Міністерство освіти і науки України

Львівський національний університет Ім. І. Франка

Факультет прикладної математики та інформатики

Кафедра дискретного аналізу та інтелектуальних систем

Паралельні та розподілені обчислення

Лабораторна робота №7

Алгоритм Прима

Роботу виконала:

Студентка ПМІ-33

Багінська Маргарита

Прийняв:

доц. Пасічник Т.В.

Львів 2023

Тема: розв'язання задачі про побудову мінімального кісткового дерево в зв’язному зваженому неорієнтованому графі за допомогою алгоритму Прима.

Мета: Реалізувати послідовний та паралельний алгоритм побудову мінімального кісткового дерево в зв’язному зваженому неорієнтованому графі за допомогою алгоритму Прима.

Хід роботи

Допоміжна функція:

```
private static Dictionary<int, Dictionary<int, int>> GenerateGraph(int size)
{
    Random random = new Random();
    var graph = new Dictionary<int, Dictionary<int, int>>();

    for (int i = 0; i < size; i++)
    {
        graph[i] = new Dictionary<int, int>();
        int numberOfEdges = random.Next(1, 11); // Each vertex has 1 to 10 edges

        for (int j = 0; j < numberOfEdges; j++)
        {
            int randomVertex = random.Next(size);
            if (randomVertex != i && !graph[i].ContainsKey(randomVertex))
            {
                int weight = random.Next(1, 101); // Edge weight between 1 and 100
                graph[i][randomVertex] = weight;
            }
        }
    }

    return graph;
}
```

Послідовний алгоритм

```
        private static TimeSpan RunPrimsAlgorithm(Dictionary<int, Dictionary<int, int>> graph, int a)
{
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();

    int n = graph.Count;

    Dictionary<int, int> parent = new Dictionary<int, int>();
    Dictionary<int, int> key = new Dictionary<int, int>();
    Dictionary<int, bool> mstSet = new Dictionary<int, bool>();

    foreach (var vertex in graph.Keys)
    {
        key[vertex] = INF;
        mstSet[vertex] = false;
    }

    key[a] = 0;
    parent[a] = -1;

    for (int count = 0; count < n - 1; count++)
    {
        int u = -1;
        int minKey = INF;

        foreach (var vertex in graph.Keys)
        {
            if (!mstSet[vertex] && key[vertex] < minKey)
            {
                minKey = key[vertex];
                u = vertex;
            }
        }

        if (u != -1)
        {
            mstSet[u] = true;

            if (graph.ContainsKey(u))
            {
                foreach (var edge in graph[u])
                {
                    int v = edge.Key;
                    int weight = edge.Value;

                    if (!mstSet[v] && weight < key[v])
                    {
                        parent[v] = u;
                        key[v] = weight;
                    }
                }
            }
        }
    }

    //PrintMST(parent, graph);

    stopwatch.Stop();
    Console.WriteLine($"Sequential time ~ {stopwatch.Elapsed}");
    return stopwatch.Elapsed;
}
```

Опис:

Ініціалізується три словники: **parent**, **key** і **mstSet**.

Parent відстежує батьківську вершину кожної вершини в MST.

Key зберігає ключові значення (мінімальна вага з’єднання з вершинами) для кожної вершини в графі.

mstSet вказує, чи була вершина вже включена в MST. MstSet встановлюються ключі для всіх вершин на нескінченність (INF) також статус кожної вершини на false (не включено в MST).

Встановлюється ключ початкової вершини a на 0, а її батьківської вершини на -1.

Запускається основний цикл, який виконуватиметься, доки кожна вершина не буде включена в MST. Кількість додавання вершин становить n - 1.

У циклі ініціалізуються змінні **u** - поточний вузол і **minKey** - мінімальне значення ключа.

Перебираємо кожну вершину, щоб знайти вершину з найменшим значенням ключа. Оновлюється u і minKey з ідентифікатором вершини та відповідним значенням ключа.

Якщо знайдено вузол u із мінімальним значенням ключа, познається як включений у MST, оновивши його статус mstSety true.

Перебирається u сусідні вузли та оновлюються їхні ключі (key) та інформація про батьків, якщо вага ребра менша за поточне значення ключа для цього сусіда.

Ця реалізація відповідає класичному алгоритму Прима та створює MST за допомогою послідовного підходу. Він добре працює для відносно невеликих графів і демонструє часову складність O(n^2), де n — кількість вершин.

