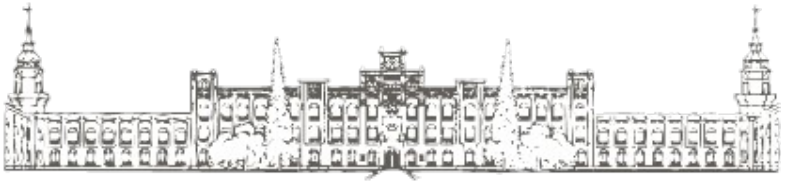


МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
КАФЕДРА ІНФОРМАТИКИ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Курсова робота з освітнього компоненту
«Технології паралельних обчислень. Курсова робота»
Тема: Алгоритм коду Прюфера та його паралельна реалізація з використанням мови Java

Керівник:
проф. Стеценко Інна В’ячеславівна

«Допущено до захисту»
_____

«___» _____ 2024 р.

Захищено з оцінкою
_____

Члени комісії:

_____

_____

Виконавець:
Прищепа Владислав Станіславович
студент групи ІП-11
залікова книжка № _____
_____

«23» травня 2024 р.

Інна СТЕЦЕНКО

Олександр Дифучин

Київ – 2024

ЗАВДАННЯ

1. Виконати огляд існуючих реалізацій алгоритму коду Прюфера, послідовних та паралельних. Зробити висновок про актуальність дослідження.

2. Виконати розробку послідовного алгоритму коду Прюфера використанням мови Java. Дослідити швидкодію алгоритму при зростанні складності обчислень та зробити висновки про необхідність паралельної реалізації алгоритму.

3. Виконати розробку паралельного алгоритму коду Прюфера з використанням мови Java.

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

5. Виконати дослідження швидкодії паралельного 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.

АНОТАЦІЯ

Пояснювальна записка до курсової роботи: 57 сторінок, 29 рисунків, 4 таблиці.

Об’єкт дослідження: задача паралельного формування коду Прюфера для згенерованого зв’язного не зацикленого графу (дерева).

Мета роботи: теоретично дослідити паралельні методи формування коду Прюфера за допомогою алгоритму коду Прюфера; переглянути відомі їхні реалізації; спроектувати, реалізувати, протестувати послідовний та паралельний алгоритми; дослідити ефективність паралелізації програми;

Виконана програмна реалізація паралельного та послідовного алгоритму кодування дерева у код, проведено аналіз їх ефективності.

Ключові слова: КОДУВАННЯ ДЕРЕВА, ЗВ’ЯЗНИЙ НЕ ЗАЦИКЛЕНИЙ ГРАФ, КОД ПРЮФЕРА, ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ.

ЗМІСТ

ВСТУП

У сучасному інформаційному суспільстві важлива роль відводиться оптимізації алгоритмів для вирішення різноманітних завдань, зокрема, задач обробки та представлення дерев. Алгоритм коду Прюфера [1] визначається як один із найефективніших та широко застосовуваних для представлення довільних дерев у компактній формі. Використання коду Прюфера виявляється актуальним у багатьох сферах, таких як біоінформатика, теорія мереж, криптографія та комбінаторика.

З появою багатоядерних процесорів та розподілених систем виникає потреба в розробці ефективних паралельних алгоритмів, спрямованих на прискорення обчислень. Саме у цьому контексті виникає ідея дослідження можливості паралельної реалізації алгоритму коду Прюфера за допомогою мови програмування Java [2]. Об'єктом даної курсової роботи є вивчення та аналіз підходів до паралельної реалізації алгоритму коду Прюфера для дерев різних розмірностей. Враховуючи особливості алгоритму, основним завданням є розробка ефективної паралельної реалізації.

Також в рамках роботи буде розглянуто та проаналізовано ефективність різних методів паралельного виконання алгоритму з урахуванням його особливостей. Окрім цього, планується розробка та аналіз нового підходу до паралельної реалізації, який сприяє швидкому кодуванню та декодуванню дерев, особливо при збільшенні розмірності вхідних даних. Отримані результати і висновки будуть важливим внеском у розуміння проблеми паралельного програмування та оптимізації алгоритмів в сучасних умовах розвитку технологій.

1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ

Алгоритм коду Прюфера достатньо складно розпаралелити, тому окрім класичної реалізації алгоритму коду Прюфера паралельну версію цього алгоритму відшукати вкрай важко. У даній роботі будуть розглянуті послідовна реалізація, а також запропонована власна варіація паралельного алгоритму.

1.1 Послідовний алгоритм коду Прюфера

Алгоритм коду Прюфера є ефективним методом для представлення дереву компактній формі. Для цього послідовно обходяться всі листи дерева від того, що з найменшим індексом, до того, що з найбільшим. Під час кожної ітерації у код Прюфера послідовно вписується індекс вершини-сусіда, а сам листок витирається із дерева. Процес повторюється, доки дерево не буде складатися лише з двох вершин. Важливо зазначити, що для даного алгоритму підходять виключно дерева, кожна вершина якого має свій унікальний індекс [3].

1.2 Паралельний алгоритм коду Прюфера

Загалом виконати паралельну реалізацію даного алгоритму важко, адже кожна наступна ітерація може бути виконана виключно після виконання попередньої ітерації [4]. Але, так, як дерево у коді задається у вигляді матриць зв’язків, то можна виконати асинхронно підрахунок сусідніх вершин для кожної вершини графа перед виконанням основної частини алгоритму. До того ж, можна спробувати розпаралелити пошук листка з наймолодшим індексом у дереві.

2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

У рамках даного розділу проводиться детальний огляд процесу розробки послідовного алгоритму коду Прюфера. Визначаються основні кроки та етапи, необхідні для створення функціонального та ефективного алгоритмічного рішення. При цьому здійснюється аналіз особливостей графових структур, що може впливати на вибір оптимальних стратегій алгоритму.

Основний акцент розділу робиться на вивченні та порівнянні часових та просторових характеристик розробленого послідовного алгоритму. Це дозволяє визначити його ефективність та потенційні області оптимізації.

2.1 Проектування послідовного алгоритму

Відповідно до теорії, описаної у розділі 1.1, було розроблено алгоритм коду Прюфера. Він використовує симетричну матрицю зв’язків для збереження поточного стану графа, де індекси матриці — індекси вершин графа; масив кількості сусідів вершини — для збереження поточних кількостей зв’язків з іншими вершинами; масив індексів — масив, у який записуватиметься праильний код Прюфера.

2.2 Реалізація послідовного алгоритму

Для відображення графа використовуватиметься квадратна матриця, де ключами будуть індекси вершин, а значеннями будуть нулі та одиниці, що означатимуть відсутність та наявність зв’язку відповідно.

Послідовний алгоритм коду Прюфера реалізований у вигляді методу graph2PruferCode. На рисунку 2.1 він створює масиви елементів коду Прюфера та кількості сусідів кожної вершини.

```
public static int[] graph2PruferCode(int[][] graph) {  1 usage
    int n = graph.length;
    int[] res = new int[n - 2];
    int[] degrees = new int[n];
```

Рисунок 2.1 — Визначення метода graph2PruferCode

У цьому методі (рис. 2.2) спочатку підраховується загальна кількість сусідів кожної вершини. Результати записуються у масив degrees у відповідну комірку для кожної вершини за індексом. Далі починається основний цикл програми, а саме до тих пір, поки масив елементів коду Прюфера res не буде заповнений, спочатку у degrees шукається вершина з найменшим індексом, яка має одного сусіда. Потім, у рядку матриці graph з відповідним індексом знайденої вершини, шукається індекс вершини-сусіда. Цей віднайдений індекс записується у кінець послідовності res, а значення у масиві degrees відшуканих вершин зменшується на 1, а у матриці graph елементи з ключами, що рівні індексам віднайдених вершин, приймають значення нуль замість одиниці.

2.3 Тестування послідовного алгоритму

Тестування алгоритму проводиться вручну. Для початку потрібно визначити метод створення графа. Правильний граф ми будемо генерувати із правильного коду Прюфера. На рисунку 2.3 можна побачити, що згенерований код Прюфера — це масив із N-2 елементів, значення яких належать [0; N).

Генерувати правильний граф ми будемо методом pruferCode2Graph із згенерованого коду Прюфера. На рисунку 2.4 можна побачити, що спочатку генеруються пуста матриця зв’язків розміром NxN та масив кількості вершин-сусідів degrees розміром N, який заповнений 1. Далі обходиться код Прюфера і кожній вершині, індекс якої присутній серед елементів коду, додається у відповідний елемент масиву degrees один стільки разів, скільки цей індекс зустрічається у коді. Далі відшукується індекс першої вершини з одним сусідом. Після цього починається основний цикл. Для кожного елемента у коді Прюфера створюється ребро між поточним листком та вершиною із вказаним у коді індексом; відповідні елементи масиву degrees зменшуються на одиницю; якщо вершина із даним індексом стала листком і її індекс менше, ніж індекс попереднього листка, то ця вершина стає наступним листком, інакше — продовжуємо з індексу попереднього листка шукати наступний листок. Після виконання залишилися лише дві вершини, які треба з’єднати. Для цього

шукаємо перший листок серед масиву degrees та об’єднуємо цю вершину із вершиною з індексом N – 1.

```
// Calculate degrees of vertices
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (graph[i][j] == 1) {
            degrees[i]++;
        }
    }
}

for (int k = 0; k < n - 2; k++) {
    int leaf = -1;
    // Find a leaf vertex
    for (int i = 0; i < n; i++) {
        if (degrees[i] == 1) {
            leaf = i;
            break;
        }
    }

    int neighbor = -1;
    // Find a neighbor of the leaf
    for (int j = 0; j < n; j++) {
        if (graph[leaf][j] == 1) {
            neighbor = j;
            break;
        }
    }

    res[k] = neighbor;
    // Update degrees and remove the edge
    degrees[neighbor]--;
    degrees[leaf]--;
    graph[leaf][neighbor] = 0;
    graph[neighbor][leaf] = 0;
}
return res;
```

