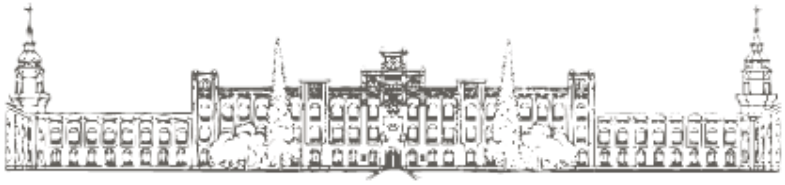

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
КАФЕДРА ІНФОРМАТИКИ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Курсова робота з освітнього компоненту
«Технології паралельних обчислень. Курсова робота»
Тема: **Алгоритм A* та його паралельна реалізація з використанням мови C#**

Керівник:
проф. Стеценко Інна В'ячеславівна

Виконавець:
Черпак Андрій Вадимович
студент групи ІП-01
залікова книжка № _____

«Допущено до захисту»
_____

«___» _____ 2023 р.

_____

«23» травня 2023 р.

Захищено з оцінкою
_____

Члени комісії:

_____ Інна СТЕЦЕНКО

_____ Олександра ДИФУЧИНА

Київ – 2023

ЗАВДАННЯ

1. Виконати огляд існуючих реалізацій алгоритму A*, послідовних та паралельних. Зробити висновок про актуальність дослідження.
2. Виконати розробку послідовного алгоритму A*використанням мови C#. Дослідити швидкодію алгоритму при зростанні складності обчислень та зробити висновки про необхідність паралельної реалізації алгоритму.
3. Виконати розробку паралельного алгоритму A*використанням мови C#.
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: 47 сторінок, 6 рисунків, 3 таблиці.

Об’єкт дослідження: задача паралельного пошуку найкоротшого шляху у графі.

Мета роботи: теоретично дослідити паралельні методи пошуку шляху у графі за допомогою алгоритму A*; переглянути відомі їх реалізації; спроектувати, реалізувати, протестувати послідовний та паралельний алгоритми; дослідити ефективність паралелізації програми;

Виконана програмна реалізація паралельного та послідовного алгоритму пошуку шляху у графі, проведено аналіз їх ефективності.

Ключові слова: ПОШУК ШЛЯХУ У ГРАФІ, A*, HDA*, ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ

ЗМІСТ

ВСТУП

Задачі пошуку найкоротших шляхів у графі знаходить широке застосування у різних областях, включаючи штучний інтелект, робототехніку, комп'ютерні ігри та навігацію. Одним з найбільш ефективних алгоритмів для вирішення даної задачі є A*. З появою багатоядерних процесорів та розподілених систем зросла потреба в розробці паралельних алгоритмів, що дозволяють виконувати складні обчислення швидше та ефективніше. Враховуючи природу алгоритму A*, можна зробити припущення про його потенціал для паралельної реалізації з метою прискорення обчислень.

В рамках даної курсової роботи буде розглянуто кілька підходів до паралельної реалізації алгоритму A* та проведено аналіз їх ефективності для графів різних розмірностей. Оскільки даний алгоритм на кожній новій ітерації опирається на результати виконання минулих ітерації, на перший погляд не так просто визначити, як же правильно його розпаралелити. Мною буде здійснено спробу просто опрацьовувати одну чергу у кілька потоків, а потім буде пояснено, чому такий підхід не дає особливого прискорення, та ще й отриманий результат інколи є близьким до найкращого, але не найкращим, що нівелює одну з основних переваг алгоритму A*. Також буде розроблено алгоритм, який шляхом паралелізації обчислень знаходить справді найкоротший маршрут за коротший проміжок часу. Ефективність зростає зі збільшенням розмірності матриці.

1 ОПИС ПОСЛІДОВНОГО АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ

Як вже було згадано мною, алгоритм A* не так просто розпаралелити, оскільки кожна його наступна ітерація обов’язково відштовхується від результатів роботи минулих ітерацій. Тому окрім класичної реалізації A* існує лише кілька паралельних версій, жодна з яких не дає відчутного прискорення. В даній роботі буде розглянуто лише послідовну версію та HDA* (Hash Distributed A*), а також створені мною варіації.

1.1 Класичний A*

Алгоритм A* належить до евристичний алгоритмів, хоча й базується на алгоритмі Дейкстри. Ці алгоритми працюють доволі просто: початкова вершина надсилається до черги. Далі доки в черзі є хоч один елемент, з неї обирається найкраща за якимось критерієм вершина, позначається як пройдена, а до черги додаються усі не пройдені вершини, до яких можна напряму дістатися з поточної. Перед додаванням до черги у даних вершин оновлюється значення відстані від старту та попередньої вершини на шляху від старту. Остання нам потрібна, щоб потім відслідкувати сам шлях. Якщо ж поточна вершина виявиться шуканою (фінішною) – робота алгоритму завершується. Якщо на момент, коли черга виявиться порожньою, шукана вершина досі не досягнута – шляху не існує.

A* відрізняється від алгоритму Дейкстри лише критерієм вибору наступної вершини з черги. Якщо в алгоритмі Дейкстри в першу чергу ми намагаємося обирати вершини найближчі до стартової, то в A* окрім відстані від стартової точки враховується також евристична оцінка відстані до кінцевої точки. Таким чином, пошук одразу спрямований у правильний бік. При грамотно підібраній евристичній функції (та відсутності коефіцієнтів при ній) алгоритм все ще знаходить найкращий шлях, але за значно коротші проміжки часу.

1.2 HDA*

На відміну від класичної реалізації A*, HDA* використовує кілька потоків для одночасної обробки вершин. При чому, кожен потік має власну чергу. Втім нащадки розглянутої вершини додаються не обов’язково чергу потоку, що її обробив. Натомість застосовується хеш-функція, яка визначає, якому з потоків належатиме та чи інша вершина. Це дозволяє рівномірно розподілити роботу між потоками. Втім, через одночасний доступ до графу та черг виникає потреба у обов’язковій синхронізації цих потоків, що насправді доволі сильно сповільнює роботу програми. Крім того, аби гарантувати, що буде знайдено найкоротший шлях, доведеться дозволити кожному з потоків по разу пройтися по кожній вершині, що збільшить кількість оброблюваних вершин.

Докладну інформацію про даний варіант реалізації алгоритму можна знайти у статті «Parallel A* Graph Search» (№3 у списку використаних джерел).

2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

2.1 Проектування послідовного алгоритму

Відповідно до теорії описаної в пункті 1.1 було розроблено алгоритм A* для пошуку найкоротшого шляху у графі. Він використовуватиме пріоритетну чергу для зберігання досягнутих вершин, що дозволить максимально ефективно додавати їх чи обирати за заданим критерієм. Оскільки передбачається також одночасний доступ кількох потоків до черги, мною було розроблено і потокобезпечну версію даної структури. Повний код може бути знайдений на моєму GitHub.

2.2 Реалізація послідовного алгоритму

Для реалізації даного алгоритму було створено такі класи, як вершина, що містить координати у просторі, індекс у списку вершин, відстань від старту, індекс попередньої вершини на шляху від старту, евристичну оцінку відстані до фінішу, а також прапорець, що вказує на те, чи пройдено цю вершину. Задля зручності подальшого розширення програми даний клас реалізує інтерфейс IVertice. Також було реалізовано клас Graph, який містить список вершин, матрицю вагів ребер між ними, індекси стартової та кінцевої точок, а також допоміжні методи для роботи з цими даними. Даний клас також реалізує інтерфейс IGraph (оскільки надалі планується розробляти інші класи зі схожими властивостями).

```
public interface IVertice
{
    public Coordinates VerticeCoordinates { get; }

    public int OwnIndex { get; }

    public void Reset();
    public IVertice Copy();
    public void SetHeuristic(IVertice start, IVertice finish);
}
```

```
public class Vertice : IVertice
{
    public float DistanceFromStart { get; set; }
    public Coordinates VerticeCoordinates { get; }
```

```
    public float? Heuristic { get; private set; }
    public int PreviousVerticeInRouteIndex { get; set; }
    public bool IsPassed;

    public int OwnIndex { get; }
    private IGraph Graph { get; }

...

    public bool TryUpdateMinRoute(int fromVerticeIndex)
    {
        if (IsPassed || Graph[fromVerticeIndex, OwnIndex] == -1)
            return false;

        float newDistance = Graph[fromVerticeIndex, OwnIndex] +
((Vertice)Graph[fromVerticeIndex]).DistanceFromStart;
        lock (this)
        {
            if (DistanceFromStart > newDistance)
            {
                DistanceFromStart = newDistance;
                PreviousVerticeInRouteIndex = fromVerticeIndex;
                return true;
            }
        }

        return false;
    }
}
```

```
public partial interface IGraph
{
    public float[][] WeightMatrix { get; set; }
    public IVertice[] Vertices { get; }
    public int StartVerticeIndex { get; set; }
    public int FinishVerticeIndex { get; set; }

    public void Reset()
    {
        Parallel.ForEach(Vertices, v => v.Reset());
    }

    public enum FileType
    {
        Binary,
        Text
    }

    public IEnumerable<int> GetAdjacentVertices(int fromVertice, int indexLimit
= int.MaxValue)
    {
        if (!IndexIsInRange(fromVertice))
            throw new IndexOutOfRangeException("Vertice index is out of
matrix");

        var targetRow = WeightMatrix[fromVertice];
        for (int i = 0; i < targetRow.Length && i < indexLimit; i++)
        {
            if (targetRow[i] >= 0)
                yield return i;
        }
```

```
    }

    public IVertice this[int ind]
    {
        get => Vertices[ind];
    }

    public float this[int vertice1, int vertice2]
    {
        get
        {
            if (!IndexIsInRange(vertice1) || !IndexIsInRange(vertice2))
                throw new IndexOutOfRangeException("Vertice index is out of
matrix");
            return WeightMatrix[vertice1][vertice2];
        }
    }

    public void SetStartEndPoint(int startPoint, int endPointIndex);

    public bool IndexIsInRange(int index) => index >= 0 && index <
WeightMatrix.Length;
}
```