Паралельний алгоритм

```
        public static TimeSpan RunParallelPrimsAlgorithm(Dictionary<int, Dictionary<int, int>> graph, int
startNode, int maxParallelism)
        {
            Stopwatch stopWatch = new Stopwatch();
            stopWatch.Start();

            Dictionary<int, int> key = new Dictionary<int, int>();
            Dictionary<int, bool> nodeVisited = new Dictionary<int, bool>();
            Dictionary<int, int> parent = new Dictionary<int, int>();

            for (int i = 0; i < graph.Count; i++)
            {
                key[i] = int.MaxValue;
                nodeVisited[i] = false;
            }
            key[startNode] = 0;
            parent[startNode] = -1;

            int completedNodesCount = 0;

            while (completedNodesCount < graph.Count - 1)
            {
                int minimalKey = int.MaxValue;
                int currentNode = -1;

                object syncRoot = new object();

                Parallel.ForEach(Partitioner.Create(0, graph.Count, graph.Count / maxParallelism), range =>
                {
                    int localMinimalKey = int.MaxValue;
                    int localCurrentNode = -1;

                    for (int i = range.Item1; i < range.Item2; i++)
                    {
                        if (!nodeVisited[i] && key[i] < localMinimalKey)
                        {
                            localMinimalKey = key[i];
                            localCurrentNode = i;
                        }
                    }

                    lock (syncRoot)
                    {
                        if (localMinimalKey < minimalKey)
                        {
                            minimalKey = localMinimalKey;
                            currentNode = localCurrentNode;
                        }
                    }
                });

                if (currentNode == -1) break;
                nodeVisited[currentNode] = true;
                completedNodesCount++;

                var currentEdges = graph[currentNode].ToList();

                Parallel.ForEach(Partitioner.Create(0, currentEdges.Count), range =>
                {
                    for (int i = range.Item1; i < range.Item2; i++)
                    {
                        KeyValuePair<int, int> edge = currentEdges[i];

                        int v = edge.Key;
                        int weight = edge.Value;

                        if (!nodeVisited[v] && weight < key[v])
                        {
                            lock (key)
                            {
                                if (!nodeVisited[v] && weight < key[v])
                                {
                                    key[v] = weight;
                                    parent[v] = currentNode;
                                }
                            }
                        }
                    }
                });
            }

            //PrintMST(parent, graph);

            stopWatch.Stop();
            Console.WriteLine($"Parallel time ~ {stopWatch.Elapsed} with {maxParallelism} threads");
            return stopWatch.Elapsed;
        }
```

Опис:

Також ініціалізується словники: **Key**, **nodeVisited** і **parent**. Початок схожий до послідовного агоритму.

Ініціалізується змінна **completedNodesCount**, яка відстежуватиме кількість завершених вузлів у MST.

Починається основний цикл, який виконуватиметься до тих пір, поки completedNodesCount загальна кількість вузлів у графі не дорівнюватиме 1 (кожен вузол відвідується один раз).

Використовується Parallel.ForEach для ефективного розподілу роботи між доступними потоками.

Усередині цього **Parallel**.**ForEach** циклу шукається невідвідані вершини з найнижчим значенням ключа в діапазоні. Записуються локальні змінні **localMinimalKey** та **localCurrentNode**, для відстеження мінімального значення ключа та пов'язаного вузла.

Синхронізується доступ до спільних змінних **minimalKey** і **currentNode**, використовуючи **lock** оператор. Оновлюються змінні, лише якщо локальне мінімальне значення ключа менше, ніж мінімальне значення ключа, знайдене на даний момент.

Після паралельного пошуку, якщо не знайдено жодної невідвіданої вершини (тобто поточний вузол дорівнює -1), цикл припиняється.

Поточний вузол позначається як відвіданий і збільшується completedNodesCount.

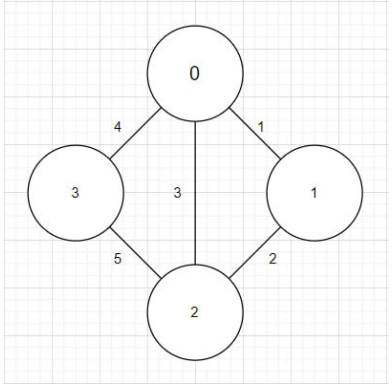
Подібним чином використовується Partitioner.Create метод with maxParallelism і Parallel.ForEach цикл для повторення країв сусідів поточного вузла. Оновлюєються ключі та інформація про батьків для цих сусідів за умови, що вершина не відвідана, а вага з'єднання менша за існуюче значення ключа.

