



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

Курсова робота

з дисципліни «Технології паралельних обчислень»

Тема: Алгоритм Беллмана-Форда на графах та його паралельна
реалізація мовою програмування Java

Керівник:

д.т.н., проф. Стеценко І.В.

«Допущено до захисту»

«__» _____ 2022 р.

Захищено з оцінкою

Члени комісії:

Виконавець:

Кочев Геннадій Геннадійович

студент групи ІП-91

залікова книжка № ІП-9113

«21» червня 2022 р.

Інна СТЕЦЕНКО

Олександра ДИФУЧИНА

ЗАВДАННЯ

- Виконати розробку паралельного алгоритму Беллмана-Форда у відповідності до варіанту завдання та обраного програмного забезпечення для реалізації. Забезпечити зручне введення даних для початку обчислень.
- Виконати тестування алгоритму, що доводить коректність результатів обчислень.
- Виконати дослідження швидкодії алгоритму при зростанні кількості даних для обчислень.
- Виконати експериментальне дослідження прискорення розробленого алгоритму при зростанні кількості даних для обчислень. Реалізація алгоритму вважається успішною, якщо прискорення більше 1.
- Зробити висновки про переваги паралельної реалізації обчислень для алгоритму, що розглядається у курсовій роботі, та програмних засобів, які використовувались.

РЕФЕРАТ

Структура та обсяг роботи. Пояснювальна записка курсової роботи складається з 5 розділів, містить 2 рисунки, 1 таблицю, 1 додаток, 6 джерел.

Розробка ставить перед собою досягнення наступних цілей:

- реалізація алгоритмів Беллмана-Форда, послідовного та паралельного, за допомогою інструментарію Java;
- аналіз впливу кількості вершин та ребер на час та ефективність виконання алгоритму;
- визначення прискорення паралельної реалізації у порівнянні з послідовною версією алгоритму;
- освоєння інструментарію Java для паралелізації алгоритму.

У розділі 1 було детально описано способи реалізації алгоритму Беллмана-Форда на графі, встановлено асимптотичну складність алгоритму, проаналізовано існуючі реалізації.

У розділі 2 було наведено програмний код реалізації послідовного алгоритму Беллмана-Форда, проаналізована швидкодія отриманої програми.

У розділі 3 було описано інструментарій Java, що обраний для виконання роботи, а також обґрунтовано його використання.

У розділі 4 була виконана паралелізація алгоритму Беллмана-Форда, наведений відповідний програмний код, обґрунтовано правильність виконання алгоритму.

У розділі 5 було спрогнозовано підвищення ефективності виконання, співставлено результат, отриманий практично, з очікуваними результатами, проведено аналіз та графічна демонстрація результатів.

КЛЮЧОВІ СЛОВА: АЛГОРИТМ БЕЛЛМАНА-ФОРДА, АЦИКЛІЧНІ ГРАФИ, JAVA MULTITHREADING, PARALLEL COMPUTATIONS, DATA PARALLELISM.

ЗМІСТ

ВСТУП	5
РОЗДІЛ 1. ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ	6
1.1 Загальні відомості	6
1.2 Постановка задачі	6
1.3 Опис алгоритму	6
1.4 Відомі паралельні реалізації алгоритму Беллмана-Форда на графах	8
РОЗДІЛ 2. РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ	9
2.1 Реалізація послідовного алгоритму	9
2.2 Аналіз швидкодії послідовного алгоритму	11
РОЗДІЛ 3. ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС	12
РОЗДІЛ 4. РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЕКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ	14
4.1 Проектування алгоритму	14
4.2 Реалізація алгоритму	15
4.3 Тестування алгоритму	18
РОЗДІЛ 5. ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ (ПОРІВНЯЛЬНИЙ АНАЛІЗ ШВИДКОСТІ ОБЧИСЛЕНЬ)	20
5.1 Теоретична оцінка показників ефективності	20
5.2 Практична оцінка показників ефективності	21
ВИСНОВКИ	24

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ**26****ДОДАТКИ****27**

Додаток А. Програмний код

27

ВСТУП

Визначення найкоротшого шляху між двома об'єктами - одна з найбільш поширених задач, що постають перед нами у процесі еволюції людства. Ще до того, як було винайдено поняття та розуміння структури даних графа, людині часто доводилось вирішувати цю проблему. З появленням комп'ютерів, смартфонів, серверів, обчислювальних пристроїв та встановленням зв'язку між ними пошук найкоротшого шляху набуває навіть більшої важливості.

Розмірність графів, кількість вершин та ребер, які обробляються на практиці - постійно збільшується. Задача пошуку найкоротшого шляху - одна з базових задач на графах і тому є дуже важливим знаходження нових рішень до неї та оптимізація вже існуючих рішень. Зважаючи на високу складність винайдення нових рішень, одним із доступних способів покращення ефективності виконання алгоритму є застосування технологій паралельних обчислень на вже відомих та перевірених часом алгоритмах.

До таких алгоритмів належить алгоритм Беллмана-Форда, винайдений у 1969 як основний алгоритм для мережі ARPANET [5]. Незважаючи на давність винайдення алгоритму, він залишається одним з найбільш популярних та ефективних для графів зі зваженими ребрами, що можуть мати від'ємну вагу. Алгоритм Беллмана-Форда знаходить шлях від однієї вершини до всіх інших у графі, послідовно декілька разів ітеруєчись за списком ребер та оновлюючи оптимальну дистанцію до вершин. Для використання підходу розподілених обчислень алгоритм потребує розбиття його окремих частин на підзадачі, які можуть виконуватися окремо, і як слідство, паралельно. Таким чином, обрана тема курсової роботи є безумовно актуальною і важливою.

Метою курсової роботи є реалізація алгоритму Беллмана-Форда у багатопоточному середовищі для прискорення обчислення найкоротшого шляху в зваженому графі з вагою ребер, що може бути від'ємною, а також аналіз можливостей обраного інструментарію для паралелізації. В якості інструментарію обрано мову програмування Java та пакет `java.util.concurrent`.

