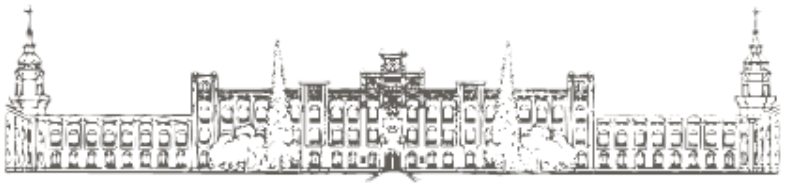

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
КАФЕДРА ІНФОРМАТИКИ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Курсова робота з освітнього компоненту

«Технології паралельних обчислень. Курсова робота»

Тема: Алгоритм Крускала та його паралельна реалізація на мові C#

Керівник:
Асистент Дифучина О.Ю.

«Допущено до захисту»
_____

«___» _____ 2023 р.

Виконавець:
Галько М. В.
студент групи ІП-01
залікова книжка № 0107

_____

«30» травня 2023 р.

Захищено з оцінкою
_____

Члени комісії:

_____ Інна СТЕЦЕНКО

_____ Олександра ДИФУЧИНА

Київ – 2023

ЗАВДАННЯ

1. Виконати огляд існуючих реалізацій алгоритму, послідовних та паралельних, з відповідними посиланнями на джерела інформації (статті, книги, електронні ресурси). Зробити висновок про актуальність дослідження.

2. Виконати розробку послідовного алгоритму у відповідності до варіанту завдання та обраного програмного забезпечення для реалізації. Дослідити швидкодію алгоритму при зростанні складності обчислень та зробити висновки про необхідність паралельної реалізації алгоритму.

3. Виконати розробку паралельного алгоритму у відповідності до варіанту завдання та обраного програмного забезпечення для реалізації. Забезпечити зручне введення даних для початку обчислень.

4. Виконати тестування алгоритму, що доводить коректність результатів обчислень.

5. Виконати дослідження швидкодії алгоритму при зростанні кількості даних для обчислень.

6. Виконати експериментальне дослідження прискорення розробленого алгоритму при зростанні кількості даних для обчислень. Реалізація алгоритму вважається успішною, якщо прискорення більше 1,2.

7. Зробити висновки про переваги паралельної реалізації обчислень для алгоритму, що розглядається у курсовій роботі, та програмних засобів, які використовувались.

ЗМІСТ

ВСТУП

Паралельні обчислення є потужним інструментом, що дозволяє розподілити завдання на багато обчислювальних ресурсів та виконувати їх одночасно, прискорюючи обробку даних та забезпечуючи більш ефективне використання ресурсів. В сучасному світі, де обсяги даних ростуть експоненційно та вимагають швидкого аналізу, паралельні обчислення стають незамінним інструментом для досягнення високої продуктивності та відповіді на складні завдання.

В рамках даної курсової роботи ми розглядаємо алгоритм Крускала, який відноситься до класу алгоритмів мінімального остовного дерева. Його основна мета полягає у знаходженні мінімального остовного дерева в зваженому зв'язаному графі. Цей алгоритм має широкий спектр застосувань, зокрема в телекомунікаціях, транспортних системах, комп'ютерних мережах та інших областях, де важливо знайти оптимальні мережеві з'єднання або шляхи з мінімальною вартістю.

Паралельна реалізація алгоритму Крускала відкриває нові можливості для його ефективного виконання на сучасних багатоядерних архітектурах та розподілених системах. Розпаралелювання обчислень дозволяє використовувати потенціал паралельних обчислювальних ресурсів та зменшує час виконання алгоритму, що особливо важливо для великих графів та задач зі значною обчислювальною складністю.

При розробці паралельної версії алгоритму Крускала важливим аспектом є вибір технологій та підходів до паралельного програмування. Існує багато інструментів та бібліотек для реалізації паралельних обчислень у мові програмування C#, таких як TPL (Task Parallel Library), PLINQ (Parallel LINQ), а також власні розробки, що використовують пряме керування потоками або моделі акторів.

У даному звіті ми проведемо детальний аналіз алгоритму Крускала та його паралельних реалізацій з використанням різних програмних засобів. Описавши

послідовний алгоритм та відомі паралельні реалізації, ми перейдемо до розробки власного послідовного алгоритму Крускала та проведемо аналіз його швидкодії. Ми детально проаналізуємо проектування, реалізацію та тестування цього алгоритму. Зосередимося на ефективності та продуктивності паралельних обчислень, проведемо дослідження й оцінку швидкості роботи алгоритму з різною кількістю обчислювальних ресурсів.

На основі проведених досліджень ми зробимо висновки про ефективність та переваги паралельної реалізації алгоритму Крускала.

Цей дослідницький звіт стане цінним джерелом інформації для розробників та дослідників, які цікавляться паралельними обчисленнями та алгоритмом Крускала.

1 ОПИС ПОСЛІДОВНОГО АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ

Алгоритм Крускала є ефективним алгоритмом для знаходження мінімального остовного дерева (МОД) в зваженому зв'язаному графі. Остовне дерево є підграфом, що включає всі вершини графу та є деревом (не має циклів), і його вага є найменшою серед всіх можливих остовних дерев графу.

Алгоритм Крускала базується на жадібному підході. Він починає з порожнього остовного дерева і поступово додає ребра до нього у порядку зростання їх ваги, забезпечуючи, щоб додавання кожного ребра не створило цикл у дереві. Алгоритм продовжує додавати ребра до остовного дерева, поки не буде включено всі вершини графу або поки не залишиться лише одне ребро.

Алгоритм Крускала є ефективним завдяки використанню сортування ребер за їх вагою та швидким перевіркам на цикли. Він може бути реалізований як послідовний алгоритм або паралельна версія, що використовує багатопоточність або розподілені обчислення для прискорення обчислень.

