаті дослідження було показано, що DRAC може забезпечити ефективне стиснення з високою якістю відтворення.

Як було описано в розділі проектування DRAC використовує архітектуру рекурентної нейронної мережі (RNN), яка дозволяє розуміти контекст даних та враховувати його при кодуванні. До складу DRAC входять два блоки – блок кодування та блок декодування, що забезпечують двосторонню обробку даних.

Блок кодування DRAC працює за допомогою RNN, яка зберігає в собі інформацію про попередні символи, які були закодовані, та використовує цю інформацію для визначення наступного символу для кодування.

Після кодування повідомлення блок кодування передає його до блоку декодування, який також працює за допомогою RNN та має доступ до інформації про попередні закодовані символи. Блок декодування здійснює розкодування закодованого повідомлення та повертає вихідну послідовність символів.

Нижче в лістингу 4.8 представлений спрощений приклад того, як DRAC можна реалізувати за допомогою Python і TensorFlow, популярного фреймворку для глибокого навчання.

Лістинг 4.8 – Реалізація DRAC з використанням Python і TensorFlow

```
import tensorflow as tf

# Define the DRAC model
class DRACModel(tf.keras.Model):
    def __init__(self):
        super(DRACModel, self).__init__()
        pass

    def call(self, inputs):
        pass

# Create an instance of the DRAC model
drac_model = DRACModel()

# Define the arithmetic coding functions
def encode_data(data, model):
    # Encode the data using DRAC
    pass

def decode_data(encoded_data, model):
```

Продовження лістингу 4.8

```
    # Decode the data using DRAC
    pass

# Example usage
# Assuming 'input_data' is the input data to be compressed
encoded_data = encode_data(input_data, drac_model)

# Assuming 'compressed_data' is the compressed data to be decompressed
decoded_data = decode_data(compressed_data, drac_model)
# 'decoded_data' is the decompressed data
```

Навчання моделі DRAC передбачає окремий процес підготовки навчальних даних, цикл навчання, розрахунок втрат і оновлення моделі, що виходить за рамки цього прикладу. Для ефективної реалізації та навчання DRAC або будь-якого іншого алгоритму стиснення без втрат на основі нейронних мереж важливо звернутися до оригінальної наукової статті та дотримуватися встановлених найкращих практик навчання нейронних мереж.

4.3 Розробка мурашиної колонії ACO

Для початку необхідно чітко сформулювати проблему, яку є потреба вирішити за допомогою ACO у розподіленій peer-to-peer мережі. Це може бути оптимізація обміну файлами між peer-to-peer користувачами з точки зору мінімізації часу завантаження, максимізації доступності файлів або мінімізації мережевого трафіку.

Далі необхідно створити модель розподіленої peer-to-peer мережі, включно з одноранговими пристроями, файлами та їхніми взаємозв'язками. Це може включати представлення мережі у вигляді графа або будь-якої

іншої відповідної структури даних, де однорангові комп'ютери є вузлами, а файли – ребрами з відповідними вагами або властивостями.

Наступним кроком є ініціалізація параметрів ACO. Необхідно визначити та ініциалізувати параметри алгоритму ACO, такі як кількість мурах, кількість ітерацій, значення альфа і бета (ваги феромонів і евристик), швидкість випаровування і початкові рівні феромонів.

Виконавши всі попередні кроки можна переходити до реалізації алгоритму ACO. Необхідно реалізувати алгоритм ACO, використовуючи ініциалізовані параметри. Зазвичай це включає декілька кроків. Перший – рух мурашок, де кожна мураха вибирає файли для спільного доступу на основі феромонів та евристичної інформації. Це можна зробити за допомогою ймовірнісних правил, таких як ймовірність вибору файлу на основі рівня феромонів та евристичної інформації (наприклад, бажаності файлу). Другий крок – це оновлення феромону. Після того, як мурахи зробили свій вибір, оновіть рівні феромонів на основі файлів, якими поділилися мурахи. Це може включати депонування феромону на файлах, до яких було надано спільний доступ, причому кількість феромону має бути пропорційною якості рішення, знайденого мурахою. Третій крок – локальний пошук. За бажанням, можна включити компонент локального пошуку для подальшої оптимізації рішення, знайденого мурахами. Це може включати дослідження сусідніх рішень або застосування інших евристик для покращення якості рішення. І останній крок в реалізації алгоритму ACO – визначення критеріїв завершення роботи алгоритму, наприклад, максимальну кількість ітерацій або поріг збіжності. Як тільки критерії завершення будуть досягнуті, зупиніть алгоритм і поверніть остаточний розв'язок.

Після виконання всіх кроків на етапі розробки алгоритму, необхідно провести оцінити продуктивність алгоритму ACO у розподіленій peer-to-peer мережі на основі визначеної проблеми та критеріїв. Допрацювати алгоритм, скоригувавши параметри, включаючи альфа-, бета-, швидкість

випаровування або інші компоненти алгоритму, на основі отриманих результатів.

Після вдосконалення алгоритму необхідно реалізувати поведінку однорангових пристроїв у мережі, включаючи обмін файлами з іншими одноранговими пристроями, отримання файлів від інших однорангових пристроїв та оновлення локальних рівнів феромонів на основі файлів, до яких надається спільний доступ. Ця поведінка повинна відповідати логікі алгоритму ACO.

