Національний технічний університет України «КПІ імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра автоматизованих систем обробки інформації та управління

**КУРСОВА РОБОТА**

з дисципліни «Паралельні та розподілені обчислення»

на тему: *«Алгоритм пошуку у ширину та його паралельна реалізація»*

Виконав студент 4 курсу Перевірив керівник

Група ІС-62 Стеценко Інна Вячеславівна

Ковинєв Кирило (прізвище, ім'я, по батькові)

Олексійович \_\_\_\_\_\_\_\_\_\_ підпис

\_\_\_\_\_\_\_\_\_\_ підпис Робота захищена з оцінкою

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Номер залікової книжки Члени комісії по захисту:

\_\_\_\_\_\_\_\_ІС-6210\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ підпис

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(прізвище, ім'я, по батькові)

\_\_\_\_\_\_\_\_\_\_ підпис

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(прізвище, ім'я, по батькові)

**Київ 2020**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Національний технічний університет України*  *«Київський політехнічний інститут імені Ігоря Сікорського»* | | | | | | | | | | | |
| (назва вищого навчального закладу) | | | | | | | | | | | |
| Кафедра | *АСОІУ* | | | | | | | | | | |
| Дисципліна | | *Паралельні та розподілені обчислення* | | | | | | | | | |
| Спеціальність | | | 122 Комп'ютерні науки та інформаційні технології | | | | | | | | |
| *Курс* | | | | 4 | | | Група | | ІС-62 | *Семестр* | 8 |
| **ЗАВДАННЯ**  НА КУРСОВУ РОБОТУ СТУДЕНТУ | | | | | | | | | | | |
| *Ковинєву Кирилу Олексійовичу* | | | | | | | | | | | |
| 1. Тема роботи *Алгоритм пошуку у ширину та його паралельна реалізація* | | | | | | | | | | | |
| 2. Термін здачі студентом закінченої роботи | | | | | | | | *09 квітня 2020 року* | | | |
| 3. Завдання курсової роботи | | | | | |  | | | | | |
| Розробити паралельний алгоритм пошуку у ширину та проаналізувати його ефективність | | | | | | | | | | | |
| 4. Зміст розрахунково-пояснювальної записки | | | | | | | | | | | |
| Анотація. Ключові слова. Вступ.  1. Опис алгоритму та його відомих паралельних реалізацій.  2. Вибір програмного забезпечення для розробки паралельних обчислень та його короткий опис  3. Розробка паралельних обчислень алгоритму з використанням обраного програмного забезпечення: проектування, реалізація, тестування  4 Дослідження ефективності паралельних обчислень алгоритму  Висновки. Список використаних джерел. Додатки. | | | | | | | | | | | |
|  | | | | | | | | | | | |
| 6. Дата видачі завдання | | | | | *5 лютого 2020 року* | | | | | | |

КАЛЕНДАРНИЙ ПЛАН

|  |  |  |  |
| --- | --- | --- | --- |
| № | Назва етапів курсового проекту (роботи) | Термін виконання етапів роботи | Примітка |
| 1 | Отримання індивідуального завдання на курсову роботу | 05.02.2020 |  |
| 2 | Розробка послідовного алгоритму у відповідності до завдання | 12.02.2020 |  |
| 3 | Розробка паралельного алгоритму | 19.02.2020 |  |
| 4 | Реалізація паралельного алгоритму | 04.03.2020 |  |
| 5 | Тестування паралельного алгоритму та доведення його коректності | 11.03.2020 |  |
| 6 | Експериментальне дослідження ефективності паралельного алгоритму | 18.03.2020 |  |
| 7 | Оформлення пояснювальної записки | 25.03.2020 |  |
| 8 | Захист курсового проекту (роботи) | 10.04.2020 |  |

Студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис)

Керівник \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (прізвище, ім'я, по батькові)

«09» квітня 2020 р.