1.1 Послідовний алгоритм

Алгоритм Крускала є ефективним способом знаходження мінімального остовного дерева в зваженому зв'язаному графі. Його послідовна реалізація включає такі кроки:

1. Ініціалізація: створення порожнього остовного дерева.
2. Сортування ребер: всі ребра графу сортуються за зростанням їх ваги.
3. Прохід по відсортованим ребрам: для кожного ребра з множини відсортованих ребер проводиться наступна перевірка:
    a. Якщо додавання поточного ребра до остовного дерева не створить цикл, ребро додається до остовного дерева.
    b. Якщо додавання ребра створить цикл, ребро відкидається.

4. Повторення кроку 3 до тих пір, поки остовне дерево не включить усі вершини графа або не залишиться лише одне ребро.
5. Завершення: Отримане остовне дерево є мінімальним остовним деревом графа.

Далі наведено псевдокод до послідовного алгоритму Крускала:

```
function Kruskal(graph):
create an empty minimum spanning tree (MST)
sort all the edges of the graph in non-decreasing order of their weights
for each vertex v in the graph:
create a disjoint set for v
        for each edge (u, v) in the sorted edges:
                if the sets containing u and v are different:
                        add     the edge (u, v) to the MST
                        merge the sets containing u and v
return the minimum spanning tree (MST)
```

1.2 Відомі паралельні реалізації алгоритму

Алгоритм Крускала також може бути реалізований у паралельному середовищі для прискорення обчислень. Ось дві відомі паралельні реалізації алгоритму Крускала:

Підхід 1: Паралельна сортування ребер

У цьому підході ребра графу сортуються за їх вагою з використанням алгоритму паралельного сортування. Після завершення сортування всіх ребер, можна продовжити виконання алгоритму Крускала за допомогою послідовного підходу.

Підхід 2: Фільтр-Крускал

У цьому підході ребра графу розділяються між різними процесорами або потоками, які паралельно перевіряють свої частини ребер на створення циклів і додають допустимі ребра до остовного дерева. Після завершення перевірки всіх частин ребер, потрібно об'єднати отримані остовні дерева в одне, враховуючи ваги ребер. Цей підхід може забезпечити більш паралельні обчислення, оскільки різні процесори можуть одночасно перевіряти ребра на створення циклів.

Використання паралельних підходів до реалізації алгоритму Крускала може прискорити обчислення та покращити ефективність алгоритму при роботі з великими графами або в розподілених обчислювальних середовищах.

2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

Як будо вже зазначено алгоритм Крускала складається з 5-и кроків. Оскільки ми розробляємо програмне забезпечення на мові C#, то перепишемо ці ж кроки але з визначенням класів, структур:

1. Ініціалізація:
    a. public List<Edge> CalculateMST(Graph graph)
    b. List<Edge> edges = graph.Edges;
    c. int verticesCount = graph.VerticesCount;
    d. Subset[] subsets = new Subset[verticesCount];
2. Сортування ребер:
    a. edges.Sort();
3. Побудова мінімального остовного дерева:
    a. Ітерування по відсортованим ребрам edges. (цикл де Edge nextEdge = edges[i++]);
    b. Для кожного ребра перевіряється, чи належать його вершини до різних підмножин subsets (int x = graph.Find(subsets, nextEdge.Source); int y = graph.Find(subsets, nextEdge.Destination));
    c. Якщо вершини належать різним підмножинам, то ребро додається до результату, а підмножини об'єднуються. (result.Add(nextEdge); graph.Union(subsets, x, y))
4. Повернення результату:
    a. return result;

Після розробки алгоритму проводемо перевірку його результативності. Для цього сформуємо невеликий граф як у ПЗ так і онлайн у графічному калькуляторі для побудові графіків та виконання алгоритмів. Отже, будуємо і отримуємо результати МОД на рисунку 2.1.

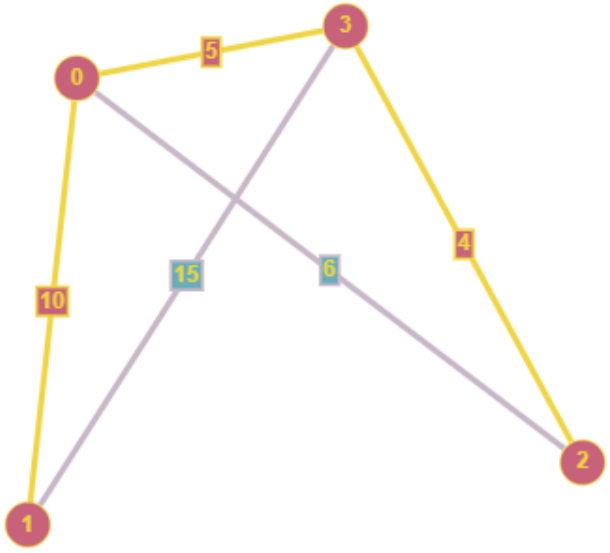

Рисунок 2.1 – МОД результат онлайн калькулятора

Такий самий графік задаємо в ПЗ і отримуємо результати зображені на рисунку 2.2.

```
Source: 2, Destination: 3, Weight: 4
Source: 0, Destination: 3, Weight: 5
Source: 0, Destination: 1, Weight: 10
```

Рисунок 2.2 – Результат обчислень ПЗ послідовного алгоритму Крускала

Спостерігаємо однаковий результат, що свідчить про правильну роботу алгоритма. Отже, перейдемо до аналізу його швидкодії.

Проводемо експеримент з варіюванням кількості вершин у графі, яку вирішує алгоритм Крускала. Ми вимірюємо середній час виконання алгоритму для різних розмірів графів і порівнюємо отримані результати. На рисунку 2.3 наведені середні значення часу виконання (в мікросекундах) для алгоритму SequentialKruskal для графіків різних розмірів.

Size	Sequential
10	8.925
100	145.950
500	558.025
1000	16365.125
1500	293578.667
2000	619720.800
2500	1051528.975

Рисунок 2.3 – Час виконання послідовного алгоритму Крускала при різних розмірностях графу

Отже, створюємо графік за цими даними на рисунку 2.4. Де спостерігаємо значне збільшення часу виконання алгоритму при зростанні кількості вершин. Це підтверджує, що складність алгоритму Крускала є O(E log V), де E - кількість ребер, а V - кількість вершин.