РОЗДІЛ 1. ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ

1.1 Загальні відомості

Алгоритм Беллмана-Форда - це алгоритм пошуку найкоротшого шляху в зваженому графі, шляхи знаходить від обраної вершини графу до всіх інших, якщо такі шляхи є. Допускає ребра з від'ємною вагою [6]. Цей алгоритм було запропоновано незалежно двома вченими - Річардом Беллманом та Лестером Фордом. Алгоритм Беллмана-Форда на графі відноситься до класу пошуку найкоротших шляхів з однієї вершини (Single Source Shortest Path - SSSP). Вперше розроблений в 1969 році та використаний для маршрутизації у роутерах. На даний час цей алгоритм знаходить використання у галузі мережевого програмування, а також у галузі комп'ютерних ігор, коли потрібно знайти оптимальний шлях між об'єктами.

1.2 Постановка задачі

Нехай маємо граф $G = (V, E)$, де V – множина вершин, E – множина пар ребер графу $e = (s, t)$ при $s, t \in V, e \in E$. Додатково, $c(e)$ – вага відповідного ребра, $\forall e \in E$. Також $dist(s, t)$ при $s, t \in V$ – найкоротша відстань між відповідними вершинами, при цьому $dist(s, t) = \infty$, якщо шлях $p(s, t)$ не існує; $dist(v, v) = 0, \forall v \in V$.

Нехай задано початкову вершину s при $s \in V$. Тоді результатом роботи алгоритму Беллмана-Форда є множина найкоротших відстаней до кожної вершини заданого графу.

1.3 Опис алгоритму

Для заданого графу $G = (V, E)$ та початкової вершини s алгоритм Беллмана-Форда знаходить найкоротший шлях від вершини s до усіх інших [2]. Задаються початкові відстані: відстань до вершини s дорівнює 0, відстань до усіх інших дорівнює нескінченності. Алгоритм ітерується $(|V| - 1)$ разів за списком усіх ребер, для кожного ребра проводить процедуру Relaxation - відстанню до вершини стає мінімальне значення між поточним значенням

відстані та сумою відстані до іншої вершини та вагою ребра, що розглядається. Таким чином, ітеративний процес зменшує значення відстані на кожній вершині.

Нижче наведено псевдокод до процедури алгоритму:

```

procedure bellman_ford(graph, start_vertex, weights) {
    start_vertex.dist = 0

    foreach vertex in graph.vertices do
        vertex.dist = infinity
    end

    for i = 1 to graph.vertices.amount - 1 do
        foreach edge (u, v) in graph.edges do
            relaxation(u, v, weights)
        end
    end
}

procedure relaxation(u, v, weights) {
    v.dist = min(v.dist, u.dist + weights(u, v))
}

```

У псевдокоді видно, що відбувається $(|V| - 1)$ ітерація, алгоритм Беллмана-Форда також здатен перевіряти заданий граф на присутність від'ємних циклів. Для цього необхідно провести ще одну ітерацію, і якщо хоча б одне значення відстані змінилось, тоді в графі наявний від'ємний цикл.

Визначимо асимптотичну складність алгоритму: відбувається $(|V| - 1)$ ітерацій, у кожній обробляється $|E|$ ребер. Якщо $|V| = |E| = n$, тоді складність алгоритму дорівнює $O(n^2)$.

1.4 Відомі паралельні реалізації алгоритму Беллмана-Форда на графах

Обговоримо “наївну” реалізацію алгоритму Беллмана-Форда за допомогою паралелізму даних та інструментарію NVIDIA GPU CUDA, що представив Ruijian An [1].

Перш за все, автор зазначає які дані можна розпаралелити для ефективного виконання алгоритму. Ініціалізація відстаней (для початкової вершини нуль, для інших - нескінченність) може бути паралелізована. Також найбільш складний крок алгоритму - Relaxation - можна виконати паралельно, принаймні частково. Кожному потоку присвоюється набір вершин та відповідних вхідних та вихідних ребер для обрахунку. Такий підхід використовує списки суміжності вершин або матриці суміжності і добре підходить для розріджених графів. Автор також використовує техніку редукції непотрібної роботи, обробляючи на кожній ітерації лише ті вершини, які були змінені.

Для імплементації автор обрав мову програмування CUDA [4], що допомагає взаємодіяти з графічним процесором. Автор використовує потоки, що організовуються у блоки, які проходять scheduling на внутрішні поточкові мультипроцесори. Для обходу обмеження в плані передачі даних до пам'яті графічного процесору, автор створює два потоки, між якими поділяє матрицю суміжності для пришвидшення виконання.

РОЗДІЛ 2. РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

2.1 Реалізація послідовного алгоритму

Було прийнято рішення реалізувати канонічний алгоритм Беллмана-Форда на графі мовою програмування Java.

Для цього спочатку потрібно визначити представлення графу, з яким будуть проводитися маніпуляції та алгоритм. Нижче наведено програмний код, що відповідає за створення класів Вершина, Ребро, Граф. Методи виведення даних про ці структури на консоль опущено, оскільки вони не є суттєвими для розуміння реалізації.

```
public class Vertex {  
    public final int id;  
    public int distance;  
  
    public Vertex(int id) {  
        this.id = id;  
    }  
}
```

```
public class Edge {  
    public int startId;  
    public int endId;  
    public int weight;  
  
    public Edge(int startId, int endId, int weight) {  
        this.startId = startId;  
        this.endId = endId;  
        this.weight = weight;  
    }  
}
```

```
public class Graph {  
    public List<Vertex> vertices;
```

```

public List<Edge> edges;

public Graph(List<Vertex> vertices, List<Edge> edges) {
    this.vertices = vertices;
    this.edges = edges;
}
}

```

Таким чином, можемо бачити, що за допомоги об'єктно-орієнтованого підходу граф представляється як список вершин та список ребер. Кожна вершина має ідентифікатор та відстань, яка обчислюється у процесі алгоритму. Ребро складається з ідентифікаторів початкової вершини, кінцевої вершини та ваги ребра.