# 

# **АНОТАЦІЯ**

Об’єктом дослідження курсової роботи є алгоритм пошуку у ширину. Подано інформацію про послідовний алгоритм пошуку. На основі послідовного алгоритму був побудований паралельний алгоритм пошуку. Була перевірена правильність роботи побудованого алгоритму за допомогою матриці відстаней від початкової вершини. Була проведена серія експериментів зі зростанням кількості вершин графу та кількості використаних потоків для дослідження швидкодії алгоритму. Розроблена можливість введення даних користувачем через консоль та надана інструкція для правильного введення. Зроблені висновки щодо результатів експериментів розробленої паралельної реалізації.

Програму розроблено вперше.

# **КЛЮЧОВІ СЛОВА**

Пошук у ширину, BFS, граф, вершини графу, паралельні обчислення, Java, паралельний алгоритм пошуку у ширину.

# **ВСТУП**

(Використання паралельних обчислень)[1] є дуже важливим на сьогоднішній день. Адже паралельні обчислення допомагають людству:

* моделювати складні взаємопов’язані об’єкти та системи;
* економити час у вирішенні задачі;
* збільшити об’єм виконаної роботи за проміжок часу;
* вирішувати складні практичні та теоретичні задачі;
* забезпечувати одночасність багатьох дій.

Задача пошуку усіх досяжних вершин або відстаней до них від одної конкретної вершини вирішується за допомогою одного з методів обходів графу. В нашому випадку ми будемо використовувати алгоритм (пошуку у ширину)[2] (англ. breadth-first search). За допомогою паралелізації алгоритму, ми спробуємо зекономити час у вирішенні задачі пошуку або збільшити об’єм роботи у великих графах.

# **РОЗДІЛ 1. ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ**

**Опис алгоритму**

Якщо задано граф G = (V, E) та початкову вершину s, алгоритм пошуку в ширину систематично обходить всі досяжні із s вершини. На першому кроці вершина s позначається, як пройдена, а в список додаються всі вершини, досяжні з s без відвідування проміжних вершин. На кожному наступному кроці всі поточні вершини списку відмічаються, як пройдені, а новий список формується із вершин, котрі є ще не пройденими сусідами поточних вершин списку. Для реалізації списку вершин найчастіше використовується черга та принцип (FIFO)[3]. Виконання алгоритму продовжується до досягнення шуканої вершини або до того часу, коли на певному кроці в список не включається жодна вершина. Другий випадок означає, що всі вершини, доступні з початкової, уже відмічені, як пройдені, а шлях до цільової вершини не знайдений.

**Відомі паралельні реалізації**

За допомогою сайту («Algowiki»)[4] ми можемо дізнатися, що обхід вершин кожного шару та їх сусідів може здійснюватися паралельно, проте конкретних прикладів реалізації та описів або назв алгоритмів немає.

# **РОЗДІЛ 2. РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ**

**Кроки алгоритму**

1. На першому кроці вершина s позначається, як пройдена, а в список додаються всі вершини, досяжні з s без відвідування проміжних вершин.
2. На кожному наступному кроці всі поточні вершини списку відмічаються, як пройдені, а новий список формується із вершин, котрі є ще не пройденими сусідами поточних вершин списку.

**Реалізація алгоритму на Java**

**void** BFS(**int** s)  
{  
 **boolean** visited[] = **new boolean**[**V**];  
 LinkedList<Integer> queue = **new** LinkedList<Integer>();  
 visited[s]=**true**;  
 queue.add(s); *// включення вершини s у чергу* **while** (queue.size() != 0)  
 {  
 s = queue.poll(); *// розглядання вершини, яка перебуває на початку черги* Iterator<Integer> i = **adj**[s].listIterator(); *// усі вершини, які досяжні з вершини s* **while** (i.hasNext())  
 {  
 **int** n = i.next();  
 **if** (!visited[n])  
 {  
 visited[n] = **true**; *// вершини, які досяжні з вершини s відмічаються, як пройдені* queue.add(n); *// вершини, які досяжні з вершини s додаються до черги* }  
 }  
 }  
}

**Аналіз швидкодії алгоритму**

Дивитися Розділ 5

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

Для розробки паралельних обчислень було вибрана мова програмування (Java)[5]. Java (вимовляється Джава) — об'єктно-орієнтована мова програмування, випущена 1995 року компанією «Sun Microsystems» як основний компонент платформи Java. З 2009 року мовою займається компанія «Oracle», яка того року придбала «Sun Microsystems». Мова значно запозичила синтаксис із C і C++. Зокрема, взято за основу об'єктну модель С++, проте її модифіковано.