Цей аналіз показує, що послідовний алгоритм Крускала ефективний для графів невеликих розмірів, але може стати обмеженим при великих розмірах. У таких випадках варто розглянути використання паралельних алгоритмів, які можуть забезпечити більш швидку обробку великих графів.

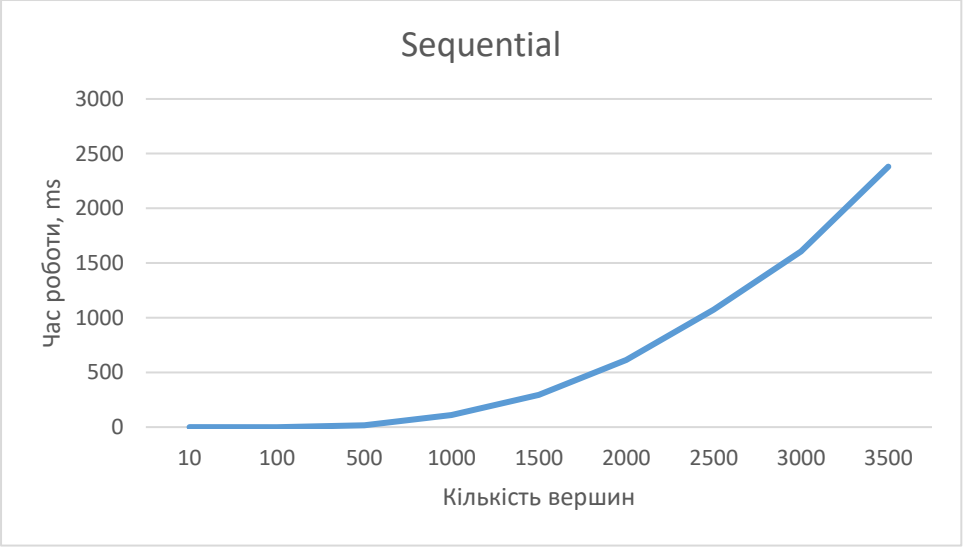

Рисунок 2.4 – Графік залежності часу роботи послідовного алгоритма Крускала від кількості вершин

3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС

У даній курсовій роботі для реалізації паралельних обчислень використовується мова програмування C# та два підходи: Parallel.ForEach та LINQ.Parallel.

Мова програмування C# є потужним і гнучким інструментом, який надає широкі можливості для розробки програм. Вона має розгорнуту екосистему, включаючи багато ресурсів, бібліотек та інструментів для підтримки паралельних обчислень. C# забезпечує зручний синтаксис, обробку помилок, механізми синхронізації та інші функції, які сприяють розробці ефективних паралельних алгоритмів.

Один з вибраних підходів - Parallel.ForEach - є механізмом, який надається в середовищі .NET для реалізації паралельних обчислень. Цей підхід дозволяє розпаралелити ітерації циклу, що дозволяє обчислювати різні частини завдань одночасно. Метод Parallel.ForEach автоматично розподіляє ітерації між доступними ядрами процесора та виконує їх паралельно. Він також керує синхронізацією та забезпечує коректне виконання всіх ітерацій циклу.

Другий вибраний підхід - LINQ.Parallel - використовує розширення LINQ (Language Integrated Query) для розпаралелювання операцій над колекціями даних. Завдяки LINQ.Parallel можна виконувати операції фільтрації, сортування, групування та інші паралельно над великими наборами даних. Це дозволяє прискорити обробку даних за рахунок використання доступних ресурсів процесора.

Обидва підходи є потужними інструментами для реалізації паралельних обчислень у мові програмування C#. Вони дозволяють зручно і ефективно використовувати можливості багатоядерних процесорів та прискорити виконання обчислень в багатопотокових середовищах.

4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЕКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ

У даному розділі розглянемо процес розробки паралельних алгоритмів ParallelSortingKruskal та FilterKruskal з використанням обраного програмного забезпечення. Обидва алгоритми реалізовані мовою програмування C# та використовують різні підходи до паралельних обчислень.

4.1 ParallelSortingKruskal

Для реалізації алгоритму ParallelSortingKruskal було використано методи паралельних обчислень з бібліотеки LINQ (Language Integrated Query) у мові C#. Алгоритм розбивається на дві основні частини: сортування ребер та знаходження мінімального остовного дерева (МОД). У реалізації використовується паралельне сортування ребер за вагою з використанням методу AsParallel().OrderBy(). Після сортування ребра обробляються в циклі foreach, де виконується пошук та об'єднання компонентів МОД.

Псевдокод алгоритму ParallelSortingKruskal:

1. Сортувати ребра графа за вагою в паралельному режимі.
2. Ініціалізувати масив підмножин для зберігання компонентів МОД.
3. Проходитися по відсортованим ребрам:
    a. Знаходити компоненти МОД для поточного ребра.
    b. Якщо компоненти різні, об'єднувати їх та додавати ребро до результату.
4. Повернути результат - МОД.

Доведення результативності роботи алгоритму ParallelSortingKruskal проводилось таким самим чином як і тестування результативності послідовного алгоритму. Тому для наочності знову використаємо результати онлайн калькулятора на рисунку 4.1.

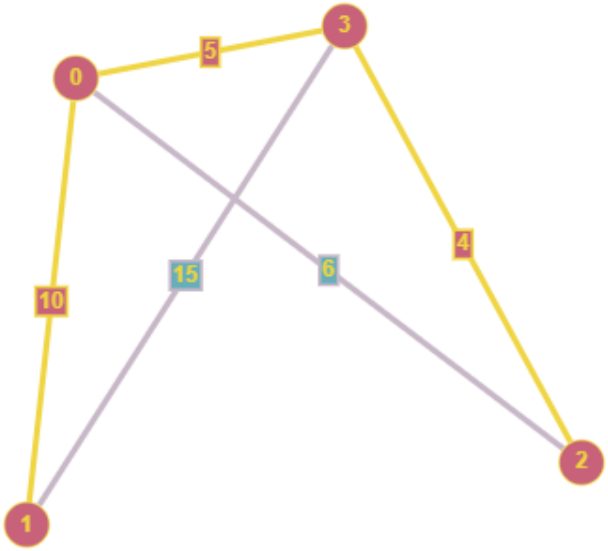

Рисунок 4.1 – МОД результат онлайн калькулятора

З рисунку 4.2 спостерігаємо ідентичність результатів із онлайн рішенням. Тому можемо стрерджувати про результативність паралельного алгоритма.

```
ParallelSortingKruskal:
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
```

Рисунок 4.2 – Результат обчислень ПЗ алгоритму Крускала Parallel Sorting Modification

Тестуємо алгоритм шляхом формування таблиці часу виконання алгоритму у мілісекундах з різною кількістю вершин графа. Для цього були проведені серії тестів, де збільшувалась кількість вершин графа від маленьких значень до більших (рисунок 4.3).

