

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 Дейкстри пошуку найкоротшого шляху та його паралельна реалізація мовою C#

|  |  |
| --- | --- |
| **Керівник**:  асистент Дифучина О.Ю.  «Допущено до захисту»  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  «\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2023 р.  Захищено з оцінкою  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Члени комісії:  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | **Виконавець**:  Чапча Святослав  Олександрович  студент групи ІТ-04  залікова книжка № ІТ0425  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  «7» травня 2023 р.  Інна СТЕЦЕНКО  Олександра ДИФУЧИНА |

**Київ – 2023**

**ЗАВДАННЯ**

1. Виконати огляд існуючих реалізацій алгоритму найкоротшого шляху Дейкстри, послідовних та паралельних, з відповідними посиланнями на джерела інформації (статті, книги, електронні ресурси). Зробити висновок про актуальність дослідження.
2. Виконати розробку послідовного алгоритму у відповідності до варіанту завдання та обраного програмного забезпечення для реалізації. Дослідити швидкодію алгоритму при зростанні складності обчислень та зробити висновки про необхідність паралельної реалізації алгоритму.
3. Виконати розробку паралельного алгоритму у відповідності до варіанту завдання та обраного програмного забезпечення для реалізації. Забезпечити зручне введення даних для початку обчислень.
4. Виконати тестування алгоритму, що доводить коректність результатів обчислень.
5. Виконати дослідження швидкодії алгоритму при зростанні кількості даних для обчислень.
6. Виконати експериментальне дослідження прискорення розробленого алгоритму при зростанні кількості даних для обчислень. Реалізація алгоритму вважається успішною, якщо прискорення більше 1,2.
7. Зробити висновки про переваги паралельної реалізації обчислень для алгоритму, що розглядається у курсовій роботі, та програмних засобів, які використовувались.

**РЕФЕРАТ**

Курсова робота спрямована на дослідження алгоритму пошуку найкоротшого шляху Дейкстри та його паралельної реалізації мовою С# з метою дослідження ефективності розпаралелювання процесів.

Курсової робота зосереджується на розробці успішних і потужних алгоритмів паралельних обчислень з використанням технологій багатопотокових та розподілених обчислень.

Основним об'єктом дослідження курсової є алгоритм Дейкстри пошуку найкоротшого шляху, послідовна та паралельна реалізації алгоритму.

Під час виконання курсової роботи буде приділена увага не лише розробці послідовного і паралельного алгоритму, але й експериментальному дослідженню ефективності паралельного алгоритму, тестуванню та підтвердженню його коректності.

Ключові слова: С#, алгоритм Дейкстри, графи, багатопотоковість, паралельність.

ЗМІСТ

[ВСТУП 5](#_Toc134531215)

[1 ОПИС ПОСЛІДОВНОГО АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ 6](#_Toc134531216)

[1.1 Опис послідовного алгоритму 6](#_Toc134531217)

[1.2 Розпаралелювання процесу переобчислення сусідніх вершин 9](#_Toc134531218)

[2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ 11](#_Toc134531219)

[3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС 14](#_Toc134531220)

[4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЕКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ 16](#_Toc134531221)

[5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ 19](#_Toc134531222)

[ВИСНОВКИ 23](#_Toc134531223)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 24](#_Toc134531224)

[ДОДАТКИ 25](#_Toc134531225)

[Додаток А. Повний лістинг коду 25](#_Toc134531226)

[Program.cs 25](#_Toc134531227)

[Methods.cs 26](#_Toc134531228)

[Dijkstra.cs 27](#_Toc134531229)

[DijkstraParallel.cs 28](#_Toc134531230)

# ВСТУП

Паралельні обчислення є доволі актуальною темою в сучасному світі і є необхідними для вирішення багатьох складних завдань, що виникають в різних сферах, таких як наука, техніка, медицина та інші. Одна з причин, чому паралельні обчислення стали настільки важливими, полягає у зростанні обсягу даних, які потрібно обробляти, і складності завдань, що стоять перед науковцями й інженерами. За допомогою паралельних обчислень можна значно зменшити час обчислень, що раніше займало багато годин або днів, тепер може бути зроблено за декілька годин або навіть хвилин.

Іншою причиною, чому паралельні обчислення дуже важливі, є можливість використання більш потужних комп'ютерів та обчислювальних кластерів, що дозволяє розв'язувати завдання в більш короткі терміни. Наприклад, при моделюванні складних процесів в природі чи при створенні штучного інтелекту застосування паралельних обчислень дозволяє значно скоротити час розрахунків та отримати більш точні результати.

Алгоритм Дейкстри є одним з найбільш важливих алгоритмів у теорії графів та дискретної математики. Він використовується для пошуку найкоротшого шляху між двома вершинами у зваженому графі. Цей алгоритм має застосування в різних областях, таких як мережеве проектування, телекомунікації, транспорт, а також в інших областях, де потрібно вирішувати задачі знаходження оптимальних маршрутів.

У зв’язку зі зростанням кількості даних та залученням більш потужних обчислювальних систем, паралельні алгоритми стають все більш актуальними. Алгоритм Дейкстри також можна розпаралелити для прискорення процесу знаходження найкоротшого шляху у графі. Це може бути особливо корисним в великих мережах з багатьма вершинами, де звичайний послідовний алгоритм Дейкстри може бути надто повільним.

Обчислення даної курсової роботи будуть проводитись на шестиядерному процесорі AMD Ryzen 5 3600.

# 1 ОПИС ПОСЛІДОВНОГО АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ

Алгоритм Дейкстри був розроблений голландським математиком Едсгером Дейкстрою в 1956 році. У той час, коли алгоритм був розроблений, не існувало ефективних методів для пошуку найкоротших шляхів між вершинами в графах, які мали б багато вершин і ребер. Він розробив алгоритм з метою знаходження найкоротшого шляху між двома точками в графі з неорієнтованими ребрами та невід’ємними вагами.

