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

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп'ютерних систем

«На правах рукопису»
УДК 004.02

До захисту допущено:
Завідувач кафедри
_____ Віталій РОМАНКЕВИЧ
«___»_____2021 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-професійною програмою

«Системне програмування і спеціалізовані комп'ютерні системи»

зі спеціальності 123 «Комп'ютерна інженерія»

на тему: «Способи розпаралелення алгоритмів класу Random Walk за допомогою фреймворків PyTorch та TensorFlow»

Виконала:
студентка ІІ курсу, групи КВ-01мп
Курдус Анастасія Олександрівна _____

Науковий керівник:
Доцент кафедри СПСКС, к.т.н., доцент
Марченко Олександр Іванович _____
_____

Рецензент:

_____

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

Київ – 2021 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп‘ютерних систем

Рівень вищої освіти – другий (магістерський)

Спеціальність – 123 «Комп‘ютерна інженерія»

Освітньо-професійна програма «Системне програмування і спеціалізовані комп‘ютерні системи»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Віталій РОМАНКЕВИЧ

«___»_____2021 р.

ЗАВДАННЯ
на магістерську дисертацію студентці

Курдус Анастасії Олександрівні

1. Тема дисертації «Способи розпаралелення алгоритмів класу Random Walk за допомогою фреймворків PyTorch та TensorFlow», науковий керівник дисертації Марченко Олександр Іванович, доцент кафедри СПСКС, к.т.н., доцент, затверджені наказом по університету від «5» 11. 2021 р. №3682-С

2. Термін подання студентом дисертації 7.12.2021

3. Об’єкт дослідження: алгоритми класу random walk, а саме LRW та RWR, які реалізуються з використанням розпаралелення за допомогою фреймворків TensorFlow та PyTorch.

4. Вихідні дані: адаптовані до розпаралелення та реалізації на фреймворках алгоритми LRW та RWR; програмно реалізовані алгоритми LRW та RWR.

5. Перелік завдань, які потрібно розробити: опис предметної області досліджень та обґрунтування використання фреймворків TensorFlow та PyTorch для реалізації алгоритмів LRW та RWR; адаптування алгоритмів LRW та RWR до реалізації за допомогою фреймворків TensorFlow та PyTorch; реалізація алгоритмів LRW та RWR за допомогою фреймворків TensorFlow та PyTorch.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу – презентація

7. Перелік публікацій:

- «Сфери використання random walks алгоритмів та їхня актуальність», VIII міжнародна науково-технічна Internet-конференція. – 2021;
- «Random Walk алгоритми», XIV науково-практична конференція магістрантів та аспірантів ПМК-2021. – 2021.

8. Дата видачі завдання 7.10.2020

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Вивчення літератури за тематикою роботи	10.10.2020	
2	Розроблення та узгодження завдання магістерської дисертації	21.10.2020	
3	Аналіз існуючих алгоритмів класу random walk	10.01.2021	
4	Аналіз існуючих фреймворків для паралельного обчислення	25.02.2021	
5	Підготовка матеріалів першого розділу магістерської дисертації	16.04.2021	
6	Підготовка матеріалів другого розділу магістерської дисертації	05.06.2021	
7	Реалізація алгоритмів LRW та RWR за допомогою фреймворків TensorFlow та PyTorch	13.09.2021	
8	Підготовка матеріалів третього розділу магістерської дисертації	27.07.2021	
9	Підготовка матеріалів четвертого розділу магістерської дисертації	13.09.2021	
10	Підготовка графічної частини магістерської дисертації	16.10.2021	
11	Оформлення документації магістерської дисертації	01.11.2021	
12	Попередній розгляд магістерської дисертації на кафедрі	01.12.2021	

Студентка Анастасія КУРДУС

Науковий керівник Олександр МАРЧЕНКО

РЕФЕРАТ

Актуальність теми. У наш час random walks алгоритми доволі популярні і використовуються у багатьох сферах. За їх допомогою покращується точність отриманих результатів, а також швидкість роботи. Random walks алгоритми успішно застосовуються в різних областях інформатики, таких як collaborative filteringre, commender system, computer vision, network embedding, link prediction, semi-supervised learning, element distinctness.

Такі алгоритми, як RWR та LRW є класичними алгоритмами групи алгоритмів random walks. Саме ці алгоритми найчастіше використовують в Computer Vision.

Computer Vision - це міждисциплінарне поле, яке розглядає те, як можна створити комп'ютери, які можуть проводити стеження, виявлення та класифікацію об'єктів. Його завдання включає методи збору, обробки, аналізу та розуміння цифрових зображень та вилучення багатомірних даних із реального світу. Найбільш важливою та актуальною галуззю застосування Computer Vision є медицина. З використанням цієї технології отримують інформацію з відеоданих, аналізуючи яку, визначають діагноз пацієнта. Також технологія використовується в промисловості для виявлення дефектів кінцевого продукту.

Тож, як видно з усього вищезазначеного, алгоритми групи random walks актуальні на сьогоднішній день і продовжують набирати популярність.

Об’єктом дослідження є алгоритми RWR та LRW, які є класичними алгоритмами групи алгоритмів random walks та фреймворки TensorFlow та PyTorch .

Предметом дослідження є алгоритми RWR та LRW, які є класичними алгоритмами групи алгоритмів random walks.

Мета роботи: покращення алгоритмів RWR та LRW за допомогою використання фреймворків TensorFlow та PyTorch, а також розробка програмного забезпечення для реалізації та тестування алгоритмів.

Методи дослідження. Метод оптимізації, експериментальний метод.

Наукова новизна полягає в наступному:

1. Запропоновано нову реалізацію алгоритмів RWR та LRW, яка відрізняється від існуючих адаптацією під використання тензорних обчислень і створена на основі фреймворків TensorFlow та PyTorch.
2. Запропоновано спосіб ропаралелення алгоритмів RWR та LRW з використанням фреймворків TensorFlow та PyTorch.
3. Виконано порівняння ефективності роботи запропонованого алгоритму з існуючими рішеннями, наведено приклади, при яких цей алгоритм показує кращі та гірші результати в порівнянні з існуючими алгоритмами.

Практична цінність отриманих в роботі результатів полягає в тому, що запропонований алгоритм дозволяє створювати швидкі програмні системи, в яких використовуються алгоритми RWR та LRW, за рахунок використання фреймворків TensorFlow та PyTorch.

Апробація роботи. Основні положення і результати роботи були представлені на XIV науковій конференції «Прикладна математика та комп'ютинг» ПМК-2021 (Київ, 17-19 листопада 2021 р.) та на VIII Міжнародній науково-технічній Internet-конференції "Сучасні методи, інформаційне, програмне та технічне забезпечення систем керування організаційно-технічними та технологічними комплексами" (Київ, 25-26 листопада 2021 р.).

Структура та обсяг роботи. Магістерська дисертація складається з вступу, чотирьох розділів та висновків.

У вступі подано загальну характеристику роботи, зроблено оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень, сформульовано мету і задачі досліджень, показано наукову новизну отриманих результатів і практичну цінність роботи.

У першому розділі розглянуто основні принципи реалізації алгоритмів RWR та LRW , їхні особливості, недоліки та переваги.

У другому розділі запропоновано метод прискорення RWR та LRW на основі використання фреймворків TensorFlow та PyTorch.

У третьому розділі наведено особливості реалізації розробленої системи.

У четвертому розділі представлено підходи до тестування системи в цілому та окремих модулів.

У висновках представлені результати проведеної роботи.

Робота представлена на 82 аркушах, містить посилання на список використаних літературних джерел.

Ключові слова: PyTorch, TensorFlow, random walks, RWR, LRW .

ABSTRACT

Actuality of theme. Nowadays, random walks algorithms are quite popular and used in many fields. They improve the accuracy of the results, as well as speed. Random walks algorithms are successfully used in various fields of computer science, such as collaborative filteringre, commender system, computer vision, network embedding, link prediction, semi-supervised learning, element distinctness.

Algorithms such as RWR and LRW are classic algorithms of the random walks group of algorithms. These algorithms are most often used in Computer Vision.

Computer Vision is an interdisciplinary field that examines how computers can be created that can track, detect, and classify objects. Its tasks include methods of collecting, processing, analyzing and understanding digital images and extracting multidimensional data from the real world. The most important and relevant area of application of Computer Vision is medicine. Using this technology, information is obtained from video data, analyzing which determines the diagnosis of the patient. The technology is also used in industry to detect defects in the final product.

So, as can be seen from all the above, the algorithms of the group random walks are relevant today and continue to gain popularity.

The object of research is the RWR and LRW algorithms, which are classic algorithms of the group of random walks algorithms and TensorFlow and PyTorch frameworks.

The subject of research is the RWR and LRW algorithms, which are classical algorithms of the group of random walks algorithms.

Objective: to improve the RWR and LRW algorithms through the use of TensorFlow and PyTorch frameworks, as well as to develop software for the implementation and testing of algorithms.

Research methods. Optimization method, experimental method.

The scientific novelty is as follows:

1. A new implementation of RWR and LRW algorithms is proposed, which differs from the existing ones by adaptation for the use of tensor calculations and is created on the basis of TensorFlow and PyTorch frameworks.

2. A method for paralleling RWR and LRW algorithms using TensorFlow and PyTorch frameworks is proposed.
3. The comparison of efficiency of work of the offered algorithm with existing decisions is executed, examples at which this algorithm shows the best and worse results in comparison with existing algorithms are resulted.

The practical value of the results obtained in this work is that the proposed algorithm allows you to create fast software systems that use RWR and LRW algorithms, through the use of TensorFlow and PyTorch frameworks.

Approbation of work. The main provisions and results of the work were presented at the XIV Scientific Conference "Applied Mathematics and Computing" PMK-2021 (Kyiv, November 17-19, 2021) and at the VIII International Scientific and Technical Internet Conference "Modern Methods, Information, Software and technical support of management systems of organizational, technical and technological complexes "(Kyiv, November 25-26, 2021).

Structure and scope of work. The master's dissertation consists of an introduction, four chapters and conclusions.

The *introduction* presents a general description of the work, assesses the current state of the problem, substantiates the relevance of research, formulates the purpose and objectives of research, shows the scientific novelty of the results and the practical value of the work.

The *first section* discusses the basic principles of implementation of RWR and LRW algorithms, their features, disadvantages and advantages.

The *second section* proposes a method for accelerating RWR and LRW based on the use of TensorFlow and PyTorch frameworks.

The *third section* presents the features of the developed system.

The *fourth section* presents approaches to testing the system as a whole and individual modules.

The *conclusions* present the results of the work.

The work is presented on 82 sheets, contains references to the list of used literature sources.

Keywords: PyTorch, TensorFlow, random walks, RWR, LRW.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ ТА ТЕРМІНІВ

CPU – Central Processing Unit

GPU – Graphics Processing Unit

LRW – Lazy Random Walk

RW – Random Walk

RWR – Random Walk With Restart

TPU – Tensor Processing Unit

ВСТУП

Random walks алгоритми стають все більш популярним у різних дисциплінах, таких як математика, інформатика, фізика, хімія, біологія, економіка тощо. Методи і технології в яких використовуються ці алгоритми набирають все більшої популярності. Random walks алгоритми використовуються в collaborative filteringre, commender system, computer vision, network embedding, link prediction, semi-supervised learning, element distinctnes. Порівняно з іншими альтернативними підходами, random walks алгоритми демонструють більшу точність та кращу швидкодію. Так як ці алгоритмі відносно нові, то практичних реалізацій досить мало. Проаналізувавши random walks алгоритми, стало зрозуміло, що теоретично, їх досить легко розбити на незалежні етапи та виконувати паралельно. Для реалізації розпаралелювання було вирішено використати фреймворки для паралельних обчислень TensorFlow та PyTorch. Найчастіше ці бібліотеки використовується для розробки нейронних мереж та для симуляторів фізичних процесів. Мета моєї магістерської дисертації — розробити random walks алгоритми з використанням TensorFlow та PyTorch фреймворків і проаналізувати які покращення дає така реалізація. Опираючись на вже існуючі реалізації різних алгоритмів з використанням вище зазначених фреймворків, передбачається збільшення ефективності роботи random walks алгоритмів.

1. ОПИС АЛГОРИТМІВ RANDOM WALK ТА ОБҐРУНТУВАННЯ ТЕМИ МАГІСТЕРСЬКОЇ ДИСЕРТАЦІЇ

В цьому розділі будуть описані основні random walk алгоритми та сфери в яких вони використовуються.

1.1 Random Walks алгоритми

Суть random walks алгоритмів в тому, що описується траєкторія, яку утворюють послідовні випадкові кроки, які виконуються в якомусь математичному просторі. Прикладом, який пояснює, що таке random walk є випадкове блукання по числовій прямій цілих чисел. Воно починається в точці 0 і на кожному кроці зсувається на +1 або на -1 з однаковою ймовірністю. Ще одним прикладом являється броунівський рух (траєкторія руху молекули в рідині або газі) і таких прикладів досить багато в реальному житті. Ці приклади показують, що модель випадкового блукання (random walk) може використовуватися в різних галузях: екологія, психологія, інформатика, фізика, хімія, біологія, економіка і соціологія.

1.1.1 Базові алгоритми

Серед базових random walks алгоритмів виділяють: Page Rank, Personalized Page Rank, RWR, LRW, B_LIN та K-dash [1].

Page Rank – алгоритм оснований на розрахунку ваги веб-сторінки шляхом підрахунку документів, які посилаються на неї. Алгоритм оцінює важливість вершини графа на основі посилань в графі. Кожній з вершин призначається певне числове значення. Це значення визначає «важливість» або «авторитетність» цієї вершини серед інших [1]. Моделюється поведінка користувача, який випадковий

чином переходить за певними посиланнями в графі. Користувач може не пройти по посиланню з ймовірністю α, а перестрибнути на іншу випадкову вершину. Метод оцінює частоту потрапляння в кожну вершину. Причому алгоритм може застосовуватися, як до веб-сторінок, так і до будь-якого комплекту об'єктів, які зв'язуються між собою посиланнями. Чим більше посилань на сторінку, тим вона «важливіше». При цьому вага сторінки «A» залежить від ваги посилання, яка передається сторінкою «B». Схема, яка показує принцип роботи алгоритму Page Rank зображена на рис. 1.1.

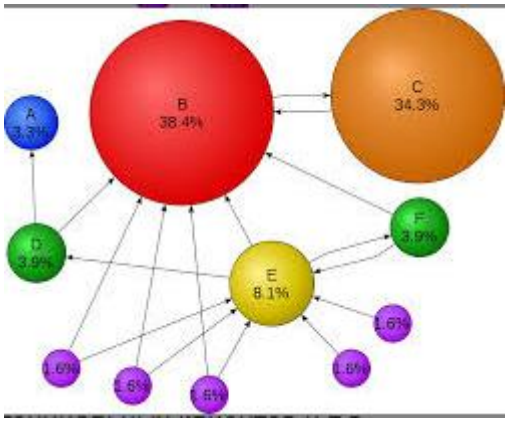

Рисунок 1.1 – Схема роботи алгоритму Page Rank

Формульне представлення random walk алгоритму Page Rank можна записати так (1.1):

$$R_{t+1}=M^T R_t \qquad (1.1)$$

де R - вектор Page Rank, M^T це матриця ймовірностей переходу, t це кількість кроків [1].

Personalized Page Rank – алгоритм майже в точності аналогічний попередньому алгоритму, за винятком того, що замість стрибка на випадкову вершину, користувач

завжди потрапляє в заздалегідь задану вершину. Формульне представлення алгоритму Personalized Page Rank виглядає так (1.2) [1]:

$$R_{t+1}=(1-\alpha)M^T R_t + \alpha \mathrm{p} \quad (1.2)$$

де α - коефіцієнт загасання, p - персоналізований Page Rank вектор, що відображає важливість кожного вузла на графіку для конкретного користувача [1].

Random Walk with Restart (RWR) – алгоритм використовується для реалізації можливості отримати рекомендації, які основані на різноманітних даних соціальної мережі, таких як історія прослуханих треків, соціальні зв'язки, схожість смаків в різних темах і багато іншого. Ці дані представлені в алгоритмі у вигляді графа відносин між об'єктами, де об'єктами є користувачі та те, що їх цікавить (музика, фото, інша людина і т.д.). Ці об'єкти і зв'язки між ними є вершинами і дугами графа відповідно, ваги дуг визначаються силою зв'язності. Починаючи з вершини x, RWR на кожному наступному кроці виконує перехід з поточної вершини до наступної по дугам, причому ймовірність переходу прямо пропорційна вазі дуги. Важливо зазначити, що кожного разу з фіксованою ймовірністю α, обхід може повернутися в початкову вершину x. Формульне представлення алгоритму можна відобразити так (1.3) [1]:

$$P^{t+1}=(1-a)Sp^t+a\mathrm{q} \quad (1.3)$$

де, p^tвектор-стовпець, в якому p_i^t означає ймовірність того, що на кроці t RWR знаходиться в вершині i. Нехай q вектор-стовпець, в якому qx = 1, а решта елементи дорівнюють нулю, і S - нормалізована за стовпцями матриця суміжності графа [1].

