Міністерство освіти і науки України
Львівський національний університет імені Івана Франка
Факультет прикладної математики та інформатики

Звіт

Лабораторна робота №7

Тема: «Алгоритм Прима»

з дисципліни "Паралельні та розподільні обчислення"

Виконав студент групи ПМі-31
Яцуляк Андрій

Львів 2023 р.

Мета: Для зваженого зв’язного неорієнтованого графа G, використовуючи послідовний та паралельний алгоритми Прима, з довільно заданої вершини a побудувати мінімальне кісткове дерево.

Теоретичний матеріал

Граф — це структура, що складається з набору об’єктів, у якому деякі пари об’єктів у певному сенсі «пов’язані». Об’єкти відповідають математичним абстракціям, які називаються вершинами, а кожна з пов’язаних пар вершин називається ребром. Як правило, граф зображується у вигляді діаграми як набір точок або кіл для вершин, з’єднаних лініями або кривими для ребер. Графи є одним з об’єктів вивчення дискретної математики.

Графом G = (V, E) називають сукупність двох множин: скінченої непорожньої множини V **вершин** і скінченої множини E **ребер**, які з'єднують пари вершин. Ребра зображаються невпорядкованими парами вершин (u, v).

У графі можуть бути **петлі** — ребра, що починаються і закінчуються в одній вершині, а також повторювані ребра (кратні, або паралельні). Якщо в графі немає петель і кратних ребер, то такий граф називають **простим**. Якщо граф містить кратні ребра, то граф називають **мультиграфом**.

Ребра вважаються неорієнтованими в тому сенсі, що пари (u, v) та (v,u) вважаються одним і тим самим ребром.

Зваженим називають простий граф, кожному ребру e якого приписано дійсне число w(e). Це число називають **вагою** ребра e.

Хід роботи

Завдання виконав мовою програмування Java у середовищі IntelliJ IDEA. Написав повноцінну програму для роботи з зваженими орієнтованими графами.

Задається граф списком ребер. Кожне ребро має є 3 поля: source, destination, weight.

```java
public class Graph {
    31 usages
    record Edge(int source, int destination, int weight) {...}
    19 usages
    private final int numOfVertex;
    12 usages
    private final ArrayList<Edge> edges = new ArrayList<>();
    3 usages
    private static int threadsNumber = 1;
    3 usages
    public Graph(int numOfVertex) { this.numOfVertex = numOfVertex; }
    30 usages
    public void setEdge(int source, int destination, int weight){...}
    1 usage
    public void removeEdge(int source, int destination){...}
    1 usage
    private boolean isEdge(int source, int destination){...}
    no usages
    public int getNumberOfVertices() { return numOfVertex; }
    no usages
    public int getNumberOfEdges() { return this.edges.size(); }
    1 usage
    public void fillGraph(int numberOfEdges) {...}
    18 usages
    private static class VertexDistancePair implements Comparable<VertexDistancePair> {...}
    @Override
    public String toString() {...}
    6 usages
    record ReturnObject(ArrayList<Edge> edges, int sumOfPaths) {...}
    no usages
    public static int getThreadsNumber() { return threadsNumber; }
    1 usage
    public static void setThreadsNumber(int threadsNumber) { Graph.threadsNumber = threadsNumber; }
    2 usages
    public ReturnObject PrimsAlgorithm(int startVertex) {...}
    2 usages
    public ReturnObject PrimsAlgorithmParallel(int startVertex) {...}
}
```

Робота з графами

Для розпаралелення я використав фреймворк ForkJoinPool. Основний цикл алгоритму Прима продовжує виконуватися, але тепер, замість послідовної обробки кожної вершини, алгоритм ідентифікує поточну вершину та створює паралельне завдання для обробки її сусідів. Сусіди поточної вершини обробляються одночасно. Для кожного сусіда алгоритм перевіряє, чи можна його додати в пріоритетну чергу, оновлюючи інформацію про мінімальну відстань. Однак такий підхід не завжди даватиме прискорення, особливо для великих графів, враховуючи що іноді послідовно вибрати ребра з найменшою вагою є набагато швидшим варіантом, ніж одночасна обробка

кількох вершин. Я створив зв’язний граф з 200 вершинами та 3000 неорієнтованими ребрами з випадковою вагою від 1 до 100.

```
Sequential time: 5867100 nanoseconds

Parallel time for 2 threads: 9916200 nanoseconds
Speedup: 0.5916681793428934
Efficiency: 0.2958340896714467

Parallel time for 3 threads: 6003300 nanoseconds
Speedup: 0.9773124781370246
Efficiency: 0.325770826045674 87

Parallel time for 4 threads: 4790200 nanoseconds
Speedup: 1.2248131602020793
Efficiency: 0.3062032900505198

Parallel time for 8 threads: 4542400 nanoseconds
Speedup: 1.2916299753434308
Efficiency: 0.16145374691792885

Parallel time for 16 threads: 4827200 nanoseconds
Speedup: 1.2154250911501492
Efficiency: 0.07596406819688432

Parallel time for 32 threads: 3700900 nanoseconds
Speedup: 1.5853170850333702
Efficiency: 0.04954115890729282

Parallel time for 64 threads: 4466500 nanoseconds
Speedup: 1.313578864883018
Efficiency: 0.020524669763797156
```

Отже, 2 і 3 потоки використовувати недоцільно, оскільки це лише сповільнює роботу програми. Найкращого прискорення в 1.58 вдалося досягти при 32-х потоках, але найкраща ефективність досягається при повільнішій роботи, при 2-х та 3-х потоках.

Висновок. Під час виконання лабораторної роботи я написав програму для побудови мінімального кісткового дерева у зваженому неорієнтованому графі, використовуючи алгоритм Прима (послідовний та паралельний), обчислив прискорення та ефективність для різної кількості потоків та навчився аналізувати ці дані.