Алгоритм Дейкстри був одним з перших алгоритмів, який широко використовувався для пошуку найкоротшого шляху в мережах з багатьма вузлами. Він був основою для розвитку багатьох інших алгоритмів, які застосовуються для різних видів мереж та графів.

## 1.1 Опис послідовного алгоритму

Алгоритм Дейкстри є алгоритмом пошуку найкоротшого шляху в графі з не від'ємними вагами ребер. Головною ідеєю алгоритму є поступове розширення "хмари" вершин, починаючи з початкової вершини і додавання до хмари вершин з найменшою відстанню від початкової вершини. Якщо вага нової вершини є меншою за поточну відстань до вершини в таблиці відстаней, то відстань оновлюється.

**Часова складність алгоритму** Дейкстри залежить від використаної структури даних для зберігання вершин графа. Якщо використовувати структуру даних "черга з пріоритетом", то часова складність становитиме O((E+V) log V), де E - кількість ребер, а V - кількість вершин у графі. Якщо використати матрицю суміжностей, то часова складність становитиме O(V^2). У випадку, якщо граф розріджений, тобто кількість ребер значно менше, ніж кількість вершин, можна використати структуру даних "куча Фібоначчі", що дасть часову складність O(E + V log V).

**Просторова складність алгоритму** Дейкстри також залежить від використаної структури даних. У випадку зі структурою "черга з пріоритетом" просторова складність становитиме O(V), використання матриці суміжностей потребує простору O(V^2), а використання структури "куча Фібоначчі" потребує O(V).

**Основні етапи алгоритму** Дейкстри можна розписати наступним чином:

1. Ініціалізація: встановлюємо відстані для кожної вершини від початкової вершини як нескінченні, окрім самої початкової вершини, для якої встановлюємо відстань рівну нулю. Всі вершини позначаються як "невідвідані".
2. Вибір початкової вершини: обираємо початкову вершину і розглядаємо всі зв'язані з нею ребра.
3. Оновлення відстаней: для кожної вершини, з'єднаної з початковою, оновлюємо її відстань, враховуючи вагу ребра, що їх з'єднує. Якщо нова відстань є меншою за попередню, замінюємо попередню відстань на нову.
4. Позначення як відвіданої: позначаємо початкову вершину як відвідану.
5. Вибір наступної вершини: обираємо наступну невідвідану вершину з найменшою відстанню від початкової вершини.
6. Повторення: повторюємо кроки 3-5 для всіх невідвіданих вершин, доки не будуть оброблені всі вершини або не буде знайдено шлях до кінцевої вершини.
7. Вивід результату: після завершення алгоритму Дейкстри ми знаходимо найкоротші шляхи від початкової вершини до всіх інших вершин.

**Псевдокод алгортиму** пошуку найкоротшого шляху Дейкстри можемо побачити на рисунку 1.1:

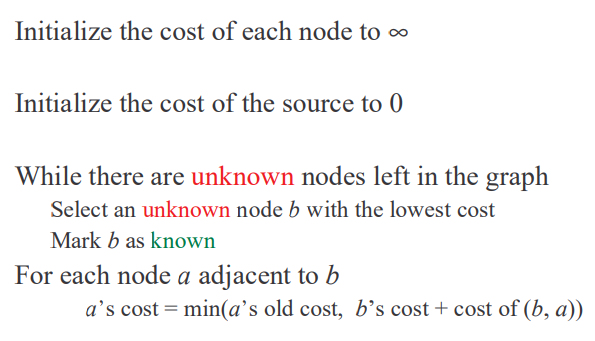


Рисунок 1. – Псевдокод алгоритму Дейкстри

**Візуально** алгоритм можна переглянути на рисунку 1.2:

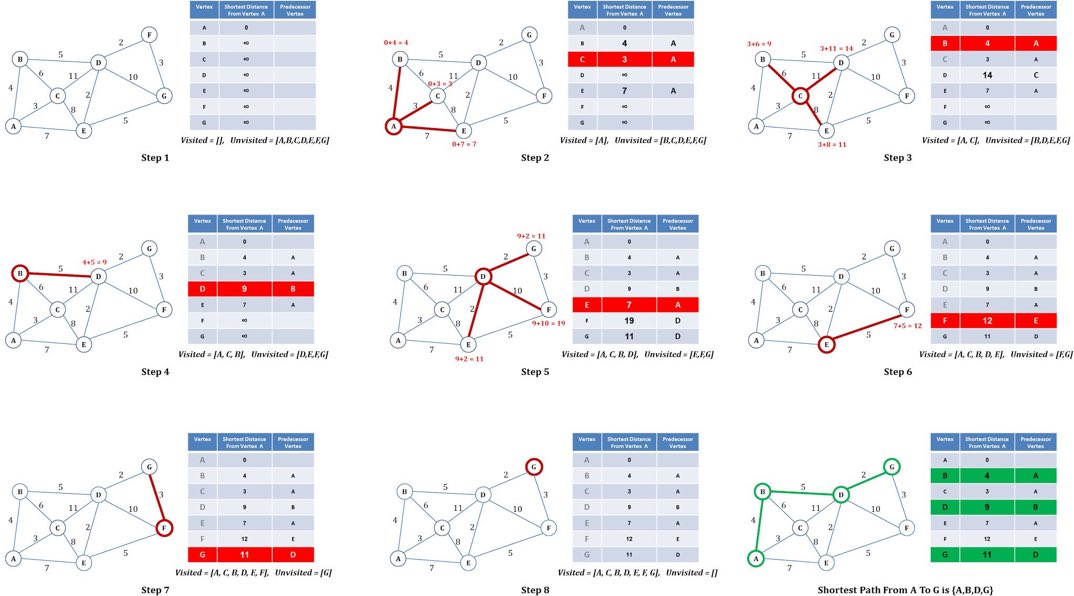


Рисунок 1. – Візуалізація послідовного алгоритму Дейкстри

Розглянемо етапи роботи послідовного алгоритму, які зазначені на рисунку 1.2:

Крок 1: алгоритм встановлює відстані від вихідної вершини A до решти вершин як нескінченність, а відстань від початкової ноди до самого себе як 0.