Усі наші алгоритми пошуку шляху реалізуватимуть спільний інтерфейс IPathSearchingAlgo. Більшість з них також реалізуватимуть ISingleSidePathSearchingAlgo, код яких наведемо нижче. Як бачимо, дані інтерфейси оголошують та частково реалізують спільні методи, характерні усім розглянутим мною варіаціям A*.

```
public abstract class IPathSearchingAlgo
{
    public int ChildrenCalculatedCounter;
    protected int StartPoint { get; }
    protected int EndPoint { get; }
    protected IGraph _graph { get; }
    protected IPathSearchingAlgo(IGraph graph, int startpoinIndex, int
finishIndex)
    {
        StartPoint = startpoinIndex;
        EndPoint = finishIndex;
        graph.SetStartEndPoint(startpoinIndex, finishIndex);
        _graph = graph;
        ChildrenCalculatedCounter = 0;
    }

    public abstract Task<IVertice?> SearchPath();

    public abstract Stack<int> TraceRoute(IVertice lastVertice);
    public abstract float GetDistance(IVertice lastVertice);
    public int GetGraphSize() => _graph.Vertices.Length;
}
```

```
public abstract class ISingleSidePathSearchingAlgo: IPathSearchingAlgo
{
    protected ISingleSidePathSearchingAlgo(Graph graph, int startpoinIndex, int
finishIndex) : base(graph,
        startpoinIndex, finishIndex)
    {
        ((Vertice)graph[startpoinIndex]).DistanceFromStart = 0;
    }
    public override Stack<int> TraceRoute(IVertice lastVertice)
    {
        Stack<int> route = new Stack<int>(100);
        route.Push(EndPoint);
        Vertice current = (Vertice)_graph[EndPoint];
        while (current.PreviousVerticeInRouteIndex != -1)
        {
            route.Push(current.PreviousVerticeInRouteIndex);
            current =
(Vertice)_graph.Vertices[current.PreviousVerticeInRouteIndex];
        }

        return route;
    }

    protected List<(Vertice vertice, float newDistance)>
CalculateChildren(Vertice parent)
    {
        List<(Vertice vertice, float newDistance)> updateDistances =
            new List<(Vertice vertice, float newDistance)>();
        foreach (var adjIndex in _graph.GetAdjacentVertices(parent.OwnIndex))
        {
            Vertice child = (Vertice)_graph[adjIndex];
            Interlocked.Increment(ref ChildrenCalculatedCounter);
            if (child.IsPassed)
                continue;

            float newDistance = _graph[parent.OwnIndex, child.OwnIndex] +
parent.DistanceFromStart;
            if (child.DistanceFromStart > newDistance)
            {
                updateDistances.Add((child, newDistance));
            }
        }

        return updateDistances;
    }

    public override float GetDistance(IVertice lastVertice)
    {
        return ((Vertice)lastVertice).DistanceFromStart;
    }
}
```

Перейдемо безпосередньо до реалізації класичного алгоритму A*. Єдиний власний метод даного класу – асинхронна (для сумісності з іншими, паралельними реалізаціями) функція SearchPath(), яка фактично і реалізує всю необхідну нам логіку. Із черги, де початково міститься лише стартова вершина,

у порядку зростання сум відстаней від старту та оцінок відстаней до фінішу дістаються вершини. Деякі вершини могли потрапити у чергу двічі, і якщо на момент виходу вони вже пройдені – опрацьовувати їх ще раз немає сенсу.

Далі отриману вершину позначають як пройдену і перевіряють, чи є вона шуканою (фінішною). Якщо так – алгоритм завершує роботу, повертаючи знайдену вершину. У протилежному випадку ми її розширюємо, тобто для усіх її нащадків ми перевіряємо, чи не пройдена вона та чи буде маршрут, що пролягає через поточну вершину, вигіднішим, ніж попередній знайдений, якщо такий був. Якщо так – значення відстані від старту та індексу попередньої вершини у шляху буде оновлено, а саму вершину додано до черги з пріоритетом, який дорівнює сумі нової відстані від старту та сталої евристичної оцінки відстані до фінішу.

І, як бачимо, якщо черга скінчилася, тобто ми пройшли по усім вершинам, доступним з початкової, а шуканої вершини так і не досягнуто – то шляху просто-напросто не існує.

```
public class ConcurrentAStar : ISingleSidePathSearchingAlgo
{
    public ConcurrentAStar(Graph graph, int startpoinIndex, int finishIndex) :
base(graph, startpoinIndex, finishIndex) { }

    public override async Task<IVertice?> SearchPath()
    {
        PriorityQueue<int> verticeQueue = new PriorityQueue<int>();
        Vertice currentVertice;
        verticeQueue.Enqueue(StartPoint, 0);
        while (verticeQueue.Count > 0)
        {
            currentVertice = (Vertice)_graph[verticeQueue.Dequeue()];
            if (currentVertice.IsPassed)
                continue;

            currentVertice.IsPassed = true;
            if (currentVertice.OwnIndex == EndPoint)
                return currentVertice;

            foreach (var adjIndex in
 graph.GetAdjacentVertices(currentVertice.OwnIndex))
            {
                Vertice child = (Vertice)_graph[adjIndex];
                Interlocked.Increment(ref ChildrenCalculatedCounter);
                if (child.TryUpdateMinRoute(currentVertice.OwnIndex))
                    verticeQueue.Enqueue(child.OwnIndex, child.DistanceFromStart
+ child.Heuristic!.Value);
            }
```

```
        }
        return null;
    }
}
```

Повний код алгоритму знаходиться у Додатку А.

2.3 Тестування послідовного алгоритму

Протестуємо алгоритм запустивши його на графі з такою матрицею вагів:

-	*1039,46*	-	*210,85*	-
1039,46	-	*562,47*	-	-
-	*562,47*	-	-	-
210,85	-	-	-	*268,78*
-	-	-	*268,78*	-

Запускаємо програму, обравши за вхідні точки вершини 1 і 4 (нумерація починається з 0), та отримуємо результат:

```
        Матриця вагів:
-       1039,46 -       210,85  -
1039,46 -       562,47  -       -
-       562,47  -       -       -
210,85  -       -       -       268,78
-       -       -       268,78  -
Path found:
        1 --> 0 --> 3 --> 4
```

Рисунок. 2.1. – Знаходження найкоротшого шляху у графі 5*5

Бачимо, що результат виконання цілком коректний. Повторимо тест на графі більшої розмірності, наприклад 10*10. Вхідними точками знову виберемо вершини 1 та 4.

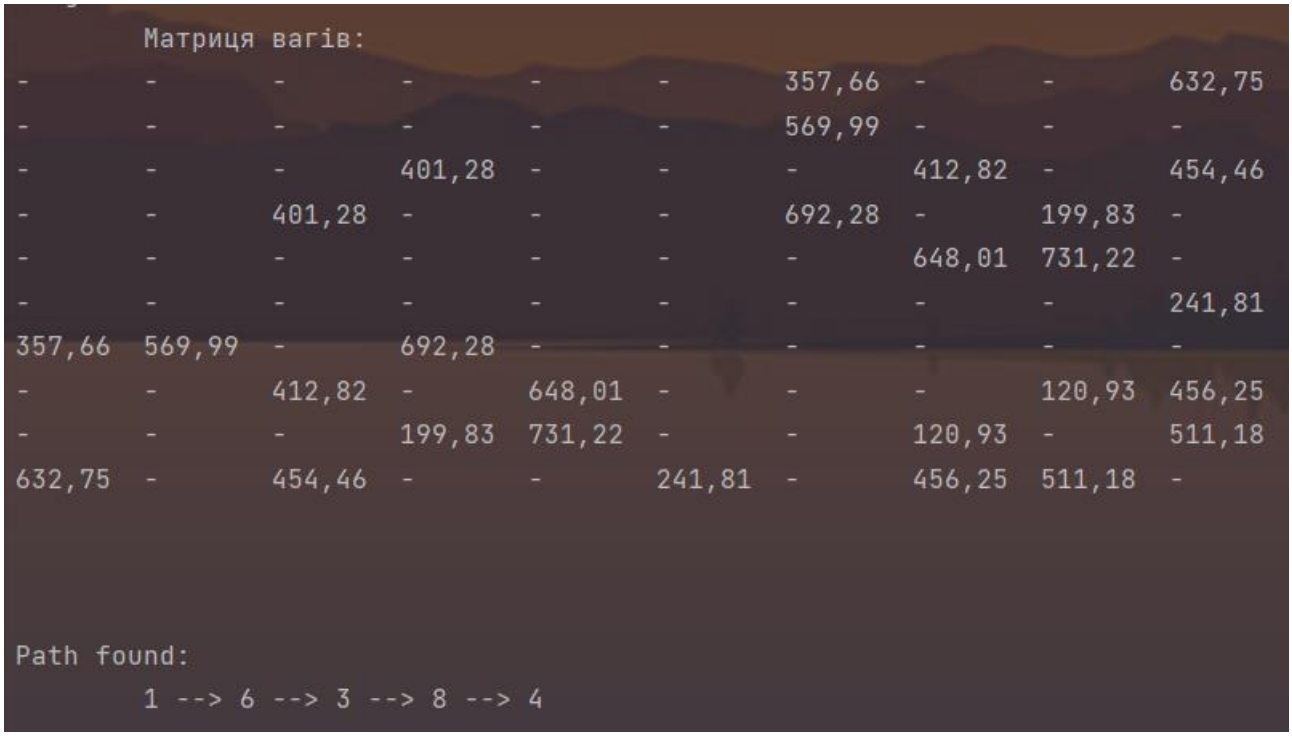

Рисунок. 2.2. – Знаходження найкоротшого шляху у графі 10*10

Як бачимо, результат виконання програми справді правильний. Тобто алгоритм працює коректно.

Результати тестування швидкодії послідовного алгоритму для випадково згенерованих графів з коефіцієнтом зв’язності 0,01 наведено в таблиці 2.1.

Алгоритм генерації графів такий: спершу випадковим чином генеруються координати N вершин. Потім генерується матриця суміжностей розміром N*N з певним коефіцієнтом зв’язності, який фактично означає ймовірність того, що дві випадково обрані вершини зв’язані. У нашому випадку для графів розміром до 1000 застосоувався коефіцієнт зв’язності 0.3, а для більших матриць – 0.01. Після цього будується матриця вагів шляхом знаходження відстаней між пов’язаними вершинами. Точки входу обираються випадковим чином.

Таблиця 2.1. – Швидкодія послідовного алгоритму

Кількість вершин графу	Час послідовного алгоритму, мс
1000	5
2000	7

Продовження таблиці 2.1

Кількість вершин графу	Час послідовного алгоритму, мс
3000	6
4000	14
5000	12
6000	11
7000	10
8000	13
9000	17
10000	17
20000	22
30000	35
40000	51
50000	63
60000	109

Варто зауважити, що час виконання залежить не лише від розміру, а й від довжини шляху між обраними точками. У моєму випадку шлях ніколи не був довшим за 7 вершин, а інколи взагалі складався лише з 2х вершин, тобто вхідна і вихідна точки були суміжними. Очевидно, у таких випадках час виконання був дуже малий незалежно від розміру матриці. Тому я не включав у статистику випадки, коли довжина шляху складала 3 вершини чи менше.