Також важливим кроком є тестування та оптимізація. Важливо протестувати реалізований алгоритм ACO у мережі та оптимізувати його дані, налаштувавши параметри або змінивши поведінку однорангових користувачів для досягнення бажаної продуктивності та результатів.

І фінальним етапом використання алгоритму ACO є його перевірка та розгортання. Важливо перевірити алгоритм ACO у мережі, порівнявши його з іншими методами або використовуючи реальні дані. Після перевірки необхідно розгорнути алгоритм ACO у розподіленій peer-to-peer мережі і відстежувати його роботу в режимі реального часу.

Приведено високорівневий інтерфейс, який кожен агент має для маніпуляції та імплементації алгоритму (лістинг 4.9).

Лістинг 4.9 – Інтерфейс для маніпуляції та імплементації алгоритму

```
# Method signatures for ACO in P2P distributed networks

# Peer class representing a peer in the P2P network
class Peer:
    def __init__(self, peer_id, files, neighbors):
        # Constructor for Peer class
        pass

    def share_files(self, num_files):
        # Method to share files among peers
        Pass
```

Продовження лістингу 4.9

```
    def receive_files(self, files_to_receive):
        # Method to receive files from other peers
        pass

# ACOFileSharing class for implementing ACO algorithm
class ACOFileSharing:
    def __init__(self, peers, num_ants, num_iterations, 
alpha, beta, evaporation_rate, initial_pheromone):
        # Constructor for ACOFileSharing class
        pass

    def run(self):
# Method to run the ACO algorithm for file sharing optimization
        pass

    def select_files_to_share(self, peer):
# Method to select files to share based on pheromone and 
heuristic information
        pass

   def update_pheromone(self):
# Method to update pheromone based on files shared by peers
        pass

# Helper functions
def randomly_select_files(num_files, files):
# Helper function to randomly select files from the list of files
    pass

def calculate_desirability(files):
# Helper function to calculate desirability of sharing each file
    Pass
```

ВИСНОВКИ

У результаті виконання роботи було розроблено архітектуру додатка та сам додаток, який реалізує роботу одно учасника в peer-to-peer мережі та призначений для неперервної роботи під великим навантаженням.

У результаті виконання роботи було виявлено, що алгоритми машинного навчання, зокрема алгоритм оптимізації маршрутизації та арифметичний кодер з застосуванням нейронних мереж, добре підходять для розподілених мереж і можуть суттєво вплинути на оптимізацію peer-to-peer мереж. Зокрема, ACO-алгоритм дозволяє виявляти найкоротші шляхи між вузлами в мережі, що покращує ефективність маршрутизації та зменшує затримки в передачі даних. Крім того, ACO підвищує масштабованість мережі, забезпечуючи здатність до адаптації до змінних умов мережі та оптимальну маршрутизацію у більш розширених мережах. Використання DRAC у peer-to-peer мережах допомогло зменшити обсяг передаваних даних, що дозволило ефективніше використовувати пропускну здатність мережі та підвищило швидкість передачі даних. Однак емпірично було встановлено, що в середовищі з великою кількістю відмов учасників, алгоритм оптимізації маршрутизації потребує значно більше часу та ресурсів для покращення маршрутизації мережі.

Під час розробки програмного забезпечення, було вивчено алгоритм побудови нейронної мережі. Розроблений алгоритм було інтегровано в додаток у вигляді модулю, за інтерцептора.