Крок 2: алгоритм вимірює орієнтовні відстані до кожного з невідвіданих сусідів початкової вершини, B, C і E у прикладі. Алгоритм позначає поточну вихідну вершину A як відвідану, витягує її з невідвіданого масиву та поміщає в відвіданий масив. Потім алгоритм визначає сусідню вершину з найкоротшою відстанню як наступну вершину, яку потрібно відвідати, вершину C у прикладі, і переходить до наступного кроку.

Крок 3: алгоритм вимірює попередні відстані невідвіданих сусідів вершини C, тобто B, D і E, відносно вихідної початкової вершини A. Алгоритм розслабляє вершину D за допомогою нової попередньої відстані та встановлює її попередню вершину шляху як C. Алгоритм не оновлює орієнтовні відстані вершин B і E, оскільки їх відстані, виміряні через вершину C, більші, ніж раніше призначені орієнтовні відстані. Як і на кроці 2, алгоритм позначає поточну вершину C як відвідану, висовує її з невідвіданого масиву та розміщує в відвіданому масиві. Потім алгоритм визначає невідвідану вершину з найкоротшою відстанню як наступну вершину, яку потрібно відвідати, вершину B у прикладі, і переходить до наступного кроку.

Алгоритм повторюватиме такі кроки, доки всі вершини не будуть позначені як відвідані або поки не залишиться пов’язаних вершин для оцінки. Потім можна визначити найкоротший шлях від вихідної вершини до будь-якої іншої вершини, шукаючи вершини-попередники з оцінюваної таблиці. Наприклад, щоб визначити найкоротший шлях від вершини A до вершини G, можна переглянути таблицю, щоб знайти вершину-попередницю G, яка є D. Вершиною-попередницею D є B, а вершиною-попередницею B є A. Таким чином, найкоротший шлях від вершини A до вершини G це {A,B,D,G} з найкоротшою відстанню 11.

## 1.2 Розпаралелювання процесу переобчислення сусідніх вершин

Оскільки порядок перевірки вершин є важливим в алгоритмі Дейкстри, єдине, що ми можемо розпаралелити в послідовному алгоритмі, та так щоб він не втрачав своєї суті – це переобчислення сусідніх вершин під час відвідування однієї із них. Основна зміна у порівнянні з послідовним алгоритмом полягає у тому, що ми розподіляємо сусідів вершини між виділеними потоками і процес переобчислення сусідніх вершин проходить паралельно.

**Часова складність алгоритму:** О(V^2/P + V log(P))

**Просторова складність алгоритму:** О(V^2/P)

**Етапи роботи алгоритму:**

1. Ініціалізація: встановлюємо відстані для кожної вершини від початкової вершини як нескінченні, окрім самої початкової вершини, для якої встановлюємо відстань рівну нулю. Всі вершини позначаються як "невідвідані".
2. Вибір початкової вершини: обираємо початкову вершину і розглядаємо всі зв'язані з нею ребра.
3. Розподіляємо вершини, які потрібно буде пройти між виділеними потоками.
4. Оновлення відстаней: кожен потік для кожної отриманої вершини, з'єднаної з початковою, оновлює її відстань, враховуючи вагу ребра, що їх з'єднує. Якщо нова відстань є меншою за попередню, замінюємо попередню відстань на нову.
5. Позначення як відвіданої: позначаємо початкову вершину як відвідану.
6. Вибір наступної вершини: обираємо наступну невідвідану вершину з найменшою відстанню від початкової вершини.
7. Повторення: повторюємо кроки 3-5 для всіх невідвіданих вершин, доки не будуть оброблені всі вершини або не буде знайдено шлях до кінцевої вершини.

Вивід результату: після завершення алгоритму Дейкстри ми знаходимо найкоротші шляхи від початкової вершини до всіх інших вершин.

# 2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

Нижче наведено функціональний алгоритм пошуку найкоротшого шляху Дейкстра, реалізований мовою програмування С#.

Щоб розробити алгоритм, представимо граф у вигляді матриці суміжності у якій кожен рядок буде відповідати вершині, а кожен елемент у цьому рядку – це вартість між вершинами.

Для того, щоб перевірити алгоритм на коректність роботи, розробимо метод, який буде генерувати випадкову матрицю суміжності за заданим розміром. Нижче на рисунку 2.1 можемо побачити реалізацію цього методу:

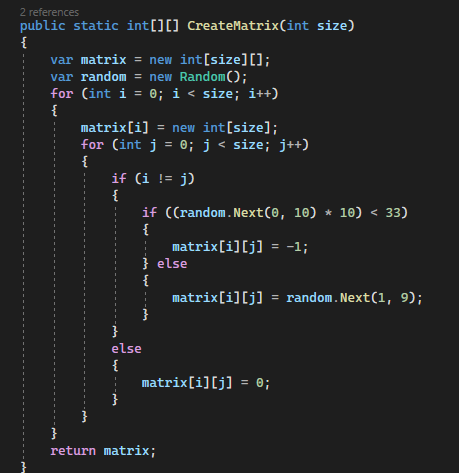


Рисунок 2. – Метод для генерації рандомної матриці суміжності

Створення рандомної матриці суміжності працює через клас Random. У ході генерації значень, ми присвоюємо значення 0 у тих випадках, коли індекс вершини та індекс вартості дорівнюють одне одному, тобто ми маємо діагональ із нулів. Також ми маємо тридцяти відсотковий шанс, що між вершинами не буде ніякого зв`язку і буде присвоєно значення вартості -1.

Тепер перейдемо до розробки самого алгоритму. Перед тим як запустити дані до основного циклу, ми маємо створити масиви та підготувати їх. Нижче на рисунку 2.2 наведена підготовка даних до основного циклу:

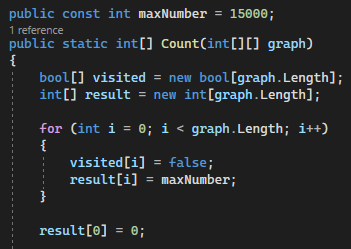


Рисунок 2.2 – Формуємо масиви, які зберігатимуть перевірені вершини та вартість шляху.