Рисунок 2.2 — Реалізація методу graph2PruferCode

```
int[] code = new int[n - 2];
Random random = new Random();
for (int i = 0; i < n - 2; i++) {
    code[i] = random.nextInt(n);
}
```

Рисунок 2.3 — Генерація початкового коду Прюфера

```
public static int[][] pruferCode2Graph(int[] pruferCode) {  1 usage
    int n = pruferCode.length + 2;
    int[][] graph = new int[n][n];
    int[] degree = new int[n];
    Arrays.fill(degree,  val: 1);
    for (int v : pruferCode)
        ++degree[v];
    int ptr = 0;
    while (degree[ptr] != 1)
        ++ptr;
    int leaf = ptr;
    for (int v : pruferCode) {
        graph[leaf][v] = 1;
        graph[v][leaf] = 1;
        --degree[leaf];
        --degree[v];
        if (degree[v] == 1 && v < ptr) {
            leaf = v;
        } else {
            for (++ptr; ptr < n && degree[ptr] != 1; ++ptr);
            leaf = ptr;
        }
    }
    for (int v = 0; v < n - 1; v++) {
        if (degree[v] == 1) {
            graph[v][n - 1] = 1;
            graph[n - 1][v] = 1;
        }
    }
    return graph;
}
```

Рисунок 2.4 — Реалізація методу pruferCode2Graph

Корекність відшуканого коду перевіряється вручну шляхом попарного порівняння відповідних значень згенерованого та розрахованого кодів Прюфера (рис. 2.5). У тесті виводиться згенерований код Прюфера, відповідне дерево у вигляді матриці зв’язків, розрахований код Прюфера, час виконання розрахунку та коректність коду Прюфера.

```
System.out.println("Prufer code generated!");
System.out.println("Prufer code: " + Arrays.toString(code));

int[][] graph = pruferCode2Graph(code);
System.out.println("Tree represented as adjacency matrix:");
for (int[] row : graph) {
    System.out.println(Arrays.toString(row));
}
long startTime = System.nanoTime();
int[] ncode = graph2PruferCode(graph);
long endTime = System.nanoTime();
System.out.println("Generated Prufer code from tree: " + Arrays.toString(ncode));
long duration = (endTime - startTime);
System.out.println("Time of execution: " + duration + " ns");
boolean flag = true;
for (int i = 0; i < n - 2; i++) {
    if (code[i] != ncode[i]) {
        flag = false;
        break;
    }
}
if (flag) {
    System.out.println("Prufer code is correct!");
} else {
    System.out.println("Prufer code is incorrect!");
}
```

Рисунок 2.5 — Реалізація тестування послідовного алгоритму

На рисунку 2.6 наведено результат виконання тестування послідовного алгоритму.

```
Enter the quantity of nodes in graph:
7
Prufer code generated!
Prufer code: [2, 4, 3, 5, 0]
Tree represented as adjacency matrix:
[0, 0, 0, 0, 0, 1, 1]
[0, 0, 1, 0, 0, 0, 0]
[0, 1, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 1, 1, 0]
[0, 0, 1, 1, 0, 0, 0]
[1, 0, 0, 1, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0]
Generated Prufer code from tree: [2, 4, 3, 5, 0]
Time of execution: 5200 ns
Prufer code is correct!
```

Рисунок 2.6 — Тестування послідовного алгоритму

З рисунку 2.6 видно, що для графу із семи вершин згенерувалася послідовність Прюфера 2, 4, 3, 5, 0. По матриці зв’язків видно, що за згенерованим кодом утворився наступний граф: 1-2-4-3-5-0-6. За цим графом було розраховано наступний код Прюфера: 2, 4, 3, 5, 0. Отже, програма працює коректно.

Тепер проведемо тестування алгоритму для графів більших розмірів і наведемо залежність часу від розміру графа. Для кращої репрезентації результатів побудуємо графік, який наведений на рисунку 2.7. Як бачимо, зі збільшенням розмірності дерева час також зростає. Проведення тестування на матрицях більшої розмірності не вдається, оскільки є обмеження в оперативній пам’яті.

Таблиця 2.1 — Тестування послідовного алгоритму

Кількість вершин у дереві	Час виконання у наносекундах
10000	134183700
12000	187824700
14000	251636700
16000	326679200
18000	407850600
20000	506843900
22000	601785800
24000	719884200
26000	863100600
28000	971877300
30000	1112121200

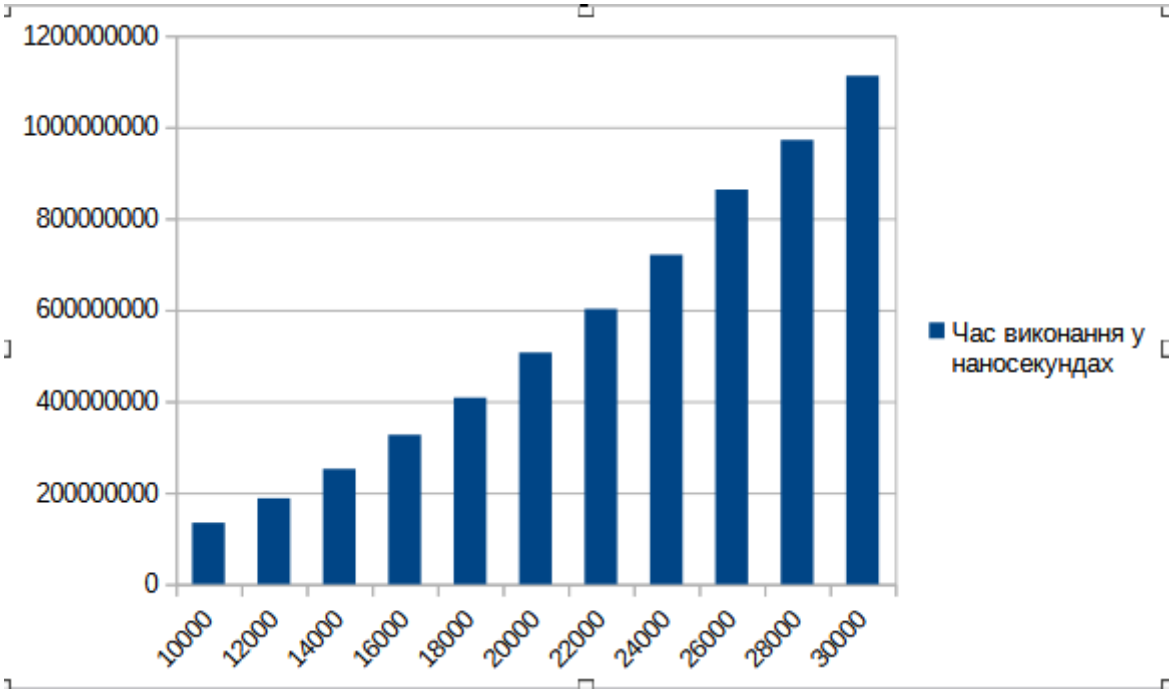


Рисунок 2.7 — Графік залежності часу від розміру графа для послідовного алгоритму

Бачимо, що різниця у часі для дерев з розмірами 10000 та 30000 відрізняється доволі суттєво, тому є сенс розпалалелити алгоритм.

2.4 Висновок

У даному розділі було проведено детально розробку алгоритму коду Прюфера. Також описали процеси тестування: створення графа, вимірювання часу виконання алгоритму від розміру графів.

Як побачили час виконання помітно зростає зі збільшенням розміру графа, тому потенціал до покращення з допомогою паралелезації існує.

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

Для реалізації паралельного алгоритму коду Прюфера я обрав Java. Вона є потужною мовою програмування, яка забезпечує високу продуктивність та надає зручні засоби для розробки багатопотокових додатків.

Java включає в стандартну бібліотеку потужні засоби паралельного програмування, які дозволяють ефективно використовувати багатоядерні процесори та підвищувати продуктивність додатків. Одним з основних інструментів є потоки (threads), які забезпечують можливість виконання декількох завдань одночасно в межах одного процесу. Потоки дозволяють реалізувати паралелізм, що сприяє швидшому виконанню складних обчислювальних задач.

Для полегшення управління потоками та іншими засобами паралельного програмування, Java включає пакет java.util.concurrent [5], який містить класи та інтерфейси для створення високопродуктивних багатопотокових додатків. Цей пакет включає інструменти для управління потоками, такі як ExecutorService, що спрощує запуск та керування потоками, а також інші утиліти, такі як інтерфейс Future.

Також варто розглянути засоби, які існують, але не будуть використані у даній роботі.

MPJ Express [6] — це потужний інструмент для паралельного програмування на Java, що реалізує стандарт MPI (Message Passing Interface). Він забезпечує розробникам можливість створювати розподілені та паралельні додатки, використовуючи знайомі концепції обміну повідомленнями. MPJ Express дозволяє ефективно виконувати обчислення на кластерних системах і суперкомп'ютерах, підтримуючи як багатопроцесорні, так і багатоядерні архітектури. Однією з переваг MPJ Express є те, що він інтегрується зі стандартним інструментарієм Java, що робить його зручним для розробників,

які вже знайомі з екосистемою цієї мови. Використання MPJ Express може значно підвищити продуктивність паралельного виконання алгоритмів, таких як алгоритм коду Прюфера, завдяки його здатності ефективно керувати обміном даними між процесами у розподілених системах. Але через простоту використання пакету java.util.concurrent та відсутність необхідності застосовувати декілька процесів, адже задача підрахунку кількості вершин сусідів та пошуку найменшого листка не є занадто складною задачою, щоб її розподіляти на декілька процесів, MPJ Express не буде використовуватися в даній роботі.