Втім, ми бачимо, що зі збільшенням кількості вершин у графі час все-таки зростає, хоч і не сильно. На жаль, провести тестування на матрицях ще більших розмірностей мені не вдалося через обмеження оперативної пам’яті. Проте бачимо, що час роботи алгоритму на графах 50000*50000 та 60000*60000 відрізняється ледь не вдвічі. Тоді є сенс спробувати розпаралелити даний алгоритм.

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

Для реалізації паралельного алгоритму A* я обрав C#. Це потужна та гнучка мовою, що надає багато можливостей для паралельного програмування. Вона підтримує різні механізми паралельного виконання, такі як потоки, таски, асинхронні функції і т.д..

Одним з найпростіших рішень для паралельного програмування в C# є Parallel Extensions. Ця бібліотека надає зручні та ефективні інструменти для паралельних обчислень, зокрема Parallel.For та Parallel.ForEach, які дозволяють легко розпаралелити виконання циклів.

Task Parallel Library (TPL є ще однією потужною бібліотекою для паралельного програмування в C#. Вона надає спрощений інтерфейс для створення та керування завданнями (tasks) та їхнім паралельним виконанням. TPL також дозволяє автоматично керувати вбудованим пулом потоків.

Насправді це далеко не повний список доступних інструментів для реалізації паралельних обчислень. Наприклад, варта уваги ще бібліотека MPI.NET, яка надає інтерфейс для взаємодії між вузлами кластера через протокол передачі повідомлень (Message Passing Interface). Втім, ця та багато інших технологій не будуть використані у даній роботі, тому на цьому можна зупинитися.

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

4.1 Варіанти паралельної реалізації

На відміну від безлічі інших алгоритмів, розглядаючи можливості паралелізації A* знайти щось вартісне непросто. Під час виконання дано роботи я насправді намагався реалізувати безліч різних варіантів розпаралелювання. Я не стану детально їх пояснювати та наводити код, лиш коротко опишу саму ідею та пов’язані з нею проблеми. Також у наступних розділах буде наведено статистику, таку як час роботи даних реалізацій алгоритму на графах різних розмірностей.

Першим, що спало на думку, було використання Parallel.ForEach для одночасної обробки нащадків певної вершини. Втім, у такому випадку виникає потреба синхронізації пріоритетної черги. Крім того, час ініціалізації тасків співставний з часом їх виконання, тому при паралелізації таких дрібних шматочків коду прискорення не спостерігалося. Навіть навпаки: у більшості випадків час роботи алгоритму навіть трохи зростав. Тому довелося шукати інші шляхи.

Далі мені спало на думку спробувати опрацьовувати елементи черги у кілька потоків. Це дозволило б нам уникнути постійної ініціалізації дрібних тасків. Втім, у такому випадку нам довелося додати велику кількість синхронізації. По-перше, особливості внутрішньої будови пріоритетної черги не дозволяють нам одночасно додавати чи витягувати кілька елементів, оскільки при кожній з цих дій відбувається так звана хіпізація – переребудова купи для збереження її властивостей. І коли кілька потоків намагаються одночасно додавати і витягувати велику кількість елементів з черги, швидкодія обмежується потребою дочекатися минулої операції перед початком наступної. Окрім синхронізації черги нам доводиться також блокувати самі вершини, аби уникнути так званих «перегонів», або Race conditions. Але це ще не все. При

такій реалізації не виконується умова, що з черги завжди беруться лише найвигідніші нам вершини, тобто ми інколи опрацьовуємо ті вершини, які послідовна реалізація взагалі не розглядала б. Це створює додаткове навантаження на чергу, що ще більше сповільнює роботу. Але найбільш критичною проблемою такої реалізації є те, що інколи один з потоків знайде якийсь шлях ще до того, як інший потік покладе у чергу вершину, шлях з якої був би коротшим. Таким чином ми постійно отримуємо шлях близький до найкоротшого, але не завжди найкоротший. Це нівелює одну з найважливіших переваг A*.

Проблему спільного використання черги чудово вирішує такий варіант реалізації A*, як HDA*, описаний у розділі 1. Він використовує окремі черги для кожного з потоків, а вершини розподіляються між чергами за допомогою хеш-функції. Втім, це не вирішує жодну з інших проблем паралельного пошуку шляху. І, як буде видно по статистиці, достатнього прискорення така реалізація не дасть.

Одним з найкращих рішень мені видавався такий варіант реалізації, коли працює з чергою та модифікує вершини лише один потік, що позбавляє нас потреби у синхронізації. Просто нехай при додаванні у чергу елемента на паралельне виконання запускається таск, який опрацює нащадків даного елемента і визначить, які з них треба модифікувати та якими значеннями. Таким чином уся «брудна робота» виконуватиметься пулом потоків, а основний потік лише використовуватиме результати для модифікації вершин та додавання їх до черги. Втім, на мій превеликий подив, така реалізація виявилася найгіршою. По-перше, як і при використанні Parallel.ForEach, час на створення та запуск завдань був більшим аніж фактичний час виконання. По-друге, якщо запускати таски для кожної з вершин, що додаються до черги, ми виконуємо надто багато зайвої роботи, оскільки більшість з результатів ніколи не буде використана. А якщо на момент отримання вершини з черги відповідний їй таск ще не завершив роботу (а то й взагалі не був запущений) –

нам доведеться довго чекати, так як таски виконуються пулом потоків стохастично, зовсім не відштовхуючись від того, наскільки потрібні зараз ті чи інші результати. Тому час роботи такого алгоритму у сотні разів перевищував час виконання класичної реалізації.

Пізніше у цілях оптимізації я спробував запускати таски лише для найкращих з отриманих нащадків. Це сильно допомогло прискорити роботу, але послідовний алгоритм все ще лишався швидшим.

4.2 Проектування та реалізація паралельного алгоритму

Після усіх вище описаних спроб і провалів я зрозумів, що, найймовірніше, єдиною можливістю паралелізувати знаходження найкоротшого шляху є всього лиш одночасних пошук шляху зі стартової вершини до кінцевої і навпаки. Тобто ми просто двічі запустимо на одному й тому ж графі, просто з двох боків, класичний A*. Тоді умовою дострокового припинення роботи буде не потрапляння у кінцеву точку, а перетин областей пошуку двох потоків. Щоправда, для цього доведеться модифікувати клас вершини, аби він міг зберігати свій стан відносно обох запущених алгоритмів пошуку. Відповідно, клас Graph, що містив методи для роботи з вершинами, теж доведеться змінити. Таким чином з'явилися класи BilateralGraph та BilateralVertice, що реалізують інтерфейси IGraph та IVertice відповідно.

```
public class BilateralVertice: IVertice
{
    public float[] DistanceFromStart { get; set; }
    public Coordinates VerticeCoordinates { get; }
    public int OwnIndex { get; }
    public float?[] Heuristic { get; private set; }
    public int[] PreviousVerticeInRouteIndex { get; set; }
    public bool[] IsPassed;
    private IGraph _graph;

...

    public bool TryUpdateMinRoute(int fromVerticeIndex, int processIndex)
    {
        if (IsPassed[processIndex] || _graph[fromVerticeIndex, OwnIndex] == -1)
            return false;

        float newDistance = _graph[fromVerticeIndex, OwnIndex] +
((BilateralVertice)_graph[fromVerticeIndex]).DistanceFromStart[processIndex];
        lock (this)
```

```
        {
            if (DistanceFromStart[processIndex] > newDistance)
            {
                DistanceFromStart[processIndex] = newDistance;
                PreviousVerticeInRouteIndex[processIndex] = fromVerticeIndex;
                return true;
            }
        }
        return false;
    }

    public void SetHeuristic(IVertice start, IVertice finish)
    {
        Heuristic = new float?[2];
        Heuristic[0] = (float)Math.Sqrt(Math.Pow(VerticeCoordinates.X -
finish.VerticeCoordinates.X, 2) +
                                        Math.Pow(VerticeCoordinates.Y -
finish.VerticeCoordinates.Y, 2));
        Heuristic[1] = (float)Math.Sqrt(Math.Pow(VerticeCoordinates.X -
start.VerticeCoordinates.X, 2) +
                                            Math.Pow(VerticeCoordinates.Y -
start.VerticeCoordinates.Y, 2));
    }

    public void Reset()
    {
        DistanceFromStart = new[] { float.MaxValue / 2, float.MaxValue / 2 };
        PreviousVerticeInRouteIndex = new[] { -1, -1 };
        Heuristic = new float?[] { null, null };
        IsPassed = new [] { false, false };
    }
}
```

```
public class BilateralGraph : IGraph
{
    public float[][] WeightMatrix { get; set; }
    public IVertice[] Vertices { get; }
    public int StartVerticeIndex { get; set; }
    public int FinishVerticeIndex { get; set; }

    public void Reset()
    {
        Parallel.ForEach(Vertices, v => v.Reset());
    }

...

    public void SetStartEndPoint(int startPointIndex, int endPointIndex)
    {
        FinishVerticeIndex = endPointIndex;
        IVertice finish = Vertices[endPointIndex];
        IVertice start = Vertices[startPointIndex];
        Parallel.ForEach(Vertices, vertice => vertice.SetHeuristic(start,
finish));
    }
}
```

При такій моделі пошуку також доведеться змінити методи для трасування знайденого шляху та пошуку його довжини. Тому доведеться