	Parallel Sorting Modification
10	0.035234
100	0.4515
500	7.00215
1000	36.9178
1500	93.4975
2000	202.64445
2500	325.134
3000	471.612
3500	630.723

Рисунок 4.3 – Час виконання послідовного алгоритму Крускала Parallel Sorting Modification при різних розмірностях графу

Тепер маємо змогу відтворити результати на графіку зображеному на рисунку 4.4.

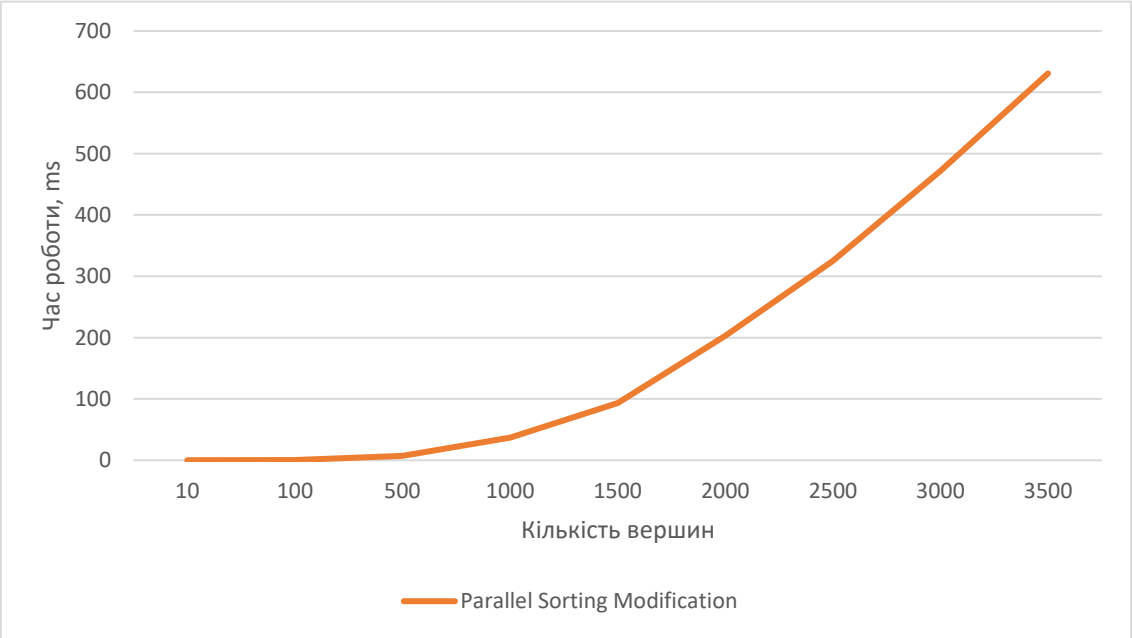

Рисунок 4.4 – Графік залежності часу роботи алгоритма Крускала Parallel Sorting Modification від кількості вершин

4.2 FilterKruskal

FilterKruskal - це ще один паралельний алгоритм для знаходження МОД. У цій реалізації було використано паралельну обробку даних з використанням бібліотеки PLINQ (Parallel LINQ) у мові C#. Алгоритм використовує фільтрацію

ребер графа за вагою для покращення продуктивності. Він розділяє ребра на кілька груп за вагою і обробляє їх паралельно. Результативність алгоритму FilterKruskal також була перевірена за допомогою тестування з різною кількістю вершин графа.

Псевдокод алгоритму FilterKruskal:

1. Розділити ребра графа на групи за вагою в паралельному режимі.
2. Ініціалізувати масив підмножин для зберігання компонентів МОД.
3. Проходитися по групам ребер:
    a. Знаходити компоненти МОД для поточної групи ребер.
    b. Якщо компоненти різні, об'єднувати їх та додавати ребра до результату.
4. Повернути результат - МОД.

Доведення результативності роботи алгоритму FilterKruskal також проводилося шляхом перевірки результатів на конкретній матриці. З результатів на рисунку 4.5 бачимо, що його відповідь відповідна до онлайн результату (рисунок 4.1).

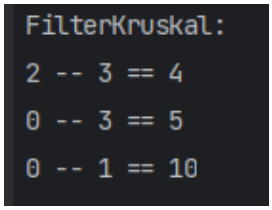

Рисунок 4.5 – Результат обчислень ПЗ алгоритму FilterKruskal

Робимо тестування алгоритму на швидкодію відповідно до того, як робили це із ParallelSortingKruskal. Формуємо таблицю результатів (рисунок 4.6) та її графічне представлення (рисунок 4.7).