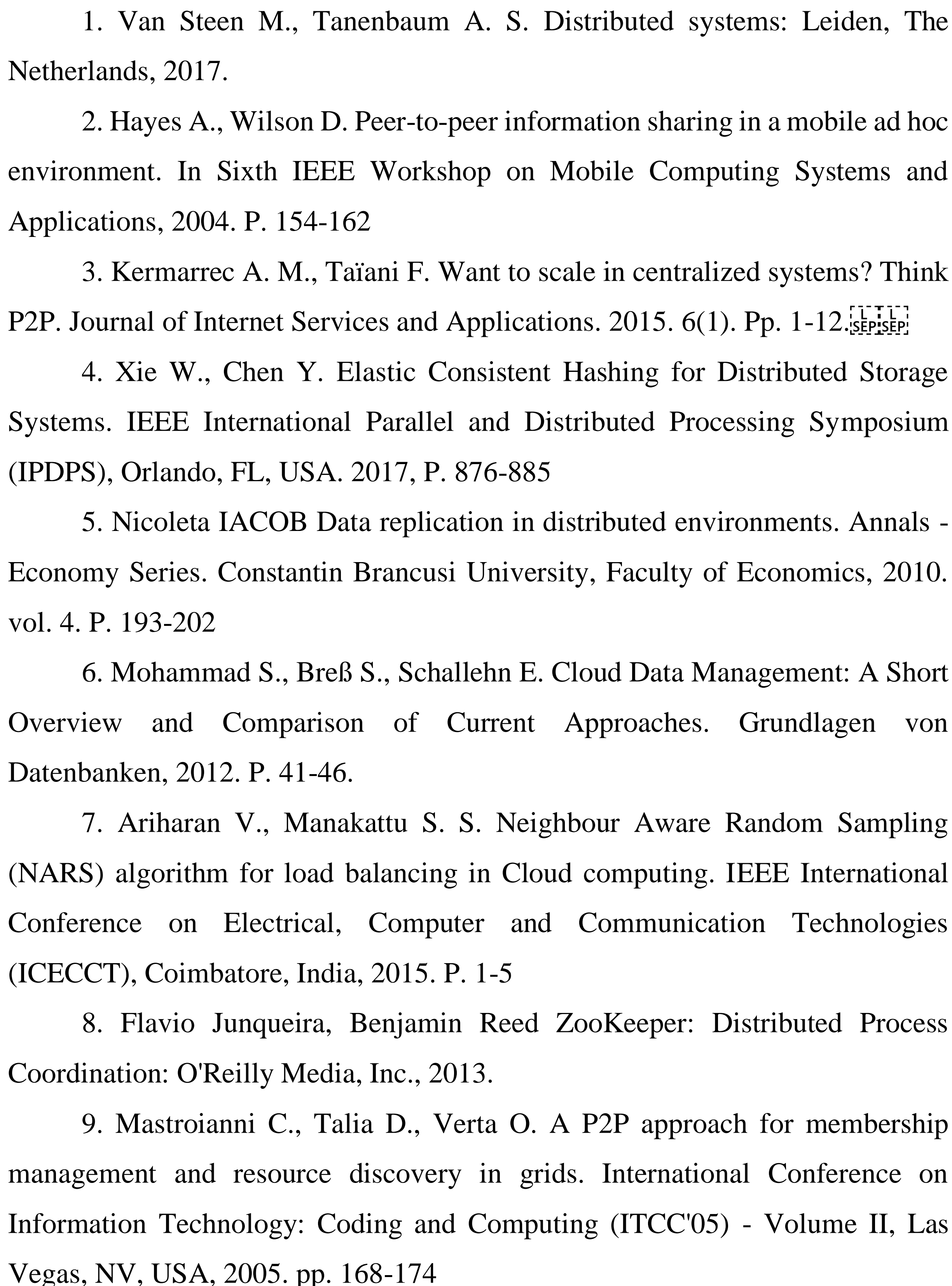
Встановлена симуляція роботи мережі на 2000 агентах, кожен з яких виконував випадкові дії запису та зчитування. Як результат, після декількох годин неперервної роботи мережі алгоритм маршрутизації вивів декілька маршрутів між підкластерами мережі, які здавались надійними та стійкими, а тому більшість трафіку проходило між цими підкластерами.

Для розробки додатку було використано декілька технологій та фреймворків, а саме .NET Core та C++ для платформи Mac OS.

Розроблений додаток, дозволяє виконувати задачі, які було поставлено, та головну функціональність, інтелектуальна складова оптимізації маршрутизації та компресії даних.

Оскільки, розробленим програмний засіб відповідає вимогам, що були поставлені для функціональності додатка, то можна зробити висновок, що атестаційна робота виконана в повному обсязі.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ



1. Van Steen M., Tanenbaum A. S. Distributed systems: Leiden, The Netherlands, 2017.

2. Hayes A., Wilson D. Peer-to-peer information sharing in a mobile ad hoc environment. In Sixth IEEE Workshop on Mobile Computing Systems and Applications, 2004. P. 154-162

3. Kermarrec A. M., Taïani F. Want to scale in centralized systems? Think P2P. Journal of Internet Services and Applications. 2015. 6(1). Pp. 1-12.

4. Xie W., Chen Y. Elastic Consistent Hashing for Distributed Storage Systems. IEEE International Parallel and Distributed Processing Symposium (IPDPS), Orlando, FL, USA. 2017, P. 876-885

5. Nicoleta IACOB Data replication in distributed environments. Annals - Economy Series. Constantin Brancusi University, Faculty of Economics, 2010. vol. 4. P. 193-202

6. Mohammad S., Breß S., Schallehn E. Cloud Data Management: A Short Overview and Comparison of Current Approaches. Grundlagen von Datenbanken, 2012. P. 41-46.

7. Ariharan V., Manakattu S. S. Neighbour Aware Random Sampling (NARS) algorithm for load balancing in Cloud computing. IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT), Coimbatore, India, 2015. P. 1-5

8. Flavio Junqueira, Benjamin Reed ZooKeeper: Distributed Process Coordination: O'Reilly Media, Inc., 2013.

9. Mastroianni C., Talia D., Verta O. A P2P approach for membership management and resource discovery in grids. International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II, Las Vegas, NV, USA, 2005. pp. 168-174

10. Hayashibara N., Defago X., Yared R., Katayama T. The/spl phi/accrual failure detector. In Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004. P. 66-78.

11. Satzger B., Pietzowski A., Trumler W., Ungerer T. A lazy monitoring approach for heartbeat-style failure detectors. Third International Conference on Availability, Reliability and Security, 2008. P. 404-409

12. Chang F., Dean J., Ghemawat S., Hsieh W. C., Wallach D. A., Burrows M., Gruber R. E. Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 2008. 26(2). P. 1-26.

13. Li Q., Wen Z., Wu Z., Hu S., Wang N., Li Y., He B. A survey on federated learning systems: vision, hype and reality for data privacy and protection. IEEE Transactions on Knowledge and Data Engineering, 2021.

14. Mills G. A., Pomary P., Togo E., Sowah R. A. Detection and Management of P2P Traffic in Networks using Artificial Neural Networks. Journal of Network and Systems Management, 2022. 30(2), P. 26.