Я вибрав цю мову, тому що:

* є досвід роботи на мові програмування Java;
* Java дуже схожа на мову програмування C#, яка є основною для мене;
* технології багатопоточності були вивчені мною саме на Java.

Консольний доданок буде розроблений у (IntellliJ IDEA)[6]. IntelliJ IDEA — комерційне інтегроване середовище розробки для різних мов програмування (Java, Python, Scala, PHP та ін.) від компанії JetBrains.

Я вибрав цю середу розробки, тому що:

* є безкоштовна версія «Community Edition»;
* зручна для використання;
* є версія для моєї операційної системи.

# **РОЗДІЛ 4. РОЗРОБКА ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЕКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ.**

**Проектування**

Ідея розпаралелювання полягає у тому, щоб кожний окремий потік отримав вершину (відмінну від інших вершин інших потоків) та виконав пошук у ширину за допомогою послідовного алгоритму. Використовуючи суспільний масив відвіданих вершин, потоки не будуть оброблювати вершини, які були оброблені іншим потоком. Використовуючи локальні черги, потоки не будуть красти вершини у інших потоків, а будуть оброблювати тільки свою ділянку графу. Виникає проблема видачі початкових вершин для потоків. Не можна задавати випадкові точки потокам, тому що виникає ймовірність, що потік буде оброблювати вершини, до яких немає шляху від початкової вершини. Для вирішення цієї задачі я використовую послідовний алгоритм пошуку у ширину до моменту, коли розмір черги не буде дорівнювати кількості потоків. Далі потоки беруть собі вершини як початкові з черги, та починають виконувати свою роботу.

**Кроки алгоритму**

1. На першому кроці вершина s позначається, як пройдена, а в список додаються всі вершини, досяжні з s без відвідування проміжних вершин.
2. На кожному наступному кроці всі поточні вершини списку відмічаються, як пройдені, а новий список формується із вершин, котрі є ще не пройденими сусідами поточних вершин списку. Якщо розмір списку >= кількості потоків, переходимо до кроку 3.
3. Для кожного потоку: вершина v, яка взята зі списку, позначається, як пройдена, а в локальний список потоку додаються всі вершини, досяжні з v без відвідування проміжних вершин.
4. На кожному наступному кроці всі поточні вершини локального списку потоку відмічаються, як пройдені, а новий локальний список формується із вершин, котрі є ще не пройденими сусідами поточних вершин списку.

*Примітка: Для одного потоку, цей алгоритм буде працювати як послідовний.*

**Реалізація**

ParallelBFSClass – клас для виконання паралельного пошуку у ширину (послідовного для одного потоку). Виконує кроки 1 та 2, та запускає виконання роботи потоків.

GraphThread – клас для виконання пошуку у ширину потоком. Виконує кроки 3 та 4.

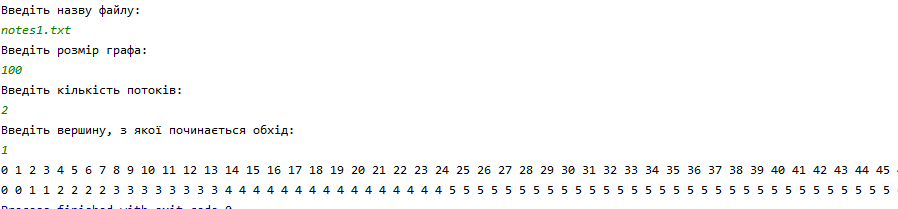
Graph – клас графу, який містить інформацію про зв’язність вершин, кількість вершин, дистанцію до інших вершин від початкової вершини.

ReadFileThread – клас для зчитування матриці суміжності графу та запису інформації до класу Graph.

Main – головний клас для роботи програми, виводить інформацію, (зчитує додаткову інформацію)[7].

**Тестування**

Для тестування вибрано граф у вигляді бінарного дерева з кількістю вершин 100. Для перевірки правильності роботи алгоритму звіримо отриману матрицю відстаней від початкової точки.