Обравши Java та використовуючи її стандартні бібліотеки, я забезпечу оптимальну реалізацію паралельного алгоритму коду Прюфера, зберігаючи баланс між продуктивністю, економією обчислювальних ресурсів та зручністю розробки.

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

У рамках даного розділу проводиться детальний огляд процесу розробки паралельного алгоритму коду Прюфера. Як і в розділі 2 визначаються основні кроки та етапи, необхідні для створення функціонального та ефективного алгоритмічного рішення. Детально розглядаються можливість розпаралелювання, вибір структур даних та оптимізаційні підходи, що можуть покращити продуктивність алгоритму. Також буде проведений аналіз результатів тестування та визначені потенційні області оптимізації для забезпечення оптимальної швидкодії паралельного алгоритму коду Прюфера.

4.1 Варіанти паралельної реалізації

4.1.1 Паралельний підрахунок кількості вершин-сусідів

З опису алгоритму в пункті 2.2 стає очевидним, що для прискорення виконання алгоритму треба розпаралелити підрахунок кількості вершин-сусідів перед виконанням основного алгоритму.

Для цього нам треба використати декілька потоків. Найпростішим буде просто взяти та для кожного рядка матриці підрахувати кількість вершин-сусідів асинхроно (рис. 4.1). В такому випадку результуючий масив буде оновлюватися у паралельному потоці.

```
int[] degreesfour = new int[n];
long startTimefour = System.nanoTime();
IntStream.range(0, n).parallel().forEach(i -> {
    for (int j = 0; j < n; j++) {
        if (graph[i][j] == 1) {
            degreesfour[i]++;
        }
    }
});
long endTimefour = System.nanoTime();
long durationfour = (endTimefour - startTimefour);
System.out.println("Time of simple parallel execution: " + durationfour + " ns");
```

Рисунок 4.1 — Реалізація простого варіанту розпаралелення підрахунку кількості вершин

Наведений варіант доволі простий, але не достатньо контрольований в плані управління потоками. Для виправлення цього недоліку, використаємо ExecutorService з фіксованою кількістю потоків (рис. 4.2).

```
int[] degreesone = new int[n];
ExecutorService executor = Executors.newFixedThreadPool(threads);

long startTimeone = System.nanoTime();
for (int i = 0; i < graph.length; i++) {
    int rowIndex = i;
    executor.submit(() -> {
        int count = 0;
        for (int j = 0; j < graph[rowIndex].length; j++) {
            if (graph[rowIndex][j] == 1) {
                count++;
            }
        }
        degreesone[rowIndex] = count;
    });
}

executor.shutdown();
try {
    executor.awaitTermination( timeout: 1, TimeUnit.MINUTES);
} catch (InterruptedException e) {
    e.printStackTrace();
}
long endTimeone = System.nanoTime();
long durationone = (endTimeone - startTimeone);
```

Рисунок 4.2 – Реалізація розпаралелення підрахунку кількості вершин з фіксованою кількістю потоків

Запропонований варіант одночасно виконує фіксовану кількість потоків, що дозволяє контролювати процес виконання алгоритму краще. Як варіант, ще буде розглянуто варіант (рис. 4.3), де за допомогою Future потоки спочатку підраховуватимуть загальну кількість вершин-сусідів, а потім результати обчислень будуть збиратися та оброблятися Future. За рахунок того, що Future обробляє виключення, які можуть виникнути під час виконання задач, це робить програму більш надійною та керованою.

```
int[] degreestwo = new int[n];
ExecutorService executortwo = Executors.newFixedThreadPool(threads);
List<Future<Integer>> futures = new ArrayList<>();
long startTimetwo = System.nanoTime();
for (int i = 0; i < graph.length; i++) {
    int rowIndex = i;
    Future<Integer> future = executortwo.submit(() -> {
        int count = 0;
        for (int j = 0; j < graph[rowIndex].length; j++) {
            if (graph[rowIndex][j] == 1) {
                count++;
            }
        }
        return count;
    });
    futures.add(future);
}

executortwo.shutdown();

try {
    for (int i = 0; i < futures.size(); i++) {
        degreestwo[i] = futures.get(i).get();
    }
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
long endTimetwo = System.nanoTime();
long durationtwo = (endTimetwo - startTimetwo);
```

Рисунок 4.3 — Реалізація розпаралелення підрахунку кількості вершин з фіксованою кількістю потоків та обробкою результатів розрахунку після завершення виконання потоків.

Кожен наступний варіант складніший та теоретично часозатратніший за попередній, але контрольованіший та надійніший. У пункті 4.2 буде проведено тестування, де серед цих запропонованих варіантів буде обраний для кінцевого паралельного алгоритму той варіант, що виконає задачу правильно за найкоротший час.

4.1.2 Паралельний пошук листка

З опису алгоритму в пункті 2.2 стає очевидним, що розпаралелити сам основний алгоритм складно. Так, як кожна ітерація основного циклу залежить від результатів виконання попередньої ітерації, то потоки мусять бути синхронізованими, а так, як кожна ітерація виконує дії над спільними даними в самому кінці, то прискорення від розпаралеленої синхронізованої версії самого алгоритму не буде.

Попри це, в кожній ітерації основного циклу спочатку виконується пошук найменшого листка у масиві кількості вершин-сусідів, а потім у рядку матриці

зв’язків із ключем рівним індексу листка шукається індекс сусіда, що по своїй суті є аналогічною дією. Пошук листка у масиві можна розпаралелити, адже пошук не потребує синхронізації потоків. Ідея (рис. 4.4) полягає у тому, щоб створити пул потоків за допомогою ForkJoinPool з фіксованою кількістю потоків, що дозволить нам ефективно використовувати потоки, адже ForkJoinPool працює за принципом «розділяй та володарюй», що означає, що вільні потоки будуть самі брати на себе не виконані підзадачі, що рівномірно розподілить кількість підзадач між потоками. Самі завдання будуть представлені у вигляді класу SearchTask, що наслідує RecursiveTask. Кожен екземпляр класу приймає індекси початкового та кінцевого елемента масиву та сам масив. Якщо кількість елементів масиву більша за 1000, то масив ділиться пополам і далі створюються екземпляри класу з цими підмасивами, інакше шукається індекс шуканої вершини з урахуванням того, що перший елемент масиву в екземплярі класу може не бути першим елементом загального масиву. Після цього повертається індекс шуканої вершини.

```
static class SearchTask extends RecursiveTask<Integer> {  5 usages
    private final int[] array;  4 usages
    private final int start;  5 usages
    private final int end;  5 usages
    private final int threshold = 1000;  1 usage

    SearchTask(int[] array, int start, int end) {  3 usages
        this.array = array;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Integer compute() {
        if (end - start <= threshold) {
            for (int i = start; i < end; i++) {
                if (array[i] == 1) {
                    return i;
                }
            }
            return -1;
        } else {
            int mid = (start + end) / 2;
            SearchTask leftTask = new SearchTask(array, start, mid);
            SearchTask rightTask = new SearchTask(array, mid, end);

            leftTask.fork();
            int rightResult = rightTask.compute();
            int leftResult = leftTask.join();

            return (leftResult != -1) ? leftResult : rightResult;
        }
    }
}

public static int parallelSearch(int[] array) {  1 usage
    ForkJoinPool pool = new ForkJoinPool( parallelism: 8);
    return pool.invoke(new SearchTask(array,  start: 0, array.length));
```

Рисунок 4.4 – Реалізація розпаралелення пошуку найменшого листка у масиві

4.2 Тестування паралельних алгоритмів

У даному розділі буде проведено тестування наведених у пункті 4.1 паралельних алгоритмів.

Проведемо за таблицею 4.1 тестування алгоритмів паралельної реалізації підрахунку кількості вершин-сусідів. На рисунках 4.5 — 4.9 зображено алгоритм тесту, а на рисунках 4.10 — 4.12 — результат.

Таблиця 4.1 — Тестування алгоритмів паралельної реалізації підрахунку кількості вершин-сусідів

Тест	Тестування алгоритмів паралельної реалізації підрахунку кількості вершин-сусідів
Номер тесту	1
Початковий стан	Маємо початкові данні
Вхідні данні	Кількість потоків 8, розмір дерева — 1000/5000/10000 вершин
Опис проведення тесту	Генеруємо код Прюфера для дерева з кількістю вершин 1000/5000/10000, потім розраховуємо саме дерево, далі проводимо послідовний та паралельні підрахунки кількостей вершин-сусідів, порівнюємо коди Прюфера послідовного та паралельних алгоритмів між собою.
Очікуваний результат	Всі три варіанти паралельного алгоритму правильно розрахують кількості вершин-сусідів.
Фактичний результат	Всі три варіанти паралельного алгоритму правильно розрахували кількості вершин-сусідів.

```java
public static void main(String[] args) {
    int n = 1000;
    int threads = 8;
    int[] code = new int[n - 2];
    Random random = new Random();
    for (int i = 0; i < n - 2; i++) {
        code[i] = random.nextInt(n);
    }
    System.out.println("Prufer code generated!");

    int[][] graph = pruferCode2Graph(code);
    int[] degrees = new int[n];
    long startTime = System.nanoTime();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (graph[i][j] == 1) {
                degrees[i]++;
            }
        }
    }
    long endTime = System.nanoTime();
    long duration = (endTime - startTime);
    //System.out.println("Time of sequential execution: " + duration + " ns");
```