Перейдемо до розгляду реалізації послідовного алгоритму Беллмана-Форда.

```

public void sequentialAlgorithm() {
    for (Vertex v : graph.vertices) {
        v.distance = Integer.MAX_VALUE - MAX_EDGE_WEIGHT;
    }
    graph.vertices.get(startId).distance = 0;

    for (int i = 1; i < graph.vertices.size(); i++) {
        for (Edge e : graph.edges) {
            graph.vertices.get(e.endId).distance = Math.min(
                graph.vertices.get(e.endId).distance,
                graph.vertices.get(e.startId).distance + e.weight
            );
        }
    }
}

```

Як можна побачити, реалізація повністю відповідає псевдокоду, що наведений у попередньому розділі. Ініціалізуються значення дистанції для кожної з вершин у графі. Проводиться $(|graph.vertices.size()| - 1)$ ітерацій. У кожній ітерації для кожного ребра виконується обчислення Relaxation, що

являє собою знаходження мінімального значення між поточним значенням та сумою дистанції від початкової вершини і ваги ребра.

2.2 Аналіз швидкодії послідовного алгоритму

Для аналізу використаємо Big-O Notation. Нехай кількість вершин у графі дорівнює V , кількість ребер - E . Розберемо складність алгоритму для кожного кроку:

- 1) Ініціалізація значень - V операцій присвоєння
- 2) Релаксація вершин - $V * E$ операцій обчислення мінімального та присвоєння.

Таким чином, маємо складність алгоритму $O(V * E) = O(n^2)$.

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

У даній курсовій роботі стоїть задача розробити та реалізувати паралельний алгоритм. З огляду на необхідність визначення інструментарію для виконання цієї задачі було обрано мову програмування Java. Обґрунтуємо вибір таким чином:

- Java - кроссплатформенна мова з відкритим вихідним кодом (OpenJDK), це дозволяє запускати створені програми на будь-якій операційній системі (Windows, Linux, FreeBSD, MacOS);
- Java - мова, що спонукає до використання принципів ООП (об'єктно-орієнтованого програмування), що робить створені програми структурованими та зручними для написання та розширення;
- Java - мова, що використовується для прикладів та лабораторних робіт на курсі “Технології розподілених обчислень”, до якого ця робота має пряме відношення, як підсумок знань, отриманих за семестр.

Використовується оточення OpenJDK 17 LTS, пакет `java.util.concurrent`. При проектуванні паралельного алгоритму постала необхідність паралельної обробки груп вершин та ребер, тому, за рекомендації викладача-керівника курсової роботи був обраний пул потоків як основний засіб паралелізації. Вибір пояснюється різною кількістю задач на кожній ітерації.

Нижче наведено утиліти, застосовані у процесі виконання роботи.

Таблиця 3.1 - Перелік утиліт та їх короткий опис

Назва утиліти	Опис застосування
IntelliJ IDEA	Середовище розробки програм мовою Java. Створено компанією JetBrains, підтримує автоматичну збірку проектів на Maven.
Google Docs	Офісний пакет для створення документів, сумісних з Microsoft Office.

Google Sheets	Офісний пакет для створення інтерактивних таблиць, використано для групування статистики та побудови аналітичних графіків.
---------------	--

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

4.1 Проектування алгоритму

Як вже стало зрозуміло за псевдокодом та послідовною реалізацією, алгоритм Беллмана-Форда в загальному вигляді представляє собою деяке число ітерацій по всіх ребрах графа зі зміною параметра дистанції для кожної з вершин.

Постає питання, як можна покращити роботу алгоритму засобами паралельних обчислень. У ході аналізу робіт науковців щодо алгоритму Беллмана-Форда та ефективної співпраці з керівником роботи, був знайдений підхід до вирішення такої проблеми. Можна перераховувати декілька вершин одночасно - тобто, розбити граф на групи вершин та ребер, що з'єднують лише ці вершини, їх обраховувати паралельно - за допомогою пулу потоків, а ребра, які залишаються, обробити послідовно, якщо вони не піддаються паралелізації. Також, аналізуючи роботу алгоритму, нескладно здогадатися, що при відсутності змін на ітерації по всіх ребрах продовжувати виконання алгоритму недоцільно, адже нових змін не буде.

Таким чином, встановлено, що кожен потік з доступних буде виконувати таку підзадачу, представлену псевдокодом нижче:

```
procedure thread_task(edges_subarray) {  
    for edge (start, end, weight) in edges_subarray do  
        vertices[end].distance = min(  
            vertices[end].distance,  
            vertices[start].distance + weight  
        )  
        if at least one vertex updated do  
            distanceUpdated = true  
        end  
    end  
}
```

Враховуючи підхід розподілених обчислень до виконання алгоритму, занотуємо, як буде виглядати повне виконання алгоритму за допомогою псевдокоду:

```

procedure parallelAlgorithm(numOfThreads) {
  initialize vertices.distance = infinity
  vertices[src].distance = 0

  chunkSize = vertices.amount / numOfThreads
  chunks = array of arrays of size chunkSize

  adjEdges, seqEdges = split_edges(graph.edges, chunks)
  for i := 1 to vertices.amount do
    for edges_subarray in adjEdges do
      thread_task(edges_subarray)
    end
    wait all thread_tasks to finish
    for edge in seqEdges do
      relaxation(edge)
    end
    if no vertices updated then stop running
  end
}

```

4.2 Реалізація алгоритму

Ключовими змінами у виконанні алгоритму є:

- а) розбиття графу на ребра, що можна обчислити паралельно, та ребра що потрібно опрацювати послідовно;
- б) використання пулу потоків для виконання підзадач обробки групи ребер та вершин;
- в) додавання контролю за змінами у обчисленні вершин - змін не відбувається - алгоритм припиняється.

Обговоримо ці зміни одна за одною та продемонструємо код фрагментами для кращого розуміння ситуації.

Нижче представлено код розбиття списку усіх ребер на дві частини - ребра з паралельною обробкою та без неї:

```
// prepare chunks containers
List<List<Edge>> adjEdges = new ArrayList<>();
int chunkSize = graph.vertices.size() / numberOfThreads;
List<IntPair> chunks = new ArrayList<>();
for (int i = 0; i * chunkSize < graph.vertices.size(); i++) {
    chunks.add(new IntPair(i * chunkSize, i * chunkSize + chunkSize));
    adjEdges.add(new ArrayList<>());
}

// split edges into parallelizable and seq
List<Edge> seqEdges = new ArrayList<>(graph.edges);
for (Edge e : graph.edges) {
    for (int i = 0; i < chunks.size(); i++) {
        IntPair pair = chunks.get(i);
        if (e.startId >= pair.x && e.startId < pair.y && e.endId >= pair.x
        && e.endId < pair.y) {
            adjEdges.get(i).add(e);
            seqEdges.remove(e);
            break;
        }
    }
}
```

Як можна побачити, процес розбиття складається з двох частин - підготовка контейнерів для ребер, та власне розбиття.