**Рисунок 4.1** Результат тестування для 2 потоків.

На рисунку 4.1 ми можемо побачити виведення матриці відстаней, яка є вірною для нашого графа (для кожної вершини є дві дочірні вершини, 0 не має зв’язків, 1 – початкова вершина).

Можемо зробити висновок, що паралельний пошук в ширину працює правильно.

# **РОЗДІЛ 5. ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ**

Для ефективного дослідження залишимо лише виведення до консолі часу роботи алгоритму.

Кількість потоків: 1, 2, 4, 8.

Кількість вершин у графі: 100, 200, 500, 1000, 2000.

Час вимірюється у наносекундах.

**Таблиця 5.1**

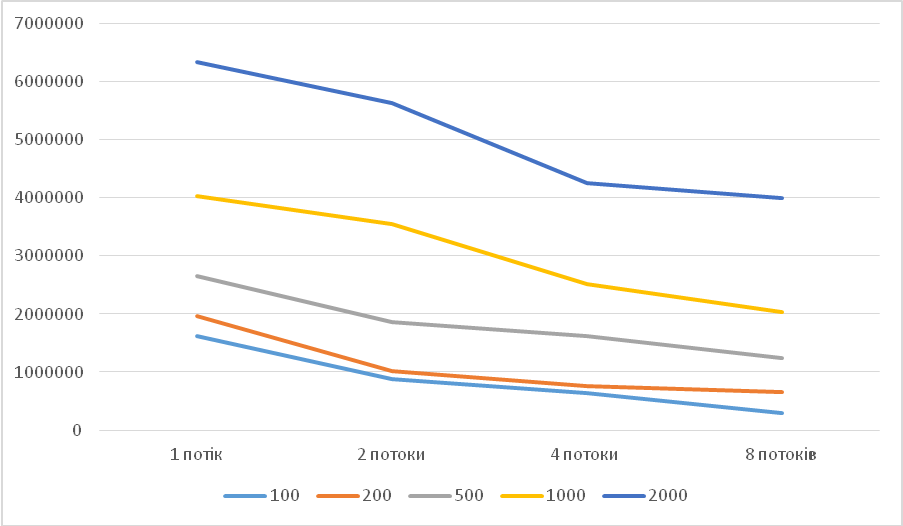
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **100** | **200** | **500** | **1000** | **2000** |
| **1** | 1619200 | 1960200 | 2651199 | 4025899 | 6350900 |
| **2** | 882200 | 1009999 | 1869850 | 3542949 | 5643450 |
| **4** | 642225 | 753275 | 1619925 | 2513924 | 4263175 |
| **8** | 290649 | 654975 | 1239737 | 2041437 | 3996900 |

Результати роботи експериментів

**Таблиця 5.2**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **100** | **200** | **500** | **1000** | **2000** |
| **1** | 1 | 1 | 1 | 1 | 1 |
| **2** | 1,835411 | 1,940794 | 1,417867 | 1,136313 | 1,125358 |
| **4** | 2,521235 | 2,602237 | 1,636618 | 1,60144 | 1,489711 |
| **8** | 5,570981 | 2,992786 | 2,138517 | 1,972091 | 1,588956 |

Прискорення порівняно з послідовним алгоритмом



**Рисунок 5.1** Графік часу виконання роботи потоками



**Рисунок 5.2** Діаграма прискорення роботи паралельного алгоритму

Так як прискорення роботи паралельного алгоритму >1, можу зробити висновок, що алгоритм працює правильно. Зі збільшенням кількості потоків ми можемо бачити зменшення часу роботи.

# **ВИСНОВКИ**

У даній курсовій роботі був розроблений паралельний алгоритм пошуку у ширину. Він вирішує задачу пошуку досяжних вершин та відстаней до них від початкової вершини. Завдяки паралельній реалізації ми можемо:

* виконати пошук швидше: наприклад пошук у графі, де 100 вершин. 2 потоки справляться з задачею за 882000 наносекунд, коли 1 потік буде виконувати її 1619000 наносекунд
* виконати пошук у більшому графі: коли 1 потік виконує пошук у графі, де 100 вершин, два потоки можуть виконати цю ж задачу у графі, де 200 вершин.