Lazy Random Walk (LRW) – використовується для вирішення проблем сегментації зображення. Особливістю цього алгоритму є те, що перехід завжди відбувається з ймовірністю ½. Тобто ймовірність того, що ходунок знаходиться на поточній

вершині завжди дорівнює ½, але і ймовірність, що він перейде на сусідню вершину також рівна ½. Формульне представлення алгоритму виглядає так (1.4) [1]:

$$p_{t+1}(i) = 1/2\left(p\middle|t(i)\right) + 1/2 \quad (1.4)$$

B_LIN та K-dash це доволі нові алгоритми, які були створенні з метою покращення швидкості роботи алгоритмів random walk. Досягається ця мета за рахунок того, що ці алгоритми обчислюють близькість наступної вершини за допомогою розрідженої матриці, а також пропускає непотрібні обчислення під час пошуку найближчих вершин [1]. Теоретичний аналіз цих алгоритмів показує, що вони гарантують високу точність отриманих результатів. Результати теоретичного аналізу підтверджуються і на практиці, адже при практичній перевірці, ці алгоритми показали високу ефективність та продемонстрували, що потрібні вершину можуть бути знайдені швидше і точність при цьому не зменшується [1].

1.1.2 Алгоритми Quantum Walks

Quantum Walks є підрозділом random walks, але від класичних random walks ці алгоритми відрізняються тим, що не сходяться до обмежувальних розподілів, і через силу квантових перешкод вони можуть поширюватися значно швидше, ніж їх класичні еквіваленти. Алгоритми Quantum Walks умовно можна розділити на дві групи: Continuous-time Quantum Walk та Discrete Quantum Walk [1].

Continuous-time Quantum Walk – цю модель можна уявити на прикладі алгоритму дерева рішень. Систематично досліджується ціле дерево за імовірнісним правилом. Розглядаються вузли дерева як квантові стани в просторі Гільберта. Потім на основі отриманих результатів будується гамільтонова функція, яка допомагає дослідити еволюцію часу квантової системи. Прикладом алгоритму, який підпадає під опис цієї моделі є Quantum Decision Tree [1].

Discrete Quantum Walk – умовно кажучи, дискретні квантові прогулянки можна реалізувати шляхом заміни кожного стану класичного неорієнтованого графа на квантовий стан за допомогою використанням допоміжного стану “ coin ” для визначення напрямку, яким буде рухатися ходунок квантової прогулянки. До цієї групи входить такий алгоритм, як Quantum PageRank [1].

1.1.3 Використання Random Walk алгоритмів

У наш час random walks алгоритми доволі популярні і використовуються у багатьох сферах. За їх допомогою покращується точність отриманих результатів, а також швидкість роботи [2].

Collaborative Filtering – це метод автоматичного прогнозування інтересів користувача шляхом збору інформації від багатьох користувачів. Передбачається, що дві людини, які мають однаковий інтерес до одного питання, матимуть однаковий інтерес і з інших питань. У великій кількості літератури зафіксовано успішне використання random walks алгоритмів для вирішення задачі спільного фільтрування. Суть цих агоритмів полягає в тому, що беручи за основу минулий вибір індивідууму, можна спрогнозувати майбутній вибір. Це доволі актуальна задача в наш час, адже велика кількість компаній хоче розуміти, за який продукт їхній клієнт хоче і готовий платити в майбутньому [2].

Recommender System – це підклас системи фільтрації інформації, яка намагається видавати актуальні, для певного користувача, рекомендації [3]. Цей підклас найбільш актуальний для соціальних мереж та пошуку в інтернеті загалом. Так, наприклад, метод MVCWalker [4] на основі random wakls, розроблений для знаходження співробітників, які потенціально задовольняють вимоги певної компанії. Суть методу полягає в тому, що встановлюються певні якості, якими повинен володіти претендент на посаду і використовуючи random walk алгоритм, реалізовується пошук кандидатів у мережі, щоб отримати список рекомендацій потенціальних співробіт-

ників. Це доволі актуальна проблема, яку вирішують random walks алгоритми, адже кількість робітників велика і знаходження спеціаліста, який відповідає всім вимогам, довгий і трудомісткий процес та і загалом, проблема знаходження потрібної інформації серед всієї представленої в інтернеті дуже яскраво виражена на сьогоднішній день [2].

Link prediction – це задача прогнозування зв’язку між двома об’єктами в мережі [5]. Для вирішення цієї проблеми було запропоновано безліч алгоритмів, але вони працюють повільно і точність результатів залишає бажати кращого. Тоді було вирішено використати random walks алгоритми для цієї задачі і було виявлено, що збільшилася точність прогнозування, а також вони мають значно кращу швидкодію. Link prediction також допомагає з'ясувати потенційний зв'язок між мікроРНК та хворобами [5]. Оцінюється гетерогенна мережа мікроРНК-Хвороба як дві підмережі, що перекриваються: підмережа мікроРНК та підмережа хвороба. Далі використовується алгоритм random walk, а саме RWR для прогнозування кандидатів мікроРНК, які потенційно можуть бути пов'язані з хворобами. Були проведенні дослідження і результати показали, що такий метод має хороші показники прогнозування [2].

Computer Vision - це міждисциплінарне поле, яке розглядає те, як можна створити комп'ютери, які можуть проводити стеження, виявлення та класифікацію об'єктів [6]. Його завдання включає методи збору, обробки, аналізу та розуміння цифрових зображень та вилучення багатомірних даних із реального світу. Найбільш важливою та актуальною галуззю застосування Computer Vision є медицина. З використанням цієї технології отримують інформацію з відеоданих, аналізуючи яку, визначають діагноз пацієнта. Також технологія використовується в промисловості для виявлення дефектів кінцевого продукту. Суть застосування random walk полягає в наступному: піксельне сегментування зображення за допомогою random walk алгоритму LRW [6]. Ініціалізуються вихідні позиції та запускається алгоритм LRW на вхідному зображенні, щоб отримати ймовірності кожного пікселя. Тоді кордони почат-

кових пікселів отримуються за допомогою ймовірностей та часу комутації. Алгоритм може дуже добре сегментувати розмиті кордони та складні області текстур [2].

1.2 Аналіз існуючих фреймворків для паралельних обчислень

Проаналізуємо популярні фреймворку для паралельних обчислень, а саме: tensorflow, pytorch, cntk.

1.2.1 TensorFlow

TensorFlow – це фреймворк з відкритим кодом, який використовується для чисельних обчислень коли потрібно забезпечити високу ефективність. Архітектура цього фреймворку настільки гнучка (рис. 1.2), що дозволяє виконувати обчислення на різних платформах, таких як: TPU ,процесори, графічні процесори. Фреймворк сумісний з 64 бітною версією Linux, MacOS, Windows , а також мобільними платформами Android та IOS [7].

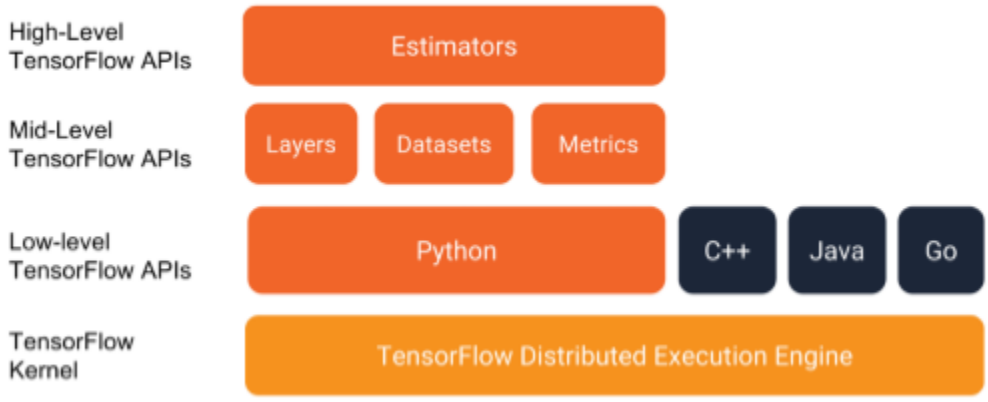

Рисунок 1.2 - Архітектура фреймворку TensorFlow

Найчастіше використовується для машинного навчання та глибокого навчання. Граф потоку даних це вираження обчислень у TensorFlow (рис. 1.3). Граф потоку даних це граф вершини якого це одиниці обчислення, а ребра це ті дані, що поступають на вхід/вихід обчислювальної одиниці.

Рисунок 1.3 – Фрагмент графа потоку даних

Використання графа потоку даних для виконання обчислень має ряд переваг [7]:

- Незалежність виконання – суть полягає в тому, що виконання кожного вузла графа може бути делеговане на різні пристрої. При цьому про зв’язок між вузлами турбується сам TensorFlow, саме він відповідальний за правильність обрахунку всього графу.
- Незалежність від мови та платформи – така незалежність досягається за допомогою використання протоколу буферизації Protobuf. Це дозволяє використовувати модель, яка створена в Python з використанням TensorFlow, для іншої програми, яка реалізована на будь-якій іншій мові програмування.
- Наочність – ця перевага добре проглядається при аналізуванні графу. Аналіз графу допомагає оптимізувати обчислення, адже можна легко виявити верши-

ни, які не використовуються і видалити їх, чим зменшити розмір графу, або можна знайти коротші шляхи для оптимізації і прискорення обрахунків.

- Розпаралелювання – TensorFlow має функціонал для виявлення задач, які можуть бути розпаралеленні. Приклад такого розпаралелювання для операцій додавання та множення можна побачити на рис. 1.4.

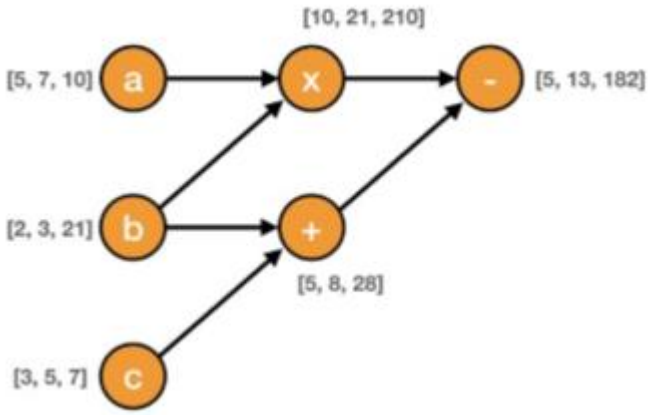

Рисунок 1.4 – Приклад графу потоку даних з реалізацією розпаралелювання

Вершинами графу потоку даних є операції, яка має однин або декілька входів і виходів. Ребра цього графу бувають двох типів: нормальні та спеціальні. Нормальні – це такі для яких вихід однієї операції є входом для іншої, вони є носіями тензорів. Спеціальні – це ті, які навпаки не переносять дані і вихід операції не являється входом для іншої, вони слугують для того, щоб забезпечити послідовне виконання операцій.

Граф потоку має низку основних понять, а саме [7]:

- Операція – це представлення того, як тензори будуть взаємодіяти між собою. Прикладами слугують зрозумілі операції додавання/множення матриць.
- Кернел – співвідносить тип операції та пристрій на якому ця операція буде реалізована та виконана. Наприклад на центральному та графічному процесорах може бути викована операція додавання матриць.

- Сесія – дозволяє реалізувати зв’язок між клієнтскою програмою та системою TensorFlow.

Назва фреймворку походить від слова тензор, яке означає багатовимірний масив над яким нейронні мережі виконують певні операції. Тензори відрізняються від стандартного для більшості мов програмування вигляду багатовимірних масивів даних тим, що по-перше обчислення з використанням тензорів прискорюється за рахунок спеціальних процесорів (тензорних або графічних), а по-друге це те, що тензори не можуть бути перезаписані, їх можна лише створювати. Атрибути обов`язкові для тензора це ступінь, тип даних та форма.

Так, наприклад, ступінь вказує на вимірність тензора. Він може мати ступінь від 0 до n (рис. 1.5).

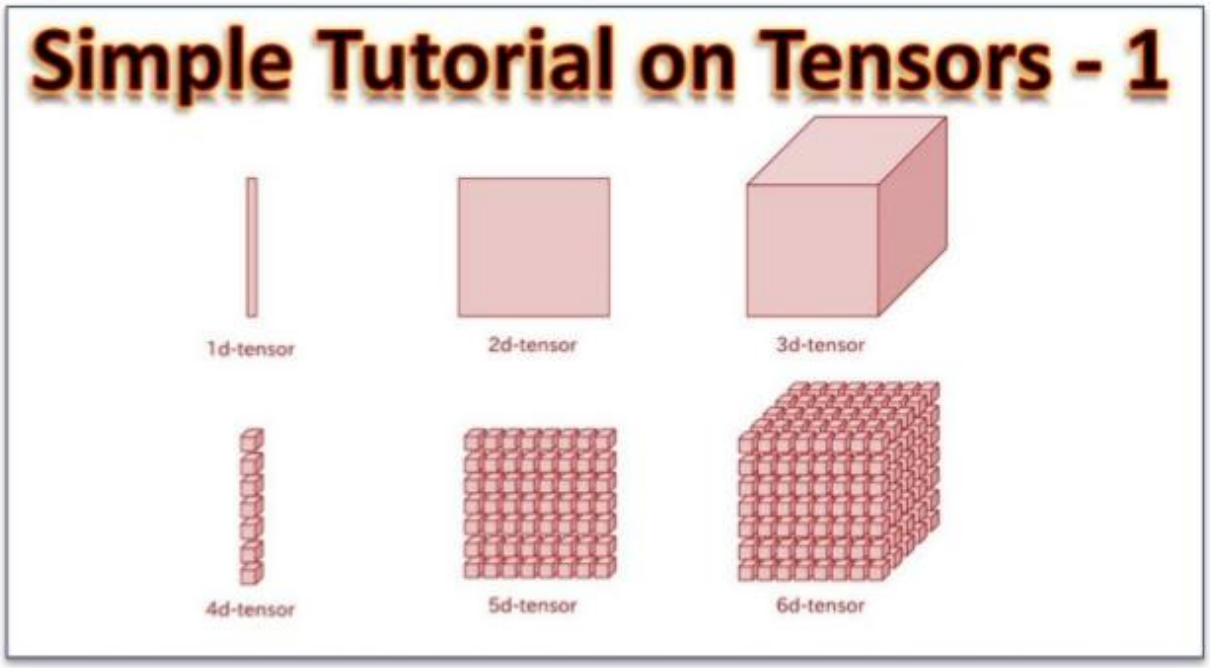

Рисунок 1.5 – Тензори різних ступенів

Тензор ступеня 0 це скаляр, 1 це вектор, 2 це матриця, 3 це кубічний тензор, а n це n-вимірний тензор.

Атрибут тип даних відповідає за визначення того типу даних, які зберігаються у кожній комірці тензора. Найчастіше це типи float, int, string, та інші.

Атрибут форма відповідає за вимірність математичного об'єкту. Для матриці це буде вектор [a, b], де a — кількість рядків матриці, а b – кількість стовпців матриці.

Загалом TensorFlow має ряд переваг, а саме:

- Абстракція – фреймворк TensorFlow самостійно реалізовує всі низькорівневі етапи алгоритму. Розробнику потрібно контролювати лише загальну логіку алгоритму.
- Зручний додатковий інструментарій для зручності аналізу розробленого алгоритму.
- Постійна підтримка фреймворку зі сторони Google.
- Доступність фреймворку для всіх операційних систем.
- Можливість використання одного и того самого коду на різних пристроях без внесення змін в цей код.

1.2.2 PyTorch

PyTorch – це фреймворк для мови програмування Python, який використовується для машинного навчання. Він має вихідний код у відкритому доступі та реалізований на базі Torch. Torch – це бібліотека схожа на MATLAB. Вона, зазвичай, використовується з мовою програмування Lua і також має відкритий вихідний код [8].

PyTorch має дві основні високо рівневі моделі:

- глибокі нейронні мережі на базі autodiff
- тензорні обрахунки (PyTorch підтримує різні типи тензорів)

Autodiff – це автоматичне диференціювання. У бібліотеках глибокого навчання є механізми обчислення градієнта помилки та зворотного розповсюдження помилки через обчислювальний граф. Цей механізм, званий автоградієнтом PyTorch, легко дос-

тупний і інтуїтивно зрозумілий. Змінний клас - головний компонент автоградієнтної системи PyTorch. Змінний клас обгортає тензор і дозволяє автоматично обчислювати градієнт на тензорі за виклику функції .backward(). Об'єкт містить дані з тензора, градієнт тензора (одноразово порахований по відношенню до деякого іншого значення) і містить посилання на будь-яку функцію, створену змінною (якщо це функція створена користувачем, посилання буде порожнім).

PyTorch тензори – це звичайний n-мірний масив, який можна використовувати певних числових обрахунків. Проте ці тезори мають певну особливість, а саме їх можна використовувати і на CPU, і на GPU. Для того,щоб використовувати ці тензори на графічному процесорі достатньо просто привести тензор до типу даних cuda.

PyTorch має ряд модулів, а саме [8]:

- модуль Autograd
- модуль Optim
- модуль nn

Модуль Autograd – це метод, який використовується при створенні нейронних мереж, так як дозволяє вираховувати диференціальні зміни параметрів одночасно з проходом. Цей модуль базується на методі автоматичної диференціації, який використовується в PyTorch. Суть полягає в тому, що здійснюється запис обчислень, які були отримані в прямому напрямку проходження, а потім проводиться відтворення у заоротньому порядку для обчислення градієнтів.