Розміром одного контейнеру для ребер стає загальна кількість вершин, поділена на число потоків, що надаються пулу потоків, тому кожен потік буде виконувати приблизно однакову кількість обчислень. Для наглядності наведемо приклад - граф на 20 вершин, 4 потоки - при такому розбитті кожен потік буде обробляти ребра між 5 вершинами: перший з першої по п'яту, другий з шостої по десяту і так далі.

Додаємо до контейнерів паралельно оброблюваних ребер за таким принципом: якщо ребро з'єднує вершини з однієї групи, додаємо його до відповідного контейнеру та видаляємо з послідовно оброблюваних. На додачу до минулого прикладу, якщо ребро з'єднує вершини 1 та 2, додаємо його до першого контейнеру, де усі ребра між вершинами від 1 до 5.

Далі представимо код використання пулу потоків для паралелізації обчислень, а також використання контролю за змінами у дистанціях до вершин.

```

ExecutorService executorService = Executors.newFixedThreadPool(numberOfThreads);
List<Callable<String>> tasks = new ArrayList<>();
AtomicBoolean distUpdated = new AtomicBoolean(false);
for (int i = 1; i < graph.vertices.size(); i++) {
    distUpdated.set(false);
    tasks.clear();
    for (List<Edge> queue : adjEdges) {
        tasks.add(() -> {
            for (Edge e : queue) {
                int distBefore = graph.vertices.get(e.endId).distance;
                graph.vertices.get(e.endId).distance = Math.min(
                    graph.vertices.get(e.endId).distance,
                    graph.vertices.get(e.startId).distance + e.weight
                );

                if (distBefore != graph.vertices.get(e.endId).distance &&
!distUpdated.get()) {
                    distUpdated.set(true);
                }
            }
            return "null";
        });
    }
    executorService.invokeAll(tasks);

    for (Edge e : seqEdges) {
        int distBefore = graph.vertices.get(e.endId).distance;
        graph.vertices.get(e.endId).distance = Math.min(
            graph.vertices.get(e.endId).distance,
            graph.vertices.get(e.startId).distance + e.weight
        );
        if (distBefore != graph.vertices.get(e.endId).distance &&
!distUpdated.get()) {
            distUpdated.set(true);
        }
    }

    if (!distUpdated.get()) {
        break;
    }
}
executorService.shutdown();

```

Як можна побачити, використовується FixedThreadPool з класу Executors пакету java.util.concurrent. Створюється список задач, які імплементують інтерфейс Callable для виконання для допомогою тредпула. Запускається цикл

ітерацій по кількості вершин, але у ньому замість цикла по всіх вершинах, запускається цикл, що створює задачі, які додаються до черги тредпула, а після додавання виконуються паралельно, та очікується повне виконання всіх задач. Задача представляє собою лямбду, яка модифікує значення distance у вершин за обробки відповідних ребер. Після цього запускається послідовний цикл по вершинах, що залишились. Також можна помітити використання булевої змінної distUpdated з атомарним синхронізованим доступом. Саме вона контролює необхідність виконання наступних ітерацій, якщо вона залишається false, то наступних ітерацій не відбувається.

4.3 Тестування алгоритму

Алгоритм Беллмана-Форда стабільно виконується та гарантує обчислення найкоротшої дистанції від початкової вершини до всіх інших. Паралельна реалізація алгоритму у сенсі виконуваних операцій по суті не відрізняється від послідовної реалізації, тому паралельна реалізація також повинна гарантувати коректність проведених обчислень [3]. Для перевірки коректності влаштуємо порівняння результатів виконання, якщо вони однакові, то все працює правильно. Для автоматизації цієї перевірки створимо метод AlgoRunner.runBothCompareResults.

Його код наведено нижче:

```
public void runBothCompareResults() throws IOException, InterruptedException {
    for (String fileName: fileNames) {
        GraphReader gr = new GraphReader(fileName);
        Graph g = gr.readAsEdgeList();
        BellmanFord bf = new BellmanFord(g, 0);

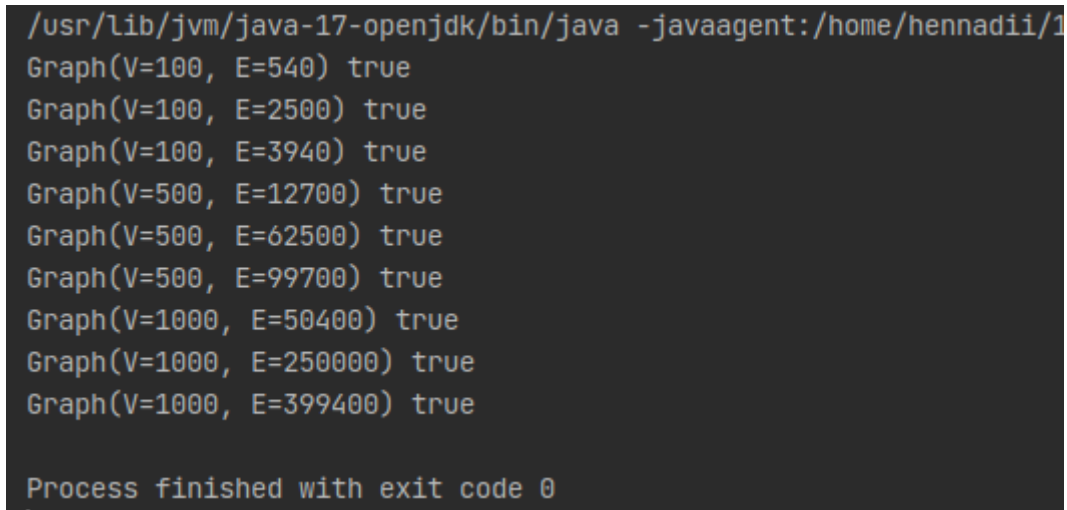
        bf.sequentialAlgorithm();
        List<Integer> seqDistances = collectVertices(g);

        bf.parallelAlgorithm(4);
        List<Integer> parDistances = collectVertices(g);

        boolean passed = seqDistances.equals(parDistances);
        System.out.println(g.info() + " " + passed);
    }
}
```

Для усіх файлів, що містять опис графу, зчитуємо їх, створюємо граф, запускаємо послідовний алгоритм, після закінчення збираємо значення отриманих дистанцій до списку, запускаємо паралельний алгоритм, збираємо його результат, і порівнюємо два результати, якщо списки однакові, то обчислення виконані однаково вірно.