15. Boutaba R., Salahuddin M. A., Limam N., Ayoubi S., Shahriar N., Estrada-Solano F., Caicedo O. M. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. Journal of Internet Services and Applications, 2018. 9(1). P. 1-99.

16. Hassanzadeh-Nazarabadi Y., Küpçü A., Özkasap Ö. Decentralized and locality aware replication method for DHT-based P2P storage systems. Future Generation Computer Systems, 2018. Vol. 84. P. 32-46.

17. Shen H., Liu G., & Chandler H. Swarm intelligence based file replication and consistency maintenance in structured P2P file sharing systems. IEEE Transactions on Computers, 2015. Vol. 64(10). P. 2953-2967.

18. Chakraborty A., Kar A. K. Swarm intelligence: A review of algorithms. Nature-inspired computing and optimization: Theory and applications, 2017. P. 475-494.

19. Blum C., Merkle, D. Swarm intelligence: introduction and applications. Springer Science & Business Media, 2008.

20. Dorigo M., Birattari M., Stutzle T. Ant colony optimization. IEEE computational intelligence magazine, 2006. Vol. 1(4). P. 28-39.

21. Dorigo M., Stützle, T. Ant colony optimization: overview and recent advances. Springer International Publishing, 2019. P. 311-351

22. Davidović T. Bee colony optimization Part I: The algorithm overview. Yugoslav Journal of Operations Research, 2016. Vol. 25(1).

23. Wang D., Tan D., Liu L. Particle swarm optimization algorithm: an overview. Soft computing, 2018. Vol. 22. P. 387-408.

24. Howard P. G., Vitter J. S. Arithmetic coding for data compression. Proceedings of the IEEE, 2001. Vol. 82(6). P. 857-865.

25. Su R., Cheng Z., Sun H., Katto, J. Scalable learned image compression with a recurrent neural networks-based hyperprior. In 2020 IEEE International Conference on Image Processing (ICIP), 2020. P. 3369-3373.

26. Ming Y., Cao S., Zhang R., Li Z., Chen Y., Song Y., Qu H. Understanding hidden memories of recurrent neural networks. In 2017 IEEE conference on visual analytics science and technology (VAST), 2017. P. 13-24.

27. Clarke I., Sandberg O., Wiley B., Hong T. W. Freenet: A distributed anonymous information storage and retrieval system. In Designing privacy enhancing technologies: international workshop on design issues in anonymity and unobservability Berkeley, CA, USA, July 25–26, 2001. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. P. 46-66.

28. Kan G. Gnutella. Peer-to-Peer: Ökonomische, technologische und juristische Perspektiven, 2002. P. 189-199.

29. Pouwelse J., Garbacki P., Epema D., & Sips, H. The bittorrent p2p file-sharing system: Measurements and analysis. In Peer-to-Peer Systems IV: 4th International Workshop, IPTPS 2005, Ithaca, NY, USA, February 24-25, 2005. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. P. 205-216.

30. Sarmady S. A peer-to-peer Dictionary Using Chord DHT. University of Sains Malaysia, Technical Report, 2007.

31. García-Pérez Á., Gotsman A., Meshman Y., & Sergey I. Paxos consensus, deconstructed and abstracted. In Programming Languages and Systems: 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018. Springer International Publishing, 2018. P. 912-939.

32. Hu J., Liu K. Raft consensus mechanism and the applications. In Journal of physics: conference series, 2020. Vol. 1544. No. 1. P. 012079

33. Liestyowati D. Public key cryptography. In Journal of Physics: Conference Series, 2020. Vol. 1477. No. 5. P. 052062

34. Washbourne L. A survey of P2P network security. IEEE, 2015.

35. Felber P., Kropf P., Schiller E., & Serbu S. Survey on load balancing in peer-to-peer distributed hash tables. IEEE Communications Surveys & Tutorials, 2013. Vol 16(1). P. 473-492.

36. Forestiero A., Leonardi E., Mastroianni C., Meo M. Self-chord: A bio-inspired P2P framework for self-organizing distributed systems. IEEE/ACM Transactions on Networking, 2010. Vol. 18(5). P. 1651-1664.

37. Shan B., Fang Y. DRAC: a delta recurrent neural network-based arithmetic coding algorithm for edge computing. Complex Intell. Syst, 2022.Vol 8. P. 3675–3681.