Модуль Optim – це модуль, яких увібрав в себе всі найпопулярніші і найбільш використовувані при побудові нейронних мереж алгоритми оптимізації.

Модуль nn – це модуль аналогічний модулю autograd, він також дозволяє знаходити графи обчислень, працювати з градієнтами, проте на більш високому рівні абстракції.

1.2.3 CNTK

CNTK (Microsoft Cognitive Toolkit) – це стандартизований інструментарій з відкритим кодом, який використовується для обробки великої кількості даних. Використовує спрямований граф для опису нейронних мереж [9]. За допомогою CNTK також можна реалізовувати та поєднувати такі типи моделей, як DNN-канали, CNN та RNN/LSTM (рис. 1.6). CNTK має можливість реалізувати розпаралелювання з залученням різної кількості графічних процесорів та серверів. BrainScript це мова за допомогою якої CNTK може застосовуватися, як незалежний інструмент машинного навчання. Проте CNTK також може бути представлений, як бібліотека для таких популярних мов програмування, як Python, C# та C++.

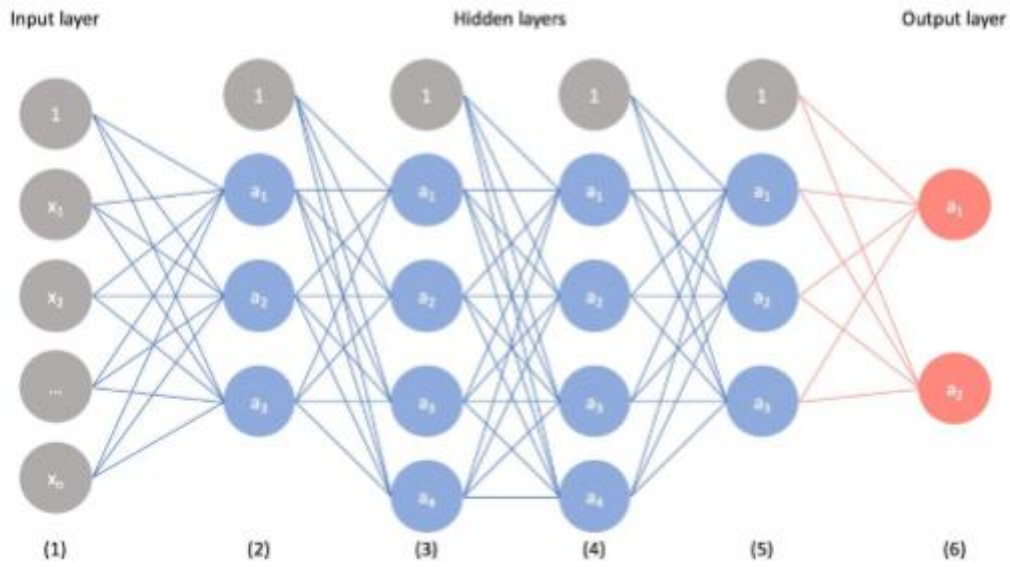

Рисунок 1.6 – Приклад LSTM мережі

CNTK представляє нейрону мережу, як певний порядок обчислювальних кроків, які реалізуються за допомогою орієнтованого графу. Кінцеві вузли цього графу це вхідні розміри мережі, інші вузли реалізуються, як матричні операції на вхідних умовах.

CNTK, як бібліотека для Python має ряд переваг [9]:

- Нативний інтерфейс – за рахунок простого і зрозумілого інтерфейсу користувач може без проблем передавати потрібні дані в обчислювальне ядро.
- Адаптивність – така перевага реалізується за рахунок гнучкості при створенні та тренуванні нейронної мережі.
- Зчитувачі даних – тут перевага полягає в тому, що ці зчитувачі є вбудованими. Це допомагає позбавитися надмірного копіювання даних.

До недоліків можна віднести лише те, що використання CNTK з мобільними пристроями доволі обмежене.

1.2.4 Вибір технологій для реалізації алгоритмів random walk

Для реалізації алгоритмів LRW та RWR, які відносяться до класу алгоритмів random walk, було обрано два фреймворки, а саме: TensorFlow та PyTorch. Ці фреймворки, як вже описувалось вище, працюють с тензорами (основна одиниця даних для представлених фреймворків), які представляють собою матрицю. Тому TensorFlow та PyTorch ідеально підходять для роботи з матрицями. Вони можуть швидко виконувати всі операції з матрицями: додавання матриць, множення матриць між собою та множення матриці на число, транспонувати матриці і багато іншого. Також досить важливим є той факт, що ці фреймворки виконують всі вище вказанні та інші операції над матрицями паралельно, що в свою чергу значно прискорює час виконання алгоритму, який реалізований з використанням фреймворків TensorFlow та PyTorch. Беручи до уваги всі вказані аспекти роботи та побудови фреймворків TensorFlow та PyTorch, а також зважаючи на суть алгоритмів LRW та RWR, які базуються на матрицях і операціях над цими матрицями, стає зрозуміло, що реалізація цих алгоритмів за допомогою фреймворків TensorFlow та PyTorch до-

сить обгрунтоване та дасть позитивні результати в вирішенні задачі покращення роботи алгоритмів RWR та LRW класу random walk.

1.3 Обгрунтування теми магістерської дисертації

Як було описано в попередніх розділах, алгоритми класу random walk доволі популярні в різних сферах, таких як математика, інформатика, хімія, фізика та багато інших. Серед найпопулярніших методів в яких використовуються алгоритми класу random walk можна відзначити:

- Collaborative Filtering
- Recommender System
- Link prediction
- Computer Vision

В свою чергу ці методи мають широке практичне застосування в багатьох, доволі важливих, галузях. Тому покращення алгоритмів, які лежать в основі цих методів досить потрібна і актуальна задача. Адже це дозолить швидше працювати самим методам, що в свою чергу вплине на швидкість роботи всієї системи, де ці методи використовуються.

Аналізуючи варіанти можливих рішень поставленої задачі, стало зрозуміло, що саме розпаралелення алгоритмів класу random walk, дозволить їм працювати швидше. Також важливим моментом став вибір інструментів, які допоможуть в реалізації поставлених цілей, адже важливо щоб алгоритми не лише швидко працювали, але і їх інтеграція з обраним інструментом реалізації була легкою і гнучкою. Тому, серед всіх фреймворків для паралельних обчислень, було прийнято рішення обрати саме TensorFlow та PyTorch. Основною причиною такого вибору, стало те, що вони побудовані на тензорах і мають внутрішню реалізацію розпаралелення матричних операцій, що чудово поєднується з основною ідеєю алгоритмів random walk.

Висновки до розділу

Тож, як видно з інформації, яка подана у першому розділі магістерської дисертації, random walks алгоритми успішно застосовуються в різних областях інформатики, таких як collaborative filteringre, commender system, computer vision, network embedding, link prediction, semi-supervised learning, element distinctness [1]. Вони проникають у все більше важливі галузі та набирають свою популярність. Тому покращення цих алгоритмів це досить актуальна проблема і задача. Саме це підштовхнуло мене до вибору теми магістерської дисертації, адже з використанням таких фреймворків, як TensorFlow та PyTorch, за допомогою розпаралелювання можна досягти досить хороших результатів в вирішенні поставленої задачі.

2. АДАПТУВАННЯ RANDOM WALK АЛГОРИТМІВ ДО ВИКОРИСТАННЯМ ФРЕЙМВОРКІВ PYTORCH ТА TENSORFLOW

Так як, суть даних алгоритмів, як і більшості інших алгоритмів, описана теоретично, то для реалізації на будь-якій мові програмування їх треба адаптувати. Адаптування теоретично описаного алгоритму означає чітке виокремлення певних етапів з яких складається алгоритм. Тобто потрібно чітко розуміти, які будуть вхідні дані, яким чином вони будуть оброблятися, який функціонал обраної мови програмування чи фреймворку буде потрібен для реалізації поставленої задачі та врешті-решт, яким повинен бути фінальний результат роботи, вже реалізованого, алгоритму. В цьому розділі буде детально розглянуто та проаналізовано саме питання адаптування алгоритмів LRW та RWR, які належать до класу random walk.

2.1 Адаптування алгоритму LRW до реалізації за допомогою фреймворків PyTorch та Tensorflow

В цьому підпункті буде детально розглянуто адаптування до реалізації за допомогою фреймворків PyTorch та Tensorflow саме алгоритму LRW. Буде наведено теоретичний опис алгоритму класу random walk, а потім описано процес адаптування алгоритму LRW до реалізації з використанням фреймворків PyTorch та Tensorflow шляхом виокремлення чітких етапів.

2.1.1 Загальний опис алгоритму LRW

Основою алгоритму LRW є базовий підхід random walk (рис. 2.1).

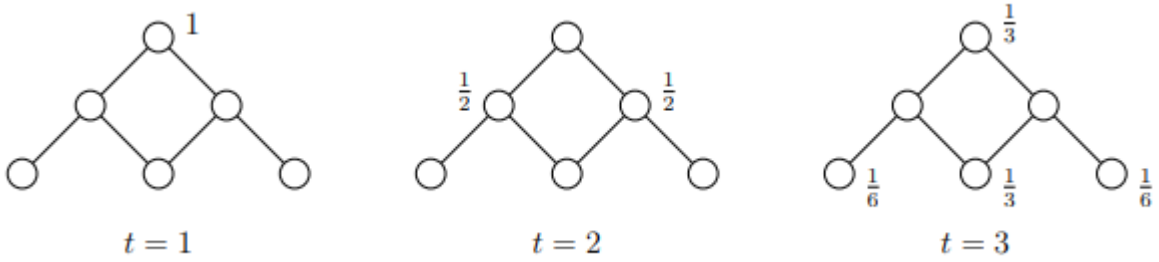

Рисунок 2.1 – Простий приклад стандартного random walk

Він полягає в наступному: нехай G = (V, E) — неорієнтований граф. Розглянемо випадковий процес, який починається з деякої вершини v∈V(G), і багаторазово переміщується до сусідньої для поточної вершини, яка обрана випадковим чином. Для t ≥ 0 і для u∈V(G), нехай $p_t(u)$ позначає ймовірність того, що ви перебуваєте у вершині u в момент t. Формульно можна представити це, як (2.1):

$$\sum_{u \in V(G)} p_t(u) = 1 \qquad (2.1)$$

Далі розуміючи, що ви знаходитесь у вершині u в момент t, можна зробити висновок, що тоді в момент t + 1 ви можете перейти до кожного сусіда v з поточної вершини u з ймовірність 1/d(u), де d(u) позначає ступінь u. Формульне представлення виразимо так (2.2):

$$p_{t+1} = \sum_{(u,v) \in E(G)} p_t(u) * \left(\frac{1}{d(u)}\right) \qquad (2.2)$$

Тоді, виходячи з цього, можемо виразити це використовуючи матричну нотацію (2.3):

$$[W(G)]_{i,j} = \begin{Bmatrix} \frac{1}{d(j)} if(i,j) \in E(G) \\ 0 \end{Bmatrix} \quad (2.3)$$

де, $[W(G)]_{i,j}$ це ймовірність переходу від вершини і до вершини j. Тому остаточна формула,яка виражає матрицю W має такий вигляд (2.4):

$$W_G = D^{-1} * A \quad (2.4)$$

де, A — це матриця суміжності G, а D — це степенева матриця (діагональна матриця, котра виражає степінь кожної вершини певного графа) .

Іншим підходом для переходу на сусідню вершину, який і являє собою lazy random walk є заміна певної рівномірної ймовірності на сталу ймовірність, яка дорівнює 1/2. На кожному кроці lazy random walk буде мати таку ймовірність як:

- залишитися в поточній вершині
- зробити перехід на сусідню, обрану випадковим чином, вершину

Важливо розуміти, що ймовірність вибору кожного з цих пунктів дорівнює 1/2.

Тому для lazy random walk алгоритму, формула, яка описує отримання матриці ваг (W) буде мати наступний вигляд (2.5):

$$W = \frac{1}{2}(I + D^{-1} * A) \quad (2.5)$$

Перехід зі стану p_tдо стану p_{t+1}буде відбуватися за наступною формулою (2.6):

$$p_{t+1} = Wp_t \quad (2.6)$$

2.1.2 Адаптований до фреймворку Tensorflow та PyTorch алгоритм LRW

Існує достатня кількість статей та інших матеріалів, які описують алгоритм lazy random walk, проте всі ці описи теоретичні, а ось практичної реалізації цього алгоритму майже немає у відкритому доступі, а реалізація з використанням фреймворків TensorFlow та PyTorch взагалі відсутня. Тому одним із важливих кроків при написанні магістерської дисертації було виявлення та чіткий поділ теоретичного алгоритму на практичні етапи і адаптування цих етапів під використання фреймворків TensorFlow та PyTorch. Першим важливим етапом було розподілення алгоритму на складові, а саме:

- вхідні дані
- допоміжні дані
- чітко обумовлені вихідні дані

Вхідні дані — це ті дані, які потрібно передати в функцію при її виклику. У випадку lazy random walk алгоритму такими даними є матриця ваг для кожної вершини графа та епсілон. Епсілон — це константа, яка являє собою умову припинення роботи алгоритму, тобто, якщо певна змінна, яка змінює своє значення на кожному етапі ітерації при роботі алгоритму, набуде значення, яке буде рівним значенню константи епсілон, то програма, яка реалізує алгоритм lazy random walk, припинить свою роботу.

Допоміжні дані — це ті дані, які потрібні для адаптування теоретичного алгоритму lazy random walk до його практичної реалізації. В даному випадку, для конкретно взятого алгоритму такими допоміжними даними є: діагональна одинична матриця, стовбець нулів, нормалізована матриця, діагональна матриця суміжностей. Діагональна одинична матриця(рис. 2.2) потрібна для встановлення початкового вектора ймовірностей. Тобто на першій ітерації ймовірність залишитися в початковій вершині приймається за одиницю.

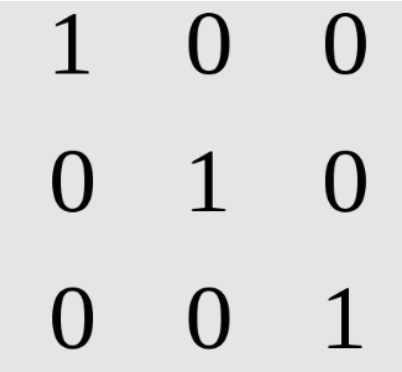

Рисунок 2.2 – Приклад одиничної діагональної матриці

Стовбець нулів — це вектор, який не бере участі в теоретичному описі алгоритму, але потрібен для програмної реалізації lazy random walk. Він використовується для початкового завдання значення епсілон. Нормалізована матриця — це матриця, яка використовується в теоретичному описі алгоритму і виступає частиною формули lazy random walk. Вона вираховується шляхом множення діагональної матриці суміжностей в степені мінус один на вхідну матрицю ваг для кожної вершини графа. Діагональна матриця суміжностей — це та матриця, яка бере участь у формуванні нормалізованої матриці.

Вихідні дані — це результат, який повинен бути отриманий після того, як програма, яка реалізовує алгоритм lazy random walk, завершить свою роботу. В даному випадку результатом буде матриця. Ця матриця є відображенням ймовірності опинитися в певній вершині графа в певний момент часу при умові виходу з початкової вершини. Умовно кажучи, якщо в результаті роботи програми, було отримано матрицю, яка має такі значення:

[0,15 0,25 0,6]

[0,35 0,25 0,4]

[0,45 0,25 0,3]

то, аналізуючи цю матрицю, можна зробити висновок, що, якщо почати з вершини i=1, j=1 ймовірність залишитися в цій вершині дорівнює 0,15, а ймовірність потрапити у вершину i=1, j=2 дорівнює 0,25, ймовірність потрапити у вершину i=1,

j=3 буде рівна 0,6 і так далі. Також важливим зауваженням є те, що на правильність отриманий результат можна перевірити додавши всі значення ймовірностей в одному рядку. Тобто, наприклад, в наведеному прикладі сума ймовірностей в першому рядку (0,15 + 0,25 + 0,6) буде дорівнювати одиниці, аналогічно у другому та третьому рядках. Для другого рядка 0,35 + 0,25 + 0,4 = 1, а для третього рядка 0,45 + 0,25 + 0,3 = 1.

Другим, не менш важливим етапом, було виявлення тих дій алгоритму lazy random walk, які можна розпаралелети. Проаналізувавши, всі дії, які виконує програма, яка реалізує алгоритм lazy random walk, стало зрозуміло, що можна створити паралельний запуск обрахунку вихідного вектора для кожної початкової вершини графа.

Третім етапом є необхідність перетворення всіх операцій, які використовуються в програмній реалізації алгоритму, на тензорні. Це важливий крок для переходу від eiger режиму до графічного режиму. Саме перехід до графічного режиму може забезпечити потрібне розпаралелення операцій.

2.2 Адаптування алгоритму RWR до реалізації за допомогою фреймворків PyTorch та Tensorflow

В цьому підпункті буде детально розглянуто адаптування до реалізації за допомогою фреймворків PyTorch та Tensorflow саме алгоритму RWR. Буде наведено теоретичний опис алгоритму класу random walk, а потім описано процес адаптування алгоритму RWR до реалізації з використанням фреймворків PyTorch та Tensorflow шляхом виокремлення чітких етапів.