створити інтерфейс ITwoSidePathSearchingAlgo як противагу SingleSidePathSearchingAlgo.

```
public abstract class ITwoSidePathSearchingAlgo: IPathSearchingAlgo
{
    protected ITwoSidePathSearchingAlgo(IGraph graph, int startpoinIndex, int
finishIndex) : base(graph,
        startpoinIndex, finishIndex)
    {
        ((BilateralVertice)graph[startpoinIndex]).DistanceFromStart[0] = 0;
        ((BilateralVertice)graph[finishIndex]).DistanceFromStart[1] = 0;
    }
    public override Stack<int> TraceRoute(IVertice lastVertice)
    {
        Stack<int> routeToEnd = new Stack<int>(100);
        Stack<int> route = new Stack<int>(100);

        BilateralVertice current = (BilateralVertice)lastVertice;
        while (current.PreviousVerticeInRouteIndex[1] != -1)
        {
            routeToEnd.Push(current.PreviousVerticeInRouteIndex[1]);
            current =
(BilateralVertice)_graph.Vertices[current.PreviousVerticeInRouteIndex[1]];
        }

        while (routeToEnd.Count > 0)
        {
            route.Push(routeToEnd.Pop());
        }

        route.Push(lastVertice.OwnIndex);
        current = (BilateralVertice)lastVertice;
        while (current.PreviousVerticeInRouteIndex[0] != -1)
        {
            route.Push(current.PreviousVerticeInRouteIndex[0]);
            current =
(BilateralVertice)_graph.Vertices[current.PreviousVerticeInRouteIndex[0]];
        }

        return route;
    }

    public override float GetDistance(IVertice lastVertice)
    {
        return ((BilateralVertice)lastVertice).DistanceFromStart.Sum();
    }
}
```

Тепер почнемо реалізувати сам алгоритм. Як я вже казав. Його задум доволі простий: два потоки одночасно шукають шлях один назустріч одному, допоки їх області пошуку не перетнуться. Враховуючи природу алгоритму A*, дані області пошуку обов’язково перетнуться, якщо шлях між обраними вершинами взагалі існує.

```
public class BilateralAStar : ITwoSidePathSearchingAlgo
{
    private BilateralVertice? meet;
    public BilateralAStar(BilateralGraph graph, int startpoinIndex, int
finishIndex) : base(graph, startpoinIndex, finishIndex) { }

    public override async Task<IVertice?> SearchPath()
    {
        List<Task<bool>> searchTasks = new List<Task<bool>>();
        searchTasks.Add(Task.Run(() => SingleThreadPathSearching(0)));
        searchTasks.Add(Task.Run(() => SingleThreadPathSearching(1)));

        await Task.WhenAny(searchTasks);
        return meet;
    }
}
```

При чому, сам пошук практично не відрізнятиметься від класичної реалізації, за винятком того, що замість досягнення кінцевої точки ми очікуємо всього лиш на перетин з областю пошуку іншого потоку, а також варто передбачити завершення роботи у тому випадку, коли інший потік знайшов рішення.

```
public async Task<bool> SingleThreadPathSearching(int procInd)
{
    PriorityQueue<int> verticeQueue = new PriorityQueue<int>();
    BilateralVertice currentVertice;
    verticeQueue.Enqueue(procInd == 0? StartPoint : EndPoint, 0);
    while (verticeQueue.Count > 0 && meet is null)
    {
        currentVertice = (BilateralVertice)_graph[verticeQueue.Dequeue()];
        if (currentVertice.IsPassed[procInd])
            continue;

        currentVertice.IsPassed[procInd] = true;
        if (currentVertice.IsPassed[(procInd+1)%2])
        {
            if (meet is null || meet.DistanceFromStart.Sum() >
currentVertice.DistanceFromStart.Sum())
                meet = currentVertice;
            return true;
        }

        foreach (var adjIndex in
_graph.GetAdjacentVertices(currentVertice.OwnIndex))
        {
            BilateralVertice child = (BilateralVertice)_graph[adjIndex];
            Interlocked.Increment(ref ChildrenCalculatedCounter);
            if (child.TryUpdateMinRoute(currentVertice.OwnIndex, procInd))
                verticeQueue.Enqueue(child.OwnIndex,
                    child.DistanceFromStart[procInd] +
child.Heuristic[procInd].Value);
        }
    }
```

```
    return false;
}
```

Повний код цього та інших паралельних алгоритмів у Додатку Б.

4.3 Тестування паралельного алгоритму

Ми вже переконалися у коректності роботи послідовного алгоритму. Тепер існує два шляхи тестування паралельних реалізацій: перевірка вручну або порівняння з результатами виконання послідовного алгоритму. Варто зауважити, що інколи існує два найкращих шляхи однакової довжини, і немає нічого страшного, якщо різні реалізації алгоритму повернуть різні з них. Головне – аби збігалася довжина шляху. У цілях тестування запустимо алгоритм на графі 60000*60000 та порівняємо результати (рисунок 4.1). При таких розмірностях ймовірність випадкового абсолютно точного збігу не допускатимемо.

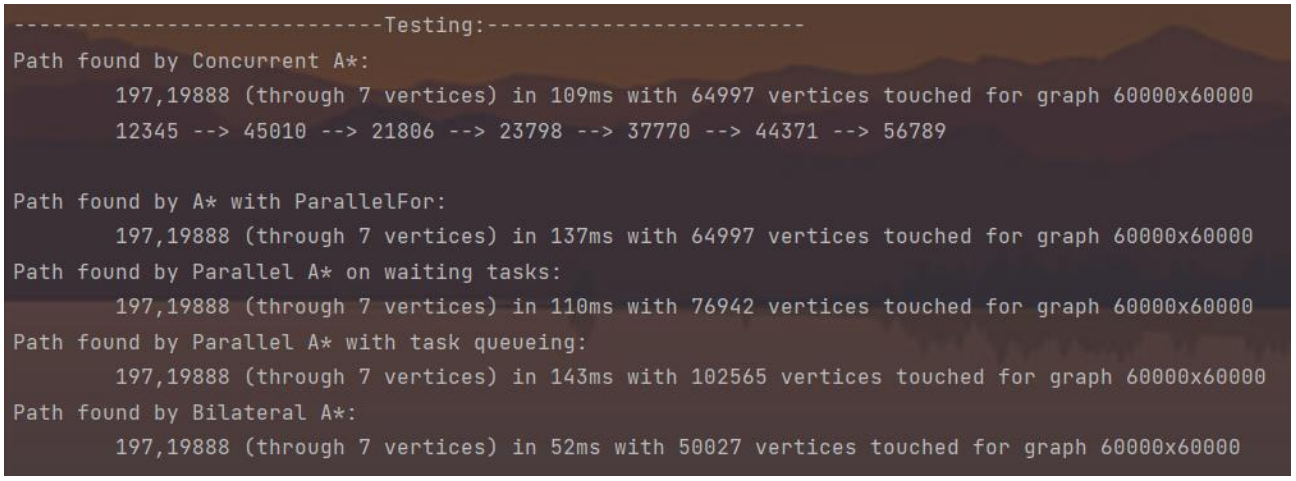

Рисунок 4.1. – Результат тестування різних алгоритмів на одному й тому ж графі розмірністю 60000*60000

Як бачимо зі скріншота, усі алгоритми повернули однаковий результат (як по відстані, так і по кількості вершин), щоправда, за різний час і з різною кількістю розглянутих вершин. Отже, усі реалізації працюють коректно.

Насправді дане зображення чудово ілюструє і повністю підтверджує мої слова про проблеми тих чи інших реалізацій. Наприклад, про те, що накладні витрати на ініціалізацію тасків переважають користь від паралелізації з

використанням Parallel.ForEach. Крім того, тут чудово видно, що підхід з запуском тасків при закиданні елемента у чергу призводить до сильного зростання кількості переглянутих вершин (примітка – на скріншоті цей метод уже оптимізований, щоб запускати таски лише до найбільш вигідних вершин, а до того ситуація була просто жахлива).

І, як бачимо, для такої розмірності графа найоптимальнішим все ж виявився двосторонній пошук, який шляхом паралельної обробки вершин (а також зменшення кількості розглянутих) дозволив нам отримати прискорення більше ніж у два рази! У наступних розділах порівняємо швидкодію алгоритмів детальніше.

5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ

Задля дослідження ефективності паралельних реалізацій алгоритму було проведено серію експериментів для графів різних розмірностей. В межах одного експерименту усі алгоритми тестувалися на одних і тих же вхідних даних, аби бути в рівних умовах. Важливу роль відіграє також те, що складність обчислень залежить далеко не лише від розміру графу, а й від підібраних точок входу та виходу. Тому запуск кожного алгоритму на різних графах був би нераціональним і не дозволив би нам об’єктивно оцінити ефективність алгоритмів.

Також відмітимо, що статистика, зібрана з графів, шлях у яких складається менше як з 3 вершин, не є інформативною, тому такі випадки не включалися у загальну статистику. У таблиці 5.1 наведені результати тестування кожного з розроблених мною алгоритмів на графах різних розмірностей.

Дослідження проводилися на ноутбуці Asus ROG GU501GM з шестиядерним процесором Intel Core i7-8750H CPU @ 2.20GHz (12 логічних процесорів) та 32ГБ оперативної пам’яті.

Таблиця 5.1. – Залежність часу виконання алгоритмів від розмірів графу.

Кількість вершин графу	Середній час послідовного A*, ms	Середній час A* з ParallelFor, ms	Середній час багатопоточ-ного A* зі спільною чергою, ms	Середній час одно-поточного A* з чергою тасків, ms	Середній час двосторон-нього A*, ms
1000	5	21	6	10	5
2000	7	26	7	11	5
3000	6	20	6	12	5

Продовдення таблиці 5.1.

Кількість вершин графу	Середній час послідовного A*, ms	Середній час A* з ParallelFor, ms	Середній час багатопоточ-ного A* зі спільною чергою, ms	Середній час одно-поточного A* з чергою тасків, ms	Середній час двосторон-нього A*, ms
4000	14	26	10	16	6
5000	12	22	7	11	6
6000	11	34	12	15	7
7000	10	37	12	16	8
8000	13	45	15	17	11
9000	17	71	19	23	14
10000	17	52	19	21	14
20000	22	58	28	29	17
30000	35	63	30	41	29
40000	51	127	210	159	28
50000	63	81	201	182	30
60000	109	137	110	143	52

Для наочності отриманих результатів побудуємо графіки по даній таблиці. На рисунку 5.1 представлений графік залежності часу виконання від розмірностей графу