Результат виконаної перевірки наведено нижче:



```
/usr/lib/jvm/java-17-openjdk/bin/java -javaagent:/home/hennadii/1
Graph(V=100, E=540) true
Graph(V=100, E=2500) true
Graph(V=100, E=3940) true
Graph(V=500, E=12700) true
Graph(V=500, E=62500) true
Graph(V=500, E=99700) true
Graph(V=1000, E=50400) true
Graph(V=1000, E=250000) true
Graph(V=1000, E=399400) true

Process finished with exit code 0
```

Рисунок 4.1 - Порівняння результатів послідовної та паралельної реалізацій

РОЗДІЛ 5. ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ (ПОРІВНЯЛЬНИЙ АНАЛІЗ ШВИДКОСТІ ОБЧИСЛЕНЬ)

5.1 Теоретична оцінка показників ефективності

Для визначення теоретичної ефективності порівняємо значення складності послідовної та паралельної реалізації.

T_1 - послідовна реалізація, один потік

T_p - паралельна реалізація, P потоків

Як вже було встановлено, складність послідовного алгоритму $T_1 = O(n^2)$, квадратична, тепер визначимо для паралельного алгоритму.

Нехай паралельна реалізація використовує P потоків, тоді потрібно визначити складність таких фрагментів паралельного алгоритму:

- розбиття усіх ребер на ті, що оброблюються паралельно, та на ті, що послідовно;
- виконання ітерацій по ребрах графу.

Для розбиття графу потрібно створити контейнери - P операцій, для поміщення ребер до контейнера виконується для кожного ребра перевірка на те, що воно підходить до контейнера, у гіршому випадку, розбиття складає $|E| * P$ операцій.

Наступний крок є суттєвим у виконанні алгоритму, відбувається у гіршому випадку $|V|$ ітерацій, хоча їх може бути менше, особливо для розріджених (sparse) графів, а саме на таких графах паралелізм даних працює найефективніше у алгоритмі Беллмана-Форда, за словами Ruijian An [1]. Число ребер дорівнює суммі тих ребер, що можуть бути оброблені паралельно та послідовно, для простоти запишемо $|E| = |parEdges| + |seqEdges|$. Паралельно виконуються $(|parEdges|/P)$ операцій, після цього виконується $|seqEdges|$ операцій. Таким чином, маємо загальний результат складності алгоритму у гіршому випадку:

$$T_p = |E| * P + |V|(|parEdges|/P + |seqEdges|)$$

Для того, щоб покращити результат виконання паралельної реалізації, потрібно мінімізувати кількість ітерацій, щоб вона не дорівнювала максимальному числу - кількості вершин, для цього використовується контроль за змінами у дистанціях вершин. А також потрібно максимізувати значення $|parEdges|$, щоб якомога більше вершин оброблялись паралельно, а послідовних вершин майже не залишалось. У такій проблемі стають у нагоді розріджені графи - графи, у яких квадрат кількості вершин значно перевищує кількість ребер, тоді ступінь паралелізації підвищується. Звісно, використання більшого числа потоків також дасть швидкість виконання.

5.2 Практична оцінка показників ефективності

Для того, щоб оцінити у реальному часі ефективність виконання паралельної реалізації алгоритму Беллмана-Форда, потрібно згенерувати графи різної розмірності, бажано, підвищеної розрідженості. Генерація графів відбувалась за допомогою самостійно створеного скрипта мовою Python із подальшим збереженням у текстовий файл списку ребер. За допомогою псевдокоду пояснимо, як відбувалась генерація:

```
procedure generate_graph(vertices_num, density) {
  init graph as empty list of edges(start, end, weight)
  for v := 0 to vertices_num - 1 do
    graph.add(edge(v, v+1, random weight))
    possible_ends = list(v+2 to vertices_num)

    for times := 0 to possible_ends.size * density do
      end_vertex = possible_ends.pop(random vertex)
      graph.add(edge(v, end_vertex, random weight))
    end
  end
}
```

Генерація графів працює таким чином, що для кожної вершини обов'язково створюється ребро, що веде до наступної за ідентифікатором

вершини, після чого від кожної вершини створюється деяке число ребер, що ведуть до випадкових вершин, що також більше за ідентифікатором. Таким чином, генератор графів створює граф у вигляді так би мовити сітки, що є наведеною від вершин з маленьким значенням ідентифікатора до вершин з відповідним більшим значенням. Параметр процедури density має значення від 0 до 1 та визначає до скількох випадкових вершин провести ребра від поточної вершини, чим менше значення цього параметру, тим більш розрідженим виходить граф.

Для порівняльного тестування обох реалізацій було створено графи з параметром density дорівнюючим 0,1 розмірністю 100, 500, 1000, 2500 вершин. Заміри проводилися на машині з 6-ядерним процесором Ryzen 7 на частоті 1.8 ГГц.