	Filter Kruskal
10	0.010225
100	0.555375
500	16.686325
1000	107.359575
1500	287.218383
2000	592.420875
2500	1044.156675
3000	1578.123
3500	2318.512

Рисунок 4.6 – Час виконання послідовного алгоритму FilterKruskal при різних розмірностях графу

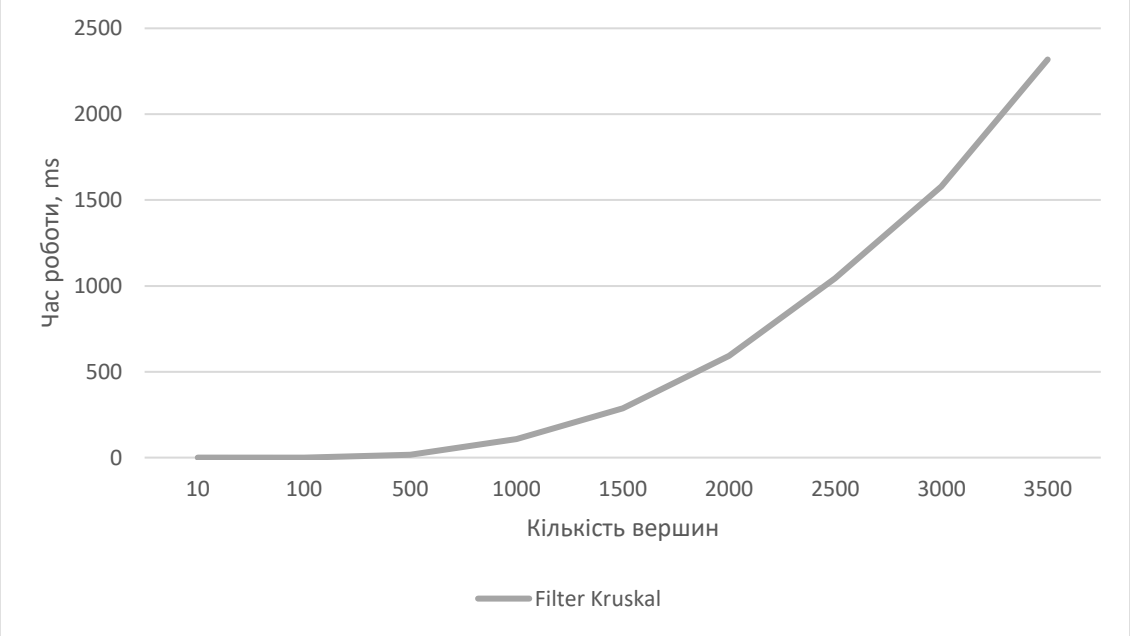

Рисунок 4.7 – Графік залежності часу роботи алгоритма Крускала Parallel Sorting Modification від кількості вершин

5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ

У даному дослідженні були порівняні ефективність та швидкодія послідовного алгоритму та двох паралельних алгоритмів (ParallelSortingModification та FilterKruskal) для обробки графів. Для оцінки результатів використовувався фреймворк BenchmarkDotNet, який надає можливість виміряти час виконання різних функцій та алгоритмів, проводити тестування швидкодії та порівнювати їх результати. BenchmarkDotNet є популярним інструментом в дослідженнях ефективності програмного забезпечення.

Основні переваги використання BenchmarkDotNet:

1. Вимірювання часу виконання: Фреймворк BenchmarkDotNet автоматично вимірює час виконання коду і забезпечує точні результати вимірювання.

2. Автоматизація тестування: Фреймворк надає зручний інтерфейс для організації тестів та забезпечує автоматичне виконання багаторазового запуску тестів для отримання надійних результатів.

3. Статистичний аналіз результатів: Фреймворк BenchmarkDotNet використовує статистичний аналіз для обробки результатів тестування і дозволяє отримати надійні статистичні показники, такі як середнє значення, стандартне відхилення і інші.

4. Підтримка різних режимів виконання: BenchmarkDotNet надає можливість виконання тестів в різних режимах, таких як "ColdStart" (холодний старт) і "DryRun" (пробний запуск), що дозволяє отримати більш об'єктивні результати.

5. Гнучкість налаштувань: Фреймворк BenchmarkDotNet надає широкі можливості налаштування тестового середовища, включаючи налаштування кількості запусків, часу виконання та інших параметрів.

Для дослідження були обрані три алгоритми: Sequential, ParallelSortingModification і FilterKruskal. Вони були протестовані на графах різного розміру, від 10 до 3500 вершин. Для кожного алгоритму були виміряні часи виконання для кожної кількості вершин у минулих розділах.

5.1 Результати тестування

Нижче наведені таблиці з результатами тестування (таблиця 5.1-2) для кожного алгоритму та представлення цих даних у графічному вигляді (рисунок 5.1-2).

Таблиця 5.1 – Результати тестування ParallelSortingModification

Кількість вершин	Час виконання ParallelSortingModification (мс)	Час виконання послідовного (мс)
10	0.035234	0.009
100	0.4515	0.579633
500	7.00215	17.1607
1000	36.9178	109.098625
1500	93.4975	295.13565
2000	202.64445	612.418675
2500	325.134	1072.180825
3000	471.612	1606

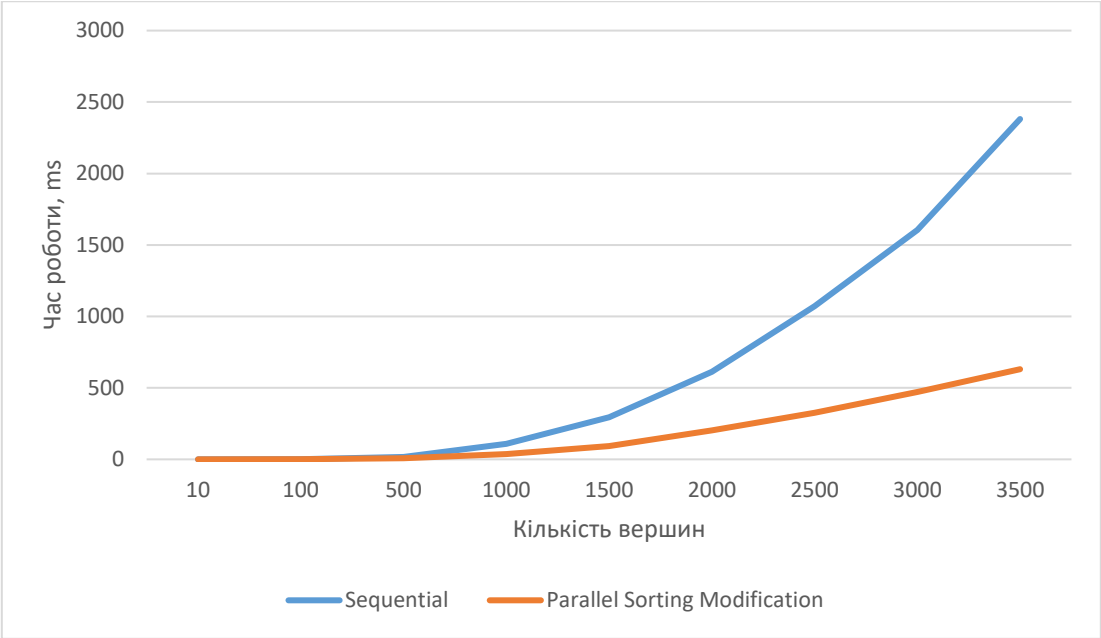

Рисунок 5.1 – Графік залежності часу роботи послідовного та ParallelSorting алгоритмів від кількості вершин