38. Freeman A .NET Core for professionals. Redmond: Oreilly, 2019.

39. Docker Tutorial. URL: https://www.docker.com (дата звернення: 21.04.2023).

ДОДАТОК А

Імплементація ACO алгоритму

```
from threading import Thread
class ant_colony:
 class ant(Thread):
  def __init__(self, init_location, possible_locations, pheromone_map, distance_callback, alpha, beta, first_pass=False):
   """
   initialized an ant, to traverse the map
   init_location -> marks where in the map that the ant starts
   possible_locations -> a list of possible nodes the ant can go to
    when used internally, gives a list of possible locations the ant can traverse to _minus those nodes already visited_
   pheromone_map -> map of pheromone values for each traversal between each node
   distance_callback -> is a function to calculate the distance between two nodes
   alpha -> a parameter from the ACO algorithm to control the influence of the amount of pheromone when making a choice in _pick_path()
   beta -> a parameters from ACO that controls the influence of the distance to the next node in _pick_path()
   first_pass -> if this is a first pass on a map, then do some steps differently, noted in methods below

   route -> a list that is updated with the labels of the nodes that the ant has traversed
   pheromone_trail -> a list of pheromone amounts deposited along the ants trail, maps to each traversal in route
   distance_traveled -> total distance tranveled along the steps in route
```

```
        location -> marks where the ant currently is
        tour_complete -> flag to indicate the ant has completed
its traversal
         used by get_route() and get_distance_traveled()
        """
        Thread.__init__(self)

        self.init_location = init_location
        self.possible_locations = possible_locations
        self.route = []
        self.distance_traveled = 0.0
        self.location = init_location
        self.pheromone_map = pheromone_map
        self.distance_callback = distance_callback
        self.alpha = alpha
        self.beta = beta
        self.first_pass = first_pass

        #append start location to route, before doing random
walk
        self._update_route(init_location)

        self.tour_complete = False

       def run(self):
        """
        until self.possible_locations is empty (the ant has
visited all nodes)
         _pick_path() to find a next node to traverse to
         _traverse() to:
          _update_route() (to show latest traversal)
          _update_distance_traveled() (after traversal)
        return the ants route and its distance, for use in
ant_colony:
         do pheromone updates
```

```
         check for new possible optimal solution with this ants
latest tour
        """
        while self.possible_locations:
         next = self._pick_path()
         self._traverse(self.location, next)

        self.tour_complete = True

       def _pick_path(self):
        """
        source:
https://en.wikipedia.org/wiki/Ant_colony_optimization_algorith
ms#Edge_selection
        implements the path selection algorithm of ACO
        calculate the attractiveness of each possible transition
from the current location
        then randomly choose a next path, based on its
attractiveness
        """
        #on the first pass (no pheromones), then we can just
choice() to find the next one
        if self.first_pass:
         import random
         return random.choice(self.possible_locations)

        attractiveness = dict()
        sum_total = 0.0
        #for each possible location, find its attractiveness
(it's (pheromone amount)*1/distance [tau*eta, from the
algortihm])
        #sum all attrativeness amounts for calculating
probability of each route in the next step
        for possible_next_location in self.possible_locations:
```

```
          #NOTE: do all calculations as float, otherwise we get
integer division at times for really hard to track down bugs
          pheromone_amount                                    =
float(self.pheromone_map[self.location][possible_next_location
])
          distance = float(self.distance_callback(self.location,
possible_next_location))

          #tau^alpha * eta^beta
          attractiveness[possible_next_location]              =
pow(pheromone_amount, self.alpha)*pow(1/distance, self.beta)
          sum_total += attractiveness[possible_next_location]

         #it is possible to have small values for pheromone amount
/ distance, such that with rounding errors this is equal to zero
         #rare, but handle when it happens
         if sum_total == 0.0:
      #increment all zero's, such that they are the smallest non-
zero values supported by the system
          #source: http://stackoverflow.com/a/10426033/5343977
          def next_up(x):
           import math
           import struct
           # NaNs and positive infinity map to themselves.
           if math.isnan(x) or (math.isinf(x) and x > 0):
            return x

           # 0.0 and -0.0 both map to the smallest +ve float.
           if x == 0.0:
            x = 0.0
      n = struct.unpack('<q', struct.pack('<d', x))[0]

           if n >= 0:
            n += 1
           else:
```

```
            n -= 1
           return struct.unpack('<d', struct.pack('<q', n))[0]

          for key in attractiveness:
           attractiveness[key] = next_up(attractiveness[key])
          sum_total = next_up(sum_total)

         #cumulative    probability    behavior,    inspired    by:
http://stackoverflow.com/a/3679747/5343977
         #randomly choose the next path
         import random
         toss = random.random()

         cummulative = 0
         for possible_next_location in attractiveness:
          weight   =   (attractiveness[possible_next_location]   /
sum_total)
          if toss <= weight + cummulative:
           return possible_next_location
          cummulative += weight

        def _traverse(self, start, end):
         """
         _update_route() to show new traversal
         _update_distance_traveled()   to   record   new   distance
traveled
         self.location update to new location
         called from run()
         """
         self._update_route(end)
         self._update_distance_traveled(start, end)
         self.location = end

        def _update_route(self, new):
         """
```

```
        add new node to self.route
        remove new node form self.possible_location
        called from _traverse() & __init__()
        """
        self.route.append(new)
        self.possible_locations.remove(new)