Представимо час виконання у вигляді таблиці:

Таблиця 5.1 - Час виконання у залежності від кількості потоків та вершин, мс

Реалізація	100 вершин	500 вершин	1000 вершин	2500 вершин
Послідовний алгоритм	3,2	47,0	484,2	7460,4
Паралельний, 2 потоки	3,4	32,8	197,0	6468,9
Паралельний, 4 потоки	3,4	19,2	135,0	5432,8
Паралельний, 8 потоків	3,9	15,9	95,8	2519,4
Паралельний, 16 потоків	4,3	15,8	67,6	1425,8

Як можна зрозуміти, на маленьких графах, як-от на 100 вершин, паралельна імплементація дає лише зайвий overhead у вигляді FixedThreadPool. Починаючи з графу на 500 вершин та більше, можна помітити відчутне прискорення зі збільшенням кількості потоків, що обробляють ребра. Порівнюючи швидкість послідовного алгоритму із паралельним на 16 потоків,

маємо такі значення прискорення: на 500 вершин - в 2.97 раз, на 1000 вершин - в 7.16 раз, на 2500 вершин - в 5.23 раз. Можна помітити, що зі зростанням кількості вершин від 1000 до 2500, прискорення трохи падає, це дає нам зрозуміти, що паралельна реалізація добре працює саме на графах середнього розміру (1000 - 2500 вершин) та підвищеної розрідженості. Для закріплення розуміння, наскільки прискорення є дієвим, побудуємо графік, що наглядно продемонструє це:

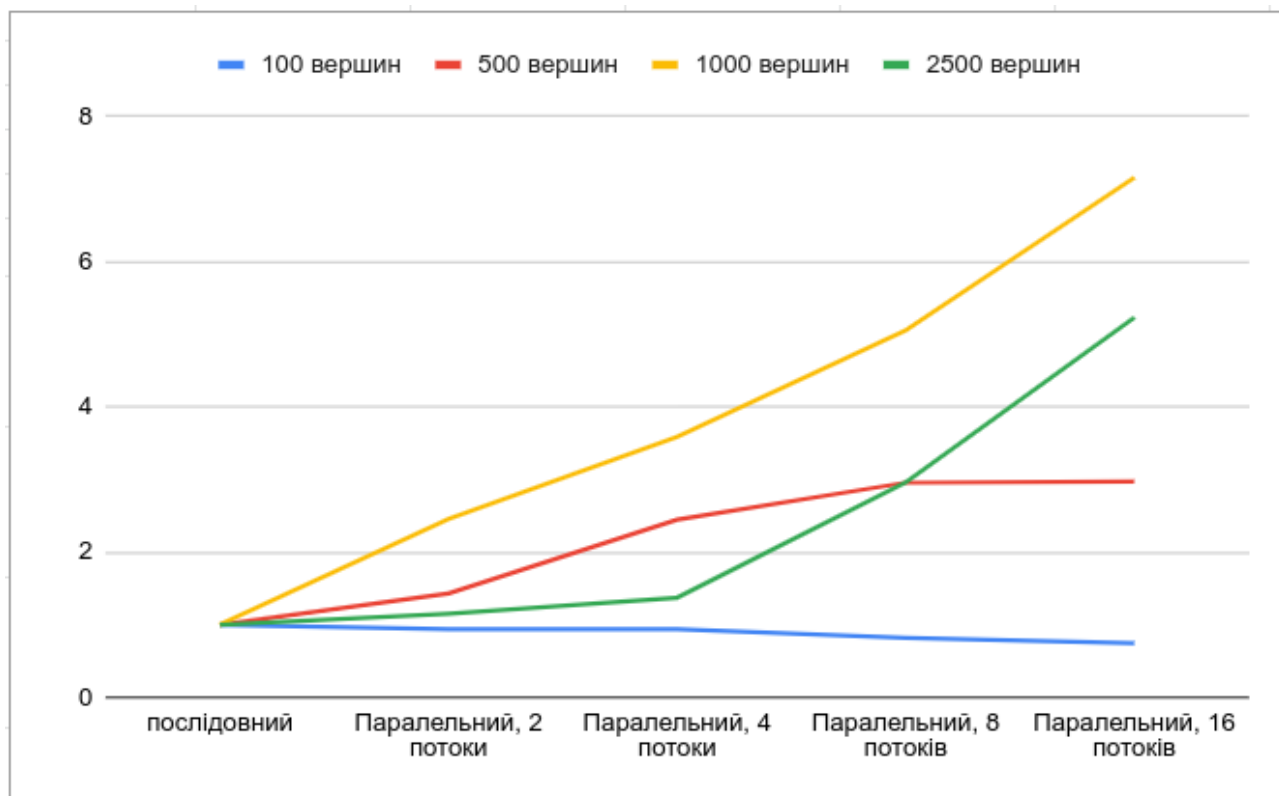


Рисунок 5.1 - Залежність прискорення від додавання потоків

Графік показує, що додавання потоків, а саме 16 потоків пришвидшило виконання алгоритму Беллмана-Форда для графу на 100 вершин, за ним наступним є граф на 2500 вершин, для 500 вершин прискорення теж було більше за 1, але не було настільки виразним, як для попередньо зазначених графів.

ВИСНОВКИ

У ході виконання курсової роботи було ґрунтовно проаналізовано обрану тему - алгоритм Беллмана-Форда на графі. Було детально описано принцип виконання та мету алгоритму, розроблено та виконано послідовну реалізацію алгоритму, оцінено її якість роботи, спроектовано та оптимізовано паралельну реалізацію алгоритму, наведено оцінку складності та прискорення, а також доведено покращення паралельної версії у порівнянні з послідовною.

Для виконання поставленої задачі було прочитано та опрацьовано декілька робіт світових науковців, що займались вирішенням проблеми паралелізації алгоритму Беллмана-Форда на графі. Було встановлено, що найбільш ефективним для цього алгоритму є використання паралелізму даних, а саме розділення ребер на групи, що можуть оброблятися паралельно.

Програмна реалізація алгоритму Беллмана-Форда у послідовному та паралельному варіантах була здійснена за допомогою мови програмування Java, зокрема пакету `java.util.concurrent`. Було детально пояснено вибір програмного забезпечення, що взятий за основу до програми.

Однією з цілей виконання курсової роботи було поставлено досягнення прискорення паралельної реалізації алгоритму щодо послідовної більше 1. Така ціль була досягнута, для розріджених графів розмірністю 1000 та 2500 вершин прискорення сягало більше ніж у 7 та 5 разів відповідно за додавання 16 потоків. Звісно, прискорення на практиці не є ідеальним та потребує подальшого обмірковування. Програма витрачає деяку кількість часу на створення пулу потоків, його обробку, та отримання результатів, не варто також забувати про недосконалість Java Thread Scheduler, що є частиною Java Virtual Machine та Java Runtime Environment, і саме тому імплементація потребує доробки та подальшої оптимізації.