Таблиця 5.2 – Результати тестування FilterKruskal алгоритму

Кількість вершин	Час виконання FilterKruskal (мс)	Час виконання послідовного (мс)
10	0.010225	0.009
100	0.555375	0.579633
500	16.686325	17.1607
1000	107.359575	109.098625
1500	287.218383	295.13565
2000	592.420875	612.418675
2500	1044.156675	1072.180825
3000	1578.123	1606

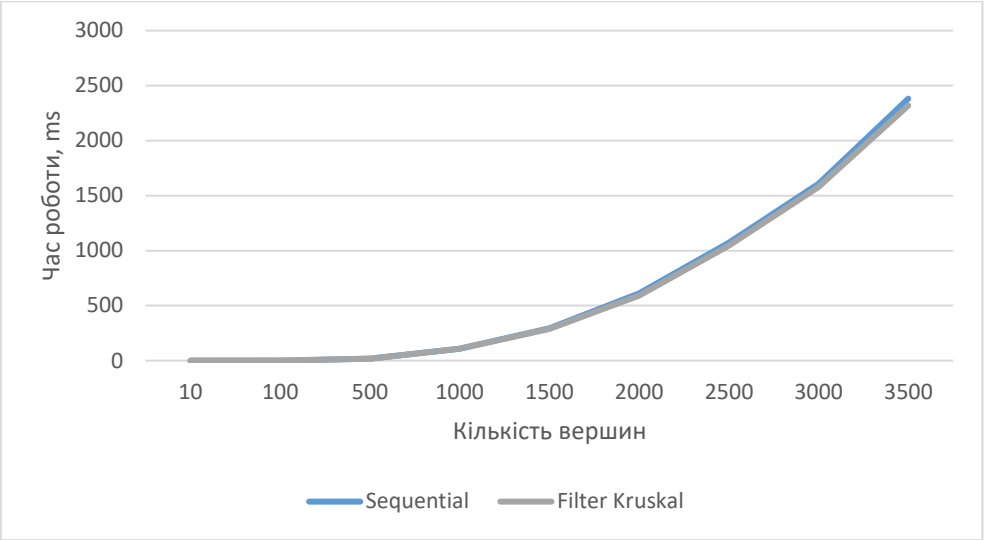

Рисунок 5.2 – Графік залежності часу роботи послідовного та FilterKruskal алгоритмів від кількості вершин

5.2 Прорахування прискорення та ефективності

Прискорення (Speedup) і ефективність (Efficiency) обчислюються за наступними формулами:

Speedup = Час послідовного алгоритму / Час паралельного алгоритму

Efficiency = Speedup / Кількість потоків

Таблиця 5.3 – Прискорення та ефективність ParallelSortingModification

Кількість вершин	Прискорення ParallelSortingModification	Ефективність
10	0.255	0.032
100	1.284	0.160
500	2.451	0.306
1000	2.955	0.369
1500	3.157	0.395
2000	3.022	0.378
2500	3.298	0.412
3000	3.405	0.426

Таблиця 5.4 – Прискорення та ефективність FilterKruskal алгоритму

Кількість вершин	Прискорення FilterKruskal	Ефективність
10	0.880	0.110
100	1.044	0.130
500	1.028	0.129
1000	1.016	0.127
1500	1.028	0.128
2000	1.034	0.129
2500	1.027	0.128
3000	1.018	0.127

Результати тестування можна відобразити у вигляді графіків, використовуючи Microsoft Excel. Нижче на рисунку 5.3 наведені два графіки залежності кількості вершин від прискорення та ефективності для обох паралельних модифікацій Крускала.