Під час підготовки даних, ми заповнюємо масив перевірених вершин та масив вартостей шляхів початковими значеннями. Це означатиме, що жоден з вершин не був перевірений, а теоретична мінімальна вартість шляху дорівнює великому значенню (більшому ніж можуть дати графи). Також ми ставимо вартість 0 для початкової вершини.

Після цього ми переходимо до основного циклу, кількість ітерацій якого дорівнює кількості вершин. У цьому циклі, ми можемо поділити роботу алгоритму на умовні дві частини. Нижче на рисунку 2.3 наведена перша частина алгоритму:

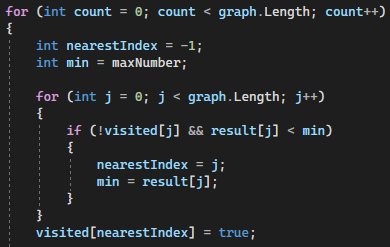


Рисунок 2.3 – Перша частина алгоритму, яка визначає вершину з мінімальною вартістю шляху

Перша частина алгоритму слугує для того, щоб визначити сусідів якої саме вершини ми будемо обраховувати. На першій ітерації – це початкова вершина, оскільки її вартість дорівнює нулю. Після того, як ми визначили вершину, ми позначаємо її пройденою.

Нижче на рисунку 2.4 зображена друга частина алгоритму:

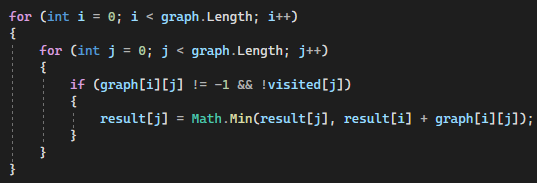


Рисунок 2.4 – Друга частина алгоритму, яка обраховує сусідів обраної вершини

Друга частина алгоритму слугує для того, щоб обрахувати сусідів обраної вершини. Двома циклами ми проходимось по всьому графу і обраховуємо ті вершини, які мають зв`язку з обраною вершиною, та ще не були пройдені. Якщо виявлена сума вартостей менша за ту, що була записана раніше, тоді вона записується у результуючий масив.

Після того як цикл пройшовся по кожній вершині, він повертає результуючий масив, який містить вартість шляху від початкової вершини до інших вершин.

Для запуску та наступним експериментам оформимо головний клас Program.cs. Результат можемо побачити на рисунку 2.5:

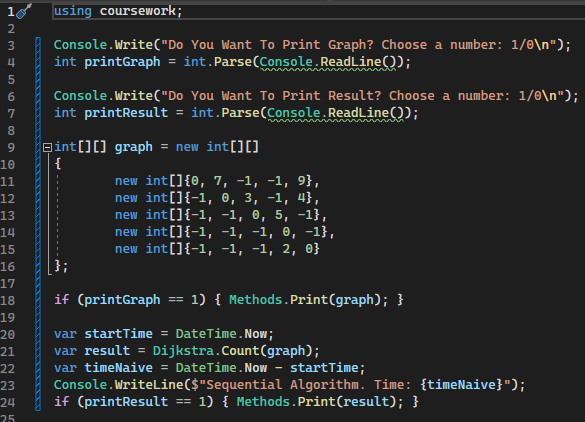


Рисунок 2.5 – Program.cs

Тепер перевіримо програму на коректність роботи. Запустимо її. Результат роботи програми зображено на рисунку 2.6:

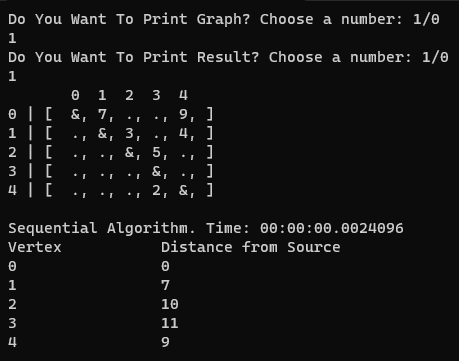


Рисунок 2.6 – Результат виконання роботи

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

С# є високорівневою мовою програмування, яка має вбудовану підтримку багатопотоковості і паралелізму, що робить її чудовим вибором для реалізації паралельних алгоритмів, таких як алгоритм Дейкстри. Додатково, С# має платформу .NET, яка надає можливості для розподіленого обчислення, що дозволяє розпаралелити обчислення на кількох машинах. Крім того, С# є однією з найпопулярніших мов програмування, що має велику спільноту розробників і багато інструментів розробки, що полегшують процес розробки та підтримки коду.

Однією з головних переваг багатопоточності у C# є підвищення ефективності виконання програм, оскільки можна розділити обчислення на декілька потоків, що виконуються паралельно. Це особливо корисно у випадках, коли потрібно обробити великі обсяги даних або виконати складні операції. Також варто зазначити, що С# підтримує адаптивний інтерфейс користувача та обробляє запити одночасно як на стороні сервера, так і на стороні клієнта. Наприклад, обробка однорангових мереж.

У C# реалізація багатопоточності здійснюється за допомогою бібліотеки System.Threading.Thread. Цей клас дає змогу створювати і управляти потоками виконання у програмі.

Основні методи управління потоками в C# включають наступні:

* Thread.Start(): метод для створення нового потоку і запуску його виконання.
* Thread.Join(): метод для блокування поточного потоку до завершення виконання іншого потоку.
* Thread.Sleep(): метод для призупинення виконання поточного потоку на певний час.
* Thread.Abort(): метод для примусового припинення виконання потоку.
* Thread.Suspend(): метод для призупинення виконання потоку.
* Thread.Resume(): метод для продовження виконання призупиненого потоку.