Рисунок 4.5 – Реалізація послідовного підрахунку кількості вершин-сусідів для порівняння з результатами паралельних варіантів підрахунку

```java
int[] degreesone = new int[n];
ExecutorService executor = Executors.newFixedThreadPool(threads);

long startTimeone = System.nanoTime();
for (int i = 0; i < graph.length; i++) {
    int rowIndex = i;
    executor.submit(() -> {
        int count = 0;
        for (int j = 0; j < graph[rowIndex].length; j++) {
            if (graph[rowIndex][j] == 1) {
                count++;
            }
        }
        degreesone[rowIndex] = count;
    });
}

executor.shutdown();
try {
    executor.awaitTermination( timeout: 1, TimeUnit.MINUTES);
} catch (InterruptedException e) {
    e.printStackTrace();
}
long endTimeone = System.nanoTime();
long durationone = (endTimeone - startTimeone);
System.out.println("Time of controlled parallel execution: " + durationone + " ns");
boolean flag = true;
for (int i = 0; i < n; i++) {
    if (degreesone[i] != degrees[i]) {
        flag = false;
        break;
    }
}
```

Рисунок 4.6 – Реалізація паралельного підрахунку кількості вершин-сусідів з фіксованою кількістю потоків та порівняння результатів із послідовним підрахунком на коректність

```java
if (flag) {
    System.out.println("Degrees for code is correct!");
} else {
    System.out.println("Degrees for code is incorrect!");
}

int[] degreestwo = new int[n];
ExecutorService executortwo = Executors.newFixedThreadPool(threads);
List<Future<Integer>> futures = new ArrayList<>();
long startTimetwo = System.nanoTime();
for (int i = 0; i < graph.length; i++) {
    int rowIndex = i;
    Future<Integer> future = executortwo.submit(() -> {
        int count = 0;
        for (int j = 0; j < graph[rowIndex].length; j++) {
            if (graph[rowIndex][j] == 1) {
                count++;
            }
        }
        return count;
    });
    futures.add(future);
}

executortwo.shutdown();

try {
    for (int i = 0; i < futures.size(); i++) {
        degreestwo[i] = futures.get(i).get();
    }
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
```

Рисунок 4.7 – Реалізація паралельного підрахунку кількості вершин-сусідів з фіксованою кількістю потоків та обробкою результатів після завершення виконання потоків.

```java
long endTimetwo = System.nanoTime();
long durationtwo = (endTimetwo - startTimetwo);
System.out.println("Time of parallel execution: " + durationtwo + " ns");
boolean flagtwo = true;
for (int i = 0; i < n; i++) {
    if (degreestwo[i] != degrees[i]) {
        flagtwo = false;
        break;
    }
}
if (flagtwo) {
    System.out.println("Degrees for code is correct!");
} else {
    System.out.println("Degrees for code is incorrect!");
}

int[] degreesfour = new int[n];
long startTimefour = System.nanoTime();
IntStream.range(0, n).parallel().forEach(i -> {
    for (int j = 0; j < n; j++) {
        if (graph[i][j] == 1) {
            degreesfour[i]++;
        }
    }
});
long endTimefour = System.nanoTime();
long durationfour = (endTimefour - startTimefour);
System.out.println("Time of simple parallel execution: " + durationfour + " ns");
boolean flagfour = true;
for (int i = 0; i < n; i++) {
    if (degreesfour[i] != degrees[i]) {
        flagfour = false;
        break;
    }
}
```

Рисунок 4.8 – Реалізація паралельного асинхроного підрахунку кількості вершин-сусідів та порівняння результатів із оброкою результатів після завершення потоків, а потім асинхроного варіантів із послідовною реалізацією

```
    if (flagfour) {
        System.out.println("Degrees for code is correct!");
    } else {
        System.out.println("Degrees for code is incorrect!");
    }

}
```

Рисунок 4.9 – Реалізація порівняння результатів паралельного та послідовного підрахунків кількості вершин-сусідів

```
Prufer code generated!
Time of controlled parallel execution: 23941800 ns
Degrees for code is correct!
Time of parallel execution: 19595800 ns
Degrees for code is correct!
Time of simple parallel execution: 44327700 ns
Degrees for code is correct!

Process finished with exit code 0
```

Рисунок 4.10 – Результат тестування паралельних підрахунків кількості вершин-сусідів для 1000 вершин

```
Prufer code generated!
Time of controlled parallel execution: 37695000 ns
Degrees for code is correct!
Time of parallel execution: 29354800 ns
Degrees for code is correct!
Time of simple parallel execution: 37906800 ns
Degrees for code is correct!

Process finished with exit code 0
```

Рисунок 4.11 – Результат тестування паралельних підрахунків кількості вершин-сусідів для 5000 вершин

```
Prufer code generated!
Time of controlled parallel execution: 48285800 ns
Degrees for code is correct!
Time of parallel execution: 42375200 ns
Degrees for code is correct!
Time of simple parallel execution: 83150200 ns
Degrees for code is correct!

Process finished with exit code 0
```

Рисунок 4.12 – Результат тестування паралельних підрахунків кількості вершин-сусідів для 10000 вершин

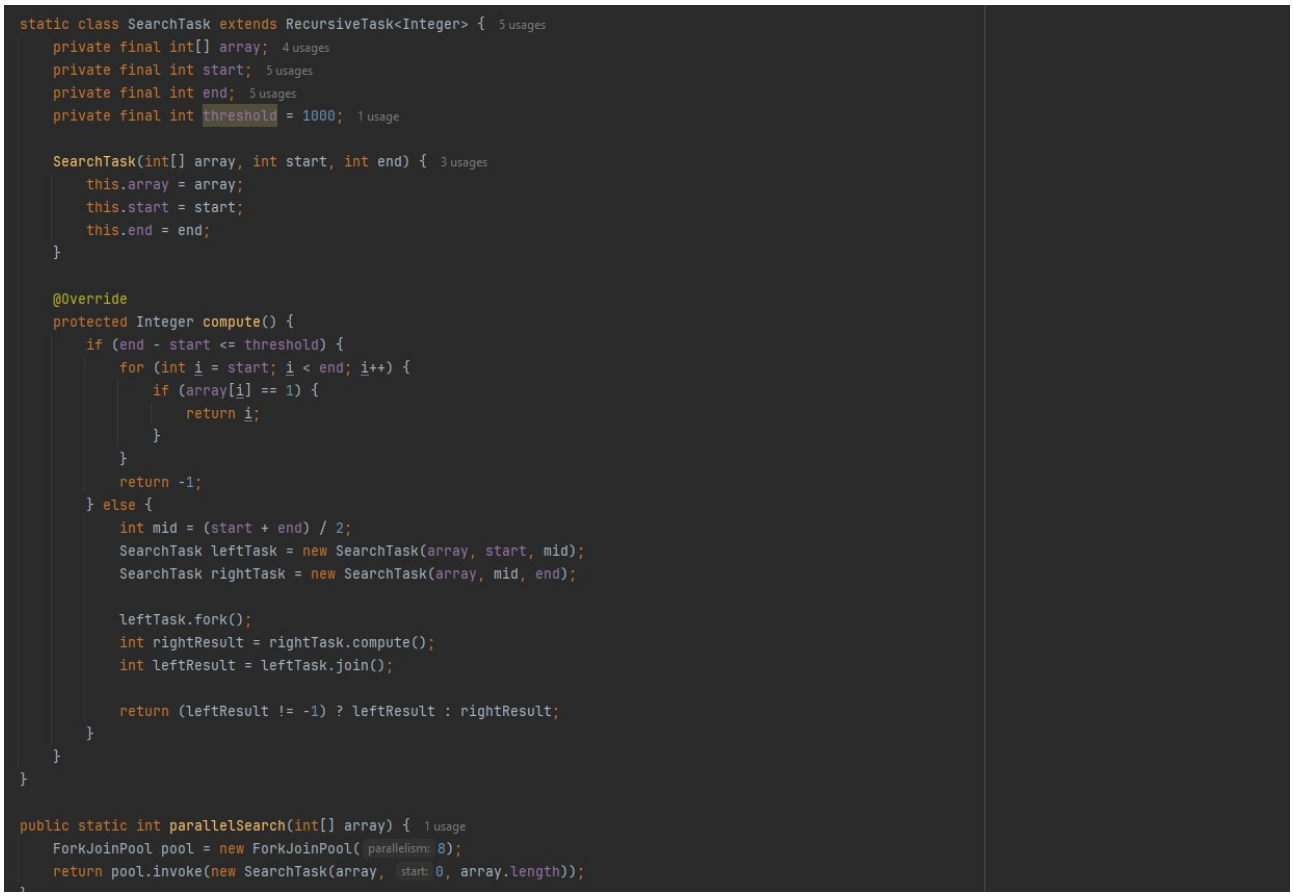
Бачимо, що всі три варіанти розпаралеленого алгоритму працюють коректно. Враховуючи те, що спочатку йде варіант з фіксованою кількістю потоків, потім — з фіксованою кількістю потоків та обробкою результатів розрахунку після завершення виконання потоків, а в кінці — простий варіант паралельного алгоритму (відповідно до рисунків 4.5 — 4.9), то виходить, що варіант з фіксованою кількістю потоків та обробкою результатів розрахунку після завершення виконання потоків є найшвидшим серед трьох запропонованих. Отже, в остаточному варіанті паралельного алгоритму коду Прюфера буде використано цей варіант алгоритму.

Проведемо за таблицею 4.2 тестування алгоритму паралельної реалізації пошуку найменшого листка дерева. На рисунках 4.13 — 4.15 зображено алгоритм тесту, а на рисунку 4.16 — результат.