       def _update_distance_traveled(self, start, end):
        """
        use        self.distance_callback        to        update
self.distance_traveled
        """
        self.distance_traveled                              +=
float(self.distance_callback(start, end))

       def get_route(self):
        if self.tour_complete:
         return self.route
        return None

       def get_distance_traveled(self):
        if self.tour_complete:
         return self.distance_traveled
        return None

      def __init__(self, nodes, distance_callback, start=None,
ant_count=50,              alpha=.5,              beta=1.2,
pheromone_evaporation_coefficient=.40,
pheromone_constant=1000.0, iterations=80):
       """
       initializes an ant colony (houses a number of worker ants
that will traverse a map to find an optimal route as per ACO
[Ant Colony Optimization])
```

```
        source:
https://en.wikipedia.org/wiki/Ant_colony_optimization_algorith
ms

        nodes -> is assumed to be a dict() mapping node ids to
values
         that are understandable by distance_callback

        distance_callback -> is assumed to take a pair of
coordinates and return the distance between them
         populated into distance_matrix on each call to
get_distance()

        start -> if set, then is assumed to be the node where all
ants start their traversal
         if unset, then assumed to be the first key of nodes when
sorted()

        distance_matrix -> holds values of distances calculated
between nodes
         populated on demand by _get_distance()

        pheromone_map -> holds final values of pheromones
         used by ants to determine traversals
         pheromone dissipation happens to these values first,
before adding pheromone values from the ants during their
traversal
         (in ant_updated_pheromone_map)

        ant_updated_pheromone_map -> a matrix to hold the
pheromone values that the ants lay down
         not used to dissipate, values from here are added to
pheromone_map after dissipation step
         (reset for each traversal)
```

```
        alpha -> a parameter from the ACO algorithm to control the influence of the amount of pheromone when an ant makes a choice

        beta -> a parameters from ACO that controls the influence of the distance to the next node in ant choice making

        pheromone_constant -> a parameter used in depositing pheromones on the map (Q in ACO algorithm)
         used by _update_pheromone_map()

        pheromone_evaporation_coefficient -> a parameter used in removing pheromone values from the pheromone_map (rho in ACO algorithm)
         used by _update_pheromone_map()

        ants -> holds worker ants
         they traverse the map as per ACO
         notable properties:
          total distance traveled
          route

        first_pass -> flags a first pass for the ants, which triggers unique behavior

        iterations -> how many iterations to let the ants traverse the map

        shortest_distance -> the shortest distance seen from an ant traversal
      shortets_path_seen -> the shortest path seen from a traversal (shortest_distance is the distance along this path)
        """
        #nodes
        if type(nodes) is not dict:
```

```
         raise TypeError("nodes must be dict")

        if len(nodes) < 1:
         raise ValueError("there must be at least one node in
dict nodes")

        #create internal mapping and mapping for return to caller
        self.id_to_key, self.nodes = self._init_nodes(nodes)
        #create matrix to hold distance calculations between
nodes
        self.distance_matrix = self._init_matrix(len(nodes))
        #create matrix for master pheromone map, that records
pheromone amounts along routes
        self.pheromone_map = self._init_matrix(len(nodes))
        #create a matrix for ants to add their pheromones to,
before adding those to pheromone_map during the
update_pheromone_map step
        self.ant_updated_pheromone_map =
self._init_matrix(len(nodes))

        #distance_callback
        if not callable(distance_callback):
         raise TypeError("distance_callback is not callable,
should be method")

        self.distance_callback = distance_callback

        #start
        if start is None:
         self.start = 0
        else:
         self.start = None
         #init start to internal id of node id passed
         for key, value in self.id_to_key.items():
          if value == start:
```

```
            self.start = key

        #if we didn't find a key in the nodes passed in, then
raise
        if self.start is None:
            raise KeyError("Key: " + str(start) + " not found in
the nodes dict passed.")

        #ant_count
        if type(ant_count) is not int:
            raise TypeError("ant_count must be int")

        if ant_count < 1:
            raise ValueError("ant_count must be >= 1")

        self.ant_count = ant_count

        #alpha
        if (type(alpha) is not int) and type(alpha) is not float:
            raise TypeError("alpha must be int or float")

        if alpha < 0:
            raise ValueError("alpha must be >= 0")

        self.alpha = float(alpha)

        #beta
        if (type(beta) is not int) and type(beta) is not float:
            raise TypeError("beta must be int or float")

        if beta < 1:
            raise ValueError("beta must be >= 1")

        self.beta = float(beta)
```

```
        #pheromone_evaporation_coefficient
        if (type(pheromone_evaporation_coefficient) is not int)
and type(pheromone_evaporation_coefficient) is not float:
         raise TypeError("pheromone_evaporation_coefficient must
be int or float")

        self.pheromone_evaporation_coefficient =
float(pheromone_evaporation_coefficient)

        #pheromone_constant
        if (type(pheromone_constant) is not int) and
type(pheromone_constant) is not float:
         raise TypeError("pheromone_constant must be int or
float")

        self.pheromone_constant = float(pheromone_constant)

        #iterations
        if (type(iterations) is not int):
         raise TypeError("iterations must be int")

        if iterations < 0:
         raise ValueError("iterations must be >= 0")

        self.iterations = iterations

        #other internal variable init
        self.first_pass = True
        self.ants = self._init_ants(self.start)
        self.shortest_distance = None
        self.shortest_path_seen = None

       def _get_distance(self, start, end):
        """
```

```
        uses the distance_callback to return the distance between
nodes