2.2.1 Загальний опис алгоритму RWR

Алгоритм random walk with restart, як і попредній алгоритм lazy random walk, базується на стандартному підході random walk. Тобто для RWR алгоритму також актуальні формули (2.7) та (2.8):

$$\sum_{u \in V(G)} p_t(u) = 1 \tag{2.7}$$

$$W_G = D^{-1} * A \tag{2.8}$$

Проте, як і алгоритм lazy random walk, алгоритм random walk with restart, має свої особливості. Цей алгоритм, як і всі алгоритми класу random walk досліджує зв'язок між кожним кроком, який ви б зробили, і його відстань від початковою точкою з якої почався шлях (рис. 2.3).

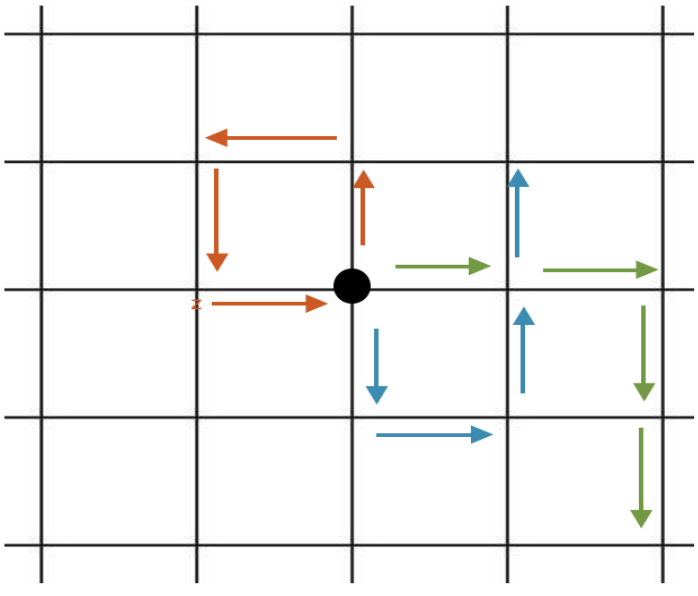

Рисунок 2.3 – Схематичне зображення випадкових переходів до сусідніх вершин

На рисунку 2.3 початковою точкою виступає чорне коло, то ймовірність рухатися від нього в будь якому напрямку є рівною в будь-який момент часу. На

зображенні також показано 3 можливі шляхи після 4-х часових кроків, кожна з яких має різну кінцеву точку відносно початкової координати. Зрозуміло, що після моделювання цього процесу певну, велику кількість разів із достатньо великим кроком у часі, можна створити розподіл ймовірностей, пов'язаний з результатом кожного випадкового блукання, що закінчується в певному місці. Тобто після багаторазового моделювання описаного процесу, буде чітко видно в які вершини графу можна потрапити частіше, а в які потрапити ймовірність менша.

Random walk with restart в загальному випадку відрізняється тим, що має один додатковий компонент, а саме ймовірність перезапуску. По суті, це вказує на те, що для кожного кроку, зробленого в будь-якому напрямку і в будь-який момент часу, існує ймовірність, пов'язана з поверненням до початкової позиції. Формульно RWR можна представити так (2.9):

$$r = c*W*r+ (1-c)*e \tag{2.9}$$

де, *c* - число в діапазоні (0,1), *W* — матриця ймовірностей переходу, де *W[i,j]* позначає ймовірність переходу від j до *i* , e — початковий вектор, тобто *e[i] = 1,* якщо *i* - початковий вузол, інакше *0,* а *r* - вектор-стовпець, де *r[i]* позначає ймовірність перебування у вузлі *i.*

Припустимо, що ми знаходимося лише у початковому вузлі *e*. Тоді ймовірність *r* опинитися в будь-якому вузлі *i* при наступному ході буде задана як (2.10):

$$r_0[i] = W[i]*e \tag{2.10}$$

де, *W[i]* представляє і-й рядок матриці W, тобто ймовірності переходу до вузла *i* з усіх вузлів.

Отже, без перезапуску ймовірність опинитися у вузлі *i* після *k* переміщень можна просто задати за допомогою (2.11):

$$r_k[i] = W[i]^k * e = W[i] * r_{k-1} \quad (2.11)$$

Тепер, беручи до уваги перезапуск, ми отримуємо (2.12):

$$r_k[i] = cW[i] * r_{k-1} + (1-c) * e \quad (2.12)$$

Отже, завдання полягає в тому, щоб знайти таке значення p_k, щоб подальше моделювання не вплинуло на ймовірності. Отже, ми припускаємо, що $r_{k-1} = r_k$. Таким чином, ми можемо записати вищезгадане рівняння, яке і являє собою головне рівняння алгоритму RWR (2.13):

$$r = c * W * r + (1-c) * e \quad (2.13)$$

Тобто, як видно, єдина відмінність між звичайним PageRank і RWR полягає в тому, що зазвичай ймовірність переходу на іншу сусідню вершину приймається рівномірною, тобто ми замінюємо вектор *e* на вектор-стовбець всі елементи якого дорівнюють 1, тоді, як для алгоритму RWR вектор *e* містить одиницю лише для початкової вершини.

Даний алгоритм має як ряд переваг, так і ряд недоліків.

Переваги:

- Забезпечує чітке уявлення подібності між двома вузлами у зваженому та незваженому графі
- Має різноманітні застосування і може використовуватися разом з іншими стохастичними моделями, такими як ланцюги Маркова

Недоліки:

- Використання однакової ймовірності перезапуску для кожного вузла
- Не існує систематичного способу вибору ймовірності перезапуску для кожної програми

2.2.2 Адаптований до фреймворків Tensorflow та PyTorch алгоритм RWR

Адаптування алгоритму random walk with restart до реалізації на фреймворків TensorFlow та PyTorch дуже подібний до вже описаного адаптування алгоритму lazy random walk. Проте існують і певні відмінності, так першим етапом також виступає виділення таких елементів як:

- вхідні дані
- допоміжні дані
- вихідні дані

проте, сам контекст цих груп різниться.

Вхідні дані — тоді як для lazy random walk до таких даних можна було віднести тільки вхідну матрицю ваг для кожної вершини графа та константу епсілон, то для алгоритму random walk with restart, цей список доповнився ще однією константою, яка задається, як початкова умова. Ця константа це значення ймовірності повернення в початкову точку в будь-який момент часу на будь-якій ітерації роботи програми, яка реалізує алгоритм random walk with restart. Важливо зазначити, що значення цієї константи лежить в діапазоні від нуля до одиниці.

Допоміжні дані — всі допоміжні дані, які використовувалися для адаптування алгоритму lazy random walk до реалізації за допомогою фреймворку TensorFlow та PyTorch , використовуються і для адаптування алгоритму random walk with restart до реалізації за допомогою фреймворків TensorFlow та PyTorch . Тобто для цього алгоритму також потрібні: діагональна одинична матриця, стовбець нулів,

нормалізована матриця, діагональна матриця суміжностей. Вони виконують абсолютно такі самі функції для адаптування алгоритму random walk with restart, як і для адаптування алгоритму lazy random walk.

Вихідні дані — зрозуміло, що для алгоритмів lazy random walk та random walk with restart вихідні дані будуть однакові, адже ці алгоритми належать до одного класу random walk. Тому результатом роботи програми, яка реалізує алгоритм random walk with restart за допомогою фреймворків TensorFlow та PyTorch , також буде матриця ймовірностей, яка визначає ймовірність потрапити в певну вершину графа з певної початкової точки.

Стосовно інших двох етапів, а саме:

- визначення теоретично можливих процесів, які можна розпаралелети
- заміна всіх операцій на тензорні для переходу з режиму eiger до графічного режиму

то вони залишаються без змін і для випадку адаптування алгоритму random walk with restart до практичної реалізації за допомогою такого фреймворку як TensorFlow.

2.3 Теоретична оцінка покращень за рахунок використання фреймворків PyTorch та Tensorflow

В цьому пункті магістерської дисертації буде зроблена теоретична оцінка покращень, які можна отримати, як результат використання фреймворків TensorFlow та PyTorch для реалізації алгоритмів lazy random walk та random walk with restart, які відносяться до класу алгоритмів random walk. В подальшому ця теоретична оцінка буде порівнюватися з фактичними результатами, які будуть отримані шляхом програмної реалізації алгоритмів lazy random walk та random walk with restart за допомогою фреймворків TensorFlow та PyTorch.

2.3.1 Теоретична оцінка покращень за рахунок використання фреймворку Tensorflow

Роздумуючи над темою магістерської дисертації, було розглянуто та вивчено велику кількість інформації, яка стосується алгоритмів lazy random walk та random walk with restart, що відносяться до класу random walk, а також зроблено огляд різноманітних інструментів для практичної реалізації вищезазначених алгоритмів random walk. Результатом опрацювання всієї знайденої інформації, стало прийняття рішення про використання саме фреймворку TensorFlow та фреймворку PyTorch для реалізації алгоритмів lazy random walk та random walk with restart класу random walk.

Фреймворк TensorFlow був обраний для реалізації алгоритмів lazy random walk та random walk with restart адже, з вищенаведеного опису цих алгоритмів класу random walk, можна зробити висновок, що прискорити ці алгоритми можна за рахунок розпаралелення матричних операцій. Проаналізувавши принцип роботи фреймворку TensorFlow, видно, що саме цей фреймворк здатен виконувати всі матричні операції паралельно, адже він побудований на тензорах. Тому від реалізації алгоритмів lazy random walk та random walk with restart, які відносяться до класу random walk, за допомогою фреймворку TensorFlow, теоретично очікується прискорення роботи алгоритмів, яке досягається саме за рахунок паралельного виконання матричних операцій.

2.3.2 Теоретична оцінка покращень за рахунок використання фреймворку PyTorch

Стосовно покращень, які очікуються від програмної реалізації алгоритмів lazy random walk та random walk with restart за допомогою використання фреймворку PyTorch, то, теоретично, вони повторюють очікування від реалізації цих алгоритмів за допомогою використання фреймворку TensorFlow. Це зрозуміло, адже структура

на якій базується фреймворк PyTorch подібна до тієї на якій базується фреймворк TensorFlow. Фреймворк PyTorch, аналогічно до фреймворку TensorFlow, виконує всі матричні операції паралельно, що теоретично повинно прискорити роботу алгоритмів lazy random walk та random walk with restart.

Висновки до розділу

Головна мета використання фреймворків TensorFlow та PyTorch для програмної реалізації алгоритмів lazy random walk та random walk with restart, які належать класу алгоритмів random walk, це прискорення роботи цих алгоритмів за рахунок паралельного виконання матричних операцій. Зрозуміло, що для подібної реалізації, теоретично описані алгоритми, потребували певного адаптування, що і було виконано в рамках цієї магистерської дисертації. Також була проведена теоретична оцінка майбутніх покращень, які можуть бути отриманні після реалізації алгоритмів lazy random walk та random walk with restart за допомогою використання фреймворків TensorFlow та PyTorch. З описаної в цьому розділі інформації, видно, що вищезазначені фреймворки мають багато спільного, тому ще однією метою даної магістерської дисертації є порівняння цих двох фреймворків та виявлення їхніх спільних та відмінних аспектів в ході реалізації, на їхній основі, алгоритмів класу random walk.

3. СПОСОБИ РОЗПАРАЛЕЛЕННЯ АЛГОРИТМІВ RANDOM WALK ЗА ДОПОМОГОЮ ФРЕЙМВОРКІВ TENSORFLOW ТА PYTORCH

В цьому розділі магістерської дисертації будуть запропоновані способи розпаралелення алгоритмів lazy random walk та random walk with restart за допомогою використання фреймворків TensorFlow та PyTorch та буде детально описано програмну реалізацію алгоритмів. Також описані всі вбудовані функції TensorFlow та PyTorch фреймворків, які використовувалися для реалізації вищезазначених алгоритмів.

Основна ідея використання фреймворків TensorFlow та PyTorch полягає в розпаралелені алгоритмів lazy random walk та random walk with restart, проте також важливо було зробити код гнучким настільки, наскільки це можливо. Тому загальна структура коду являє собою сукупність класів, кожен з яких втилює певну локіку. Спільним класом для обох алгоритмів на обох фреймворках є базовий клас Rw(рис. 3.1).

```
class Rw:
  def run_static(self):
    raise Exception("Method 'run' with static graph computation hasn't been implemented.")

  def run_dynamic(self):
    raise Exception("Method 'run' with dynamic graph computation hasn't been implemented.")

  def calculate(self, ones_p, old_p):
    raise Exception("Method 'calculate' hasn't been implemented.")
```

Рисунок 3.1 - Базовий клас для подальшої реалізації алгоритмів на фреймворах Tensorflow та Pytorch

Він містить в собі 3 абстрактних методи, які повинні бути реалізовані в залежності від фреймворку та алгоритму:

1. run_static — метод, який вираховує розподілення ймовірностей, в залежності від вершини, з якої почався random walk і повертає матрицю ймовірностей. Цей метод базується на статичному графі обчислення. Якщо фреймворк не надає можливості реалізувати такий метод, під час його виклику показується помилка про це.
2. run_dynamic — на відміну від попереднього методу, цей базується на динамічному графі обчислення.
3. calculate — метод, який повертає нове розподілення ймовірностей.
   Вхідними параметрами є:
   - ones_p — вектор, в якому елемент, який має такий самий номер, як і той з якого почався обхід, має значення рівне одиниці, всі інші дорівнюють нулю.
   - old_p — вектор, який містить розподілення ймовірностей, що були вирахувані на попередній ітерації.

3.1 Програмна реалізація алгоритмів RWR та LRW

В цьому підрозділі буде покроково описано спільні елементи в реалізації алгоритмів lazy random walk та random walk with restart за допомогою використання фреймворків TensorFlow та PyTorch. Також будуть описані вбудовані функції фреймворків TensorFlow та PyTorch і пояснена ціль використання цих функцій для програмної реалізації алгоритмів lazy random walk та random walk with restart. На рис. 3.2 приведена програмна реалізація спільного класу для алгоритмів lazy random walk та random walk with restart за допомогою використання фреймворку TensorFlow, а на рис. 3.3 приведений код спільного класу релізований за допомогою фреймворку PyTorch. Як видно з цих рисунків, класи RwTensorflow та RwPyTorch в свою чергу наслідується від іншого класу, а саме Rw, про який вже розповідалося вище.

```
class RwTensorflow(Rw):
  def __init__(self, device, weight_matrix, epsilon):
    self.device = device
    with tf.device(self.device):
      self.weight_matrix = tf.constant(weight_matrix, dtype='float64')
      self.epsilon = tf.constant(epsilon, dtype='float64')
      self.shape = tf.shape(self.weight_matrix)[0]
      self.i_matrix = tf.linalg.diag(tf.ones(self.shape, dtype='float64'))
      self.d_matrix = tf.linalg.diag(tf.math.reduce_sum(self.weight_matrix, 1))
      self.normalized_matrix = tf.matmul(tf.linalg.inv(self.d_matrix), self.weight_matrix)
      self.zeros_column = tf.zeros([self.shape, 1], dtype='float64')

  @tf.function
  def run_static(self):
    with tf.device(self.device):
      compute_fn = tf.function(self.compute)
      calculate_fn = tf.function(self.calculate)
      return tf.map_fn(lambda ones_p: compute_fn(ones_p, calculate_fn), self.i_matrix)

  def run_dynamic(self):
    with tf.device(self.device):
      return tf.map_fn(lambda ones_p: self.compute(ones_p, self.calculate), self.i_matrix)

  def compute(self, ones_p, calculate_fn):
    p, old_p = tf.while_loop(lambda p, old_p: tf.math.greater(tf.norm(tf.math.subtract(p, old_p), 1), self.epsilon),
                             lambda p, old_p: (calculate_fn(ones_p, p), p),
                             (tf.reshape(ones_p,[self.shape,1]), self.zeros_column))
    return tf.reshape(p, [self.shape])
```

Рисунок 3.2 - Програмна реалізація спільного класу для алгоритмів lazy random walk та random walk with restart за допомогою використання фреймворку TensorFlow.

```
class RwTorch(Rw):
  def __init__(self, device, weight_matrix, epsilon):
    self.device = 'cuda' if device == 'gpu' else 'cpu'
    self.weight_matrix = torch.tensor(weight_matrix, dtype=torch.float64, device=self.device)
    self.shape = self.weight_matrix.shape[0]
    self.epsilon = torch.tensor(epsilon, dtype=torch.float64, device=self.device)
    self.i_matrix = torch.diag(torch.ones(self.shape, dtype=torch.float64, device=self.device))
    self.d_matrix = torch.diag(torch.sum(self.weight_matrix, 1))
    self.normalized_matrix = torch.matmul(torch.inverse(self.d_matrix), self.weight_matrix)

  def run_dynamic(self):
    return torch.stack([self.compute(ones_p) for ones_p in self.i_matrix])

  def compute(self, ones_p):
    p = torch.reshape(ones_p,[self.shape,1])
    while True:
      old_p = p
      p = self.calculate(ones_p, old_p)
      err = torch.linalg.norm(torch.sub(p, old_p), 1)
      if err < self.epsilon:
        break;
    return torch.reshape(p, [self.shape])
```

Рисунок 3.3 - Програмна реалізація спільного класу з використанням PyTorch

Далі більш детально будуть описані вбудовані функції фреймворків TensorFlow та PyTorch, які використовувались для реалізації вищевказаних спільних класів.

import tensorflow as tf — таким чином фреймворк TensorFlow імпортується в програму і далі може використовуватися за допомогою tf. Наприклад, для того, щоб задати константу потрібно написати наступне: tf.constant.