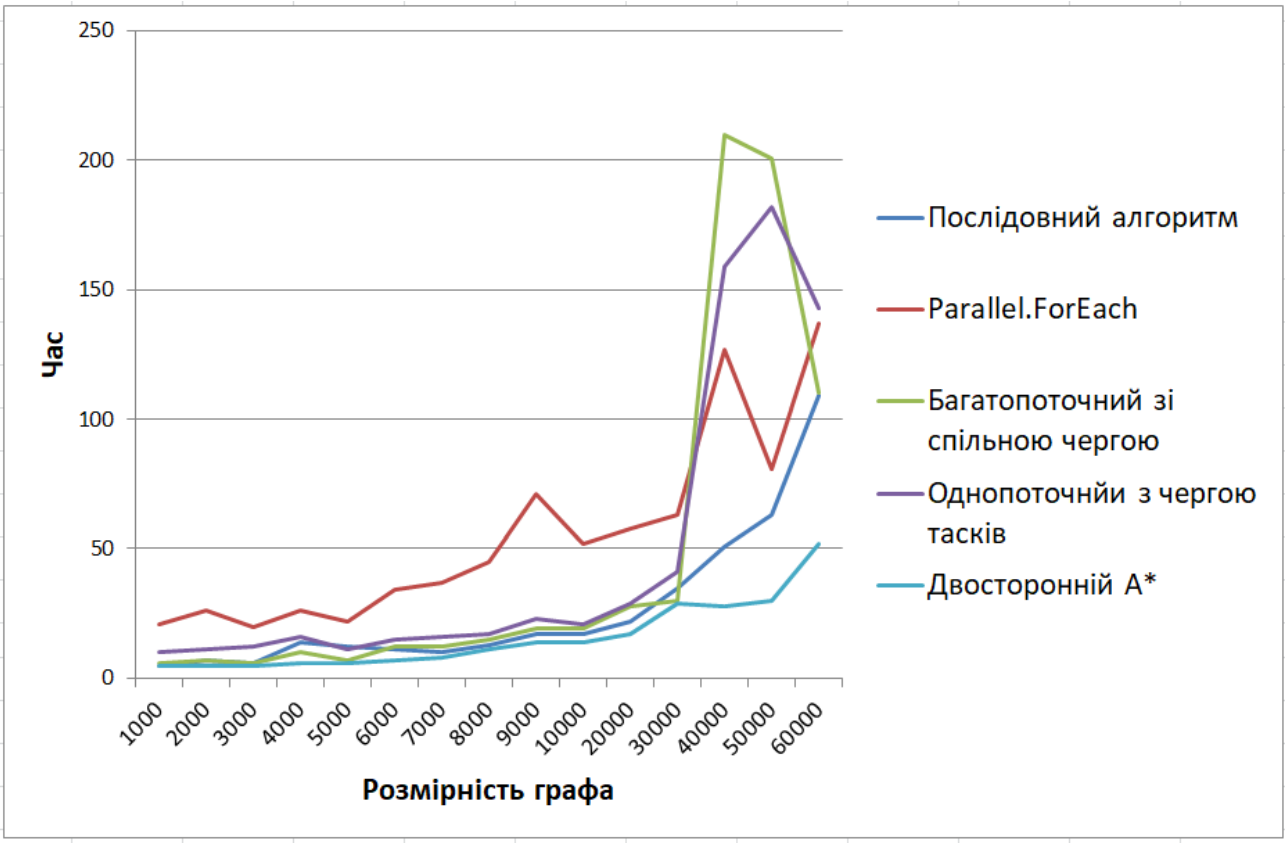

Рисунок 5.1. – Залежності часу виконання різних варіантів реалізації алгоритму А* від розміру графу

Як бачимо, двосторонній пошук виявився найшвидшим незалежно від розміру графу, в той час як інші паралельні варіанти реалізації явно поступаються класичній версії. Тепер обрахуємо прискорення кожного з алгоритмів (таблиця 5.1) та побудуємо їх графіки (рисунок 5.2).

Таблиця 5.2. – Залежність прискорення паралельних алгоритмів від розмірів графу.

Кількість елементів	Прискорення А* з ParallelFor, ms	Прискорення багатопоточ-ного А* зі спільною чергою, ms	Прискорення одно-поточного А* з чергою тасків, ms	Присокрення двосторон-нього А*, ms
1000	0,23809524	0,8333	0,5	1

Продовження таблиці 5.2

Кількість елементів	Прискорення A* з ParallelFor, ms	Прискорення багатопоточ-ного A* зі спільною чергою, ms	Прискорення одно-поточного A* з чергою тасків, ms	Присокрення двосторон-нього A*, ms
1000	0,23809524	0,8333	0,5	1
2000	0,26923077	1	0,6364	1,4
3000	0,3	1	0,5	1,2
4000	0,53846154	1,4	0,875	2,3333
5000	0,54545455	1,7143	1,0909	2
6000	0,32352941	0,9167	0,7333	1,5714
7000	0,27027027	0,8333	0,625	1,25
8000	0,28888889	0,8667	0,7647	1,1818
9000	0,23943662	0,8947	0,7391	1,2143
10000	0,32692308	0,8947	0,8095	1,2143
20000	0,37931034	0,7857	0,7586	1,2941
30000	0,55555556	1,1667	0,8537	1,2069
40000	0,4015748	0,2429	0,3208	1,8214
50000	0,77777778	0,3134	0,3462	2,1
60000	0,79562044	0,9909	0,7622	2,0962

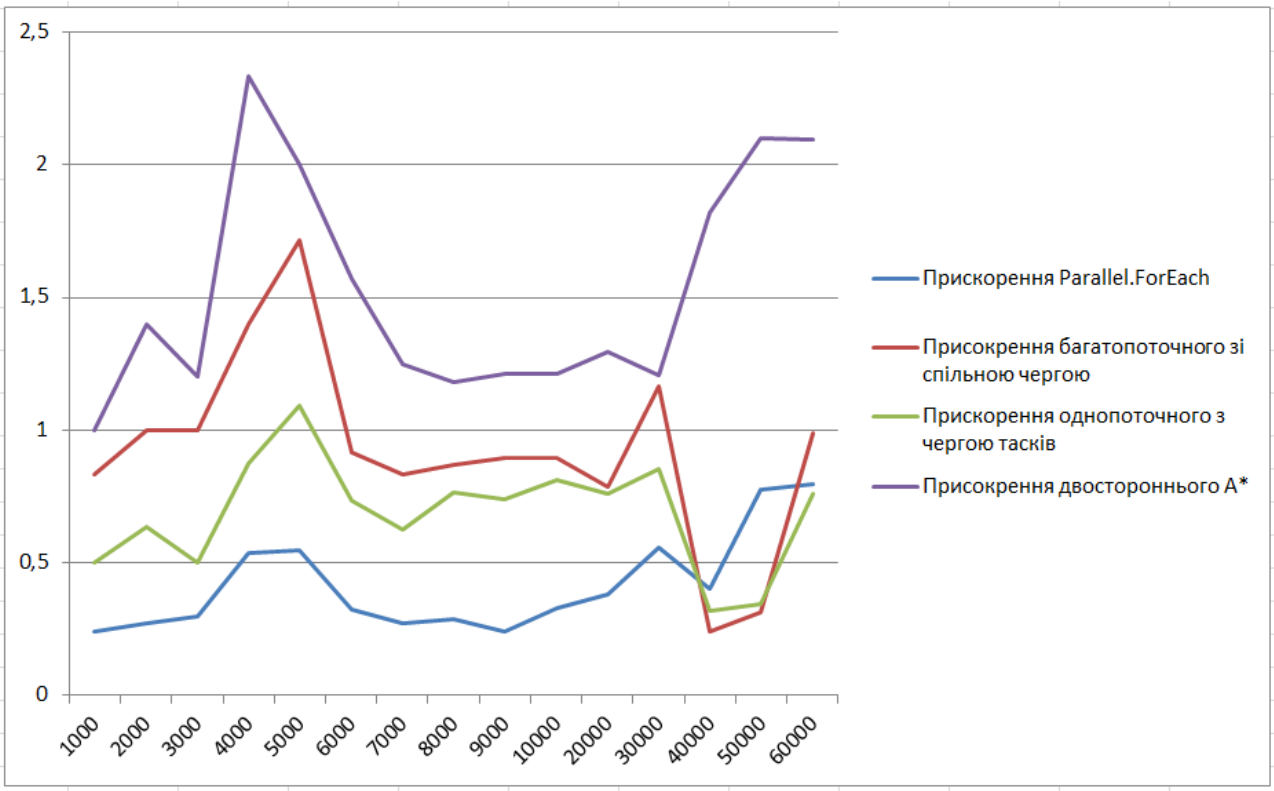

Рисунок 5.2. – Графік залежності прискорення паралельних реалізацій алгоритму A* від розміру графу

Отже, як бачимо, прискорення двостороннього A* при достатніх розмірах графа навіть перевищує 2, не говорячи вже про 1.2, тому дану курсову роботу можна вважати успішно виконаною.

```
----------------------------Testing:------------------------
Path found by Concurrent A*:
        117,1136 (through 4 vertices) in 65ms with 47897 vertices touched for graph 40000x40000
        0 --> 20425 --> 16666 --> 6789

Path found by A* with ParallelFor:
        117,1136 (through 4 vertices) in 51ms with 47897 vertices touched for graph 40000x40000
Path found by Parallel A* on waiting tasks:
        117,11608 (through 3 vertices) in 173ms with 240306 vertices touched for graph 40000x40000
Path found by Parallel A* with task queueing:
        117,1136 (through 4 vertices) in 86ms with 313261 vertices touched for graph 40000x40000
Path found by Bilateral A*:
        117,113594 (through 4 vertices) in 50ms with 71806 vertices touched for graph 40000x40000

Process finished with exit code 0.
```

Рисунок 5.3. – Ще один приклад тестування часу роботи різних варіантів реалізації алгоритму A* для графу розмірністю 40000*40000

ВИСНОВКИ

Під час виконання даної курсової роботи мною було досліджено та описано алгоритм A* у його класичній та паралельній реалізації, спроектовано, описано, реалізовано та проаналізовано ще 4 мої власні паралельні реалізації алгоритму A*. Більшість з цих алгоритмів не змогли дати достатнього прискорення порівняно з класичною версією, причини чого теж були ґрунтовно проаналізовані та пояснені. Втім, один з підходів дозволив отримати прискорення до 2х разів за рахунок зменшення кількості переглянутих вершин та паралелізації обчислень.

Для кожного з розглянутих варіантів реалізації алгоритму було здійснено перевірку коректності роботи, а також побудовано збірну порівняльну таблицю залежності часу виконання кожного з алгоритмів від розмірності графа, на якому проводиться пошук. На основі цієї таблиці було знайдено прискорення тих чи інших алгоритмів та побудовано їх графіки. Ми також вкотре довели, що час роботи залежить не стільки від розмірів графа, як від обраних вхідних точок, тому всі дослідження в межах одного експерименту проводилися з абсолютно однаковими вхідними даними.