Таблиця 4.2 — Тестування алгоритму паралельної реалізації пошуку найменшого листка графа

Тест	Тестування алгоритму паралельної реалізації пошуку найменшого листка графа
Номер тесту	2
Початковий стан	Маємо початкові данні
Вхідні данні	Кількість потоків 8, розмір дерева — 30000 вершин.
Опис проведення тесту	Генеруємо код Прюфера для дерева з кількістю вершин 30000, потім розраховуємо сам граф, далі проводимо послідовний підрахунок кількостей вершин-сусідів, фіксуємо час послідовного повного обходу масиву кількостей вершин-сусідів, шукаємо послідовно найменший листок у масиві, потім шукаємо найменший листок за допомогою паралельного алгоритму і фіксуємо час виконання.

Очікуваний результат	Паралельний алгоритм впорається із пошуком найменшого листка і зробить це швидше, ніж послідовний алгоритм виконає повний обхід масиву.
Фактичний результат	Паралельний алгоритм впорався із пошуком найменшого листка, але зробив це значно повільніше, ніж послідовний алгоритм виконав повний обхід масиву.



```
static class SearchTask extends RecursiveTask<Integer> {  5 usages
    private final int[] array;  4 usages
    private final int start;  5 usages
    private final int end;  5 usages
    private final int threshold = 1000;  1 usage

    SearchTask(int[] array, int start, int end) {  3 usages
        this.array = array;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Integer compute() {
        if (end - start <= threshold) {
            for (int i = start; i < end; i++) {
                if (array[i] == 1) {
                    return i;
                }
            }
            return -1;
        } else {
            int mid = (start + end) / 2;
            SearchTask leftTask = new SearchTask(array, start, mid);
            SearchTask rightTask = new SearchTask(array, mid, end);

            leftTask.fork();
            int rightResult = rightTask.compute();
            int leftResult = leftTask.join();

            return (leftResult != -1) ? leftResult : rightResult;
        }
    }
}

public static int parallelSearch(int[] array) {  1 usage
    ForkJoinPool pool = new ForkJoinPool( parallelism: 8);
    return pool.invoke(new SearchTask(array,  start: 0, array.length));
```

Рисунок 4.13 – Реалізація паралеленого пошуку найменшого листка у масиві

```
public static void main(String[] args) throws ExecutionException, InterruptedException{
    int n = 30000;
    int[] code = new int[n - 2];
    Random random = new Random();
    for (int i = 0; i < n - 2; i++) {
        code[i] = random.nextInt(n);
    }
    System.out.println("Prufer code generated!");

    int[][] graph = pruferCode2Graph(code);
    int[] degrees = new int[n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (graph[i][j] == 1) {
                degrees[i]++;
            }
        }
    }
    long startTime = System.nanoTime();
    int leaf = -1;
    for (int i = 0; i < n; i++) {
        if (degrees[i] == 1) {
            leaf = i;
        }
    }
    long endTime = System.nanoTime();
    long duration = (endTime - startTime);
    System.out.println("Time of worst sequentional execution: " + duration + " ns");
    for (int i = 0; i < n; i++) {
        if (degrees[i] == 1) {
            leaf = i;
            break;
        }
    }
```

Рисунок 4.14 – Реалізація тестування паралельного пошуку найменшого листка

```
    long startTimeOne = System.nanoTime();
    int indexOne = parallelSearch(degrees);
    long endTimeOne = System.nanoTime();
    long durationOne = (endTimeOne - startTimeOne);
    System.out.println("Time of parallel execution: " + durationOne + " ns");
    if (leaf == indexOne) {
        System.out.println("Search is correct!");
    } else {
        System.out.println("Search is incorrect!");
    }

}
```

Рисунок 4.15 – Реалізація тестування паралельного пошуку найменшого листка

```
Prufer code generated!
Time of worst sequentional execution: 94800 ns
Time of parallel execution: 2916200 ns
Search is correct!

Process finished with exit code 0
```

Рисунок 4.16 – Результат тестування паралельного пошуку найменшого листка

Видно, що паралельний алгоритм пошуку впорався із завданням, але зробив це у десятки разів повільніше, ніж послідовний алгоритм обійшов весь масив (умовно відшукав листок з найбільшим індексом, що знаходиться десь в кінці масиву). Можливо, при більшому розміру масиву паралельний алгоритм може впоратися краще, ніж послідовний, але враховуючи те, що у нас обмеження на кількість вершин у графі 30000, то такий результат паралельного алгоритму є вкрай незадовільним. Через це у остаточноиу паралельному алгоритмі коду Прюфера залишиться послідовна реалізація пошуку листка.

4.3 Остаточний паралельний алгоритм коду Прюфера

Враховуючи результати тестування в розділі 4.2, маємо наступний найефективніший паралельний алгоритм коду Прюфера на рисунках 4.16 та 4.17.

```
public static int[] graph2PruferCode(int[][] graph) {  1 usage
    int n = graph.length;
    int[] res = new int[n - 2];
    int[] degrees = new int[n];

    ExecutorService executor = Executors.newFixedThreadPool( nThreads: 8);
    List<Future<Integer>> futures = new ArrayList<>();
    for (int i = 0; i < graph.length; i++) {
        int rowIndex = i;
        Future<Integer> future = executor.submit(() -> {
            int count = 0;
            for (int j = 0; j < graph[rowIndex].length; j++) {
                if (graph[rowIndex][j] == 1) {
                    count++;
                }
            }
            return count;
        });
        futures.add(future);
    }

    executor.shutdown();

    try {
        for (int i = 0; i < futures.size(); i++) {
            degrees[i] = futures.get(i).get();
        }
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
```

Рисунок 4.16 – Реалізація паралельного алгоритму кода Прюфера

```
    for (int k = 0; k < n - 2; k++) {
        int leaf = -1;
        // Find a leaf vertex
        for (int i = 0; i < n; i++) {
            if (degrees[i] == 1) {
                leaf = i;
                break;
            }
        }

        int neighbor = -1;
        // Find a neighbor of the leaf
        for (int j = 0; j < n; j++) {
            if (graph[leaf][j] == 1) {
                neighbor = j;
                break;
            }
        }

        res[k] = neighbor;
        // Update degrees and remove the edge
        degrees[neighbor]--;
        degrees[leaf]--;
        graph[leaf][neighbor] = 0;
        graph[neighbor][leaf] = 0;
    }
    return res;
}
```

Рисунок 4.17 – Реалізація паралельного алгоритму кода Прюфера

5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ

Для усередненої оцінки ефективності алгоритмів для кожної комбінації параметрів усі тести було провдено 3 рази. Тести проводилися на деревах достатньо великого розміру, щоб можна було помітити суттєву різницю. Дерева генеруються довільні, з довільним ступенем вузлів у кожній вершині, балансованість відсутня, глибина дерева довільна. Для кожної комбінації кількості потоків та розмірів дерева було згенеровано дерево три рази і для кожного обчислено код Прюфера, а потім зафіксовано час виконання та вирахувано середнє арифметичне часу. Час для послідовної реалізації взято з таблиці 2.1. Потім було розраховане прискорення для кожної комбінації кількості потоків та розмірів дерева.

Для тестування було використано комп’ютер з наступними характеристиками:

1) ОС – Windows 10 Pro;

2) тип системи — x64;

3) процесор – AMD Ryzen 7 2700X Eight-Core Processor, 3700 МГц, 8 ядер.

У таблиці 5.1 наведено результати тестування ефективності паралельного алгоритму відносно послідовного.

Таблиця 5.1 — Тестування ефективності паралельного алгоритму відносно послідовного

Розмір графа	Послідовний, нс	Кількість потоків	Час виконання, нс	Прискорення
10000	134183700	2	129928733.3	1.032748466
		3	114323000	1.173724447
		4	112159966.7	1.19636002

		5	112282666.7	1.195052664
		6	115829133.3	1.158462437
		7	117269866.7	1.144230004
		8	125913200	1.065684138
12000	187824700	2	158463066.6	1.185290074
		3	145040300	1.294982843
		4	144465733.3	1.300133227
		5	145482833.3	1.291043732
		6	147372533.3	1.274489186
		7	149504666.7	1.256313292
		8	157467400	1.192784665
14000	251636700	2	202533733.3	1.242443399
		3	186313100	1.350611954
		4	187830533.3	1.339700716
		5	184652333.3	1.362759384
		6	184566933.3	1.363389939
		7	186281333.3	1.350842275
		8	194181133.3	1.295886453
16000	326679200	2	247239800	1.321305065
		3	228983800	1.426647649
		4	226392966.7	1.442974156
		5	222632533.3	1.467347091
		6	225372833.3	1.449505671
		7	233036933.3	1.401834445

		8	234075466.7	1.395614861
18000	407850600	2	304825666.6	1.337979851
		3	290941066.7	1.401832353
		4	275328266.7	1.481324838
		5	272936400	1.494306366
		6	275006500	1.483058037
		7	278568133.3	1.464096396
		8	279791500	1.457694748
20000	506843900	2	363394600	1.394748023
		3	339342000	1.493607924
		4	351813800	1.440659519
		5	334241100	1.516402082
		6	324441633.3	1.562203638
		7	332532400	1.524194033
		8	330618333.3	1.533018133
22000	601785800	2	430934100	1.396468277
		3	398429433.3	1.51039494
		4	390008266.7	1.543007806
		5	385488166.7	1.561100578
		6	385859633.3	1.559597709
		7	388419300	1.549320026
		8	403618266.7	1.490977613
24000	719884200	2	502796333.3	1.431761038

		3	465988033.3	1.544855551
		4	457589666.7	1.573209039
		5	452820400	1.589778641
		6	450047200	1.599574889
		7	449870600	1.600202814
		8	452475866.7	1.590989162
26000	863100600	2	584052433.3	1.477779307
		3	541175266.7	1.594863352
		4	527856966.7	1.635103171
		5	524147066.7	1.646676391
		6	521316400	1.655617587
		7	521609300	1.654687905
		8	531712766.7	1.623245959
28000	971877300	2	667371300	1.456276738
		3	620214266.7	1.56700249
		4	602983466.7	1.611781008
		5	600158866.7	1.619366728
		6	597334466.7	1.627023643
		7	601640600	1.615378517
		8	596889233.3	1.628237277
30000	1112121200	2	763730566.6	1.456169556
		3	713822866.7	1.55797923
		4	684833900	1.623928372

		5	710081100	1.566188989
		6	681808200	1.631134973
		7	680338766.7	1.634657989
		8	682667600	1.629081562

На рисунках 5.1, 5.2 показано залежність часу від розміру графа, а на рисунках 5.3 та 5.4 — прискорення від розміру графа при різній кількості потоків.