Крім того, C# надає ряд класів для синхронізації доступу до ресурсів між потоками, такі як Monitor, Mutex, Semaphore, AutoResetEvent та інші. Використання цих класів дозволяє уникнути проблем зі зміною спільних ресурсів з декількох потоків одночасно.

Для того щоб виконати поставлену задачу, ми використаємо бібліотеку System.Threading.Tasks, яка спрощує роботу з багатопроцесорними, багатоядерними системами. Окрім цього, вона спрощує роботу по створенню нових потоків. В основі цієї бібліотеки лежить концепція задач, кожна з яких описує окрему операцію. В бібліотеці класів задача представлена спеціальним класом Task, який описує окрему задачу, яка запускається асинхронно в одному з потоків із пулу потоків. Хоча її і можна запускати синхронно в поточному потоці.

Основні методи управління потоками в бібліотеці System.Threading.Tasks включають наступні:

* Task.Run - запускає виконання задачі в іншому потоці.
* Task.WaitAll - забезпечує, що всі задачі будуть виконані перед продовженням виконання програми.
* Task.WhenAll - повертає задачу, яка буде завершена тільки після завершення всіх переданих їй задач.
* Task.Delay - затримує виконання задачі на певний час.
* Task.Factory.StartNew - створює і запускає нову задачу.

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

Для розробки паралельного алгоритму бітонічного сортування мовою програмування C# будемо використовувати як базу вже розроблений послідовний метод.

Спочатку почнемо з проєктування паралельного алгоритму:

* Мета алгоритму – отримати найкоротші шляхи від початкової вершини до кожної присутньої на графі, використовувати менше енергії пристрою та працювати швидше за послідовний алгоритм.
* Паралельний алгоритм може отримувати граф вільного розміру та повертати результат не втрачаючи даних
* Характеристики алгоритму: алгоритм не отримує нічого нового, чого не мав послідовний алгоритм. При паралельному програмуванні час обчислень зменшується зі збільшенням кількості процесорів. Прискорення збільшується зі збільшенням кількості вершин
* Послідовність дій: відправляємо кількість потоків до паралельного алгоритму –> ділимо кількість вершин на кількість потоків і отримуємо приблизну роботу на кожен потік -> запускаємо потоки -> очікуємо завершення виконання потоків.

Перейдемо до реалізації. Єдина частина, що змінюється у порівнянні з послідовним алгоритмом – це та, де алгоритм обраховує сусідні вершини. Нижче на рисунку 4.1 ми можемо побачити підготовку та виклик потоків, а на рисунку 4.2 частину, яка працює паралельно:

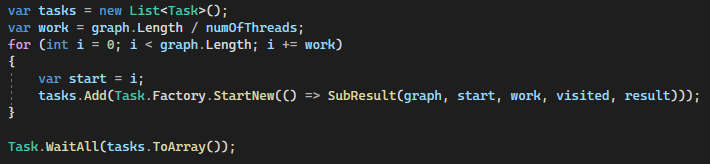


Рисунок 4. – Підготовка, виклик та очікування завершення роботи потоків

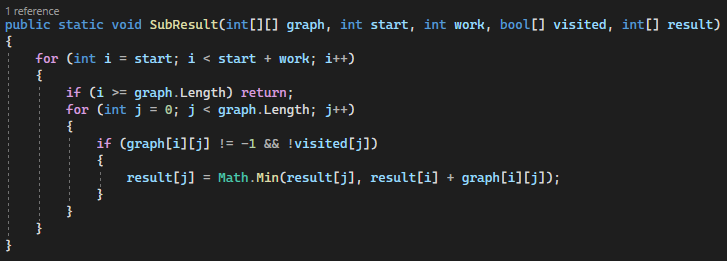


Рисунок 4. – Програмна частина, яка запускається паралельно

Спочатку ми створюємо лист завдань, які у майбутньому будуть містити завдання, які ми створимо. Потім ми ділимо роботу між потоками, а якщо залишається зайва частинка, то додаємо її потроху до кожного потоку у циклах. Після визову завдання через метод Task.Factory.StartNew(), який повертає тип Task, ми додаємо завдання до раніше створеного листу.

Оскільки C# за стандартом використовує усі свободні логічні процесори комп`ютера, то ми обираємо кількість потоків саме розділенням роботи, бо ті потоки, які не отримали роботи, просто не будуть використовуватись.

Після створення усіх завдань ми визиваємо метод Task.WaitAll(), щоб дочекатись результату роботи потоків перед продовженням виконання програми. Після проходження по всім циклам, ми отримуємо масив, що зберігає вартість шляхів від початкової вершини до інших.

Тепер перевіримо програму на коректність роботи. Запустимо її. Результат роботи побачимо на рисунку 4.3:

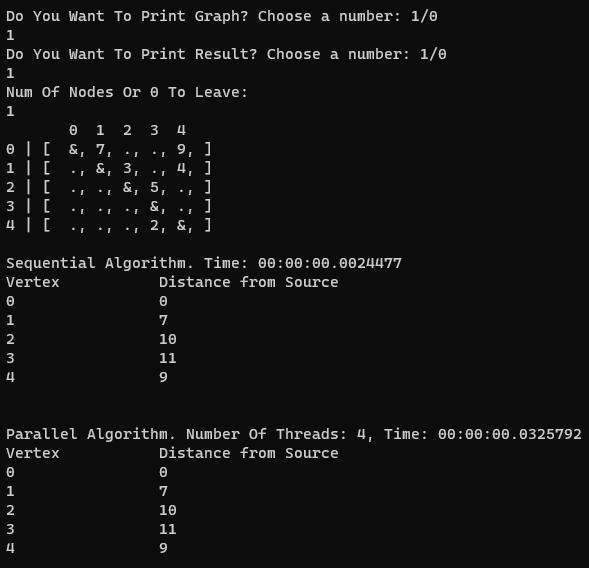


Рисунок 4. – Результат виконання роботи

Як бачимо, результат роботи послідовного та паралельного алгоритму збігаються. Якщо ми подивимось на матрицю суміжності, то можемо побачити, що результати правильні. Наприклад, щоб дістатись від 0 вершини до 3 вершини, ми маємо спочатку пройти від 0 до 4 вершини, а потім з 4 до 3 вершини. Вартість цього шляху як раз і буде 9 + 2 = 11.