Аналогом такого імпорту для фреймворку PyTorch є import torch.

import timeit — таким чином в програму імпортується бібліотека timeit, яка використовується для вимірювання часу виконання програмою алгоритму. Ця бібліотека використовується, як для вимірювання часу виконання програми, котра реалізована за допомогою фреймворка TensorFlow, так і для програми, яка реалізована за допомогою фреймворку PyTorch.

tf.constant — використовується для створення тензора-константи із тензорного об’єкту. Аналогом цієї функції для фреймворку PyTorch виступає функція torch.tensor. В програмній реалізації алгоритмів lazy random walk та random walk with restart ці функції використовуються для приведення типів констант.

В загальному tf.constant має наступний вигляд:

tf.constant(value, dtype, shape, name)

де атрибути означають наступне:

- value – це значення константи
- dtype – тип цієї константи; якщо тип не вказано, то за замовчуванням він виводиться з типу value
- shape – задає форму константи. Наприклад, якщо tf.constant має наступну структуру tf.constant([1, 2, 3, 4, 5, 6], shape=[2, 3]), то як результат буде отримано константу, яка являє собою матрицю розміром 2 на 3 (рис. 3.4)

```
array([[1, 2, 3],
       [4, 5, 6]], dtype=int32)
```

Рисунок 3.4 - Кінцева форма константи з атрибутом shape=[2, 3]

- name – задає додаткову назву тензора

А функція torch.tensor в загальному виглядає так:

torch.tensor(data, dtype, requires_grad, pin_memory)

де атрибути означають наступне:

- data – задає вхідні дані, може бути списком, масивом, об`єктом
- dtype – задає тип вихідної матриці
- requires_grad — якщо задано, як autograd то всі операції, які виконувалися над матрицею будуть записані
- pin_memory — якщо задано, як true то вихідна матриця буде записана в кеш

На рис. 3.5 показано приклад використання функції torch.tensor.

```
>>> torch.tensor([[0.1, 1.2], [2.2, 3.1], [4.9, 5.2]])
tensor([[ 0.1000,  1.2000],
        [ 2.2000,  3.1000],
        [ 4.9000,  5.2000]])
```

Рисунок 3.5 - Приклад використання torch.tensor

tf.shape — використовується для отримання одновимірного ціллочисельного тезора, який приймає форму, що задана атрибутом input. В PyTorch функції-аналога немає, адже дане значення дістається з тензора через крапку (як поле об`єкта). В програмній реалізації алгоритмів lazy random walk та random walk with restart використовуеться для отримання розмірності матриць. Приклад роботи функції tf.shape приведено на рис. 3.6.

```
>>> t = tf.constant([[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]])
>>> tf.shape(t)
<tf.Tensor: shape=(3,), dtype=int32, numpy=array([2, 2, 3], dtype=int32)>
```

Рисунок 3.6 - Приклад використання функції tf.shape

В загальному має наступний вигляд:

tf.shape(input, out_type, name)

де атрибути означають наступне:

- input — задає форму тензора. Приймає значення Tensor чи SparseTensor
- out_type — вказує на тип отриманої інформації (необов’язковий)
- name — задає ім’я операції (необов’язковий)

tf.linalg.diag - використовується для отримання діагонального тезора із заданими за допомогою атрибурів значеннями діагоналі. Фреймворк PyTorch також має аналог цієї функції — torch.diag. В програмній реалізації алгоритмів lazy random walk та random walk with restart використовується для отримання діагональної матриці. Приклад роботи функції tf.linalg.diag приведено на рис. 3.7, а приклад роботи функції torch.diag на рис. 3.8.

```
diagonal = np.array([[1, 2, 3, 4],
                     [5, 6, 7, 8]])
tf.matrix_diag(diagonal) ==> [[[1, 0, 0, 0],
                               [0, 2, 0, 0],
                               [0, 0, 3, 0],
                               [0, 0, 0, 4]],
                              [[5, 0, 0, 0],
                               [0, 6, 0, 0],
                               [0, 0, 7, 0],
                               [0, 0, 0, 8]]]
```

Рисунок 3.7 - Приклад використання функції tf.linalg.diag

```
>>> a = torch.randn(3)
>>> a
tensor([ 0.5950,-0.0872, 2.3298])
>>> torch.diag(a)
tensor([[ 0.5950, 0.0000, 0.0000],
        [ 0.0000,-0.0872, 0.0000],
        [ 0.0000, 0.0000, 2.3298]])
>>> torch.diag(a, 1)
tensor([[ 0.0000, 0.5950, 0.0000, 0.0000],
        [ 0.0000, 0.0000,-0.0872, 0.0000],
        [ 0.0000, 0.0000, 0.0000, 2.3298],
        [ 0.0000, 0.0000, 0.0000, 0.0000]])
```

Рисунок 3.8 - Приклад використання функції torch.diag

В загальному функція tf.linalg.diag має наступний вигляд:

tf.linalg.diag(diagonal, name, k, num_rows, num_cols, padding_value, align)

де атрибути означають наступне:

- diagonal — вхідний тензор
- name — задає ім’я операції (необов’язковий)
- k — задає зміщення по діагоналі. Позитивне значення означає наддіагональ, 0 відноситься до головної діагоналі, а від’ємне значення означає субдіагоналі. k може бути одним цілим числом (для однієї діагоналі) або парою цілих чисел, що вказують нижній і верхній кінці смуги матриці.
- num_rows - кількість рядків вихідної матриці. Якщо це не надано, операція припускає, що вихідна матриця є квадратною матрицею.
- num_cols - кількість стовпців вихідної матриці. Якщо це не надано, операція припускає, що вихідна матриця є квадратною матрицею.
- padding_value - значення, яким потрібно заповнити область за межами вказаної діагональної смуги. За замовчуванням 0.
- align - деякі діагоналі коротші, ніж і їх потрібно заповнити. Це атрибут, який визначає, як мають бути вирівняні супердіагоналі та субдіагоналі

відповідно. Є чотири можливі вирівнювання: "RIGHT_LEFT", "LEFT_RIGHT", "LEFT_LEFT" і "RIGHT_RIGHT".

А torch.diag має наступний вигляд:

torch.diag(input, diagonal, out)

де атрибути означають наступне:

- input — задає вхідну матрицю
- out — вказує на тип вихідного тензора
- diagonal — задає тип діагоналі

tf.matmul - використовується для отримання нової матриці, яка є результатом множення матриці A на матрицю B. Аналог цієї функції для фреймворку PyTorch це функція torch.matmul. В програмній реалізації алгоритмів lazy random walk та random walk with restart використовувалася для множення матриць. Приклад роботи функції tf.matmul приведено на рис. 3.9,а приклад роботи функції torch.matmul на рис. 3.10

```
>>> a = tf.constant([1, 2, 3, 4, 5, 6], shape=[2, 3])
>>> a  # 2-D tensor
<tf.Tensor: shape=(2, 3), dtype=int32, numpy=
array([[1, 2, 3],
       [4, 5, 6]], dtype=int32)>
>>> b = tf.constant([7, 8, 9, 10, 11, 12], shape=[3, 2])
>>> b  # 2-D tensor
<tf.Tensor: shape=(3, 2), dtype=int32, numpy=
array([[ 7,  8],
       [ 9, 10],
       [11, 12]], dtype=int32)>
>>> c = tf.matmul(a, b)
>>> c  # `a` * `b`
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[ 58,  64],
       [139, 154]], dtype=int32)>
```

Рисунок 3.9 - Приклад використання функції tf.matmul

```
>>> tensor1 = torch.randn(10, 3, 4)
>>> tensor2 = torch.randn(4, 5)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([10, 3, 5])
```

Рисунок 3.10 - Приклад роботи функції torch.matmul

В загальному tf.linalg.matmul має наступний вигляд:

tf.linalg.matmul(a, b, transpose_a, transpose_b, adjoint_a, adjoint_b, a_is_sparse, b_is_sparse, output_type, name)

де атрибути означають наступне:

- a — матриця, яка виступає першим множником
- b - матриця, яка виступає другим множником
- transpose_a — якщо True, матриця a транспонується перед множенням
- transpose_b — якщо True, матриця b транспонується перед множенням
- adjoint_a — якщо True, a спряжено і транспонується перед множенням.
- adjoint_b — якщо True, b спряжено і транспонується перед множенням.
- a_is_sparse — якщо True, a розглядається як розріджена матриця.
- b_is_sparse — якщо True, b розглядається як розріджена матриця.
- output_type - тип вихідних даних, якщо потрібно. За замовчуванням тип output_type збігається з типом введення
- name - задає ім’я операції (необов’язковий)

Загальний вигляд функції torch.matmul такий:

torch.matmul(input, other, out)

де атрибути означають наступне:

- input — задає першу матрицю
- other — задає другу матрицю
- out — задає ім’я вихідної матриці

tf.zeros - використовується для отримання матриці всі елементи якої нулі. В реалізації за допомогою PyTorch аналогів цієї функції не використовється, адже цикл, в якому tf.zeros потрібна, не реалізується. В програмній реалізації алгоритмів lazy

random walk та random walk with restart використовувалася для обрахунку початкового епсілон. Приклад роботи функції приведено на рис. 3.11.

```
>>> tf.zeros([3, 4], tf.int32)
<tf.Tensor: shape=(3, 4), dtype=int32, numpy=
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int32)>
```

Рисунок 3.11 - Приклад роботи функції tf.zeros

В загальному має наступний вигляд:

tf.zeros(shape, dtype, name)

де атрибути означають наступне:

- shape — задає розмірність матриці
- dtype — задає тип елементів у матриці
- name — задає ім’я операції (необов’язковий)

tf.linalg.inv - використовується для отримання інверсії однієї або декількох матриць. Аналог цієї функції для фреймворку PyTorch являється функція torch.inverse. В програмній реалізації алгоритмів lazy random walk та random walk with restart використовувалася для інверсії діагональної матриці.

В загальному має наступний вигляд:

tf.linalg.inv(input, name)

де атрибути означають наступне:

- input — задає вхідну матрицю
- name — задає ім’я операції (необов’язковий)

Функція torch.inverse виглядає наступним чином:

torch.inverse(input, out)

де атрибути означають наступне:

- input — задає вхідну матрицю
- out — задає ім’я вихідної матриці (необов’язковий)

tf.math.reduce_sum - використовується для отримання суми елементів матриці по горизонталі або вертикалі. Подібні дії в реалізації за допомогою фреймворку PyTorch виконує функція torch.sum. В програмній реалізації алгоритмів lazy random walk та random walk with restart використовувалася для обрахунку діагональної матриці маг. Приклад роботи функції tf.math.reduce_sum приведено на рис. 3.12, а приклад роботи функції torch.sum приведено на рис. 3.13.

```
>>> x = tf.constant([[1, 1, 1], [1, 1, 1]])
>>> x.numpy()
array([[1, 1, 1],
       [1, 1, 1]], dtype=int32)
>>> # sum all the elements
>>> # 1 + 1 + 1 + 1 + 1+ 1 = 6
>>> tf.reduce_sum(x).numpy()
```

Рисунок 3.12 - Приклад роботи функції tf.math.reduce_sum

```
>>> a = torch.randn(4, 4)
>>> a
tensor([[ 0.0569, -0.2475,  0.0737, -0.3429],
        [-0.2993,  0.9138,  0.9337, -1.6864],
        [ 0.1132,  0.7892, -0.1003,  0.5688],
        [ 0.3637, -0.9906, -0.4752, -1.5197]])
>>> torch.sum(a, 1)
tensor([-0.4598, -0.1381,  1.3708, -2.6217])
>>> b = torch.arange(4 * 5 * 6).view(4, 5, 6)
>>> torch.sum(b, (2, 1))
tensor([  435.,  1335.,  2235.,  3135.])
```

Рисунок 3.13 - Приклад використання функції torch.sum

В загальному функція tf.math.reduce_sum має наступний вигляд:

tf.math.reduce_sum(input_tensor, axis, keepdims, name)

де атрибути означають наступне:

- input_tensor — задає початкову матрицю
- axis — задає напрямок
- keepdims — зберігає розмівність матриці
- name — задає ім’я операції (необов’язковий)

Функція torch.sum має такий вигляд:

torch.sum(input, keepdim, dtype)

де атрибути означають наступне:

- input — вхідний тензор
- keepdim — зерігати чи ні вхідний тензор
- dtype — тип елементів результуючого тензора

tf.ones - використовується для отримання матриці всі елементи якої одиниці. Функція torch.ones являється повним аналогом функції tf.ones в фреймворку PyTorch. В програмній реалізації алгоритмів lazy random walk та random walk with restart використовувалася для задання початкового значення матриці ймовірностей. Приклад роботи функції tf.ones приведено на рис. 3.14, а функції torch.ones на рис. 3.15.

```
>>> tf.ones([3, 4], tf.int32)
<tf.Tensor: shape=(3, 4), dtype=int32, numpy=
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]], dtype=int32)>
```

Рисунок 3.14 - Приклад роботи функції tf.ones

```
>>> torch.ones(2, 3)
tensor([[ 1.,  1.,  1.],
        [ 1.,  1.,  1.]])

>>> torch.ones(5)
tensor([ 1.,  1.,  1.,  1.,  1.])
```

Рисунок 3.15 - Приклад використання функції torch.ones

В загальному tf.ones має наступний вигляд:

tf.ones(shape, dtype, name)

де атрибути означають наступне:

- shape — задає розмірність матриці
- dtype — задає тип елементів матриці
- name — задає ім’я операції (необов’язковий)

Загальний вигляд torch.ones:

torch.ones(size, out, dtype, layout, requires_grad)

де атрибути означають наступне:

- size— послідовність цілих чисел, що визначають форму вихідного тензора
- out — вихідний тензор
- dtype — тип елементів вихідного тензора
- layout — бажаний тип вихідного тензора
- requires_grad - якщо autograd має записувати операції над повернутим тензором

tf.vectorized_map - використовується для виконання певних операцій над кожним елементов матриці. В програмній реалізації алгоритмів lazy random walk та

random walk with restart використовувалася для обрахунку діагональної матриці. Приклад роботи функції приведено на рис. 3.16.

```
def outer_product(a):
  return tf.tensordot(a, a, 0)

batch_size = 100
a = tf.ones((batch_size, 32, 32))
c = tf.vectorized_map(outer_product, a)
assert c.shape == (batch_size, 32, 32, 32, 32)
```

Рисунок 3.16 - Приклад роботи функції tf. vectorized_map

В загальному має наступний вигляд:

tf.vectorized_map(fn, elems, fallback_to_while_loop)

де атрибути означають наступне:

- fn — задає функцію, яка виконується над кожним елементом матриці
- elems — задає саму матрицю
- fallback_to_while_loop — якщо приймає значення true, то це означає, що при неможливості виконати векторизацію буде виконуватися while_loop.

Аналогів такої функції у фреймворка PyTorch намає, проте для реалізації подібного було використано функцію torch.stack.

torch.stack — об`єднує рядки в матрицю.

В загальному має наступний вигляд:

torch.stack(tensors, dim, out)

де атрибути означають наступне:

- tensors — задає послідовність рядків, які будуть об`єднані в матрицю
- dim — задає розмір вихідної матриці

- out — задає ім`я вихідної матриці

tf.while_loop — реалізує цикл. У фреймворку PyTorch саме функції-аналогу немає, проте такий самий цикл реалізується за допомогою структури while.