Під час програмної реалізації алгоритму було закріплено вміння програмувати мовою Java, а також отримано навички паралелізації послідовних

рішень за допомогою інструментів OpenJDK 17 та конкретно пакету `java.util.concurrent`. Були освоєні поняття потоку, пула потоків, атомарного доступу до даних, синхронізації потоків, а також паралелізм даних.

Отримані значення прискорення задовольняють умовам курсової роботи та є дуже добрими для початківця у паралельному програмуванні. Але простір для покращень прискорення алгоритму та поглиблення знань є і залишається, процес оптимізації алгоритму займає деякий час та кінцевий результат завжди може бути краще. Наведемо приклади оптимізації, що можуть бути також втілені для подальшого прискорення: обробка лише тих вершин, які були змінені на попередній ітерації алгоритму, використання GPU для виконання обчислень алгоритму та більшого ступеню паралелізації тощо.

Таким чином, у процесі виконання курсової роботи було розроблено та вдосконалено паралельну реалізацію алгоритма Беллмана-Форда на графі, що відповідає вимогам та стандартам. Роботу завершено успішно.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ruijian, An. “Parallel approach to single-source shortest path problem on large complete graphs”. RUCS, University of Toronto, 2017.
2. Kulkarni, P.R. et al. “Parallel Implementation of Dijkstra’s and Bellman Ford Algorithm”. CSE Buffalo, 2014.
3. Safari, Mohsen et al. “Automated Verification of the Parallel Bellman–Ford Algorithm”. International Static Analysis Symposium, 2021.
4. Sengo, Raj. “Bellman-Ford Single Source Shortest Path Algorithm on GPU using CUDA”, Towards Data Science, 2020.
5. Gaurav, Hajela et al. “Parallel Implementations for Solving Shortest Path Problem using Bellman-Ford”. International Journal of Computer Applications, 2014. C. 2-3.
6. Ganesh, G Surve. “Parallel implementation of Bellman-ford algorithm using CUDA architecture”. IEEE, ICECA, 2017.

ДОДАТКИ

Додаток А. Програмний код

Посилання на GitHub-репозиторій:

https://github.com/genndy007/bellman_ford_rethink

Vertex.java

```
package com.github.genndy007.bellman_ford;

public class Vertex {
    public final int id;
    public int distance;

    public Vertex(int id) {
        this.id = id;
    }

    public String toString() {
        return "V_id=" + String.valueOf(id) + " dist=" + String.valueOf(distance);
    }
}
```

Edge.java

```
package com.github.genndy007.bellman_ford;

public class Edge {
    public int startId;
    public int endId;
    public int weight;

    public Edge(int startId, int endId, int weight) {
        this.startId = startId;
        this.endId = endId;
        this.weight = weight;
    }

    public String toString() {
        return "Edge startId=" + String.valueOf(startId) +
            " endId=" + String.valueOf(endId) +
            " weight=" + String.valueOf(weight);
    }
}
```

IntPair.java

```
package com.github.genndy007.bellman_ford;
```

```

public class IntPair {
    final int x;
    final int y;

    IntPair(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public String toString() {
        return "(" + this.x + ", " + this.y + ")";
    }
}

```

Graph.java

```

package com.github.genndy007.bellman_ford;

import java.util.List;

public class Graph {
    public List<Vertex> vertices;
    public List<Edge> edges;

    public Graph(List<Vertex> vertices, List<Edge> edges) {
        this.vertices = vertices;
        this.edges = edges;
    }

    public String info() {
        return "Graph(V=" + vertices.size() + ", E=" + edges.size() + ")";
    }

    @Override
    public String toString() {
        String response = "Graph with\n vertices_ids = ";
        for (Vertex v : vertices) {
            response += String.valueOf(v.id) + " ";
        }
        response += "\n edges = ";
        for (Edge e : edges) {
            response += "(" +
                String.valueOf(e.startId) + ", " +
                String.valueOf(e.endId) + ", " +
                String.valueOf(e.weight) + ") ";
        }
        return response;
    }
}

```

```

    public void printDistances() {
        String line;
        for (Vertex v : vertices) {
            line = String.valueOf(v.id) + " " + String.valueOf(v.distance);
            System.out.println(line);
        }
    }
}

```

GraphReader.java

```

package com.github.genndy007.bellman_ford;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;

public class GraphReader {
    private final String directory = "graphs/";
    private final String filePath;

    public GraphReader(String filePath) {
        this.filePath = filePath;
    }

    private BufferedReader getBufferedReader() {
        InputStream is =
this.getClass().getClassLoader().getResourceAsStream(directory + filePath);
        if (is == null) throw new IllegalArgumentException(filePath + " not found
in resources dir");
        return new BufferedReader(new InputStreamReader(is));
    }

    public Graph readAsEdgeList() throws IOException {
        BufferedReader br = getBufferedReader();
        List<Vertex> vertices = new ArrayList<>();
        List<Edge> edges = new ArrayList<>();

        String line;
        boolean firstLine = true;
        while ((line = br.readLine()) != null) {
            if (firstLine) {
                String[] splitted = line.split("\\s+");
                int verticesNum = Integer.parseInt(splitted[0]);
                for (int id = 0; id < verticesNum; id++) {
                    vertices.add(new Vertex(id));
                }
            }
        }
    }
}

```

```

        firstLine = false;
        continue;
    }

    String[] splited = line.split("\\s+");
    int startId = Integer.parseInt(splited[0]);
    int endId = Integer.parseInt(splited[1]);
    int weight = Integer.parseInt(splited[2]);

    edges.add(new Edge(startId, endId, weight));

}

return new Graph(vertices, edges);
}
}