З вигляду графіків відразу можна побачити, що паралельний алгоритм коду Прюфера показує себе відмінно і дає прискорення трохи більше за 1.6 рази зі збільшенням розміру графів, хоча при розмірах графу у 10000 вершин прискорення менше 1.2. При збільшені потоків до 5 прискорення збільшується, після 5 — майже не збільшується.

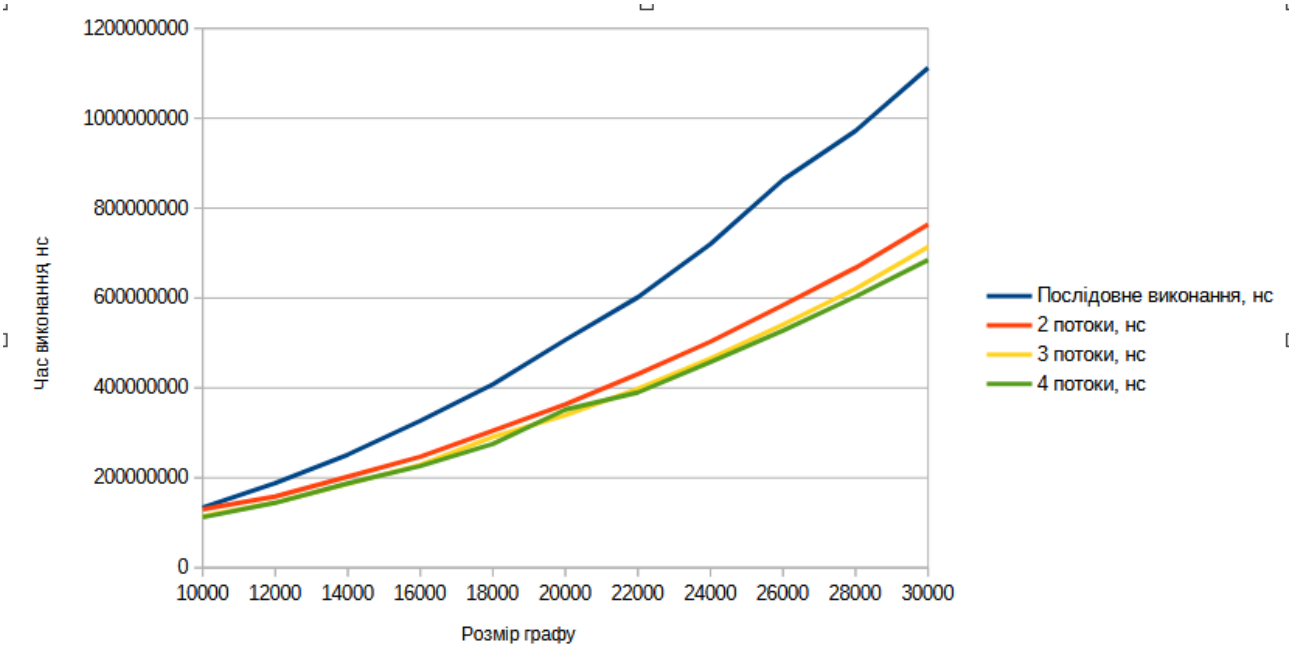


Рисунок 5.1 — Залежність часу від розміру графа для різної кількості потоків (2 - 4)

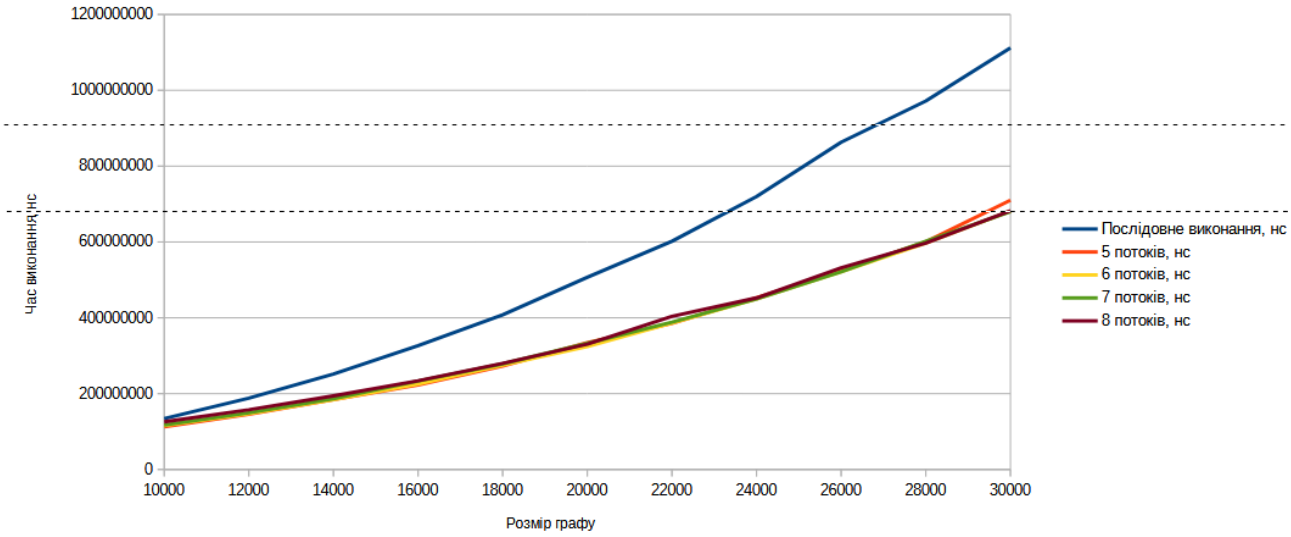


Рисунок 5.2 — Залежність часу від розміру графа для різної кількості потоків (5 - 8)

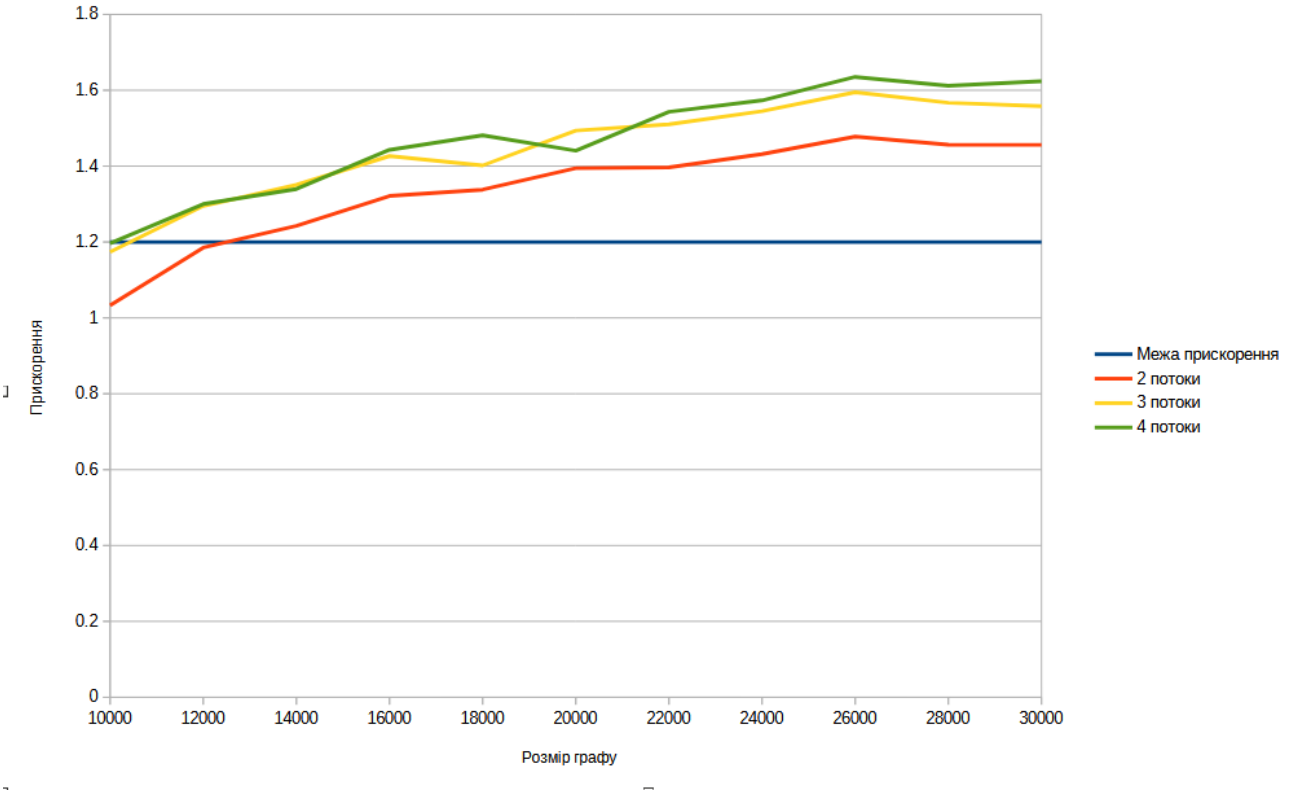


Рисунок 5.3 — Залежність прискорення від розміру графа для різної кількості потоків (2 - 4)