В загальному tf.while_loop має наступний вигляд:

tf.while_loop(cond, body, parallel_iterations, swap_memory, maximum_iterations, name)

де атрибути означають наступне:

- cond — умова завершення циклу
- body — тіло, яке виконується в циклі
- parallel_iterations — задає кількість ітерацій, які дозволені для паралельного виконання
- swap_memory — включає кеш
- maximum_iterations — визначає максимальну кількість ітерацій
- name — задає ім’я операції (необов’язковий)

tf.math.greater — повертає істинність для кожного елементу матриці. В програмній реалізації алгоритмів lazy random walk та random walk with restart використовувалася для виявлення моменту завершення роботи алгоритмів. Приклад використання функції tf.math.greater зображено на рис. 3.17.

```
x = tf.constant([5, 4, 6])
y = tf.constant([5, 2, 5])
tf.math.greater(x, y) ==> [False, True, True]

x = tf.constant([5, 4, 6])
y = tf.constant([5])
tf.math.greater(x, y) ==> [False, False, True]
```

Рисунок 3.17 - Приклад використання tf.math.greater

В загальному має наступний вигляд:

tf.math.greater(x, y, name)

де атрибути означають наступне:

- x — перша матриця елементи якої будуть порівнюватися
- y — друга матриця елементи якої будуть порівнюватися
- name — задає ім’я операції (необов’язковий)

tf.math.subtract - використовується для віднімання двух матриць поелементно. У фреймворку PyTorch ця функція заміщується функцією torch.sub. Приклад роботи функції tf.math.subtract приведено на рис. 3.18, а функції torch.sub на рис. 3.19.

```
>>> x = [1, 2, 3, 4, 5]
>>> y = 1
>>> tf.subtract(x, y)
<tf.Tensor: shape=(5,), dtype=int32, numpy=array([0, 1, 2, 3, 4], dtype=int32)>
>>> tf.subtract(y, x)
<tf.Tensor: shape=(5,), dtype=int32,
numpy=array([ 0, -1, -2, -3, -4], dtype=int32)>
```

Рисунок 3.18 - Приклад роботи tf.math.subtract

```
>>> a = torch.tensor((1, 2))
>>> b = torch.tensor((0, 1))
>>> torch.sub(a, b, alpha=2)
tensor([1, 0])
```

Рисунок 3.19 - Приклад роботи функції torch.sub

В загальному tf.math.subtract має наступний вигляд:

tf.math.subtract(x, y, name)

де атрибути означають наступне:

- x — перша матриця елементи якої будуть відніматися
- y — друга матриця елементи якої будуть відніматися
- name — задає ім’я операції (необов’язковий)

А загальний вигляд torch.sub такий:

torch.sub(input, other, alpha, out)

де атрибути означають наступне:

- input — задає першу вхідну матрицю
- other — задає другу вхідну матрицю
- alpha — задає множник для матриці other
- out — задає ім’я вихідної матриці

tf.reshape - використовується для зміни форми матриці. Аналогічну функцію має фреймворк PyTorch, це функція torch.reshape. В програмній реалізації алгоритмів lazy random walk та random walk with restart використовувалася для перетворення рядків одиничної матриці на стовпці. Приклад роботи функції tf.reshape приведено на рис. 3.20, а функції torch.reshape на рис. 3.21.

```
>>> t1 = [[1, 2, 3],
...       [4, 5, 6]]
>>> print(tf.shape(t1).numpy())
[2 3]
>>> t2 = tf.reshape(t1, [6])
>>> t2
<tf.Tensor: shape=(6,), dtype=int32,
  numpy=array([1, 2, 3, 4, 5, 6], dtype=int32)>
>>> tf.reshape(t2, [3, 2])
<tf.Tensor: shape=(3, 2), dtype=int32, numpy=
  array([[1, 2],
         [3, 4],
         [5, 6]], dtype=int32)>
```

Рисунок 3.20 - Приклад роботи tf.reshape

```
>>> a = torch.arange(4.)
>>> torch.reshape(a, (2, 2))
tensor([[ 0.,  1.],
        [ 2.,  3.]])
>>> b = torch.tensor([[0, 1], [2, 3]])
>>> torch.reshape(b, (-1,))
tensor([ 0,  1,  2,  3])
```

Рисунок 3.21 - Приклад роботи функції torch.reshape

В загальному tf.reshape має наступний вигляд:

tf.reshape(tensor, shape, name)

де атрибути означають наступне:

- tensor — вхідна матриця
- shape — задає форму вихідної матриці
- name — задає ім’я операції (необов’язковий)

Функція torch.reshape має такий вигляд, як:

torch.reshape(input, shape)

де атрибути означають наступне:

- input — задає вхідну матрицю
- shape — задає форму вихідної матриці

tf.add - використовується для поелементного додавання двух матриць. В програмній реалізації алгоритмів lazy random walk та random walk with restart використовувалася для додавання тих матриць, сумування яких передвачено у формулах алгоритмів. Приклад роботи функції tf.add приведено на рис. 3.22, а на рис. 3.23 приведено приклад роботи функції torch.add.

```
>>> x = [1, 2, 3, 4, 5]
>>> y = tf.constant([1, 2, 3, 4, 5])
>>> tf.add(x, y)
<tf.Tensor: shape=(5,), dtype=int32,
numpy=array([ 2,  4,  6,  8, 10], dtype=int32)>
```

Рисунок 3.22 - Приклад роботи tf.add

```
>>> a = torch.randn(4)
>>> a
tensor([ 0.0202,  1.0985,  1.3506, -0.6056])
>>> torch.add(a, 20)
tensor([ 20.0202,  21.0985,  21.3506,  19.3944])
```

Рисунок 3.23 - Приклад роботи функції torch.add

В загальному tf.add має наступний вигляд:

tf.add(x, y, name)

де атрибути означають наступне:

- x — перша матриця елементи якої будуть сумуватися
- y — друга матриця елементи якої будуть сумуватися
- name — задає ім’я операції (необов’язковий)

В загальному torch.add має наступний вигляд:

torch.add(input, other, alpha, out)

де атрибути означають наступне:

- input — задає першу вхідну матрицю
- other — задає другу вхідну матрицю
- alpha — задає множник для матриці other
- out — задає ім’я вихідної матриці

tf.divide - використовується для поелементного ділення двух матриць. Аналогічний функціонал є і у фреймворка PyTorch. Реалізований він за допомогою функції torch.div. В програмній реалізації алгоритмів lazy random walk та random walk with restart використовувалася для ділення тих матриць, які передвачено у формулах алгоритмів. Приклад роботи функції tf.divide приведено на рис. 3.24, а приклад роботи функції torch.div приведено на рис. 3.25.

```
>>> x = tf.constant([16, 12, 11])
>>> y = tf.constant([4, 6, 2])
>>> tf.divide(x,y)
<tf.Tensor: shape=(3,), dtype=float64,
numpy=array([4. , 2. , 5.5])>
```

Рисунок 3.24 - Приклад роботи tf.divide

```
>>> a = torch.tensor([[-0.3711, -1.9353, -0.4605, -0.2917],
...                   [ 0.1815, -1.0111,  0.9805, -1.5923],
...                   [ 0.1062,  1.4581,  0.7759, -1.2344],
...                   [-0.1830, -0.0313,  1.1908, -1.4757]])
>>> b = torch.tensor([ 0.8032,  0.2930, -0.8113, -0.2308])
>>> torch.div(a, b)
tensor([[-0.4620, -6.6051,  0.5676,  1.2639],
        [ 0.2260, -3.4509, -1.2086,  6.8990],
        [ 0.1322,  4.9764, -0.9564,  5.3484],
        [-0.2278, -0.1068, -1.4678,  6.3938]])
```

Рисунок 3.25 - Приклад роботи функції torch.div

В загальному tf.divide має наступний вигляд:

tf.divide(x, y, name)

де атрибути означають наступне:

- x — перша матриця елементи якої будуть поділені

- y — друга матриця елементи якої будуть слугувати дільниками
- name — задає ім’я операції (необов’язковий)

В загальному torch.div має наступний вигляд:

torch.div(input, other, rounding_mode, out)

де атрибути означають наступне:

- input — задає першу вхідну матрицю
- other — задає другу вхідну матрицю
- rounding_mode — задає тип округлення
- out — задає ім’я вихідної матриці

tf.norm - використовується для приведення матриці чи вектора до норми. Аналогом цієї функції у фреймворку PyTorch є функція torch.norm. В програмній реалізації алгоритмів lazy random walk та random walk with restart використовувалася для приведення вектора до скалярного вигляду. Приклад використання функції torch.norm приведений на рис. 3.26.

```
>>> a = torch.arange(9, dtype= torch.float) - 4
>>> b = a.reshape((3, 3))
>>> torch.norm(a)
tensor(7.7460)
>>> torch.norm(b)
tensor(7.7460)
```

Рисунок 3.26 - Приклад роботи функції torch.norm

В загальному tf.norm має наступний вигляд:

tf.norm(tensor, ord, name)

де атрибути означають наступне:

- tensor — вхідна матриця

- ord — порядок норми
- name — задає ім’я операції (необов’язковий)

В загальному torch.norm має наступний вигляд:

torch.norm(input, p, keepdim, out, dtype)

де атрибути означають наступне:

- input — задає вхідну матрицю
- p — задає порядок норми
- keepdim — зберігати чи ні вхідну матрицю
- out — задає ім’я вихідної матриці
- dtype — задає тип елементів вихідної матриці

Розглянемо більш детально спільний батьківський клас RwTensorflow, який реалізовано за допомогою фреймворку Tensorflow та спільний батьківський клас RwPytorch, який реалізовано за допомогою фреймворку PyTorch. Конструктор цих класів приймає спільні параметри для цих алгоритмів, а саме:

- device — назва девайсу на якому будуть знаходитися тензори, і виконуватися обчислення над ними.
- weight_matrix — квадратна матриця ваг.
- epsilon — різниця між двома векторами ймовірностей,при якій алгоритм зупиниться.

Так, як Tensorflow надає користувачу можливість обирати девайс на якому будуть виконуватися обчислення, то щоб скористатися цією можливістю було використано обгортку, яка зображена на рис. 3.27.

```
with tf.device(self.device):
```

Рисунок 3.27 - Обгортка для вибору девайсу на якому будуть виконуватися обчислення

PyTorch автоматично не копіює створені тензори між девайсами, тобто немає необхідності в обгортці тензорних операцій, як це зроблено у реалізації TensorFlow. Достатньо лише вказати девайс при ініціалізації тензора і всі подальші операції з цим тензором будуть виконуватися на вказаному девайсі.

Перш за все, так як фреймворки TensorFlow та PyTorch працюють з тензорами, то першочерговим завданням є переведення зі Python структури даних в тензорне представлення. Це досягається за рахунок використання функції tf.constant (рис. 3.28) для фреймворку TensorFlow, та функції torch.tensor (рис. 3.29) для фреймворку PyTorch.

```
self.weight_matrix = tf.constant(weight_matrix, dtype='float64')
self.epsilon = tf.constant(epsilon, dtype='float64')
```

Рисунок 3.28 - Переведення Python констант у тензорне представлення при реалізації на фреймворку TensorFlow

```
self.weight_matrix = torch.tensor(weight_matrix, dtype=torch.float64, device=self.device)
self.epsilon = torch.tensor(epsilon, dtype=torch.float64, device=self.device)
```

Рисунок 3.29 - Переведення Python констант у тензорне представлення при реалізації на фреймворку PyTorch

Для подальшого обрахунку складових формул алгоритмів lazy random walk та random walk with restart необхідно знати розмірність матриці(тобто кількість вершин в графі). Це значення записується в поле класу shape (рис. 3.30 та рис. 3.31).

```
self.shape = tf.shape(self.weight_matrix)[0]
```

Рисунок 3.30 - Знаходження кількості вершин графа в реалізації за допомогою фреймворку TensorFlow

```
self.shape = self.weight_matrix.shape[0]
```

Рисунок 3.31 - Знаходження кількості вершин графа в реалізації за допомогою фреймворку PyTorch

Аналогічно додатковим елементом для обрахунку складових формул вищезазначених алгоритмів, являється одинична діагональна матриця i_matrix, яка була отримана наступним чином при реалізації на фреймворку TensorFlow(рис. 3.32) та таким, як показано на рис. 3.33 при реалізації на фреймворку PyTorch.

```
self.i_matrix = tf.linalg.diag(tf.ones(self.shape, dtype='float64'))
```

Рисунок 3.32 - Обрахунок одиничної діагональної матриці на TensorFlow

```
self.i_matrix = torch.diag(torch.ones(self.shape, dtype=torch.float64, device=self.device))
```

Рисунок 3.33 - Обрахунок одиничної діагональної матриці на PyTorch

Також таким елементом є вектор-стовпець, розмірність якого дорівнює кількості вершин графу, а кожен елемент дорівнює нулю. Отримується вона з використанням функції tf.zeros(рис. 3.34) і використовується лише в реалізації на фреймворку TensorFlow.

```
self.zeros_column = tf.zeros([self.shape, 1], dtype='float64')
```

Рисунок 3.34 - Створення нульового вектора-стовпця

Наступним кроком, як при реалізації за допомогою фреймворка TensorFlow, так і при реалізації за допомогою фреймворка PyTorch, є нормалізація вхідної матриці ваг, тобто переведення вхідної матриці до матриці ймовірностей. Це було реалізовано за рахунок використання сукупності вбудованих функції фреймворку Tensorflow (рис. 3.35) та PyTorch (рис. 3.36), які були описані раніше, а саме tf.matmul та

tf.linalg.inv для реалізації за допомогою TensorFlow, та функцій torch.matmul та torch.inverse для реалізації за допомогою PyTorch. А також, як видно, використовувалися такі змінні як: d_matrix та weight_matrix.

```
self.normalized_matrix = tf.matmul(tf.linalg.inv(self.d_matrix), self.weight_matrix)
```

Рисунок 3.35 - Нормалізація матриці ваг для TensorFlow

```
self.normalized_matrix = torch.matmul(torch.inverse(self.d_matrix), self.weight_matrix)
```

Рисунок 3.36 - Нормалізація матриці ваг для PyTorch

де, змінна weight_matrix вже була описана вище, а d_matrix — це діагональна матриця, кожен елемент якої дорівнює сумі ваг ребер, які виходять з відповідної вершини, тобто, для нульової вершини відповідниї елемент буде знаходиться на нульовому рядку та нульовому стовпчику. Дана матриця, вираховується за допомогою функцій tf.linalg.diag і tf.math.reduce_sum для реалізації за допомогою TensorFlow (рис. 3.37) та функцій torch.diag і torch.sum для реалізації за допомогою PyTorch (рис. 3.38).

```
self.d_matrix = tf.linalg.diag(tf.math.reduce_sum(self.weight_matrix, 1))
```

Рисунок 3.37 - Обчислення діагональної матриці ваг на TensorFlow

```
self.d_matrix = torch.diag(torch.sum(self.weight_matrix, 1))
```

Рисунок 3.38 - Обчислення діагональної матриці ваг на PyTorch

Так як фреймворк Tensorflow підтримує використання як статичного так і динамічного графа обчислень, то базовий клас реалізує обидва методи run_static та run_dynamic. Розглянемо реалізацію кожного з них більш детально. На рис. 3.39 зображено реалізований метод run_static.

```
@tf.function
def run_static(self):
  with tf.device(self.device):
    return tf.vectorized_map(self.compute, self.i_matrix)
```

Рисунок 3.39 - Реалізація методу run_static

Перш за все, звернемо увагу на анотацію @tf.function, саме вона створює статичний обчислювальний граф. Аналогами анотації @tf.function, є функція tf.function, яка перетворює Python функції у такий граф. Більш детально про них буде описано далі.

Аналогічно, як і тіло конструктору, тіло даного методу було обернуто у функцію tf.device для вказаня девайсу для роботи. Основний цикл програми виконується за допомогою функції tf.vectorized_map, яка проходиться по кожному рядку матриці i_matrix, яка була описана вище. На рис. 3.40 зображено реалізацію методу run_dynamic.

```
def run_dynamic(self):
  with tf.device(self.device):
    return tf.vectorized_map(self.compute, self.i_matrix)
```

Рисунок 3.40 - Реалізація методу run_ dynamic

Реалізація даного методу дещо схожа, на ту, що була описана вище, за виключенням, того що він не анотується @tf.function.

Обидва методи run_static та run_dynamic використовують метод compute, який приймає наступні аргументи:

- ones_p — вектор, де елемент з якого починається обхід дорівнює одиниці, а всі інші нулю.
- calculate_fn — аналог методу calculate, який був описаний вище, за виключенням того, что він може бути представлений у вигляді статичного графу(у run_static методі), або самого методу(у run_dynamic методі).

На відміну від TensorFlow, на PyTorch може бути реалізовано лише метод run_dynamic (рис. 3.41).

```
def run_dynamic(self):
  return torch.stack([self.compute(ones_p) for ones_p in self.i_matrix])
```

Рисунок 3.41 - Реалізація методу run_ dynamic

Як видно з рисунків вище, кожен метод, як на TensorFlow так і на PyTorch, використовують метод compute. На рис. 3.42 зображено реалізацію методу compute на TensorFlow.

```
def compute(self, ones_p, calculate_fn):
  p, old_p = tf.while_loop(lambda p, old_p: tf.math.greater(tf.norm(tf.math.subtract(p, old_p), 1), self.epsilon),
                           lambda p, old_p: (calculate_fn(ones_p, p), p),
                           (tf.reshape(ones_p,[self.shape,1]), self.zeros_column))
  return tf.reshape(p, [self.shape])
```

Рисунок 3.42 - Реалізація методу compute на TensorFlow

Як видно в його основу покладений цикл, який реалізований за допомогою tf.while_loop. Ітеративне обчислення розподілення ймовірностей буде відбуватися до тих пір, поки різниця між новим обчисленим вектором і попереднім не стане меншим за значення epsilon. Так як зовнішня ітерація відбувається по рядкам одиничної матриці, для коректного виконання матричних операцій вектор-рядок перетворюється у вектор-стовпець за допомогою функції tf.reshape. Наприкінці даного методу відбувається зворотнє перетворення.