Для цього алгоритму було проведено тестування, де ми перевірили матрицю відстаней для графу. Завдяки тестуванню та зробленим експериментам зі зміною кількості потоків та вершин графа, я можу зробити висновок, що алгоритм працює.

# **СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ**

1. Електронний репозитарій ДВНЗ «УжНУ» [Електронний ресурс], 2020 – Режим доступу: https://dspace.uzhnu.edu.ua/jspui/bitstream/lib/16315/1/Технології%20розподілених%20систем%20та%20паралельних%20обчислень.%20Лекції.pdf
2. Wikipedia [Електронний ресурс], 2020 – Режим доступу:   
   https://uk.wikipedia.org/wiki/Пошук\_у\_ширину, вільний.
3. Foxford [Електронний ресурс], 2020 – Режим доступу:  
   https://foxford.ru/wiki/informatika/ochered, вільний.
4. Algowiki [Електронний ресурс], 2020 – Режим доступу:   
   https://algowiki-project.org/ru/Поиск\_в\_ширину\_(BFS), вільний.
5. Wikipedia [Електронний ресурс], 2020 – Режим доступу:   
   https://uk.wikipedia.org/wiki/Java, вільний.
6. Wikipedia [Електронний ресурс], 2020 – Режим доступу:   
   https://uk.wikipedia.org/wiki/IntelliJ\_IDEA, вільний.
7. Javarush [Електронний ресурс], 2020 – Режим доступу:   
   https://javarush.ru/quests/lectures/questsyntax.level03.lecture07, вільний.

# **ДОДАНОК А (ЛІСТИНГ КОДУ)**

Main

**import** java.io.IOException;  
**import** java.util.Iterator;  
**import** java.util.Scanner;  
  
**public class** Main {  
 **public static void** main(String args[])  
 {  
  
 Scanner scanner = **new** Scanner(System.***in***);  
  
 System.***out***.println(**"Введіть назву файлу:"**);  
 String name = scanner.nextLine(); *// назва файлу з матрицею суміжності* System.***out***.println(**"Введіть розмір графа:"**);  
 **int** v = scanner.nextInt(); *// розмір графа* System.***out***.println(**"Введіть кількість потоків:"**);  
 GraphThread.*thrNumb* = scanner.nextInt(); *// кількість потоків* System.***out***.println(**"Введіть вершину, з якої починається обхід:"**);  
 **int** s = scanner.nextInt(); *// вершина, з якої починається обхід* Graph g=**null**;  
 **try** {  
 g=ReadFileClass.*ReadFile*(name, v); *// зчитання матриці в граф g* } **catch** (IOException e) {  
  
 }  
 ParallelBFSClass.*ParallelBFS*(g,s);  
*// long startTime = System.nanoTime();  
// g.BFS(1);  
// long timeSpent = System.nanoTime() - startTime;  
// System.out.println(timeSpent);* System.***out***.println(**"Середній час виконання потоків: "**);  
 System.***out***.println(GraphThread.*sumTime*/GraphThread.*thrNumb*);  
 System.***out***.println(**"Максимальний час виконання потоку : "**);  
 System.***out***.println(GraphThread.*max*);  
 }  
}

ReadFileClass

**import** java.io.BufferedReader;  
**import** java.io.FileNotFoundException;  
**import** java.io.FileReader;  
**import** java.io.IOException;  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
  
**public class** ReadFileClass {  
 **private static int** *n*;  
 **public static int** [][] *myArr*;  
 **public static** Graph *g* = **new** Graph(*n*);  
  
 **public static** Graph ReadFile(String name, **int** numb) **throws** IOException {  
 *n*=numb;  
 *myArr* = **new int**[*n*][*n*];  
 *g* = **new** Graph(*n*);  
  
 BufferedReader br = **new** BufferedReader(**new** FileReader(name)); *// читання вмісту файлу в список рядків* List<String> lines = **new** ArrayList<>();  
 **while** (br.ready()) {  
 lines.add(br.readLine());  
 }  
  