# 5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ

Приступимо до аналізу швидкодії та ефективності роботи алгоритму.

Для цього створимо спеціальний метод Experiment, який зафіксує роботу алгоритмів на різних розмірах матриць та різній кількості потоків. Нижче на рисунку 5.1 можемо побачити цей метод:



Рисунок 5. – Метод Experiment()

Як ми можемо побачити, ми створили два масиви, один з яких містить 4 різних розміра, а других містить 3 різних кількості потоків. Також ми запускаємо паралельний алгоритм кожну ітерацію 3 рази і фіксуємо середнє значення за яке він впорався з отриманим графом. Нижче на рисунку 5.2 можемо побачити результат роботи програми:

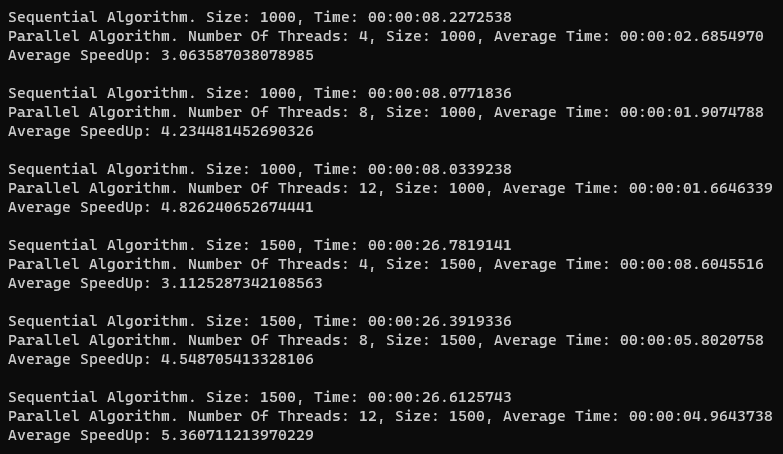


Рисунок 5. – Результат виконання програми

На основі отриманих даних, побудуємо порівняльну з різними розмірами та кількістю потоків. Позначимо час, що зайняв алгоритм як T, та прискорення отримане у результаті ділення часу послідовного алгоритму на паралельних як S. Нижче у таблиці 5.1 можемо побачити результати:

Таблиця 5.1. – Порівняльна таблиця послідовного та паралельного алгоритмів

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Кількість вершин графа | Послідовний алгоритм, с | Паралельний алгоритм | | | | | |
| 4 потоки | | 8 потоків | | 12 потоків | |
| T, с | S | T, с | S | T, с | S |
| 1000 | 8.113 | 2.685 | 3.022 | 1.907 | 4.254 | 1.665 | 4.873 |
| 1500 | 26.595 | 8.604 | 3.091 | 5.802 | 4.584 | 4.964 | 5.358 |
| 2000 | 62.128 | 20.228 | 3.071 | 13.527 | 4.593 | 11.367 | 5.466 |
| 3000 | 209.819 | 68.865 | 3.047 | 43.816 | 4.789 | 35.159 | 5.968 |

Побудуємо графік прискорення на основі отриманих результатів. Нижче на рисунку 5.3 можемо побачити отриманий графік прискорення:

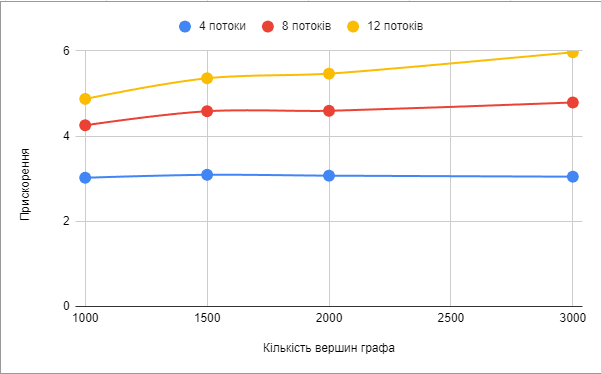


Рисунок 5. – Графік прискорення

Як можемо побачити на отриманому графіку, кількість вершин графа впливає на прискорення у залежності від кількості потоків, тобто на більшій кількості потоків прискорення з розміром зростає, а на чотирьох потоках тримається на одному й тому ж рівні. Також, маємо гарне прискорення пов`язане зі збільшенням кількості потоків. Чим більше потоків – тим кращий результат.

Але, варто зазначити, що мій комп`ютер має 12 логічних процесорів, тому при збільшенні кількості потоків вище цього числа, ми скоріш за все помітимо гірші значення прискорення, тому проведемо експеримент з 16 потоками і порівняємо отримані результати зі значеннями при 12 потоках. Нижче на рисунку 5.4 можемо помітити результати отриманого дослідження:

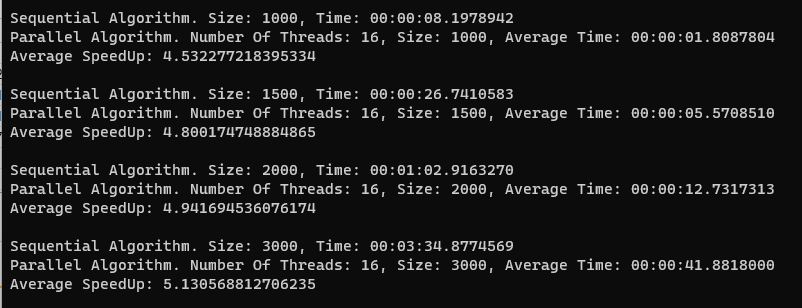


Рисунок 5. – Результат роботи програми

Створимо порівняльну таблицю, щоб дізнатись, як впливає на прискорення кількість потоків більша за кількість логічних процесорів. Нижче у таблиці 5.2 можемо помітити результати:

|  |  |  |
| --- | --- | --- |
|  | 12 потоків, с | 16 потоків, с |
| 1000 | 1.665 | 1.8087 |
| 1500 | 4.964 | 5.5708 |
| 2000 | 11.367 | 12.738 |
| 3000 | 35.159 | 41.882 |

Як ми можемо побачити, час виконання алгоритму при 16 потоків більший за час виконання при 12 потоків, тому можемо зробити висновок, що найкраще рішення використовувати кількість потоків рівну кількості логічних процесорів.