Аналогічний метод, який реалізується за допомогою фреймворку PyTorch, зображено на рис. 3.43. Логіка реалізації цього метода на PyTorch повторює логіку реалізації на TensorFlow. Різниця полягає лише у використанні умовної структури while замість tf.while_loop.

```
def compute(self, ones_p):
  p = torch.reshape(ones_p,[self.shape,1])
  while True:
    old_p = p
    p = self.calculate(ones_p, old_p)
    err = torch.linalg.norm(torch.sub(p, old_p), 1)
    if err < self.epsilon:
      break;
  return torch.reshape(p, [self.shape])
```

Рисунок 3.43 - Реалізація методу compute на PyTorch

3.1.1 Особливості реалізації алгоритму RWR за допомогою фреймворків TensorFlow та PyTorch

Розглянемо особливості реалізації алгоритму random walk with restart. Як видно з рис. 3.44 клас RwrTensorflow наслідує базовий функціонал, який реалізовується класом RwTensorflow з деякими відмінностями.

```
class RwrTensorflow(RwTensorflow):
  def __init__(self, device, weight_matrix, restart_probability, epsilon):
    super().__init__(device, weight_matrix, epsilon)
    with tf.device(self.device):
      self.restart_probability = tf.constant(restart_probability, dtype='float64')
      self.inversed_restart_probability = tf.subtract(1, self.restart_probability)
      self.normalized_matrix = tf.transpose(self.normalized_matrix)

  def calculate(self, ones_p, old_p):
    return tf.add(tf.reshape(tf.multiply(self.restart_probability, ones_p),[self.shape,1]),
                  tf.matmul(tf.multiply(self.inversed_restart_probability, self.normalized_matrix), old_p))
```

Рисунок 3.44 - Реалізація класу RwrTensorflow

В реалізації за допомогою фреймворка PyTorch, клас RwrTorch аналогічним чином наслідує базовий клас RwTorch (рис. 3.45).

```
class RwrTorch(RwTorch):
  def __init__(self, device, weight_matrix, restart_probability, epsilon):
    super().__init__(device, weight_matrix, epsilon)
    self.restart_probability = torch.tensor(restart_probability, dtype=torch.float64, device=self.device)
    self.inversed_restart_probability = torch.sub(1, self.restart_probability)
    self.normalized_matrix = torch.transpose(self.normalized_matrix,0,1)

  def calculate(self, ones_p, old_p):
    return torch.add(torch.reshape(torch.mul(self.restart_probability, ones_p),[self.shape,1]),
                     torch.matmul(torch.mul(self.inversed_restart_probability, self.normalized_matrix), old_p))
```

Рисунок 3.45 - Реалізація класу RwrTorch

Перша відмінність, пов’язана з додатковим параметром, який передається у конструктор цих класів, а саме restart_probability, який відповідає ймовірності переходу на початок обходу. Також, для ітеративного обчислення вектору ймовірностей, нормалізована матриця повинна бути транспонована.

Друга і основна відмінність, полягає саме у реалізації методу calculate. В цьому методі всі отримані в попередньому класі замінні поєднуються згідно до формули алгоритму random walk with restart, яка була згадана в попередніх розділах магістерської дисертації.

3.1.2 Особливості реалізації алгоритму LRW за допомогою фреймворків TensorFlow та PyTorch

Розглянемо особливості реалізації алгоритму lazy random walk. Як видно з рис. 3.46 клас LrwTensorflow також успадковує базовий функціонал, який реалізовується класом RwTensorflow з деякими відмінностями.

```
class LrwTensorflow(RwTensorflow):
  def __init__(self, device, weight_matrix, epsilon):
    super().__init__(device, weight_matrix, epsilon)
    with tf.device(self.device):
      self.normalized_matrix = tf.transpose(tf.divide(tf.add(self.i_matrix, self.normalized_matrix), 2))

  def calculate(self, ones_p, old_p):
      return tf.matmul(self.normalized_matrix, old_p)
```

Рисунок 3.46 - Реалізація класу LrwTensorflow

В реалізації за допомогою фреймворка PyTorch, клас LrwTorch аналогічним чином наслідує базовий клас RwTorch (рис. 3.47).

```
class LrwTorch(RwTorch):
  def __init__(self, device, weight_matrix, epsilon):
    super().__init__(device, weight_matrix, epsilon)
    self.normalized_matrix = torch.transpose(torch.div(torch.add(self.i_matrix, self.normalized_matrix), 2), 0, 1)

  def calculate(self, ones_p, old_p):
    return torch.matmul(self.normalized_matrix, old_p)
```

Рисунок 3.47 - Реалізація класу LrwTorch

Перша відмінність полягає у обрахунку нормалізованої матриці ймовірностей. Нормалізація була реалізована згідно до формули (3.1):

$$W = \frac{1}{2}(I + D^{-1} * A) \tag{3.1}$$

Друга і основна відмінність, полягає саме у реалізації методу calculate. В цьому методі всі отримані в попередньому класі замінні поєднуються згідно до формули алгоритму lazy random walk, яка була згадана в попередніх розділах магістерської дисертації.

3.2 Спосіб розпаралелення алгоритмів LRW та RWR

Основним методом для прискорення роботи алгоритмів lazy random walk та random walk with restart, які відносяться до класу random walk, виступає розпаралелення. В цьому підпункті буде детально описано спільні і відмінні аспекти реалізації розпаралелювання на фреймворках PyTorch та TensorFlow.

3.2.1 Спосіб розпаралелення алгоритмів LRW та RWR за допомогою використання фреймворку TensorFlow

Спосіб розпаралелення алгоритмів LRW та RWR за допомогою використання фреймворку TensorFlow полягає у виконанні наступних трьох кроків:

- внутрішнє розпаралелювання операцій над матрицями,
- динамічний та статичний режими роботи фреймворка
- використання функції tf.vectorized_map

Внутрішнє розпаралелювання операцій над матрицями — фреймворк TensorFlow являється фреймворком для паралельного обчислення саме за рахунок того, що має вбудоване внутрішнє розпаралелювання всіх операцій лінійної алгебри для матриць (додавання, віднімання, множення, ділення та інші). Всі ці операції виконуються паралельно на будь-яких процесорах: CPU, GPU, TPU. Проте реалізація цієї паралельності для вищевказаних процесорів досягається по-різному. Так, наприклад, для CPU при виконанні операцій над матрицями використовуються всі вільні ядра, а також віртуальні потоки. Приймаючи до уваги, що всі без винятку CPU, на сьогоднішній день, мають мінімум два обчислювальні ядра, що можуть бути використанні для виконання одночасних і незалежних команд, то стає зрозумілим, що таке вбудоване розпаралелювання дає позитивні результати при вирішені проблеми прискорення роботи алгоритмів. Розглянемо більш детально, як саме працює вбудоване розпаралелювання фреймворка TensorFlow. Будь-яка операція, яка надходить, поділяється на ще менші елементарні операції: порівняння, перезапису і т.д. Далі всі ці операції формують чергу і виконуються паралельно вільними ядрами.

Для таких процесорів, як GPU виконання подібних операцій паралелеться на кількість обчислювальних блоків. Основою паралельних обчислень на GPU процесорах є технологія CUDA. Реалізація вбудованого внутрішнього розпаралелювання має декілька етапів:

- виділяння пам’яті та занесення у неї всіх потрібних даних

- виконання ядром графічного процесора команд
- перенесення у пам’ять графічного процесора отриманого результату

Фреймворк TensorFlow автоматично обирає процесор GPU, якщо такий процесор є в наявності.

Якщо розглядати реалізацію вбудованого розпаралелювання на процесорах TPU, то вона відрізняється від реалізації на CPU та GPU. Різниця полягає в тому, що операції виконуються не з використанням регістрів, а використовується систолічна матриця множників(рис. 3.48).

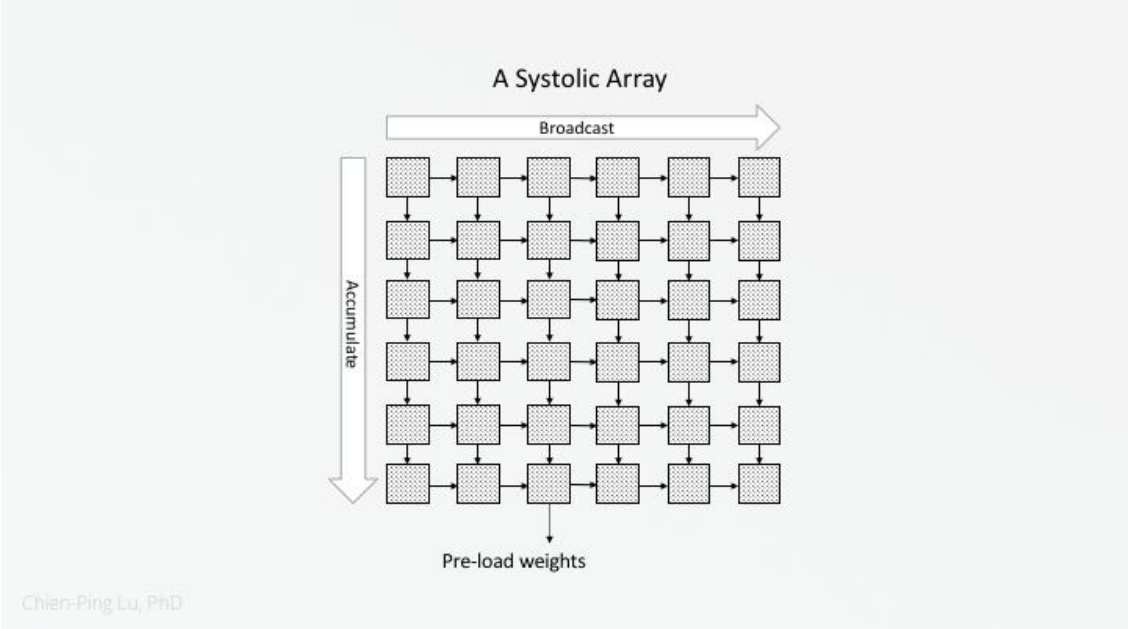

Рисунок 3.48 - Приклад систолічної матриці.

Наприклад на рис. 3.49 показано принцип виконання операції множення на систолічній матриці.

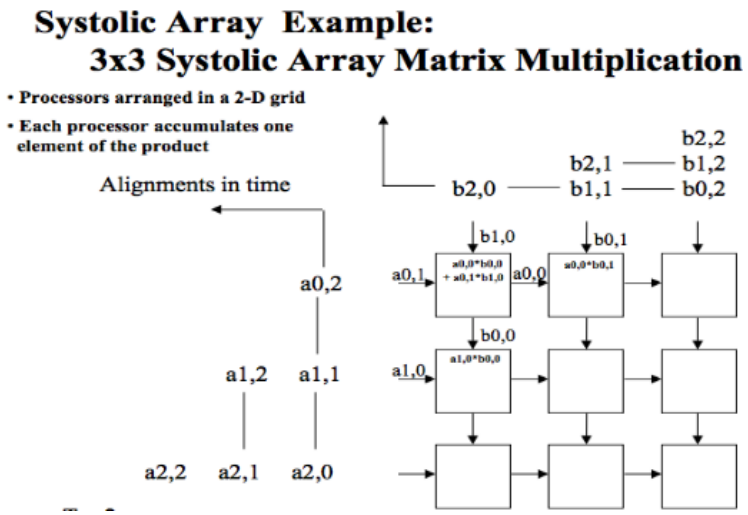

Рисунок 3.49 - Принцип виконання множення на систолічній матриці.

Головною перевагою використання таких матриць є те, що дані операндів та проміжні результати заносяться в масив процесора. Таким чином зникає необхідність звернення до основної пам'яті або внутрішнього кеша.

Важливо також зазначити, що незважаючи на те, що фреймворки самі автоматично визначають кількість потоків, які будуть виконувати задані операції, користувач може корегувати цей параметр. Так, наприклад, для фреймворку TensorFlow, такі корективи можна реалізувати використовуючи функції:

- tf.config.threading.set_intra_op_parallelism_threads(num_threads)
- tf.config.threading.set_inter_op_parallelism_threads(num_threads)

де, num_threads задає кількість потоків.

tf.config.threading.set_intra_op_parallelism_threads — визначає кількість потоків для виконання певної окремої операції. Наприклад, множення, додавання, віднімання матриць. Якщо num_threads рівне нулю, то кількість потоків залишається такою, якою обрала система.

tf.config.threading.set_inter_op_parallelism_threads — визначає кількість потоків що використовуються для паралельності між незалежними неблокуючими операціями. Аналогічно, як і для tf.config.threading.set_intra_op_parallelism_threads,

якщо значення num_threads рівне нулю, то кількість потоків залишається такою, якою обрала система.

Динамічний та статичний режими роботи — ці режими роботи базуються на тому, які обчислювальні графи використовуються. Обчислювальні графи — це абстракція, яка описує обчислення у вигляді орієнтованого ациклічного графа. Наприклад, вираз y = wx+b можна розгорнути у граф (рис. 3.50). Основною особливістю такого уявлення є паралелізм, коли операції виконуються одночасно, що прискорює роботу.

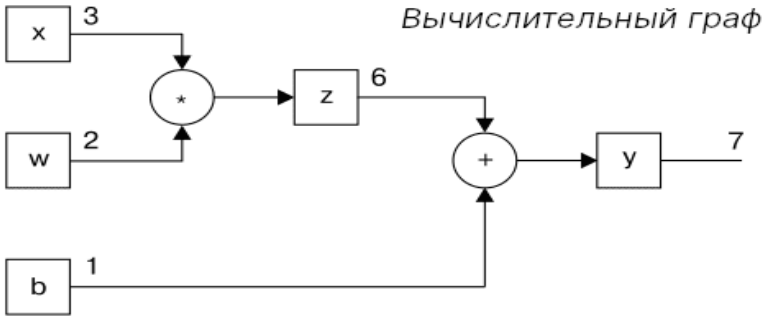

Рисунок 3.50 - Обчислювальний граф виразу y = wx+b

Ці обчислювальні графи є двох видів: динамічні та статичні. Фреймворк TensorFlow може використовувати обчислювальні графи обох видів. Розберемо в чому різниця цих обчислювальних графів та як вони реалізують розпаралелювання.

Динамічний обчислювальний граф не вимагає компіляції перед кожним виконанням. Тому можна спокійно змінювати вхідні дані у процесі роботи.

Статичний обчислювальний граф вимагає перекомпіляції за зміни вхідних параметрів. Однак дозволяє отримати високу продуктивність, коли використовується як кінцевий продукт.

Процес розпаралелення в обох випадках виглядає однаково,із отриманого на вхід коду будується граф. Потім проводиться певний аналіз і виявляються фрагменти коду, які підлягають розпаралеленню.

В програмній реалізації алгоритмів lazy random walk та random walk with restart було використано як динамічний режим роботи так и статичний режим роботи (для фреймворку TensorFlow (рис. 3.51)).

```
@tf.function
def run_static(self):
  compute_fn = tf.function(self.compute)
  calculate_fn = tf.function(self.calculate)
  return tf.map_fn(lambda ones_p: compute_fn(ones_p, calculate_fn), self.i_matrix)

def run_dynamic(self):
  return tf.map_fn(lambda ones_p: self.compute(ones_p, self.calculate), self.i_matrix)
```

Рисунок 3.51 - Використання динамічного та статичного режимів роботи для фреймворка TensorFlow

Різниця між використанням динамічного та статичного обчислювального графа зі сторони реалізації на фреймворку TensorFlow очевидна, вона полягає у використанні функції tf.function.

tf.function — створює статичний обчислювальний граф з певного коду. Також ця функція може використовуватися у іншій інтерпретації, а саме @tf.function. Різниця полягає в тому, що @tf.function зазначається перед створенням функції і інформує про те, що при компіляції ця функція буде перетворена у граф, тоді як tf.function використовується в момент виклику функції, але інформує про те сама, що і @tf.function.

Функція tf.vectorized_map — це ще одна складова при програмній реалізації розпаралелювання алгоритмів lazy random walk та random walk with restart. Проте вона може бути використана лише при реалізації цих алгоритмів за допомогою фреймворку TensorFlow (рис. 3.52).

```
return tf.vectorized_map(self.compute, self.i_matrix)
```

Рисунок 3.52 - Використання функції tf.vectorized_map для реалізації алгоритмів класу random walk за допомогою фреймворку TensorFlow

Використання цієї функції обумовлено тим, що на відміну від двох попередній складових розпаралелювання, реалізує не внутрішнє розпаралелювання, а зовнішнє. Тобто, тоді, як дві попередні складові паралелять операції, які виконуються над матрицями, за допомогою tf.vectorized_map паралелеться сама матриця, а саме одночасно виконуються дії над кожним рядком цієї матриці.

Загальний вигляд цієї фунції наступний:

tf.vectorized_map(fn, elems, fallback_to_while_loop)

де, атрибути мають наступні значення:

- fn — функція, яка буде виконуватися
- elems — матриця над елементами якої буде виконуватися fn
- fallback_to_while_loop — якщо приймає значення true, то це означає, що при неможливості виконати векторизацію буде виконуватися while_loop.

3.2.2 Спосіб розпаралелення алгоритмів LRW та RWR за допомогою використання фреймворку PyTorch

На відміну від TensorFlow, спосіб розпаралелення алгоритмів LRW та RWR за допомогою використання фреймворку PyTorch полягає у виконанні лише двух кроків:

- внутрішнє розпаралелювання операцій над матрицями,
- динамічний режим роботи фреймворка

адже аналога функції tf.vectorized_map фреймворк PyTorch немає.

. Важливо зазначити, що саме така складова, як внутрішнє розпаралелювання операцій над матрицями є спільною для фреймворків PyTorch та TensorFlow. Єдиною відмінністю є те, що фреймворк PyTorch потребує додаткових маніпуляцій зі сторони користувача, щоб обрати процесор GPU, а саме завдання девайсу при ініціалізації тензора за допомогою атрибуту device. В усьому іншому реалізації внутрішнього розпаралелення алгоритмів для фреймворків PyTorch і TensorFlow збігаються.