 **for** (**int** i = 0; i < *n*; i++) {  
 **for** (**int** j = 0; j < *n*; j++) {  
 String[] line = lines.get(i).split(**" "**); *// поділ рядків по пропуску  
 myArr*[i][j] = Integer.*parseInt*(line[j]); *// запис значень в матрицю суміжності* }  
 }  
  
 **for** (**int** i = 0; i < *n*; i++) {  
 **for** (**int** j = 0; j < *n*; j++) {  
 **if**(*myArr*[i][j]==1)  
 {  
 *g*.addEdge(i,j); *// додавання вершин в граф* }  
  
 }  
 }  
 **return** *g*;  
 }  
}

Graph

**import** java.util.Iterator;  
**import** java.util.LinkedList;  
**import** java.util.concurrent.BlockingQueue;  
**import** java.util.concurrent.LinkedBlockingDeque;  
  
**class** Graph {  
 **public int V**; *// кількість вершин в графі* **public** LinkedList<Integer> **adj**[]; *// список масивів пов'язаних вершин* **public** BlockingQueue<Integer> **queue**;  
 **public boolean visited**[]; *// масив відвіданих вершин* **public int distance**[]; *// масив дистанцій від початкової точки* Graph(**int** v) {  
 **V** = v;  
 **adj** = **new** LinkedList[v];  
 **for** (**int** i = 0; i < v; ++i)  
 **adj**[i] = **new** LinkedList();  
 **visited**= **new boolean**[**this**.**V**];  
 **queue** = **new** LinkedBlockingDeque<>();  
 **distance** = **new int**[**V**];  
 }  
  
 **void** BFS(**int** s)  
 {  
 **boolean** visited[] = **new boolean**[**V**];  
 LinkedList<Integer> queue = **new** LinkedList<Integer>();  
 visited[s]=**true**;  
 queue.add(s); *// включення вершини s у чергу* **while** (queue.size() != 0)  
 {  
 s = queue.poll(); *// розглядання вершини, яка перебуває на початку черги* Iterator<Integer> i = **adj**[s].listIterator(); *// усі вершини, які досяжні з вершини s* **while** (i.hasNext())  
 {  
 **int** n = i.next();  
 **if** (!visited[n])  
 {  
 visited[n] = **true**; *// вершини, які досяжні з вершини s відмічаються, як пройдені* queue.add(n); *// вершини, які досяжні з вершини s додаються до черги* }  
 }  
 }  
 }  
  
 **void** addEdge(**int** v, **int** w) {  
 **adj**[v].add(w);  
 } *// функція додавання зв'язку між вершинами*}

ParallelBFSClass

**import** java.util.Iterator;  
  
**public class** ParallelBFSClass {  
 **public static void** ParallelBFS(Graph g, **int** s)  
 {  
 GraphThread[] parThreads = **new** GraphThread[GraphThread.*thrNumb*]; *// масив потоків* **int** [] arrS = **new int**[GraphThread.*thrNumb*]; *// масив вершин, які будуть видані потокам  
 // робота послідовного алгоритму до моменту розпаралелювання* g.**visited**[s] = **true**;  
 g.**distance**[s]=0;  
 g.**queue**.add(s);  
 **while** (g.**queue**.size() < GraphThread.*thrNumb*) *// поки кількість вершин в черзі послідовного алгоритму лише дорівнюватиме кількості потоків* {  
 s = g.**queue**.poll();  
  
 Iterator<Integer> i = g.**adj**[s].listIterator();  
 **while** (i.hasNext()) {  
 **int** n = i.next();  
 **if** (!g.**visited**[n]) {  
 g.**visited**[n] = **true**;  
 g.**distance**[n]=g.**distance**[s]+1;  
 g.**queue**.add(n);  
 }  
 }  
 }  
 **for**(**int** i=0;i<GraphThread.*thrNumb*;i++)  
 {  
 arrS[i]=g.**queue**.poll();  
 }  
 **for**(**int** i=0;i<GraphThread.*thrNumb*;i++) *// розпаралелювання пошуку в ширину* {  
 parThreads[i]=**new** GraphThread(g,arrS[i]);  
 parThreads[i].start();  
 }  
  