        if a distance has not been calculated before, then it is
populated in distance_matrix and returned
        if a distance has been called before, then its value is
returned from distance_matrix
        """
        if not self.distance_matrix[start][end]:
         distance = self.distance_callback(self.nodes[start],
self.nodes[end])

         if (type(distance) is not int) and (type(distance) is
not float):
          raise TypeError("distance_callback should return either
int or float, saw: "+ str(type(distance)))

         self.distance_matrix[start][end] = float(distance)
         return distance
        return self.distance_matrix[start][end]

       def _init_nodes(self, nodes):
        """
        create a mapping of internal id numbers (0 .. n) to the
keys in the nodes passed
        create a mapping of the id's to the values of nodes
        we use id_to_key to return the route in the node names
the caller expects in mainloop()
        """
        id_to_key = dict()
        id_to_values = dict()

        id = 0
        for key in sorted(nodes.keys()):
         id_to_key[id] = key
      id_to_values[id] = nodes[key]
```

```
         id += 1

        return id_to_key, id_to_values

       def _init_matrix(self, size, value=0.0):
        """
        setup a matrix NxN (where n = size)
        used in both self.distance_matrix and self.pheromone_map
        as they require identical matrixes besides which value to
initialize to
        """
        ret = []
        for row in range(size):
         ret.append([float(value) for x in range(size)])
        return ret

       def _init_ants(self, start):
        """
        on first pass:
         create a number of ant objects
        on subsequent passes, just call __init__ on each to reset
them
        by default, all ants start at the first node, 0
        as per problem description:
https://www.codeeval.com/open_challenges/90/
        """
        #allocate new ants on the first pass
        if self.first_pass:
         return [self.ant(start, self.nodes.keys(),
self.pheromone_map, self._get_distance,
          self.alpha, self.beta, first_pass=True) for x in
range(self.ant_count)]
        #else, just reset them to use on another pass
        for ant in self.ants:
```

```
        ant.__init__(start,                  self.nodes.keys(),
self.pheromone_map, self._get_distance, self.alpha, self.beta)

      def _update_pheromone_map(self):
       """
       1) Update self.pheromone_map by decaying values contained
therein via the ACO algorithm
       2)    Add    pheromone_values    from    all    ants    from
ant_updated_pheromone_map
       called by:
        mainloop()
         (after all ants have traveresed)
       """
       #always a square matrix
       for start in range(len(self.pheromone_map)):
        for end in range(len(self.pheromone_map)):
         #decay the pheromone value at this location
         #tau_xy <- (1-rho)*tau_xy (ACO)
         self.pheromone_map[start][end]          =           (1-
self.pheromone_evaporation_coefficient)*self.pheromone_map[sta
rt][end]

         #then add all contributions to this location for each
ant that travered it
         #(ACO)
         #tau_xy <- tau_xy + delta tau_xy_k
         # delta tau_xy_k = Q / L_k
         self.pheromone_map[start][end]                       +=
self.ant_updated_pheromone_map[start][end]

      def _populate_ant_updated_pheromone_map(self, ant):
       """
       given  an  ant,  populate  ant_updated_pheromone_map  with
pheromone values according to ACO
       along the ant's route
```

```
        called from:
         mainloop()
         ( before _update_pheromone_map() )
        """
        route = ant.get_route()
        for i in range(len(route)-1):
         #find the pheromone over the route the ant traversed
         current_pheromone_value                                =
float(self.ant_updated_pheromone_map[route[i]][route[i+1]])

         #update the pheromone along that section of the route
         #(ACO)
         # delta tau_xy_k = Q / L_k
         new_pheromone_value                                    =
self.pheromone_constant/ant.get_distance_traveled()

         self.ant_updated_pheromone_map[route[i]][route[i+1]]  =
current_pheromone_value + new_pheromone_value
         self.ant_updated_pheromone_map[route[i+1]][route[i]]  =
current_pheromone_value + new_pheromone_value

       def mainloop(self):
        """
        Runs the worker ants, collects their returns and updates
the pheromone map with pheromone values from workers
         calls:
         _update_pheromones()
         ant.run()
        runs the simulation self.iterations times
        """

        for _ in range(self.iterations):
         #start the multi-threaded ants, calls ant.run() in a new
thread
         for ant in self.ants:
```

```
         ant.start()

        #source: http://stackoverflow.com/a/11968818/5343977
        #wait until the ants are finished, before moving on to
modifying shared resources
        for ant in self.ants:
         ant.join()

        for ant in self.ants:
         #update  ant_updated_pheromone_map  with  this  ant's
constribution of pheromones along its route
         self._populate_ant_updated_pheromone_map(ant)

         #if we haven't seen any paths yet, then populate for
comparisons later
         if not self.shortest_distance:
          self.shortest_distance = ant.get_distance_traveled()

         if not self.shortest_path_seen:
          self.shortest_path_seen = ant.get_route()

         #if we see a shorter path, then save for return
         if           ant.get_distance_traveled()            <
self.shortest_distance:
          self.shortest_distance = ant.get_distance_traveled()
          self.shortest_path_seen = ant.get_route()
     #decay current pheromone values and add all pheromone values
we saw during traversal (from ant_updated_pheromone_map)
        self._update_pheromone_map()

        #flag  that  we  finished  the  first  pass  of  the  ants
traversal
        if self.first_pass:
         self.first_pass = False
```

```
        #reset all ants to default for the next iteration
        self._init_ants(self.start)

        #reset  ant_updated_pheromone_map  to  record  pheromones
for ants on next pass
        self.ant_updated_pheromone_map                          =
self._init_matrix(len(self.nodes), value=0)

       #translate shortest path back into callers node id's
       ret = []
       for id in self.shortest_path_seen:
        ret.append(self.id_to_key[id])

       return ret
```