Це може бути пов`язано з тим, що якщо кількість потоків більша за кількість логічних процесорів, то це може призвести до збільшення конкуренції за доступ до ресурсів процесора, що може спричинити зниження продуктивності програми. Але якщо кількість потоків менша за кількість логічних процесорів, то деякі процесори можуть залишатися не використаними, що призводить до недоексплуатації ресурсів системи.

# ВИСНОВКИ

В даній курсовій роботі була створена паралельна реалізація алгоритму пошуку найкоротшого шляху Дейкстри, було порівняно ефективність з послідовним алгоритмом і в результаті чого ми можемо зробити деякі висновки щодо ефективності виконання паралельних обчислень.

Для задач, які потребують роботи з багатьма потоками, мова програмування C# є дуже ефективною. Під час виконання мого курсового проекту я вивчив дві основні бібліотеки - System.Threading.Tasks і System.Threading.Thread - які допомогли мені вирішити складні завдання, пов'язані з паралельними обчисленнями.

На основі проведених численних досліджень було встановлено, що використання паралельних обчислень значно збільшує швидкість виконання алгоритмів, підвищує їхню ефективність і дозволяє розв'язувати більш складні завдання. Одним з досліджених алгоритмів був алгоритм Дейкстри для пошуку найкоротшого шляху, і його паралельна реалізація дала результати, що підтверджують цей висновок.

Під час виконання курсової роботи був проведений експеримент у ході якого ми дізналися, що кількість потоків гарно прискорює послідовний алгоритм. Також, ми дізналися, що на прискорення не впливає розмір, а якщо і впливає, то тільки при більшій кількості потоків. Хочу зазначити, що було отримано максимальне прискорення 5.968 при 12 потоках та 3000 кількості вершин графу.

Я переконаний, що використання паралельних обчислень є дуже ефективним для багатьох програм, включаючи мою курсову роботу. Звичайно, є різні фактори, що впливають на ефективність паралелізму, і їх потрібно враховувати. Проте, фактом залишається, що алгоритми, які реалізовані з використанням паралельних обчислень, працюють швидше та більш ефективно, ніж ті, що не використовують цей підхід.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. An Implementation of Parallelizing Dijkstra’s Algorithm – [Електронний ресурс] / Zilong Ye, 2012 – Режим доступу до ресурсу: https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Ye-Fall-2012-CSE633.pdf
2. Parallelizing Dijkstra's Algorithm – [Електронний ресурс] / Mengqing He, 2021 – Режим доступу до ресурсу: https://repository.stcloudstate.edu/cgi/viewcontent.cgi?article=1044&context=csit\_etds
3. A Parallelization of Dijkstra’s Shortest Path Algorithm – [Електронний ресурс] / A. Crauser, K. Mehlhorn, U. Meyer and P. Sanders, 1998 – Режим доступу до ресурсу: https://people.mpi-inf.mpg.de/~mehlhorn/ftp/ParallelizationDijkstra.pdf
4. Dijkstra’s Algorithm Continued – [Електронний ресурс] / Paul G. Allen School, 2006 – Режим доступу до ресурсу: https://courses.cs.washington.edu/courses/cse326/06au/lectures/lect22.pdf
5. Dijkstra’s Algorithm – [Електронний ресурс] / GeeksForGeeks, 2013 – Режим доступу до ресурсу: https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
6. Опис навчальної дисципліни, її мета, предмет вивчення та результати навчання. Курсова робота з технологій паралельних обчислень. / Стеценко І. В. Київ, Україна: НТУУ "КПІ"., 2023. – Режим доступу до ресурсу: https://docs.google.com/document/d/1Ejj-rDbphRqcn4agNbFtD9-\_KDR5A-HU/edit
7. Threading in C# - [Електронний ресурс] / Joe Albahari, 2014 – Режим доступу до ресурсу: https://www.albahari.com/threading/part5.aspx#\_TaskFactory

# ДОДАТКИ

## Додаток А. Повний лістинг коду

### Program.cs

using coursework;

using System.Xml.Linq;

//Methods.Experiment();

//int[][] graph = new int[][]

//{

// new int[]{0, 7, -1, -1, 9},

// new int[]{-1, 0, 3, -1, 4},

// new int[]{-1, -1, 0, 5, -1},

// new int[]{-1, -1, -1, 0, -1},

// new int[]{-1, -1, -1, 2, 0}

//};

//int[][] graph = new int[][]

//{

// new int[]{ 0, 14, -1, -1, -1, -1, -1, 8, -1 },

// new int[]{ 4, 0, 8, -1, -1, -1, -1, 11, -1 },

// new int[]{ -1, 8, 0, 7, -1, 10, -1, -1, 2 },

// new int[]{ -1, -1, 7, 0, 9, 14, -1, -1, -1 },

// new int[]{ -1, -1, -1, 9, 0, 10, -1, -1, -1 },

// new int[]{ -1, -1, 4, -1, 10, 0, 2, -1, -1 },

// new int[]{ -1, -1, -1, 14, -1, 2, 0, 1, 6 },

// new int[]{ 8, 11, -1, -1, -1, -1, 1, 0, 7 },

// new int[]{ -1, -1, 2, -1, -1, -1, 16, 7, 0 }

//};

Console.Write("Do You Want To Print Graph? Choose a number: 1/0\n");

int printGraph = int.Parse(Console.ReadLine());