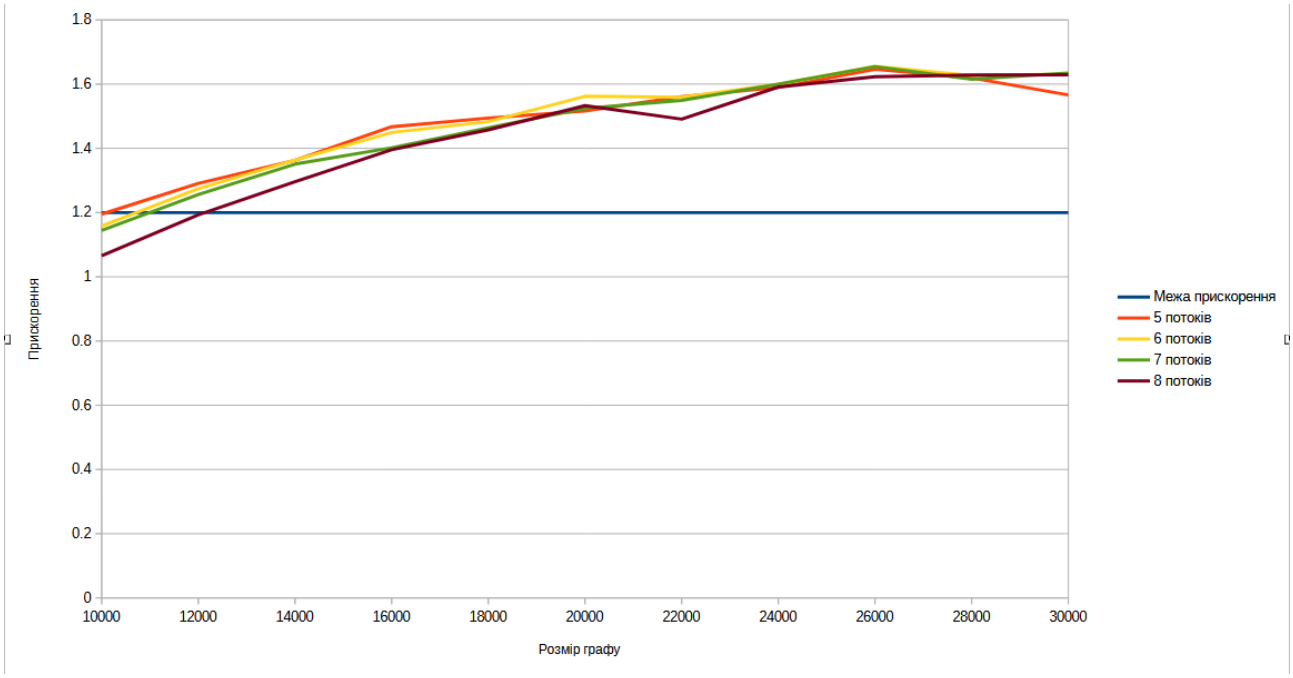


Рисунок 5.4 — Залежність прискорення від розміру графа для різної кількості потоків (5 - 8)

ВИСНОВКИ

Під час виконання курсової роботи було досліджено варіанти паралельної реалізації алгоритму кода Прюфера. Були описані основні класи, методи, описано схеми їхньої роботи. Також проведене успішне їхнє тестування, що допомогло відловити некоректну роботу і переконатися, що все працює правильно.

Наведено також результати вимірювань часу та прискорення алгоритмів у таблицях та графіках, які можна перегрянути в розділі 5.

Загалом реалізація паралельної версії виявилася успішною, і її прискорення більше за 1.2. Але ще є простір для підвищення прискорення паралельної реалізації, адже можна ще розпаралелити пошук найменшого листка у масиві.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Prüfer code [Електронний ресурс] – https://cp-algorithms.com/graph/pruefer_code.html
2. Java [Електронний ресурс] – https://dev.java/learn/
3. Java Program to Create the Prufer Code for a Tree [Електронний ресурс] – https://www.sanfoundry.com/java-program-create-prufer-code-tree/#google_vignette
4. Parallel algorithms for computing Prüfer-like codes of labeled trees [Електронний ресурс] – https://www.researchgate.net/publication/228725135_Parallel_algorithms_for_computing_Prufer-like_codes_of_labeled_trees
5. Package java.util.concurrent [Електронний ресурс] – https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html
6. MPJ Express [Електронний ресурс] – http://mpjexpress.org/