Динамічний режим роботи фреймворка — фреймворк PyTorch має в своєму арсеналі лише динамічний режим роботи (рис. 3.53), чим і відрізняється від фреймворка TensorFlow, який має, як динамічний, так і статичний режими роботи. Реалізація роботи цього режиму аналогічна тій, яка описана вище для фреймворка TensorFlow.

```
def run_dynamic(self):
  return list(map(self.compute, self.i_matrix))
```

Рисунок 3.53 - Динамічний режим роботи для фреймворка PyTorch

Висновки до розділу

В даному розділі магістерської дисертації було детально описано всі вбудовані функції TensorFlow та PyTorch фреймворків, які використовувалися для реалізації алгоритмів lazy random walk та random walk with restart, що відносяться до класу random walk, а також покроково описано саму програмну реалізацію. Також було детально описано всі складові, які застосовувалися для розпаралелювання алгоритмів класу random walk. З описаного вище можна виокремити те, що фреймворки TensorFlow та PyTorch, являючись фреймворками для паралельного обчислення, реалізують внутрішнє розпаралелювання матричних операцій без втручання розробника, проте користувачу для виконання цих розпаралелень потрібно адаптувати алгоритми таким чином, щоб використовувалися лише матричні операції.

Після роботи з фреймворками TensorFlow та PyTorch в рамках написання магістерської дисертації, можна зробити висновок, що для роботи з матрицями ці фреймворки мають всі необхідні інструменти, які дозволяють виконувати будь-які операції над матрицями і, що доволі важливо, робити потрібні обчислення досить швидко. Програмна реалізація алгоритмів random walk, з використанням вже зазначених фреймворків, виявилася зручною і, важливо зазначити, що інтеграція алгоритмів і фреймворків пройшла легко, адже в основі, як перших, так і других лежать матриці. Тому теоритична ідея, використання фреймворків TensorFlow та PyTorch для реалізації алгоритмів lazy random walk та random walk with restart, отримала підтвердження на практиці.

4. ТЕСТУВАННЯ АЛГОРИТМІВ RANDOM WALK РЕАЛІЗОВАНИХ З ВИКОРИСТАННЯМ ФРЕЙМВОРКУ PYTORCH

В даному розділі магістерської дисертації буде проведено ряд тестових експериментів і аналіз отриманих результатів.

Для реалізації тестування було додано ряд нових реалізацій, а саме: реалізація алгоритмів класу random walk за допомгою Numpy та функція генерації рандомних графів (рис. 4.1).

```
def generate_random_graph(self):
  random_matrix = numpy.random.randint(self.min_weight, self.max_weight, (self.nodes_count, self.nodes_count))
  zeros_count = self.edges_percentage * self.nodes_count * self.nodes_count
  while zeros_count > 0:
    [x, y] = numpy.random.randint(low = 0, high = self.nodes_count, size=2)
    if random_matrix[x][y] != 0:
      zeros_count=zeros_count-1;
      random_matrix[x][y] = 0
  return random_matrix.tolist()
```

Рисунок 4.1 — Функція генерації рандомних графів

Розміри графів, які використовувалися для тестування, були такими: 10, 100, 500, 1000, 5000, 10000. Такі дані було обрано для того, щоб максимально вдало показати покращення, які дає паралелезація та використання фреймворків TensorFlow та PyTorch і простежити динаміку від графів з маленькою кількістю вершин до графів з великою кількістю вершин. Також ще однією метою вибору таких розмірів графа є бажання моделювання реальних випадків при яких можна використовувати алгоритми і тестування їх поведінки в таких ситуаціях.

Для отримання максимально правдивих результатів, всі проведені тести виконувалися по декілька разів, а потім усереднені результати використовувалися для побудови графіків.

4.1 Порівняння паралельної та непаралельної реалізації алгоритмів LRW та RWR

В цьому підрозділі будуть представлені результати вимірювання часу роботи алгоритмів lazy random walk та random walk with restart на фреймворках TensorFlow та PyTorch та результати, які були отримані при непаралельній реалізації цих алгоритмів за допомогою Numpy.

4.1.1 Тестування LRW

Після запуску тесту з графом на 10 вершин для вимірювання часу роботи алгоритму lazy random walk, було отримано такі результати (рис. 4.2):

Algorithm: Lazy random walk

	TensorFlow(static mode)	TensorFlow(dynamic mode)	PyTorch	Numpy
cpu	0.804113	0.776656	0.00731241	0.00710078
gpu	0.84271	0.795142	0.0485681	

Рисунок 4.2 — Час роботи алгоритму на графі розміром 10 вершин

Результати при тестуванні алгоритму lazy random walk на графі розміром 100 вершин (рис.4.3):

Algorithm: Lazy random walk

	TensorFlow(static mode)	TensorFlow(dynamic mode)	PyTorch	Numpy
cpu	0.822705	0.804308	0.085216	0.0954527
gpu	0.890118	0.806143	0.405188	

Рисунок 4.3 — Час роботи алгоритму на графі розміром 100 вершин

Результати при тестуванні алгоритму lazy random walk на графі розміром 500 вершин (рис. 4.4):

Algorithm: Lazy random walk

	TensorFlow(static mode)	TensorFlow(dynamic mode)	PyTorch	Numpy
cpu	1.05661	1.02823	1.31642	1.59352
gpu	0.841632	0.805661	2.27538	

Рисунок 4.4 — Час роботи алгоритму на графі розміром 500 вершин

Результати при тестуванні алгоритму lazy random walk на графі розміром 1000 вершин (рис. 4.5):

Algorithm: Lazy random walk

	TensorFlow(static mode)	TensorFlow(dynamic mode)	PyTorch	Numpy
cpu	2.63793	2.56655	8.19948	8.50917
gpu	0.891159	0.909039	3.99218	

Рисунок 4.5 — Час роботи алгоритму на графі розміром 1000 вершин

4.1.2 Тестування RWR

Після запуску тесту з графом на 10 вершин для вимірювання часу роботи алгоритму RWR, було отримано такі результати (рис. 4.6):

Algorithm: Random walk with restart

	TensorFlow(static mode)	TensorFlow(dynamic mode)	PyTorch	Numpy
cpu	1.07754	1.05383	0.00399442	0.00348721
gpu	1.1085	1.07521	0.017362	

Рисунок 4.6 — Час роботи алгоритму на графі розміром 10 вершин

Результати при тестуванні алгоритму random walk with restart на графі розміром 100 вершин (рис. 4.7):

Algorithm: Random walk with restart

	TensorFlow(static mode)	TensorFlow(dynamic mode)	PyTorch	Numpy
cpu	1.15138	1.12971	0.0425426	0.0424901
gpu	1.1824	1.1179	0.134839	

Рисунок 4.7 — Час роботи алгоритму на графі розміром 100 вершин

Результати при тестуванні алгоритму random walk with restart на графі розміром 500 вершин (рис. 4.8):

Algorithm: Random walk with restart

	TensorFlow(static mode)	TensorFlow(dynamic mode)	PyTorch	Numpy
cpu	1.237	1.20271	1.27405	1.53205
gpu	1.14624	1.12045	0.643383	

Рисунок 4.8 — Час роботи алгоритму на графі розміром 500 вершин

Результати при тестуванні алгоритму random walk with restart на графі розміром 1000 вершин (рис. 4.9):

Algorithm: Random walk with restart

	TensorFlow(static mode)	TensorFlow(dynamic mode)	PyTorch	Numpy
cpu	1.61333	1.59154	8.77707	9.7948
gpu	1.14968	1.16479	1.44009	

Рисунок 4.9 — Час роботи алгоритму на графі розміром 1000 вершин

4.2 Порівняльна характеристика отриманих результатів

Наведемо графік отриманих результатів вимірювання часу роботи алгоритмів lazy random walk та random walk with restart, які реалізвані за допомогою фреймворків для паралельного обчислення TensorFlow та PyTorch і непаралельного Nampy. Таких графіків буде чотири, а саме:

- Графік часу роботи алгоритму lazy random walk (рис. 4.10)

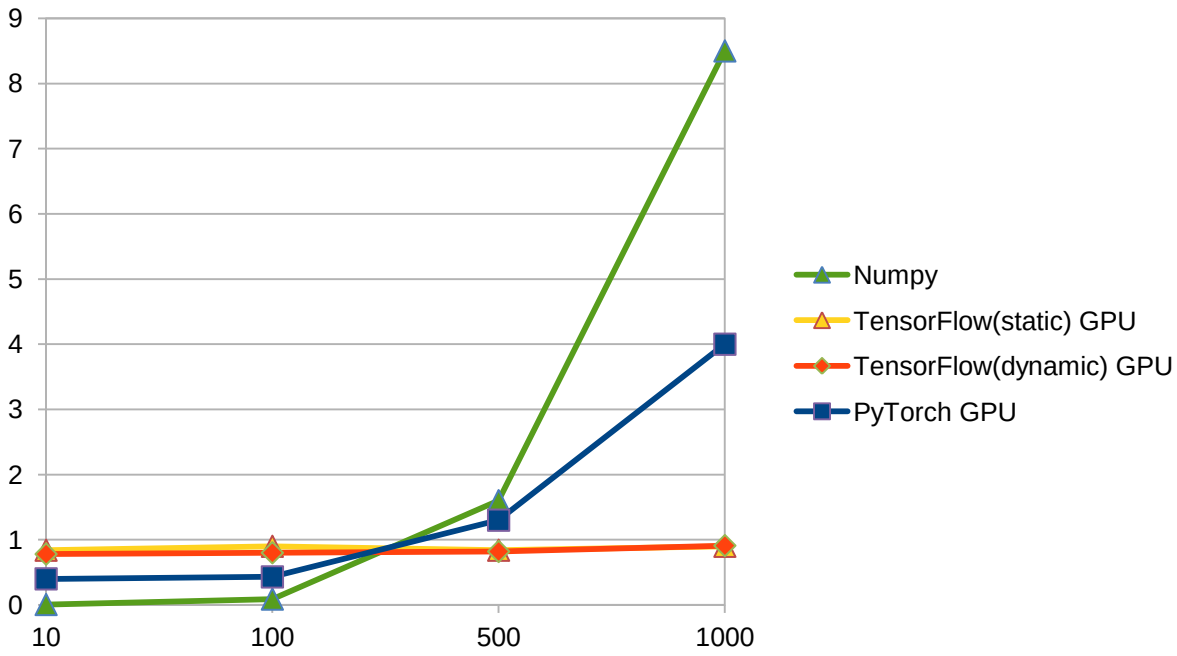

Рисунок 4.10 — Графік часу роботи алгоритму lazy random walk

- Графік часу роботи алгоритму random walk with restart (рис. 4.11)

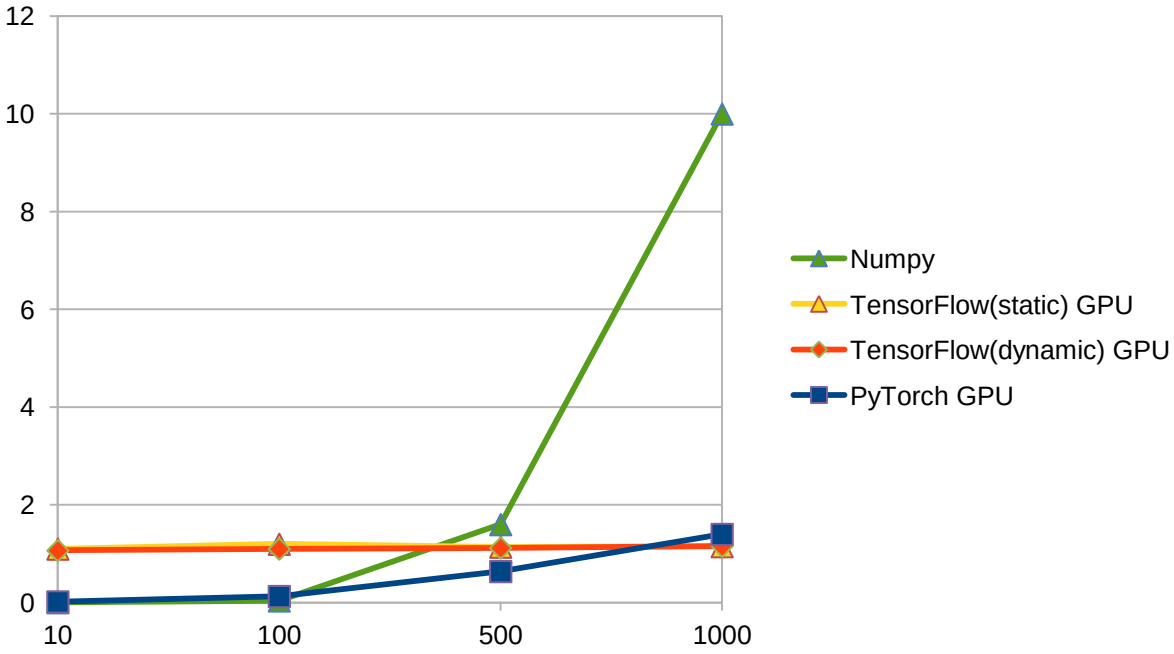

Рисунок 4.11 — Графік часу роботи алгоритму random walk with restart

Проаналізуємо отримані графіки для алгоритмів lazy random walk та random walk with restart. Як видно розпаралелення дає позитивний результат на графах з великою кількістю вершин. Це пов'язано з тим, що на малих графах час синхронізації потоків більший ніж звичайне обчислення матричних операцій.

Далі проаналізуємо роботу фреймворків TensorFlow та PyTorch для великих графів, як на CPU так і на GPU. Результати тестування показують, що TensorFlow при запуску на CPU і GPU, як в статичному режимі, так і в динамічному працює швидше ніж PyTorch. Це пояснюється використанням додаткової складової в схемі розпаралелення, а саме tf.vectorized_map. Проте, варто звернути увагу, на те, що при запуску на GPU час виконання менший. Це також легко пояснити тим, що процесор GPU має більше ядер, а отже може запускати більшу кількість потоків.

ВИСНОВКИ

Метою дослідження, яке було проведено в даній магістерській дисертації, було покращення алгоритмів lazy random walk та random walk with restart, які відносяться до класу random walk, за рахунок реалізації розпаралелення за допомогою фреймворків TensorFlow та PyTorch. Супутньою метою було порівняння ефективності роботи фреймворків між собою при реалізації, за їх допомогою, алгоритмів класу random walk.

Також було проведено аналіз результатів, отриманих при тестуванні реалізованих алгоритмів, і порівняльний, як теоретичний, так і практичний, аналіз фреймворків TensorFlow та PyTorch. Попередньо, для отримання цих результатів, було проведено адаптацію алгоритмів lazy random walk та random walk with restart для інтеграції з вищевказаними фреймворками.

Загалом, аналіз отриманих результатів показав, що теоретична ідея з розпаралеленням алгоритмів класу random walk та написання їх за допомогою фреймворків TensorFlow та PyTorch, отримала практичне підтвердження своєї ефективності для великих графів.

Також було зроблено висновок про те, що для реалізації розпаралелення фреймворк TensorFlow підходить краще, адже має більшу кількість потрібного функціоналу і дає можливість розпаралелети ті фрагменти коду, які неможливо розпаралелети з використанням PyTorch. Ще одним важливим висновком аналізу отриманих результатів є той факт, що алгоритми, які реалізовані за допомогою фреймворків TensorFlow та PyTorch, краще запускати на процесорі GPU, адже цей процесор має більше ядер, а отже може створити більше потоків, що призведе до зменшення часу виконання алгоритму.

Практична цінність отриманих в роботі результатів полягає в тому, що алгоритми класу random walk, які використовуються в різноманітних галузях, таких

як: математика, інформатика, фізика, хімія, біологія, воєнна справа і т.д. при реалізації за допомогою фреймворків TensorFlow та PyTorch будуть працювати швидше та ефективніше.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Курдус А.О., Марченко О.І. Random Walk алгоритми. XIV конференція молодих вчених ПМК-2021. С. 298 – 302.

2. Курдус А.О. Сфери використання random walks алгоритмів та їхня актуальність. VIII Міжнародна науково-технічна internet-конференція «Сучасні методи, інформаційне, програмне та технічне забезпечення систем керування організаційно-технічними та технологічними комплексами». 2021. С. 97 – 99.URL:https://drive.google.com/file/d/14AWKwUkqDgYMIuFW_OPpZaQLFCSAWiFr/view (дата звернення: 05.12.2021)

3. Robin Burke, 2011. Recommender System. URL: https://www.researchgate.net/publication/220604600_Recommender_Systems_An_Overview (дата звернення: 13.11.2021)

4. Feng Xia, 2019. Random Walks: A Review of Algorithms and Applications. URL: https://www.researchgate.net/publication/337501163_Random_Walks_A_Review_of_Algorithms_and_Applications (дата звернення: 24.10.2021)

5. Bratanic, 2021. Link prediction. URL: https://towardsdatascience.com/tagged/link-prediction (дата звернення: 23.10.2021)

6. Jean Ponce, 2021. Computer Vision. URL: https://www.springer.com/journal/11263 (дата звернення: 06.11.2021)

7. TensorFlow: URL: https://www.tensorflow.org (дата звернення: 12.11.2021)

8. PyTorch: URL: https://pytorch.org (дата звернення: 12.11.2021)

9. CNTK: URL: https://habr.com/ru/company/microsoft/blog/336552/ (дата звернення: 12.11.2021)