Неосновною, але не менш цікавою частиною роботи була реалізація власної пріоритетної черги, а також її потокобезпечного варіанту. Дані колекції по швидкодії не уступають вбудованій реалізації пріоритетної черги. Втім, швидкодія деяких алгоритмів впиралася у їх низьку пропускну спроможність.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Grama and V. Kumar, “Parallel search algorithms for discrete optimization problems,” ORSA Journal on Computing, vol. 7, no. 4, cc. 365–385, 1995.
2. Rafia, “A* algorithm for multicore graphics processors,” 2010.
3. Weinstock and R. Holladay. “Parallel A* Graph Search”[Електронний ресурс] / Ariana Weinstock and Rachel Holladay // MIT – Режим доступу до ресурсу: https://people.csail.mit.edu/rholladay/docs/parallel_search_report.pdf
4. Microsoft Learn. “How to: Write a simple Parallel.ForEach loop” [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-write-a-simple-parallel-foreach-loop
5. Microsoft Learn. “Task.Run Method” [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.run?view=net-7.0

ДОДАТКИ

Буде наведено лише найважливіші частини коду програми. Повний код можна знайти за посиланням:

https://github.com/CherpakAndrii/ParallelCourseWork

Додаток А. Код послідовного алгоритму

```
public partial interface IGraph
{
    public float[][] WeightMatrix { get; set; }
    public IVertice[] Vertices { get; }
    public int StartVerticeIndex { get; set; }
    public int FinishVerticeIndex { get; set; }

    public void Reset()
    {
        Parallel.ForEach(Vertices, v => v.Reset());
    }

    public enum FileType
    {
        Binary,
        Text
    }

    public IEnumerable<int> GetAdjacentVertices(int fromVertice, int indexLimit
= int.MaxValue)
    {
        if (!IndexIsInRange(fromVertice))
            throw new IndexOutOfRangeException("Vertice index is out of
matrix");

        var targetRow = WeightMatrix[fromVertice];
        for (int i = 0; i < targetRow.Length && i < indexLimit; i++)
        {
            if (targetRow[i] >= 0)
                yield return i;
        }
    }

    public IVertice this[int ind]
    {
        get => Vertices[ind];
    }

    public float this[int vertice1, int vertice2]
    {
        get
        {
            if (!IndexIsInRange(vertice1) || !IndexIsInRange(vertice2))
                throw new IndexOutOfRangeException("Vertice index is out of
matrix");
            return WeightMatrix[vertice1][vertice2];
        }
    }

    public void SetStartEndPoint(int startPoint, int endPointIndex);

    public bool IndexIsInRange(int index) => index >= 0 && index <
```

```
WeightMatrix.Length;
}

public class Graph : IGraph
{
    public float[][] WeightMatrix { get; set; }
    public IVertice[] Vertices { get; }
    public int StartVerticeIndex { get; set; }
    public int FinishVerticeIndex { get; set; }

    public Graph(int size)
    {
        if (size < 3)
            throw new ArgumentException("The graph must contain at least 3
vertices!");

        Vertices = GraphGenerator.GenerateVertices(size, this);
        Console.WriteLine("Vertices generated!");
        bool[][] _adjMatrix = GraphGenerator.GenerateAdjacenceMatrix(size, size
< 1000? 0.3f : 0.01f);
        Console.WriteLine("Adj matrix generated!");
        WeightMatrix = IGraph.BuildWeightMatrix(Vertices, _adjMatrix);
        Console.WriteLine("Weight matrix built!");
    }

    public void Reset()
    {
        Parallel.ForEach(Vertices, v => v.Reset());
    }

    public Graph(string sourceFilePath, IGraph.FileType fileType)
    {
        (bool[][] adjMatrix, (int, int)[] coordinates) = fileType ==
IGraph.FileType.Text
            ? IGraph.ReadFromTxtFile(sourceFilePath)
            : IGraph.ReadFromBinFile(sourceFilePath);
        Console.WriteLine("File reading done");
        Vertices = new Vertice[coordinates.Length];
        for (int i = 0; i < coordinates.Length; i++)
        {
            Vertices[i] = new Vertice(this, i, coordinates[i]);
        }
        WeightMatrix = IGraph.BuildWeightMatrix(Vertices, adjMatrix);
    }

    public IVertice this[int ind]
    {
        get => Vertices[ind];
    }

    public float this[int vertice1, int vertice2]
    {
        get
        {
            if (!IndexIsInRange(vertice1) || !IndexIsInRange(vertice2))
                throw new IndexOutOfRangeException("Vertice index is out of
matrix");
            return WeightMatrix[vertice1][vertice2];
        }
    }

    public Graph(IGraph original)
    {
```

```
        WeightMatrix = original.WeightMatrix;
        Vertices = new IVertice[original.Vertices.Length];
        for (int i = 0; i < Vertices.Length; i++)
        {
            Vertices[i] = original.Vertices[i].Copy();
        }
    }

    public void SetStartEndPoint(int startPoint, int endPointIndex)
    {
        FinishVerticeIndex = endPointIndex;
        IVertice finish = Vertices[endPointIndex];
        IVertice start = Vertices[startPoint];
        Parallel.ForEach(Vertices, vertice => vertice.SetHeuristic(start,
finish));
    }

    private bool IndexIsInRange(int index) => index >= 0 && index <
WeightMatrix.Length;
}

public interface IVertice
{
    public Coordinates VerticeCoordinates { get; }

    public int OwnIndex { get; }

    public void Reset();
    public IVertice Copy();
    public void SetHeuristic(IVertice start, IVertice finish);
}

public class Vertice : IVertice
{
    public float DistanceFromStart { get; set; }
    public Coordinates VerticeCoordinates { get; }
    public float? Heuristic { get; private set; }
    public int PreviousVerticeInRouteIndex { get; set; }
    public bool IsPassed;

    public int OwnIndex { get; }
    private IGraph Graph { get; }

    public Vertice(IGraph graph, int ownIndex, (int x, int y) coordinates)
    {
        VerticeCoordinates = new Coordinates(coordinates);
        DistanceFromStart = float.MaxValue / 2;
        PreviousVerticeInRouteIndex = -1;
        IsPassed = false;
        OwnIndex = ownIndex;
        Graph = graph;
    }

    public IVertice Copy()
    {
        return new Vertice(Graph, OwnIndex, (VerticeCoordinates.X,
VerticeCoordinates.Y));
    }

    public void SetHeuristic(IVertice start, IVertice finish)
    {
        Heuristic = (float)Math.Sqrt(Math.Pow(VerticeCoordinates.X -
finish.VerticeCoordinates.X, 2) +
```

```
                                    Math.Pow(VerticeCoordinates.Y -
finish.VerticeCoordinates.Y, 2));
    }

    public bool TryUpdateMinRoute(int fromVerticeIndex)
    {
        if (IsPassed || Graph[fromVerticeIndex, OwnIndex] == -1)
            return false;

        float newDistance = Graph[fromVerticeIndex, OwnIndex] +
((Vertice)Graph[fromVerticeIndex]).DistanceFromStart;
        lock (this)
        {
            if (DistanceFromStart > newDistance)
            {
                DistanceFromStart = newDistance;
                PreviousVerticeInRouteIndex = fromVerticeIndex;
                return true;
            }
        }

        return false;
    }

    public void Reset()
    {
        DistanceFromStart = float.MaxValue / 2;
        PreviousVerticeInRouteIndex = -1;
        Heuristic = null;
        IsPassed = false;
    }
}

public abstract class IPathSearchingAlgo
{
    public int ChildrenCalculatedCounter;
    protected int StartPoint { get; }
    protected int EndPoint { get; }
    protected IGraph _graph { get; }
    protected IPathSearchingAlgo(IGraph graph, int startpoinIndex, int
finishIndex)
    {
        StartPoint = startpoinIndex;
        EndPoint = finishIndex;
        graph.SetStartEndPoint(startpoinIndex, finishIndex);
        _graph = graph;
        ChildrenCalculatedCounter = 0;
    }

    public abstract Task<IVertice?> SearchPath();

    public abstract Stack<int> TraceRoute(IVertice lastVertice);
    public abstract float GetDistance(IVertice lastVertice);
    public int GetGraphSize() => _graph.Vertices.Length;
}

public abstract class ISingleSidePathSearchingAlgo: IPathSearchingAlgo
{
    protected ISingleSidePathSearchingAlgo(Graph graph, int startpoinIndex, int
finishIndex) : base(graph,
        startpoinIndex, finishIndex)
    {
        ((Vertice)graph[startpoinIndex]).DistanceFromStart = 0;
```

```
    }
    public override Stack<int> TraceRoute(IVertice lastVertice)
    {
        Stack<int> route = new Stack<int>(100);
        route.Push(EndPoint);
        Vertice current = (Vertice)_graph[EndPoint];
        while (current.PreviousVerticeInRouteIndex != -1)
        {
            route.Push(current.PreviousVerticeInRouteIndex);
            current =
(Vertice)_graph.Vertices[current.PreviousVerticeInRouteIndex];
        }

        return route;
    }

    protected List<(Vertice vertice, float newDistance)>
CalculateChildren(Vertice parent)
    {
        List<(Vertice vertice, float newDistance)> updateDistances =
            new List<(Vertice vertice, float newDistance)>();
        foreach (var adjIndex in _graph.GetAdjacentVertices(parent.OwnIndex))
        {
            Vertice child = (Vertice)_graph[adjIndex];
            Interlocked.Increment(ref ChildrenCalculatedCounter);
            if (child.IsPassed)
                continue;

            float newDistance = _graph[parent.OwnIndex, child.OwnIndex] +
parent.DistanceFromStart;
            if (child.DistanceFromStart > newDistance)
            {
                updateDistances.Add((child, newDistance));
            }
        }

        return updateDistances;
    }

    public override float GetDistance(IVertice lastVertice)
    {
        return ((Vertice)lastVertice).DistanceFromStart;
    }
}

public class ConcurrentAStar : ISingleSidePathSearchingAlgo
{
    public ConcurrentAStar(Graph graph, int startpoinIndex, int finishIndex) :
base(graph, startpoinIndex, finishIndex) { }

    public override async Task<IVertice?> SearchPath()
    {
        PriorityQueue<int> verticeQueue = new PriorityQueue<int>();
        Vertice currentVertice;
        verticeQueue.Enqueue(StartPoint, 0);
        while (verticeQueue.Count > 0)
        {
            currentVertice = (Vertice)_graph[verticeQueue.Dequeue()];
            if (currentVertice.IsPassed)
                continue;

            currentVertice.IsPassed = true;
            if (currentVertice.OwnIndex == EndPoint)
```

```
                return currentVertice;

            foreach (var adjIndex in
_graph.GetAdjacentVertices(currentVertice.OwnIndex))
            {
                Vertice child = (Vertice)_graph[adjIndex];
                Interlocked.Increment(ref ChildrenCalculatedCounter);
                if (child.TryUpdateMinRoute(currentVertice.OwnIndex))
                    verticeQueue.Enqueue(child.OwnIndex, child.DistanceFromStart
+ child.Heuristic!.Value);
            }
        }

        return null;
    }
}
```