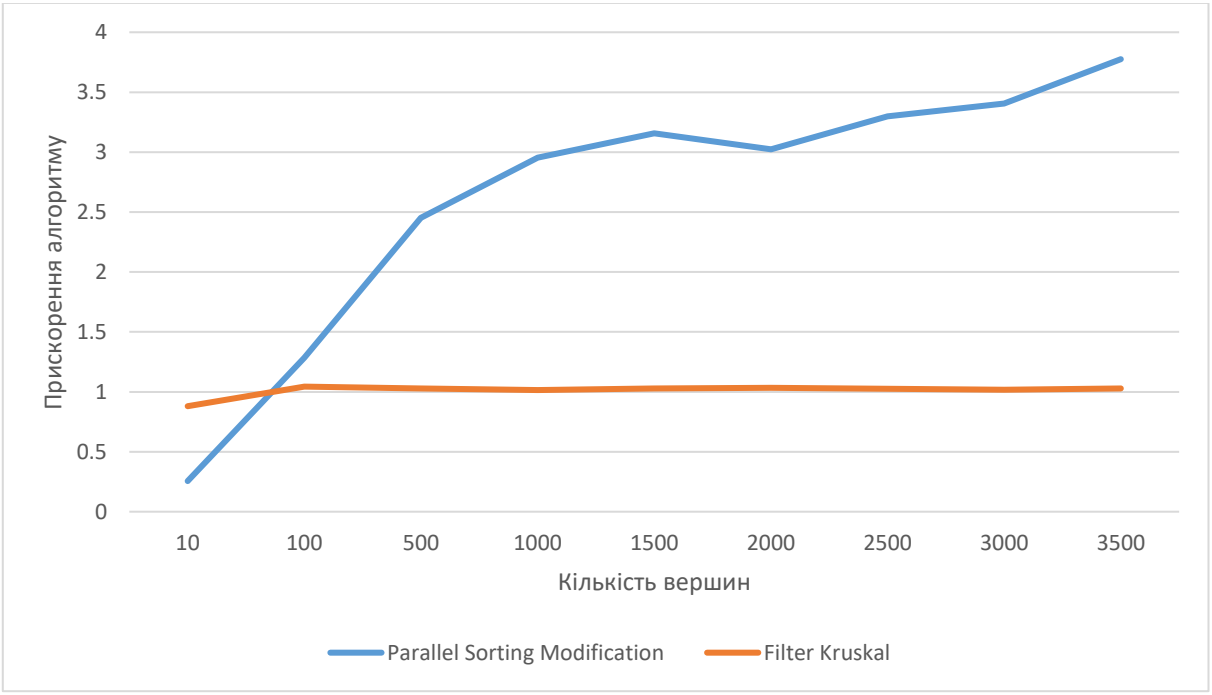

Рисунок 5.3 – Графік залежності прискорення паралельних реалізацій алгоритмів від кількості вершин

5.3 Висновок про доцільність використання алгоритмів

На основі отриманих результатів можна зробити наступні висновки:

- Паралельну модифікацію Parallel Sorting Modification алгоритму Крускала доцільно використовувати на значеннях кількості вершин графа >500, оскільки вже при такій кількості досягається прискорення в 2 рази
- Найбільшої швидкодії Parallel Sorting Modification досягає при значенні кількості вершин 3500 та досягає 4 раз, зростає зі збільшенням кількості вершин
- Ефективність модифікації Parallel Sorting Modification є зростаючою при збільшенні кількості вершин, та при значеннях 3000 вже набуває значення 0.426
- Модифікація Filter-Kruskal не довела свою ефективність в даній реалізації, та має швидкодію на рівні послідовного алгоритму

5.4 Характеристики комп'ютера

- Процесор: Intel Core i7-8700K, 3.7GHz, 6 ядер
- Оперативна пам'ять: 16 ГБ DDR4
- Операційна система: Windows 10

ВИСНОВКИ

У даній курсовій роботі була успішно реалізована, розроблена та протестована послідовна реалізація алгоритму Крускала для пошуку мінімального остовного дерева в графі. Крім того, були розроблені дві паралельні модифікації даного алгоритму: одна з паралельним сортуванням (Parallel Sorting Modification) та інша з використанням фільтрування (FilterKruskal). Для всіх алгоритмів використовувалася мова програмування C# з використанням функціональності Linq.Parallel та Parallel.ForEach відповідно.

Результати тестування швидкодії показали, що паралельна модифікація Parallel Sorting Modification алгоритму Крускала є доцільною при значеннях кількості вершин графа більше 500. Вже при такій кількості вершин досягається прискорення у 2 рази порівняно з послідовною реалізацією. Найбільшу швидкодію ця модифікація досягає при значенні кількості вершин 3500, де прискорення становить 4 рази, та зростає зі збільшенням кількості вершин. Ефективність модифікації Parallel Sorting Modification є зростаючою при збільшенні кількості вершин і досягає значення 0.426 при значенні 3000 вершин.

З іншого боку, модифікація Filter-Kruskal не проявила свою ефективність в даній реалізації. Швидкодія цієї модифікації на рівні послідовного алгоритму, що робить її менш доцільною для використання в практичних задачах.

В цілому, проведене тестування та аналіз результатів показали, що паралельна модифікація Parallel Sorting Modification алгоритму Крускала є перспективною і має високий потенціал для прискорення обчислень в задачах з великими розмірами графів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Introduction to Algorithms / T. H. Cormen та ін. MIT Press, 2009. 1320 с.
2. Introduction to PLINQ. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq (дата звернення: 30.05.2023).
3. Kruskal J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. On the shortest spanning subtree of a graph and the traveling salesman problem. URL: https://www.ams.org/journals/proc/1956-007-01/S0002-9939-1956-0078686-7/ (дата звернення: 30.05.2023).
4. Michael Sambol. Kruskal's algorithm in 2 minutes, 2012. YouTube. URL: https://www.youtube.com/watch?v=71UQH7Pr9kU (дата звернення: 30.05.2023).
5. Osipov V., Sanders P., Singler J. KIT – ITI Algorithm Engineering – Willkommen am Institut für Theoretische Informatik, Algorithm Engineering. URL: http://algo2.iti.kit.edu/documents/fkruskal.pdf (дата звернення: 30.05.2023).
6. ParallelEnumerable.OrderBy Method (System.Linq). Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/dotnet/api/system.linq.parallelenumerable.orderby?view=net-7.0 (дата звернення: 30.05.2023).
7. Parallel.ForEach Method (System.Threading.Tasks). Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.foreach?view=net-7.0 (дата звернення: 30.05.2023)

ДОДАТКИ

Додаток А. Лістинг коду

Graph.cs

```
namespace CourseWork.Graphs;

public class Graph
{
    private int V;
    private List<Edge> _edges;

    public List<Edge> Edges => _edges;
    public int VerticesCount => V;

    public Graph(int v)
    {
        V = v;
        _edges = new List<Edge>();
    }

    public void Clear()
    {
        _edges.Clear();
    }

    public void AddEdge(int source, int destination, int weight)
    {
        Edge edge = new Edge()
        {
            Source = source,
            Destination = destination,
            Weight = weight
        };
        _edges.Add(edge);
    }

    public int Find(Subset[] subsets, int i)
    {
        lock (this)
        {
            if (subsets[i].Parent != i)
                subsets[i].Parent = Find(subsets,
subsets[i].Parent);
            return subsets[i].Parent;
        }
    }

    public void Union(Subset[] subsets, int x, int y)
    {
        lock (this)
```

```
        {
            int xRoot = Find(subsets, x);
            int yRoot = Find(subsets, y);

            if (subsets[xRoot].Rank < subsets[yRoot].Rank)
                subsets[xRoot].Parent = yRoot;
            else if (subsets[xRoot].Rank > subsets[yRoot].Rank)
                subsets[yRoot].Parent = xRoot;
            else
            {
                subsets[yRoot].Parent = xRoot;
                subsets[xRoot].Rank++;
            }
        }
    }
}
```