ДОДАТОК Б

Імплементація DRAC Model

```
class Model():

    def __init__(self, args, training=True):
        self.args = args
        if not training:
            args.batch_size = 1
            args.seq_length = 1

        # choose rnn cell
        if args.model == 'gru':
            cell_fn = rnn.GRUCell
        else:
            raise Exception("model type not supported:
{}".format(args.model))

        # warp multi layered rnn cell into one cell with
dropout
        cells = []
        for _ in range(args.num_layers):
            cell = cell_fn(args.rnn_size)
            if training and (args.output_keep_prob < 1.0 or
args.input_keep_prob < 1.0):
                cell = rnn.DropoutWrapper(cell,

input_keep_prob=args.input_keep_prob,

output_keep_prob=args.output_keep_prob)
            cells.append(cell)
        self.cell = cell = rnn.MultiRNNCell(cells,
state_is_tuple=True)

        # input/target data (int32 since input is char-
level)
```

```
            self.input_data = tf.placeholder(
                tf.int32, [args.batch_size, args.seq_length])
            self.targets = tf.placeholder(
                tf.int32, [args.batch_size, args.seq_length])
            self.initial_state                                =
cell.zero_state(args.batch_size, tf.float32)

            # softmax output layer, use softmax to classify
            with tf.variable_scope('rnnlm'):
                softmax_w = tf.get_variable("softmax_w",
                                            [args.rnn_size,
args.vocab_size])
                softmax_b    =    tf.get_variable("softmax_b",
[args.vocab_size])

            # transform input to embedding
            embedding      =      tf.get_variable("embedding",
[args.vocab_size, args.rnn_size])
            inputs     =     tf.nn.embedding_lookup(embedding,
self.input_data)

            #  dropout  beta  testing:  double  check  which  one
should affect next line
            if training and args.output_keep_prob:
                inputs         =         tf.nn.dropout(inputs,
args.output_keep_prob)

            # unstack the input to fits in rnn model
            inputs = tf.split(inputs, args.seq_length, 1)
            inputs  =  [tf.squeeze(input_,  [1])  for  input_  in
inputs]

            #  loop  function  for  rnn_decoder,  which  take  the
previous  i-th  cell's  output  and  generate  the  (i+1)-th  cell's
input
```

```
            def loop(prev, _):
                prev = tf.matmul(prev, softmax_w) + softmax_b
                prev_symbol = tf.stop_gradient(tf.argmax(prev,
1))
                return      tf.nn.embedding_lookup(embedding,
prev_symbol)

            # rnn_decoder to generate the ouputs and final
state. When we are not training the model, we use the loop
function.
            outputs,                last_state                =
legacy_seq2seq.rnn_decoder(inputs,  self.initial_state,  cell,
loop_function=loop if not training else None, scope='rnnlm')
            output  =  tf.reshape(tf.concat(outputs,  1),  [-1,
args.rnn_size])

            # output layer
            self.logits   =   tf.matmul(output,   softmax_w)   +
softmax_b
            self.probs = tf.nn.softmax(self.logits)

            # loss is calculate by the log loss and taking the
average.
            loss = legacy_seq2seq.sequence_loss_by_example(
                    [self.logits],
                    [tf.reshape(self.targets, [-1])],
                    [tf.ones([args.batch_size                *
args.seq_length])])
            with tf.name_scope('cost'):
                self.cost     =     tf.reduce_sum(loss)      /
args.batch_size / args.seq_length
            self.final_state = last_state
            self.lr = tf.Variable(0.0, trainable=False)
            tvars = tf.trainable_variables()
```

```
            # calculate gradients
            grads,                      _                    =
tf.clip_by_global_norm(tf.gradients(self.cost, tvars),
                        args.grad_clip)
            with tf.name_scope('optimizer'):
                optimizer = tf.train.AdamOptimizer(self.lr)

            # apply gradient change to the all the trainable
variable.
            self.train_op                                    =
optimizer.apply_gradients(zip(grads, tvars))
        # instrument tensorboard
            tf.summary.histogram('logits', self.logits)
            tf.summary.histogram('loss', loss)
            tf.summary.scalar('train_loss', self.cost)

        def   sample(self,   sess,   chars,   vocab,   num=200,
prime='The ', sampling_type=1):
            state      =      sess.run(self.cell.zero_state(1,
tf.float32))
            for char in prime[:-1]:
                x = np.zeros((1, 1))
                x[0, 0] = vocab[char]
                feed = {self.input_data: x, self.initial_state:
state}
                [state] = sess.run([self.final_state], feed)

            def weighted_pick(weights):
                t = np.cumsum(weights)
                s = np.sum(weights)
                return(int(np.searchsorted(t,
np.random.rand(1)*s)))

            ret = prime
            char = prime[-1]
```

```
            for _ in range(num):
                x = np.zeros((1, 1))
                x[0, 0] = vocab[char]
                feed = {self.input_data: x, self.initial_state:
state}
                [probs,    state]    =    sess.run([self.probs,
self.final_state], feed)
                p = probs[0]

                if sampling_type == 0:
                    sample = np.argmax(p)
                elif sampling_type == 2:
                    if char == ' ':
                        sample = weighted_pick(p)
                    else:
                        sample = np.argmax(p)
                else:  # sampling_type == 1 default:
                    sample = weighted_pick(p)

                pred = chars[sample]
                ret += pred
                char = pred
            return ret
```

ДОДАТОК В

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	100 с.
	Інші документи	
2.	Презентаційні матеріали	16 плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Кучук А.О.			Моделі штучного інтелекту для управління та оптимізації peer-to-peer мереж	Шифр групи	Код напр./спец.
Перевірив		Волощук О.Б.				СШІм-21-1	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Філатов В.О.					