Додаток Б. Код паралельного алгоритму

```
public class BilateralVertice: IVertice
{
    public float[] DistanceFromStart { get; set; }
    public Coordinates VerticeCoordinates { get; }
    public int OwnIndex { get; }
    public float?[] Heuristic { get; private set; }
    public int[] PreviousVerticeInRouteIndex { get; set; }
    public bool[] IsPassed;
    private IGraph _graph;

    public BilateralVertice(IGraph graph, int ownIndex, (int x, int y)
coordinates)
    {
        VerticeCoordinates = new Coordinates(coordinates);
        DistanceFromStart = new [] { float.MaxValue / 2, float.MaxValue / 2 };
        PreviousVerticeInRouteIndex = new[] { -1, -1 };
        IsPassed = new[] { false, false };
        OwnIndex = ownIndex;
        _graph = graph;
    }

    public IVertice Copy()
    {
        return new BilateralVertice(_graph, OwnIndex, (VerticeCoordinates.X,
VerticeCoordinates.Y));
    }

    public bool TryUpdateMinRoute(int fromVerticeIndex, int processIndex)
    {
        if (IsPassed[processIndex] || _graph[fromVerticeIndex, OwnIndex] == -1)
            return false;

        float newDistance = _graph[fromVerticeIndex, OwnIndex] +
((BilateralVertice)_graph[fromVerticeIndex]).DistanceFromStart[processIndex];
        lock (this)
        {
            if (DistanceFromStart[processIndex] > newDistance)
            {
                DistanceFromStart[processIndex] = newDistance;
                PreviousVerticeInRouteIndex[processIndex] = fromVerticeIndex;
                return true;
            }
        }
        return false;
    }

    public void SetHeuristic(IVertice start, IVertice finish)
    {
        Heuristic = new float?[2];
        Heuristic[0] = (float)Math.Sqrt(Math.Pow(VerticeCoordinates.X -
finish.VerticeCoordinates.X, 2) +
                                        Math.Pow(VerticeCoordinates.Y -
finish.VerticeCoordinates.Y, 2));
        Heuristic[1] = (float)Math.Sqrt(Math.Pow(VerticeCoordinates.X -
start.VerticeCoordinates.X, 2) +
                                           Math.Pow(VerticeCoordinates.Y -
start.VerticeCoordinates.Y, 2));
    }

    public void Reset()
    {
```

```
        DistanceFromStart = new[] { float.MaxValue / 2, float.MaxValue / 2 };
        PreviousVerticeInRouteIndex = new[] { -1, -1 };
        Heuristic = new float?[] { null, null };
        IsPassed = new [] { false, false };
    }
}

public class BilateralGraph : IGraph
{
    public float[][] WeightMatrix { get; set; }
    public IVertice[] Vertices { get; }
    public int StartVerticeIndex { get; set; }
    public int FinishVerticeIndex { get; set; }

    public BilateralGraph(int size)
    {
        if (size < 3)
            throw new ArgumentException("The graph must contain at least 3
vertices!");

        Vertices = GraphGenerator.GenerateVertices(size, this);
        Console.WriteLine("Vertices generated!");
        bool[][] _adjMatrix = GraphGenerator.GenerateAdjacenceMatrix(size,
0.3f);
        Console.WriteLine("Adj matrix generated!");
        WeightMatrix = IGraph.BuildWeightMatrix(Vertices, _adjMatrix);
        Console.WriteLine("Weight matrix built!");
    }

    public void Reset()
    {
        Parallel.ForEach(Vertices, v => v.Reset());
    }

    public BilateralGraph(string sourceFilePath, IGraph.FileType fileType)
    {
        (bool[][] adjMatrix, (int, int)[] coordinates) = fileType ==
IGraph.FileType.Text
            ? IGraph.ReadFromTxtFile(sourceFilePath)
            : IGraph.ReadFromBinFile(sourceFilePath);
        Console.WriteLine("File reading done");
        Vertices = new BilateralVertice?[coordinates.Length];
        for (int i = 0; i < coordinates.Length; i++)
        {
            Vertices[i] = new BilateralVertice(this, i, coordinates[i]);
        }
        WeightMatrix = IGraph.BuildWeightMatrix(Vertices, adjMatrix);
    }

    public IVertice this[int ind]
    {
        get => Vertices[ind];
    }

    public float this[int vertice1, int vertice2]
    {
        get
        {
            if (!IndexIsInRange(vertice1) || !IndexIsInRange(vertice2))
                throw new IndexOutOfRangeException("Vertice index is out of
matrix");
            return WeightMatrix[vertice1][vertice2];
        }
```

```
    }

    public void SetStartEndPoint(int startPointIndex, int endPointIndex)
    {
        FinishVerticeIndex = endPointIndex;
        IVertice finish = Vertices[endPointIndex];
        IVertice start = Vertices[startPointIndex];
        Parallel.ForEach(Vertices, vertice => vertice.SetHeuristic(start,
finish));
    }

    public BilateralGraph(BilateralGraph original)
    {
        WeightMatrix = original.WeightMatrix;
        Vertices = new IVertice[original.Vertices.Length];
        for (int i = 0; i < Vertices.Length; i++)
        {
            Vertices[i] = original.Vertices[i].Copy();
        }
    }

    private bool IndexIsInRange(int index) => index >= 0 && index <
WeightMatrix.Length;
}

public abstract class ITwoSidePathSearchingAlgo: IPathSearchingAlgo
{
    protected ITwoSidePathSearchingAlgo(IGraph graph, int startpoinIndex, int
finishIndex) : base(graph,
        startpoinIndex, finishIndex)
    {
        ((BilateralVertice)graph[startpoinIndex]).DistanceFromStart[0] = 0;
        ((BilateralVertice)graph[finishIndex]).DistanceFromStart[1] = 0;
    }
    public override Stack<int> TraceRoute(IVertice lastVertice)
    {
        Stack<int> routeToEnd = new Stack<int>(100);
        Stack<int> route = new Stack<int>(100);

        BilateralVertice current = (BilateralVertice)lastVertice;
        while (current.PreviousVerticeInRouteIndex[1] != -1)
        {
            routeToEnd.Push(current.PreviousVerticeInRouteIndex[1]);
            current =
(BilateralVertice)_graph.Vertices[current.PreviousVerticeInRouteIndex[1]];
        }

        while (routeToEnd.Count > 0)
        {
            route.Push(routeToEnd.Pop());
        }

        route.Push(lastVertice.OwnIndex);
        current = (BilateralVertice)lastVertice;
        while (current.PreviousVerticeInRouteIndex[0] != -1)
        {
            route.Push(current.PreviousVerticeInRouteIndex[0]);
            current =
(BilateralVertice)_graph.Vertices[current.PreviousVerticeInRouteIndex[0]];
        }

        return route;
    }
```

```
    public override float GetDistance(IVertice lastVertice)
    {
        return ((BilateralVertice)lastVertice).DistanceFromStart.Sum();
    }
}

public class ConcurrentAStarWithParallelFor : ISingleSidePathSearchingAlgo
{
    public ConcurrentAStarWithParallelFor(Graph graph, int startpoinIndex, int
finishIndex) : base(graph,
        startpoinIndex, finishIndex) { }

    public async override Task<IVertice?> SearchPath()
    {
        BlockingPriorityQueue<int> verticeQueue = new
BlockingPriorityQueue<int>();
        verticeQueue.Enqueue(StartPoint, 0);
        Vertice currentVertice;
        while (verticeQueue.Count > 0)
        {
            currentVertice = (Vertice)_graph[verticeQueue.Dequeue()];
            if (currentVertice.IsPassed)
                continue;

            currentVertice.IsPassed = true;
            if (currentVertice.OwnIndex == EndPoint)
                return currentVertice;

            await Task.Run(() =>
            {

Parallel.ForEach( graph.GetAdjacentVertices(currentVertice.OwnIndex), adjIndex
=>
                {
                    Vertice child = (Vertice)_graph[adjIndex];
                    Interlocked.Increment(ref ChildrenCalculatedCounter);
                    if (child.TryUpdateMinRoute(currentVertice.OwnIndex))
                    {
                        verticeQueue.Enqueue(adjIndex, child.DistanceFromStart +
child.Heuristic!.Value);
                    }
                });
            });
        }

        return null;
    }
}

public class HDAStar : ISingleSidePathSearchingAlgo
{
    private readonly int _numProcessors; // Number of processors to distribute
the workload
    private readonly ConcurrentDictionary<int, BlockingPriorityQueue<int>>
_processorQueues;
    private readonly object _lock = new object();

    public HDAStar(Graph graph, int startpoinIndex, int finishIndex, int
numProcessors) : base(graph, startpoinIndex,
        finishIndex)
    {
        _numProcessors = numProcessors;
```

```
        _processorQueues = new ConcurrentDictionary<int,
BlockingPriorityQueue<int>>();
        for (int i = 0; i < _numProcessors; i++)
        {
            _processorQueues[i] = new BlockingPriorityQueue<int>();
        }
    }

    public async override Task<IVertice> SearchPath()
    {
        _processorQueues[0].Enqueue(StartPoint, 0);
        bool foundPath = false;

        while (_processorQueues.Values.Any(q => q.Count != 0))
        {
            await Task.Run(() =>
            {
                Parallel.ForEach(_processorQueues.Keys, processorId =>
                {
                    if (_processorQueues[processorId].Count != 0 && !foundPath)
                    {
                        Vertice currentVertice =
(Vertice)_graph[_processorQueues[processorId].Dequeue()];
                        if (!currentVertice.IsPassed)
                        {
                            lock (_lock)
                            {
                                currentVertice.IsPassed = true;
                            }

                            if (currentVertice.OwnIndex == EndPoint)
                            {
                                foundPath = true;
                            }

                            foreach (int adjIndex in
_graph.GetAdjacentVertices(currentVertice.OwnIndex))
                            {
                                Vertice child = (Vertice)_graph[adjIndex];
                                bool shouldEnqueue = false;
                                lock (child)
                                {
                                    if (!child.IsPassed)
                                    {
                                        shouldEnqueue =
child.TryUpdateMinRoute(currentVertice.OwnIndex);
                                    }
                                }

                                if (shouldEnqueue)
                                {
                                    int hash = ComputeHash(adjIndex);
                                    _processorQueues[hash].Enqueue(adjIndex,
                                        child.DistanceFromStart +
child.Heuristic!.Value);
                                }
                            }
                        }
                    }
                });
            });

            if (foundPath)
```

```
                break;
        }

        return foundPath ? (Vertice)_graph[EndPoint] : null;
    }

    private int ComputeHash(int vertexIndex)
    {
        return vertexIndex % _numProcessors;
    }
}

public class ParallelAStarOnTaskQueue : ISingleSidePathSearchingAlgo
{
    public ParallelAStarOnTaskQueue(Graph graph, int startpoinIndex, int
finishIndex) : base(graph, startpoinIndex, finishIndex) { }

    public override async Task<IVertice?> SearchPath()
    {
        PriorityQueue<(Vertice, Task<List<(Vertice, float)>>?)> verticeQueue =
new PriorityQueue<(Vertice, Task<List<(Vertice, float)>>?)>();
        Vertice currentVertice = (Vertice)_graph[StartPoint];
        Task<List<(Vertice, float)>>? calculateChildrenTask = Task.Run( () =>
CalculateChildren(currentVertice));
        verticeQueue.Enqueue((currentVertice, calculateChildrenTask), 0);
        while (verticeQueue.Count > 0)
        {
            (currentVertice, calculateChildrenTask) = verticeQueue.Dequeue();
            if (currentVertice.IsPassed)
                continue;

            currentVertice.IsPassed = true;
            if (currentVertice.OwnIndex == EndPoint)
                return currentVertice;

            var children = (calculateChildrenTask is not null? await
calculateChildrenTask : CalculateChildren(currentVertice));
            if (children.Count == 0)
                continue;
            var minPriorPair = children.MinBy(p => p.Item1.Heuristic!.Value +
p.Item2);
            float minPrior = minPriorPair.Item1.Heuristic!.Value +
minPriorPair.Item2;
            foreach (var pair in children)
            {
                Vertice child = pair.Item1;
                if (!child.IsPassed && child.DistanceFromStart > pair.Item2)
                {
                    child.PreviousVerticeInRouteIndex = currentVertice.OwnIndex;
                    child.DistanceFromStart = pair.Item2;
                    float prior = child.Heuristic!.Value + pair.Item2;
                    Task<List<(Vertice, float)>>? calculateNextChildrenTask =
                        prior <= minPrior+1 ? Task.Run(() =>
CalculateChildren(child)) : null;
                    verticeQueue.Enqueue((child, calculateNextChildrenTask),
prior);
                }
            }
        }

        return null;
    }
}
```

```
public class ParallelAStarOnWaitingTasks : ISingleSidePathSearchingAlgo
{
    public bool PathFound { get; set; }
    public byte NumberOfThreads { get; }
    private int _workingTasks;
    private readonly object _queueLocker;

    public ParallelAStarOnWaitingTasks(Graph graph, int startpoinIndex, int
finishIndex, byte threads=12) : base(graph, startpoinIndex, finishIndex)
    {
        PathFound = false;
        NumberOfThreads = threads;
        _workingTasks = 0;
        _queueLocker = new();
    }

    public override async Task<IVertice> SearchPath()
    {
        int childrenCalculated = 0;
        BlockingPriorityQueue<int> verticeQueue = new
BlockingPriorityQueue<int>();
        verticeQueue.Enqueue(StartPoint, 0);
        Task[] listeners = new Task[NumberOfThreads];
        for (int i = 0; i < NumberOfThreads; i++)
        {
            listeners[i] = Task.Run(() => ListenQueue(verticeQueue));
        }

        await Task.WhenAll(listeners);

        return PathFound? (Vertice)_graph[EndPoint] : null;
    }

    private async Task ListenQueue(BlockingPriorityQueue<int> queue)
    {
        Vertice current;
        while ((queue.Count > 0 || _workingTasks > 0) && !PathFound)
        {
            lock (_queueLocker)
            {
                if (queue.Count == 0)
                {
                    Monitor.Wait(_queueLocker);
                    continue;
                }
                current = (Vertice)_graph[queue.Dequeue()];
                Interlocked.Increment(ref _workingTasks);
            }

            if (current.IsPassed)
            {
                Interlocked.Decrement(ref _workingTasks);
                continue;
            }

            current.IsPassed = true;
            if (current.OwnIndex == EndPoint)
            {
                PathFound = true;
                Interlocked.Decrement(ref _workingTasks);
                return;
            }
```

```
            foreach (int adjIndex in
_graph.GetAdjacentVertices(current.OwnIndex))
            {
                Vertice child = (Vertice)_graph[adjIndex];
                Interlocked.Increment(ref ChildrenCalculatedCounter);
                if (child.TryUpdateMinRoute(current.OwnIndex))
                {
                    lock (_queueLocker)
                    {
                        queue.Enqueue(adjIndex,
child.DistanceFromStart+child.Heuristic!.Value);
                        Monitor.PulseAll(_queueLocker);
                    }
                }
            }

            Interlocked.Decrement(ref _workingTasks);
        }
        lock (_queueLocker)
        {
            Monitor.PulseAll(_queueLocker);
        }
    }
}

public class BilateralAStar : ITwoSidePathSearchingAlgo
{
    private BilateralVertice? meet;
    public BilateralAStar(BilateralGraph graph, int startpoinIndex, int
finishIndex) : base(graph, startpoinIndex, finishIndex) { }

    public override async Task<IVertice?> SearchPath()
    {
        List<Task<bool>> searchTasks = new List<Task<bool>>();
        searchTasks.Add(Task.Run(() => SingleThreadPathSearching(0)));
        searchTasks.Add(Task.Run(() => SingleThreadPathSearching(1)));

        await Task.WhenAny(searchTasks);
        return meet;
    }

    public async Task<bool> SingleThreadPathSearching(int procInd)
    {
        PriorityQueue<int> verticeQueue = new PriorityQueue<int>();
        BilateralVertice currentVertice;
        verticeQueue.Enqueue(procInd == 0? StartPoint : EndPoint, 0);
        while (verticeQueue.Count > 0 && meet is null)
        {
            currentVertice = (BilateralVertice)_graph[verticeQueue.Dequeue()];
            if (currentVertice.IsPassed[procInd])
                continue;

            currentVertice.IsPassed[procInd] = true;
            if (currentVertice.IsPassed[(procInd+1)%2])
            {
                if (meet is null || meet.DistanceFromStart.Sum() >
currentVertice.DistanceFromStart.Sum())
                    meet = currentVertice;
                return true;
            }

            foreach (var adjIndex in
```

```
_graph.GetAdjacentVertices(currentVertice.OwnIndex))
            {
                BilateralVertice child = (BilateralVertice)_graph[adjIndex];
                Interlocked.Increment(ref ChildrenCalculatedCounter);
                if (child.TryUpdateMinRoute(currentVertice.OwnIndex, procInd))
                    verticeQueue.Enqueue(child.OwnIndex,
                        child.DistanceFromStart[procInd] +
child.Heuristic[procInd].Value);
            }
        }

        return false;
    }
}
```