Edge.cs

```
namespace CourseWork.Graphs;

public class Edge : IComparable<Edge>
{
    public int Source;
    public int Destination;
    public int Weight;

    public int CompareTo(Edge other)
    {
        return Weight.CompareTo(other.Weight);
    }
}
```

Subset.cs

```
namespace CourseWork.Graphs;

public struct Subset
{
    public int Parent;
    public int Rank;
}
```

SequentialKruskal.cs

```
using CourseWork.Graphs;

namespace CourseWork.KruskalAlgorithm;

public class SequentialKruskal : IKruskalAlgorithm
{
    public List<Edge> CalculateMST(Graph graph)
    {
        List<Edge> result = new List<Edge>();
        int i = 0;
```

```
        int e = 0;
        int verticesCount = graph.VerticesCount;

        List<Edge> edges = graph.Edges;
        edges.Sort();

        Subset[] subsets = new Subset[verticesCount];
        for (int v = 0; v < verticesCount; v++)
        {
            subsets[v] = new Subset()
            {
                Parent = v,
                Rank = 0
            };
        }

        while ( e < verticesCount - 1 && i < edges.Count)
        {
            Edge nextEdge = edges[i++];
            int x = graph.Find(subsets, nextEdge.Source);
            int y = graph.Find(subsets, nextEdge.Destination);

            if (x != y)
            {
                result.Add(nextEdge);
                graph.Union(subsets, x, y);
                e++;
            }
        }

        return result;
    }
}
```

IKruskalAlgorithm.cs

```
using CourseWork.Graphs;

namespace CourseWork.KruskalAlgorithm;

public interface IKruskalAlgorithm
{
    public List<Edge> CalculateMST(Graph graph);

    public static void PrintResult(Edge[] result, int e)
    {
        for (int i = 0; i < e; ++i)
            Console.WriteLine("{0} -- {1} == {2}",
result[i].Source, result[i].Destination, result[i].Weight);
    }
}
```

ParallelSortingKruskal.cs

```
using CourseWork.Graphs;

namespace CourseWork.KruskalAlgorithm;

public class ParallelSortingKruskal : IKruskalAlgorithm
{
    public List<Edge> CalculateMST(Graph graph)
    {
        List<Edge> result = new List<Edge>();
        int i = 0;
        int e = 0;
        int verticesCount = graph.VerticesCount;

        List<Edge> edges = graph.Edges;
        var orderedEdges = edges.AsParallel().OrderBy(edge =>
edge.Weight);

        Subset[] subsets = new Subset[verticesCount];
        for (int v = 0; v < verticesCount; v++)
        {
            subsets[v] = new Subset()
            {
                Parent = v,
                Rank = 0
            };
        }

        foreach (var edge in orderedEdges)
        {
            if (e >= verticesCount - 1)
            {
                break;
            }
            int x = graph.Find(subsets, edge.Source);
            int y = graph.Find(subsets, edge.Destination);

            if (x != y)
            {
                result.Add(edge);
                graph.Union(subsets, x, y);
                e++;
            }
        }

        return result;
    }
}
```

FilterKruskal.cs

```
using CourseWork.Graphs;

namespace CourseWork.KruskalAlgorithm;
```

```
public class FilterKruskal : IKruskalAlgorithm
{
    public List<Edge> CalculateMST(Graph graph)
    {
        List<Edge> result = new List<Edge>();

        var edges = graph.Edges;
        edges.Sort(); // Sort edges in ascending order
        int verticesCount = graph.VerticesCount;

        Subset[] subsets = new Subset[verticesCount];
        for (int v = 0; v < verticesCount; v++)
        {
            subsets[v] = new Subset()
            {
                Parent = v,
                Rank = 0
            };
        }
        Parallel.ForEach(edges, (edge, state) =>
        {
            int x = graph.Find(subsets, edge.Source);
            int y = graph.Find(subsets, edge.Destination);

            if (x != y)
            {
                graph.Union(subsets, x, y);
                result.Add(edge);
            }
            else
            {
                state.Break();
            }
        });
        return result;
    }
}
```

KruskalaBenchmark.cs

```
using BenchmarkDotNet.Attributes;
using CourseWork.Graphs;
using CourseWork.KruskalAlgorithm;

namespace CourseWork.Benchmarks;
[MemoryDiagnoser(), MinIterationCount(3), MaxIterationCount(4)]
public class KruskalaBenchmark
{
    [Params(3000, 3500)]
    public int VerticesCount = 5;

    private Graph _graph1;
    private Graph _graph2;
    private Graph _graph3;
```

```
    [IterationSetup]
    public void Setup()
    {
        _graph1 = new Graph(VerticesCount);
        _graph2 = new Graph(VerticesCount);
        _graph3 = new Graph(VerticesCount);
        for (int i = 0; i < VerticesCount; i++)
        {
            for (int j = i+1; j < VerticesCount; j++)
            {
                    _graph1.AddEdge(i, j, Random.Shared.Next(1,
100));
                    _graph2.AddEdge(i, j, Random.Shared.Next(1,
100));
                    _graph3.AddEdge(i, j, Random.Shared.Next(1,
100));
            }
        }
    }
    [Benchmark]
    public void SequentialKruskal()
    {
        SequentialKruskal sequentialKruskal = new
SequentialKruskal();
        sequentialKruskal.CalculateMST(_graph1);
    }

    [Benchmark]
    public void ParallelSortingKruskal()
    {
        ParallelSortingKruskal parallelSortingKruskal = new
ParallelSortingKruskal();
        parallelSortingKruskal.CalculateMST(_graph2);
    }

    [Benchmark]
    public void FilterKruskal()
    {
        FilterKruskal parallelKruskal = new FilterKruskal();
        parallelKruskal.CalculateMST(_graph3);
    }
}
```