 **for**(GraphThread thr : parThreads)  
 {  
 **try** {  
 thr.join();  
 } **catch** (InterruptedException e) {  
  
 }  
 }  
 }  
}

GraphThread

**import** java.util.Iterator;  
**import** java.util.concurrent.BlockingQueue;  
**import** java.util.concurrent.LinkedBlockingDeque;  
  
**public class** GraphThread **extends** Thread{  
  
 **public static int** *thrNumb*; *// кількість потоків* **private int s**; *// вершина, з якої починається обхід* **private** Graph **graph**; *// граф для пошуку в ширину* **public** BlockingQueue<Integer> **localQueue**; *// локальна черга для кожного потоку* **public static int** *max*=0;  
 **public static int** *sumTime*=0;  
  
 **public** GraphThread(Graph graph, **int** s)  
 {  
 **this**.**graph**=graph;  
 **this**.**s**=s;  
 **this**.**graph**.**visited**[s] = **true**;  
 **localQueue** = **new** LinkedBlockingDeque<>() ;  
 }  
  
  
 @Override  
 **public void** run()  
 {  
 **long** startTime = System.*nanoTime*(); *// змінна для вимірювання часу работі кожного потоку* **graph**.**visited**[**s**] = **true**;  
 **localQueue**.add(**s**);  
 **while** (**localQueue**.size() != 0)  
 {  
 **s** = **localQueue**.poll();  
 Iterator<Integer> i = **graph**.**adj**[**s**].listIterator();  
 **while** (i.hasNext()) {  
 **int** n = i.next();  
 **if** (!**graph**.**visited**[n]) {  
 **graph**.**visited**[n] = **true**;  
 **graph**.**distance**[n]=**graph**.**distance**[**s**]+1;  
 **localQueue**.add(n);  
 }  
 }  
 }  
 **long** timeSpent = System.*nanoTime*() - startTime;  
 **if**(*max*<timeSpent)  
 *max*=(**int**)timeSpent;  
 *sumTime*+=timeSpent;  
*// System.out.println("Программа выполнялась " + timeSpent + " наносекунд");* }  
}

# **ДОДАНОК Б (ІНФОРМАЦІЯ ПРО ВВЕДЕННЯ ДАНИХ)**

1. Після повідомлення «Введіть назву файлу:», Вам потрібно ввести повну назву файлу разом зі шляхом до нього та його розширенням, та натиснути клавішу «Enter». Файл повинен містити матрицю суміжності Вашого графу.
2. Після повідомлення «Введіть розмір графу:», Вам потрібно ввести кількість вершин Вашого графу, яка є цілим додатним числом.
3. Після повідомлення «Введіть кількість потоків:», Вам потрібно ввести кількість потоків, які будуть здійснювати пошук у ширину. Кількість повинна бути цілим додатнім числом.
4. Після повідомлення «Введіть вершину, з якої починається обхід», Вам потрібно ввести номер вершини, яка належить вашому графу. Номер повинен бути цілим додатнім числом.

Приклад:

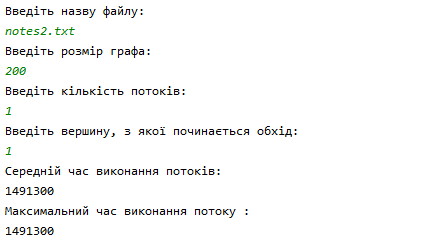


Рисунок 6.1 Приклад введеня інформації у консольний доданок

# **ДОДАНОК В (ІНФОРМАЦІЯ ПРО ТЕКСТОВІ ФАЙЛИ)**

Експерименти проводилися за допомогою графів у вигляді бінарних дерев, інформація про яких (матриці суміжностей) зберігалася у текстових файлах. Ці текстові файли додаються до курсової роботи, для можливості самостійної перевірки роботи пошуку алгоритму на таких графах.

* notes1.txt – матриця суміжності для графу, з кількістю вершин 100
* notes2.txt – матриця суміжності для графу, з кількістю вершин 200
* notes3.txt – матриця суміжності для графу, з кількістю вершин 500
* notes4.txt – матриця суміжності для графу, з кількістю вершин 1000
* notes5.txt – матриця суміжності для графу, з кількістю вершин 2000