ДОДАТОК А

КОД ПРОГРАМИ

```
//.../PruferCode.java

import java.util.Arrays;
import java.util.Random;
import java.util.Scanner;

public class PruferCode {

    public static int[][] pruferCode2Graph(int[] pruferCode) {
        int n = pruferCode.length + 2;
        int[][] graph = new int[n][n];
        int[] degree = new int[n];
        Arrays.fill(degree, 1);
        for (int v : pruferCode)
            ++degree[v];
        int ptr = 0;
        while (degree[ptr] != 1)
            ++ptr;
        int leaf = ptr;
        for (int v : pruferCode) {
            graph[leaf][v] = 1;
            graph[v][leaf] = 1;
            --degree[leaf];
            --degree[v];
            if (degree[v] == 1 && v < ptr) {
                leaf = v;
            } else {
```

```
            for (++ptr; ptr < n && degree[ptr] != 1; ++ptr);
            leaf = ptr;
        }
    }
    for (int v = 0; v < n - 1; v++) {
        if (degree[v] == 1) {
            graph[v][n - 1] = 1;
            graph[n - 1][v] = 1;
        }
    }
    return graph;
}

public static int[] graph2PruferCode(int[][] graph) {
    int n = graph.length;
    int[] res = new int[n - 2];
    int[] degrees = new int[n];

    // Calculate degrees of vertices
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (graph[i][j] == 1) {
                degrees[i]++;
            }
        }
    }

    for (int k = 0; k < n - 2; k++) {
        int leaf = -1;
        // Find a leaf vertex
```

```
        for (int i = 0; i < n; i++) {
            if (degrees[i] == 1) {
                leaf = i;
                break;
            }
        }

        int neighbor = -1;
        // Find a neighbor of the leaf
        for (int j = 0; j < n; j++) {
            if (graph[leaf][j] == 1) {
                neighbor = j;
                break;
            }
        }

        res[k] = neighbor;
        // Update degrees and remove the edge
        degrees[neighbor]--;
        degrees[leaf]--;
        graph[leaf][neighbor] = 0;
        graph[neighbor][leaf] = 0;
    }
    return res;
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter the quantity of nodes in graph: ");
    int n = sc.nextInt();
```

```
        int[] code = new int[n - 2];
        Random random = new Random();
        for (int i = 0; i < n - 2; i++) {
            code[i] = random.nextInt(n);
        }
        System.out.println("Prufer code generated!");
        System.out.println("Prufer code: " + Arrays.toString(code));

        int[][] graph = pruferCode2Graph(code);
        System.out.println("Tree represented as adjacency matrix:");
        for (int[] row : graph) {
            System.out.println(Arrays.toString(row));
        }
        long startTime = System.nanoTime();
        int[] ncode = graph2PruferCode(graph);
        long endTime = System.nanoTime();
        System.out.println("Generated Prufer code from tree: " +
Arrays.toString(ncode));
        long duration = (endTime - startTime);
        System.out.println("Time of execution: " + duration + " ns");
        boolean flag = true;
        for (int i = 0; i < n - 2; i++) {
            if (code[i] != ncode[i]) {
                flag = false;
                break;
            }
        }
        if (flag) {
            System.out.println("Prufer code is correct!");
        } else {
```

```
            System.out.println("Prufer code is incorrect!");
        }
        sc.close();
    }
}

//.../CalculusTest.java

import java.util.Arrays;
import java.util.Random;
import java.util.concurrent.*;
import java.util.*;
import java.util.stream.IntStream;

public class CalcuclusTest {

    public static int[][] pruferCode2Graph(int[] pruferCode) {
        int n = pruferCode.length + 2;
        int[][] graph = new int[n][n];
        int[] degree = new int[n];
        Arrays.fill(degree, 1);
        for (int v : pruferCode)
            ++degree[v];
        int ptr = 0;
        while (degree[ptr] != 1)
            ++ptr;
        int leaf = ptr;
        for (int v : pruferCode) {
            graph[leaf][v] = 1;
            graph[v][leaf] = 1;
```

```
            --degree[leaf];
            --degree[v];
            if (degree[v] == 1 && v < ptr) {
                leaf = v;
            } else {
                for (++ptr; ptr < n && degree[ptr] != 1; ++ptr);
                leaf = ptr;
            }
        }
        for (int v = 0; v < n - 1; v++) {
            if (degree[v] == 1) {
                graph[v][n - 1] = 1;
                graph[n - 1][v] = 1;
            }
        }
        return graph;
    }

    public static void main(String[] args) {
        int n = 10000;
        int threads = 8;
        int[] code = new int[n - 2];
        Random random = new Random();
        for (int i = 0; i < n - 2; i++) {
            code[i] = random.nextInt(n);
        }
        System.out.println("Prufer code generated!");

        int[][] graph = pruferCode2Graph(code);
        int[] degrees = new int[n];
```

```
long startTime = System.nanoTime();
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (graph[i][j] == 1) {
            degrees[i]++;
        }
    }
}
long endTime = System.nanoTime();
long duration = (endTime - startTime);
//System.out.println("Time of sequential execution: " + duration + " ns");

int[] degreesone = new int[n];
ExecutorService executor = Executors.newFixedThreadPool(threads);

long startTimeone = System.nanoTime();
for (int i = 0; i < graph.length; i++) {
    int rowIndex = i;
    executor.submit(() -> {
        int count = 0;
        for (int j = 0; j < graph[rowIndex].length; j++) {
            if (graph[rowIndex][j] == 1) {
                count++;
            }
        }
        degreesone[rowIndex] = count;
    });
}

executor.shutdown();
```

```
        try {
            executor.awaitTermination(1, TimeUnit.MINUTES);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        long endTimeone = System.nanoTime();
        long durationone = (endTimeone - startTimeone);
        System.out.println("Time of controlled parallel execution: " + durationone + "
ns");
        boolean flag = true;
        for (int i = 0; i < n; i++) {
            if (degreesone[i] != degrees[i]) {
                flag = false;
                break;
            }
        }
        if (flag) {
            System.out.println("Degrees for code is correct!");
        } else {
            System.out.println("Degrees for code is incorrect!");
        }

        int[] degreestwo = new int[n];
        ExecutorService executortwo = Executors.newFixedThreadPool(threads);
        List<Future<Integer>> futures = new ArrayList<>();
        long startTimetwo = System.nanoTime();
        for (int i = 0; i < graph.length; i++) {
            int rowIndex = i;
            Future<Integer> future = executortwo.submit(() -> {
                int count = 0;
```

```
        for (int j = 0; j < graph[rowIndex].length; j++) {
            if (graph[rowIndex][j] == 1) {
                count++;
            }
        }
        return count;
    });
    futures.add(future);
}

executortwo.shutdown();

try {
    for (int i = 0; i < futures.size(); i++) {
        degreestwo[i] = futures.get(i).get();
    }
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
long endTimetwo = System.nanoTime();
long durationtwo = (endTimetwo - startTimetwo);
System.out.println("Time of parallel execution: " + durationtwo + " ns");
boolean flagtwo = true;
for (int i = 0; i < n; i++) {
    if (degreestwo[i] != degrees[i]) {
        flagtwo = false;
        break;
    }
}
if (flagtwo) {
```

```
    System.out.println("Degrees for code is correct!");
} else {
    System.out.println("Degrees for code is incorrect!");
}

int[] degreesfour = new int[n];
long startTimefour = System.nanoTime();
IntStream.range(0, n).parallel().forEach(i -> {
    for (int j = 0; j < n; j++) {
        if (graph[i][j] == 1) {
            degreesfour[i]++;
        }
    }
});
long endTimefour = System.nanoTime();
long durationfour = (endTimefour - startTimefour);
System.out.println("Time of simple parallel execution: " + durationfour + " ns");
boolean flagfour = true;
for (int i = 0; i < n; i++) {
    if (degreesfour[i] != degrees[i]) {
        flagfour = false;
        break;
    }
}
if (flagfour) {
    System.out.println("Degrees for code is correct!");
} else {
    System.out.println("Degrees for code is incorrect!");
}
```

```
    }
}

//.../SearchTest.java

import java.util.Arrays;
import java.util.Random;
import java.util.concurrent.*;

public class SearchTest {

    public static int[][] pruferCode2Graph(int[] pruferCode) {
        int n = pruferCode.length + 2;
        int[][] graph = new int[n][n];
        int[] degree = new int[n];
        Arrays.fill(degree, 1);
        for (int v : pruferCode)
            ++degree[v];
        int ptr = 0;
        while (degree[ptr] != 1)
            ++ptr;
        int leaf = ptr;
        for (int v : pruferCode) {
            graph[leaf][v] = 1;
            graph[v][leaf] = 1;
            --degree[leaf];
            --degree[v];
            if (degree[v] == 1 && v < ptr) {
                leaf = v;
            } else {
```

```
            for (++ptr; ptr < n && degree[ptr] != 1; ++ptr);
            leaf = ptr;
        }
    }
    for (int v = 0; v < n - 1; v++) {
        if (degree[v] == 1) {
            graph[v][n - 1] = 1;
            graph[n - 1][v] = 1;
        }
    }
    return graph;
}

static class SearchTask extends RecursiveTask<Integer> {
    private final int[] array;
    private final int start;
    private final int end;
    private final int threshold = 1000;

    SearchTask(int[] array, int start, int end) {
        this.array = array;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Integer compute() {
        if (end - start <= threshold) {
            for (int i = start; i < end; i++) {
                if (array[i] == 1) {
```

```
                return i;
            }
        }
        return -1;
    } else {
        int mid = (start + end) / 2;
        SearchTask leftTask = new SearchTask(array, start, mid);
        SearchTask rightTask = new SearchTask(array, mid, end);

        leftTask.fork();
        int rightResult = rightTask.compute();
        int leftResult = leftTask.join();

        return (leftResult != -1) ? leftResult : rightResult;
    }
  }
}

public static int parallelSearch(int[] array) {
  ForkJoinPool pool = new ForkJoinPool(8);
  return pool.invoke(new SearchTask(array, 0, array.length));
}



public static void main(String[] args) throws ExecutionException,
InterruptedException{
  int n = 30000;
  int[] code = new int[n - 2];
  Random random = new Random();
```

```
for (int i = 0; i < n - 2; i++) {
    code[i] = random.nextInt(n);
}
System.out.println("Prufer code generated!");

int[][] graph = pruferCode2Graph(code);
int[] degrees = new int[n];
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (graph[i][j] == 1) {
            degrees[i]++;
        }
    }
}
long startTime = System.nanoTime();
int leaf = -1;
for (int i = 0; i < n; i++) {
    if (degrees[i] == 1) {
        leaf = i;
    }
}
long endTime = System.nanoTime();
long duration = (endTime - startTime);
System.out.println("Time of worst sequentional execution: " + duration + " ns");
for (int i = 0; i < n; i++) {
    if (degrees[i] == 1) {
        leaf = i;
        break;
    }
}
```

```
        long startTimeOne = System.nanoTime();
        int indexOne = parallelSearch(degrees);
        long endTimeOne = System.nanoTime();
        long durationOne = (endTimeOne - startTimeOne);
        System.out.println("Time of parallel execution: " + durationOne + " ns");
        if (leaf == indexOne) {
            System.out.println("Search is correct!");
        } else {
            System.out.println("Search is incorrect!");
        }

    }
}

//.../ParallelPruferCode.java

import java.util.*;
import java.util.concurrent.*;

public class ParallelPruferCode {

    public static int[][] pruferCode2Graph(int[] pruferCode) {
        int n = pruferCode.length + 2;
        int[][] graph = new int[n][n];
        int[] degree = new int[n];
        Arrays.fill(degree, 1);
        for (int v : pruferCode)
            ++degree[v];
        int ptr = 0;
```

```
    while (degree[ptr] != 1)
        ++ptr;
    int leaf = ptr;
    for (int v : pruferCode) {
        graph[leaf][v] = 1;
        graph[v][leaf] = 1;
        --degree[leaf];
        --degree[v];
        if (degree[v] == 1 && v < ptr) {
            leaf = v;
        } else {
            for (++ptr; ptr < n && degree[ptr] != 1; ++ptr);
            leaf = ptr;
        }
    }
    for (int v = 0; v < n - 1; v++) {
        if (degree[v] == 1) {
            graph[v][n - 1] = 1;
            graph[n - 1][v] = 1;
        }
    }
    return graph;
}

public static int[] graph2PruferCode(int[][] graph) {
    int n = graph.length;
    int[] res = new int[n - 2];
    int[] degrees = new int[n];

    ExecutorService executor = Executors.newFixedThreadPool(8);
```

```
List<Future<Integer>> futures = new ArrayList<>();
for (int i = 0; i < graph.length; i++) {
    int rowIndex = i;
    Future<Integer> future = executor.submit(() -> {
        int count = 0;
        for (int j = 0; j < graph[rowIndex].length; j++) {
            if (graph[rowIndex][j] == 1) {
                count++;
            }
        }
        return count;
    });
    futures.add(future);
}

executor.shutdown();

try {
    for (int i = 0; i < futures.size(); i++) {
        degrees[i] = futures.get(i).get();
    }
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}

for (int k = 0; k < n - 2; k++) {
    int leaf = -1;
    // Find a leaf vertex
    for (int i = 0; i < n; i++) {
        if (degrees[i] == 1) {
```

```
                leaf = i;
                break;
            }
        }

        int neighbor = -1;
        // Find a neighbor of the leaf
        for (int j = 0; j < n; j++) {
            if (graph[leaf][j] == 1) {
                neighbor = j;
                break;
            }
        }

        res[k] = neighbor;
        // Update degrees and remove the edge
        degrees[neighbor]--;
        degrees[leaf]--;
        graph[leaf][neighbor] = 0;
        graph[neighbor][leaf] = 0;
    }
    return res;
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter the quantity of nodes in graph: ");
    int n = sc.nextInt();
    int[] code = new int[n - 2];
    Random random = new Random();
```

```
        for (int i = 0; i < n - 2; i++) {
            code[i] = random.nextInt(n);
        }
        System.out.println("Prufer code generated!");
        System.out.println("Prufer code: " + Arrays.toString(code));

        int[][] graph = pruferCode2Graph(code);
        System.out.println("Tree represented as adjacency matrix:");
        for (int[] row : graph) {
            System.out.println(Arrays.toString(row));
        }
        long startTime = System.nanoTime();
        int[] ncode = graph2PruferCode(graph);
        long endTime = System.nanoTime();
        System.out.println("Generated Prufer code from tree: " +
Arrays.toString(ncode));
        long duration = (endTime - startTime);
        System.out.println("Time of execution: " + duration + " ns");
        boolean flag = true;
        for (int i = 0; i < n - 2; i++) {
            if (code[i] != ncode[i]) {
                flag = false;
                break;
            }
        }
        if (flag) {
            System.out.println("Prufer code is correct!");
        } else {
            System.out.println("Prufer code is incorrect!");
        }
```

```
        sc.close();
    }
}
```