```

BellmanFord.java

```

package com.github.genndy007.bellman_ford;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.atomic.AtomicBoolean;

public class BellmanFord {
    public Graph graph;
    public int startId = 0;
    private int MAX_EDGE_WEIGHT = 100;

    public BellmanFord(Graph graph, int startId) {
        this.graph = graph;
        this.startId = startId;
    }

    public void sequentialAlgorithm() {
        for (Vertex v : graph.vertices) {
            v.distance = Integer.MAX_VALUE - MAX_EDGE_WEIGHT;
        }
        graph.vertices.get(startId).distance = 0;

        for (int i = 1; i < graph.vertices.size(); i++) {
            for (Edge e : graph.edges) {
                graph.vertices.get(e.endId).distance = Math.min(

```

```

        graph.vertices.get(e.endId).distance,
        graph.vertices.get(e.startId).distance + e.weight
    );
    }
}

public void parallelAlgorithm(int numberOfThreads) throws InterruptedException
{
    for (Vertex v : graph.vertices) {
        v.distance = Integer.MAX_VALUE - 100;
    }
    graph.vertices.get(startId).distance = 0;

    // prepare chunks containers
    List<List<Edge>> adjEdges = new ArrayList<>();
    int chunkSize = graph.vertices.size() / numberOfThreads;
    List<IntPair> chunks = new ArrayList<>();
    for (int i = 0; i * chunkSize < graph.vertices.size(); i++) {
        chunks.add(new IntPair(i * chunkSize, i * chunkSize + chunkSize));
        adjEdges.add(new ArrayList<>());
    }

    // split edges into parallelizable and seq
    List<Edge> seqEdges = new ArrayList<>(graph.edges);
    for (Edge e : graph.edges) {
        for (int i = 0; i < chunks.size(); i++) {
            IntPair pair = chunks.get(i);
            if (e.startId >= pair.x && e.startId < pair.y && e.endId >= pair.x
&& e.endId < pair.y) {
                adjEdges.get(i).add(e);
                seqEdges.remove(e);
                break;
            }
        }
    }

    ExecutorService executorService =
Executors.newFixedThreadPool(numberOfThreads);
    List<Callable<String>> tasks = new ArrayList<>();
    AtomicBoolean distUpdated = new AtomicBoolean(false);
    for (int i = 1; i < graph.vertices.size(); i++) {
        distUpdated.set(false);
        tasks.clear();
        for (List<Edge> queue : adjEdges) {
            tasks.add(() -> {
                for (Edge e : queue) {
                    int distBefore = graph.vertices.get(e.endId).distance;
                    graph.vertices.get(e.endId).distance = Math.min(

```



```

        graph.vertices.get(e.endId).distance,
        graph.vertices.get(e.startId).distance + e.weight
    );

    if (distBefore != graph.vertices.get(e.endId).distance &&
!distUpdated.get()) {
        distUpdated.set(true);
    }
}
return "null";
});
}
executorService.invokeAll(tasks);

for (Edge e : seqEdges) {
    int distBefore = graph.vertices.get(e.endId).distance;
    graph.vertices.get(e.endId).distance = Math.min(
        graph.vertices.get(e.endId).distance,
        graph.vertices.get(e.startId).distance + e.weight
    );
    if (distBefore != graph.vertices.get(e.endId).distance &&
!distUpdated.get()) {
        distUpdated.set(true);
    }
}

if (!distUpdated.get()) {
    break;
}
}
executorService.shutdown();
}
}

```

AlgoRunner.java

```

package com.github.genndy007.bellman_ford;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class AlgoRunner {
    private final Integer TIME_TESTS = 5;
    private List<String> fileNames;

    public List<String> getFileNames() {
        return fileNames;
    }
}

```

```

public void runBothCompareResults() throws IOException, InterruptedException {
    for (String fileName : fileNames) {
        GraphReader gr = new GraphReader(fileName);
        Graph g = gr.readAsEdgeList();
        BellmanFord bf = new BellmanFord(g, 0);

        bf.sequentialAlgorithm();
        List<Integer> seqDistances = collectVertices(g);

        bf.parallelAlgorithm(4);
        List<Integer> parDistances = collectVertices(g);

        boolean passed = seqDistances.equals(parDistances);
        System.out.println(g.info() + " " + passed);
    }
}

public void runCheckTime(Integer size, Integer density, boolean isParallel,
Integer numOfThreads) throws IOException, InterruptedException {
    String fileName = "graph_v" + size + "_p0" + density + ".txt";
    GraphReader gr = new GraphReader(fileName);
    Graph g = gr.readAsEdgeList();
    BellmanFord bf = new BellmanFord(g, 0);

    System.out.println(isParallel ? "Parallel" : "Sequential");
    System.out.println("Graph V=" + size + " density=0." + density);

    List<Integer> execTimes = new ArrayList<>(TIME_TESTS);
    for (int i = 0; i < TIME_TESTS; i++) {
        System.out.println("Test " + i);
        long startTime = System.currentTimeMillis();
        if (isParallel) {
            bf.parallelAlgorithm(numOfThreads);
        } else {
            bf.sequentialAlgorithm();
        }
        long endTime = System.currentTimeMillis();
        int execTime = (int) (endTime - startTime);
        execTimes.add(execTime);
    }
    System.out.println(execTimes);
    double average = execTimes
        .stream()
        .mapToDouble(a -> a)
        .average()
        .orElse(0.0);
    System.out.println("Average time (ms): " + average);
}

```

```

    private void generateFileNames(List<Integer> graphSizes, List<Integer>
densities) {
        this.fileNames = new ArrayList<>(graphSizes.size() * densities.size());
        for (Integer size : graphSizes) {
            for (Integer density : densities) {
                String fileName = "graph_v" + size + "_p0" + density + ".txt";
                fileNames.add(fileName);
            }
        }
    }

    private List<Integer> collectVertices(Graph g) {
        List<Integer> distances = new ArrayList<>(g.vertices.size());
        for (Vertex v : g.vertices) {
            distances.add(v.distance);
        }
        return distances;
    }
}

```

Main.java

```

package com.github.genndy007.bellman_ford;

import java.io.IOException;

public class Main {
    public static void main(String[] args) throws IOException,
InterruptedException {
        AlgoRunner runner = new AlgoRunner();
        runner.runCheckTime(2500, 1, true, 16);
    }
}

```