Додаток В. Код для порівняння роботи різних алгоритмів

```
public static class Program
{
    public static async Task Main()
    {
        Graph g = new Graph("saved60.grph", IGraph.FileType.Binary);//*/new
Graph(60000);
        //IGraph.SaveToBinFile("saved60.grph", g);
        BilateralGraph bg = new BilateralGraph("saved60.grph",
IGraph.FileType.Binary);

        Console.WriteLine("Building graphs done\n\n----------------------------
Testing:-------------------------");

        int s = 12345, f = 56789;
        IPathSearchingAlgo algo = new ConcurrentAStar(g, s, f);
        await TestAlgo(algo, "Concurrent A*", true);
        g.Reset();
        algo = new ConcurrentAStarWithParallelFor(g, s, f);
        await TestAlgo(algo, "A* with ParallelFor");
        g.Reset();
        algo = new ParallelAStarOnWaitingTasks(g, s, f);
        await TestAlgo(algo, "Parallel A* on waiting tasks");
        g.Reset();
        algo = new ParallelAStarOnTaskQueue(g, s, f);
        await TestAlgo(algo, "Parallel A* with task queueing");
        g.Reset();
        algo = new BilateralAStar(bg, s, f);
        await TestAlgo(algo, "Bilateral A*");

        //await TestResults();
    }

    public static async Task TestAlgo(IPathSearchingAlgo algo, string algoName,
bool printAllPath = false)
    {
        Stopwatch sw = Stopwatch.StartNew();
        IVertice? found = await algo.SearchPath();
        sw.Stop();

        if (found is not null)
        {
            var route = algo.TraceRoute(found);
            Console.WriteLine($"Path found by
{algoName}:\n\t{algo.GetDistance(found)} (through {route.Count} vertices) in
{sw.ElapsedMilliseconds}ms with {algo.ChildrenCalculatedCounter} vertices
touched for graph {algo.GetGraphSize()}x{algo.GetGraphSize()}");
            if (printAllPath)
            {
                PrintPath(route);
            }
        }
        else
            Console.WriteLine($"Path not found by {algoName} in
{sw.ElapsedMilliseconds}ms with {algo.ChildrenCalculatedCounter} vertices
touched for graph {algo.GetGraphSize()}x{algo.GetGraphSize()}");
    }

    private static void PrintPath(Stack<int> route)
    {
        Console.Write('\t');
        Console.Write(route.Pop());
```

```
        while (route.Count > 0)
        {
            Console.Write(" --> "+route.Pop());
        }

        Console.WriteLine('\n');
    }

    public static async Task TestResults()
    {
        Graph g = new Graph(10);
        PrintGraph(g);

        IPathSearchingAlgo algo = new ConcurrentAStar(g, 1, 4);
        var last = await algo.SearchPath();
        if (last is  null)
        {
            Console.WriteLine("Path is not found!");
            return;
        }

        var route = algo.TraceRoute(last);
        Console.WriteLine("Path found:");
        PrintPath(route);
    }

    private static void PrintGraph(IGraph graph)
    {
        Console.WriteLine("\tМатриця ваґів:");
        foreach (var row in graph.WeightMatrix)
        {
            foreach (var element in row)
            {
                Console.Write((element > -1 ? Math.Round(element, 2).ToString()
: "-") + "\t");
            }

            Console.WriteLine();
        }

        Console.WriteLine("\n");
    }
}
```