Console.Write("Do You Want To Print Result? Choose a number: 1/0\n");

int printResult = int.Parse(Console.ReadLine());

while (true)

{

Console.WriteLine("Num Of Nodes Or 0 To Leave: ");

int nodes = int.Parse(Console.ReadLine());

if (nodes == 0) { break; }

int[][] graph = Methods.CreateMatrix(nodes);

if (printGraph == 1) { Methods.Print(graph); }

var startTime = DateTime.Now;

var result = Dijkstra.Count(graph);

var timeNaive = DateTime.Now - startTime;

Console.WriteLine($"Sequential Algorithm. Time: {timeNaive}");

if (printResult == 1) { Methods.Print(result); }

int numOfThreads = 4;

startTime = DateTime.Now;

result = DijkstraParallel.Count(graph, numOfThreads);

var timeParallel = DateTime.Now - startTime;

Console.WriteLine($"Parallel Algorithm. Number Of Threads: {numOfThreads}, Time: {timeParallel}");

if (printResult == 1) { Methods.Print(result); }

Console.WriteLine($"SpeedUp: {timeNaive / timeParallel}\n");

}

### Methods.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace coursework

{

public class Methods

{

public static int[][] CreateMatrix(int size)

{

var matrix = new int[size][];

var random = new Random();

for (int i = 0; i < size; i++)

{

matrix[i] = new int[size];

for (int j = 0; j < size; j++)

{

if (i != j)

{

if ((random.Next(0, 10) \* 10) < 33)

{

matrix[i][j] = -1;

} else

{

matrix[i][j] = random.Next(1, 9);

}

}

else

{

matrix[i][j] = 0;

}

}

}

return matrix;

}

public static void Print(int[] arr)

{

Console.Write("Vertex Distance " + "from Source\n");

for (int i = 0; i < arr.Length; i++)

{

Console.Write(i + " \t\t " + arr[i] + "\n");

}

Console.Write("\n\n");

}

public static void Print(int[][] matrix)

{

Console.Write(" ");

for (int i = 0; i < matrix.Length; i++)

{

Console.Write("{0} ", i);

}

Console.WriteLine();

for (int i = 0; i < matrix.Length; i++)

{

Console.Write("{0} | [ ", i);

for (int j = 0; j < matrix.Length; j++)

{

if (matrix[i][j] == -1)

{

Console.Write(" .,");

}

else if (matrix[i][j] == 0)

{

Console.Write(" &,");

}

else

{

Console.Write(" {0},", matrix[i][j]);

}

}

Console.Write(" ]\r\n");

}

Console.Write("\r\n");

}

public static void Experiment()

{

int[] sizes = new int[] { 1000, 1500, 2000, 3000 };

int[] numOfThreads = new int[] { 16 };

foreach (int size in sizes)

{

foreach(int numOfThread in numOfThreads)

{

int[][] graph = Methods.CreateMatrix(size);

var startTime = DateTime.Now;

var result = Dijkstra.Count(graph);

var timeNaive = DateTime.Now - startTime;

TimeSpan timeParallel = new TimeSpan();

for (int i = 0; i < 3; i++)

{

startTime = DateTime.Now;

result = DijkstraParallel.Count(graph, numOfThread);

timeParallel += DateTime.Now - startTime;

}

timeParallel = timeParallel / 3;

Console.WriteLine($"Sequential Algorithm. Size: {size}, Time: {timeNaive}");

Console.WriteLine($"Parallel Algorithm. Number Of Threads: {numOfThread}, Size: {size}, Average Time: {timeParallel}");

Console.WriteLine($"Average SpeedUp: {timeNaive / timeParallel}\n");

}

}

}

}

}

### Dijkstra.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace coursework

{

public class Dijkstra

{

public const int maxNumber = 15000;

public static int[] Count(int[][] graph)

{

bool[] visited = new bool[graph.Length];

int[] result = new int[graph.Length];

for (int i = 0; i < graph.Length; i++)

{

visited[i] = false;

result[i] = maxNumber;

}

result[0] = 0;

for (int count = 0; count < graph.Length; count++)

{

int nearestIndex = -1;

int min = maxNumber;

for (int j = 0; j < graph.Length; j++)

{

if (!visited[j] && result[j] < min)

{

nearestIndex = j;

min = result[j];

}

}

visited[nearestIndex] = true;

for (int i = 0; i < graph.Length; i++)

{

for (int j = 0; j < graph.Length; j++)

{

if (graph[i][j] != -1 && !visited[j])

{

result[j] = Math.Min(result[j], result[i] + graph[i][j]);

}

}

}

}

return result;

}

}

}

### DijkstraParallel.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace coursework

{

public class DijkstraParallel

{

public const int maxNumber = 15000;

public static int[] Count(int[][] graph, int numOfThreads)

{

bool[] visited = new bool[graph.Length];

int[] result = new int[graph.Length];

for (int i = 0; i < graph.Length; i++)

{

visited[i] = false;

result[i] = maxNumber;

}

result[0] = 0;

for (int count = 0; count < graph.Length; count++)

{

int nearestIndex = -1;

int min = maxNumber;

for (int j = 0; j < graph.Length; j++)

{

if (!visited[j] && result[j] < min)

{

nearestIndex = j;

min = result[j];

}

}

visited[nearestIndex] = true;

var tasks = new List<Task>();

var work = graph.Length / numOfThreads;

for (int i = 0; i < graph.Length; i += work)

{

var start = i;

tasks.Add(Task.Factory.StartNew(() => SubResult(graph, start, work, visited, result)));

}

Task.WaitAll(tasks.ToArray());

}

return result;

}

public static void SubResult(int[][] graph, int start, int work, bool[] visited, int[] result)

{

for (int i = start; i < start + work; i++)

{

if (i >= graph.Length) return;

for (int j = 0; j < graph.Length; j++)

{

if (graph[i][j] != -1 && !visited[j])

{

result[j] = Math.Min(result[j], result[i] + graph[i][j]);

}

}

}

}

}

}