Блокується доступ до спільних змінних key і parent, оновлюючи їх значення, щоб забезпечити безпеку потоку.

Варто зазначити, що ця паралельна реалізація алгоритму Прима гарантує правильність MST, але приріст швидкості може бути обмеженим через притаманну послідовність алгоритму.

Переконаємось у правильності обчислень наших алгоритмів:

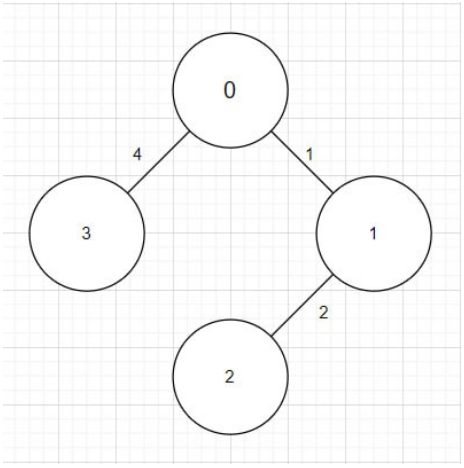
До прикладу візьмемо граф




```
var graph = new Dictionary<int, Dictionary<int, int>>()
{
{ 0, new Dictionary<int, int>() { { 1, 1 }, { 2, 3 }, { 3, 4 } } },
{ 1, new Dictionary<int, int>() { { 0, 1 }, { 2, 2 } } },
{ 2, new Dictionary<int, int>() { { 0, 3 }, { 1, 2 }, { 3, 5 } } },
{ 3, new Dictionary<int, int>() { { 0, 4 }, { 2, 5 } } },
};
```

```
Edge    Weight
0 - 1   1
1 - 2   2
0 - 3   4
Sequential time ~ 00:00:00.0012649
Edge    Weight
0 - 1   1
1 - 2   2
0 - 3   4
Parallel time ~ 00:00:00.0172898 with 2 threads
```

Можемо побачити, що обидва алгоритми працюють коректно і видали однаково правильний результат.



Отримане мінімальне кістякове дерево.

Прискорення Sp для паралельного алгоритму визначається відношенням часової складності послідовного T1 та паралельного алгоритмів для p процесорів Sp = T1 / Tp. (Sp > 1 Оптимально).

Ефективність Ep для паралельного алгоритму визначається прискоренням цього алгоритму відносно кількості процесорів: Ep= Sp/p Ідеал: Ep(n) = 1.

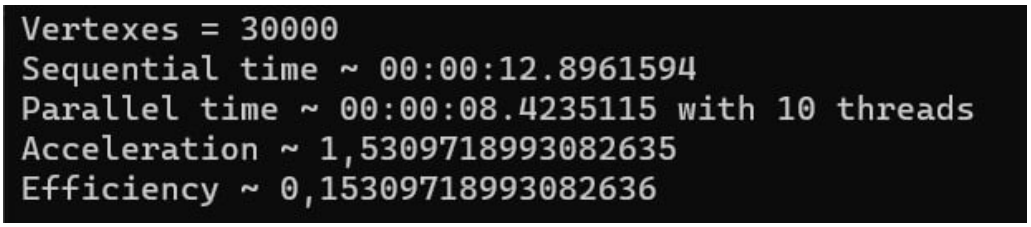
Результати

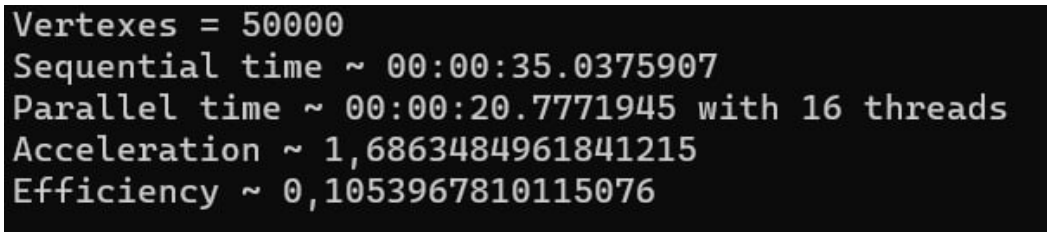
На малій розмірності графа розпаралелення не є оптимальним, як і у попередніх лабораторних роботах.

```
Vertexes = 2000
Sequential time ~ 00:00:00.0562446
Parallel time ~ 00:00:00.3011619 with 3 threads
Acceleration ~ 0,18675868361834613
Efficiency ~ 0,06225289453944871
```

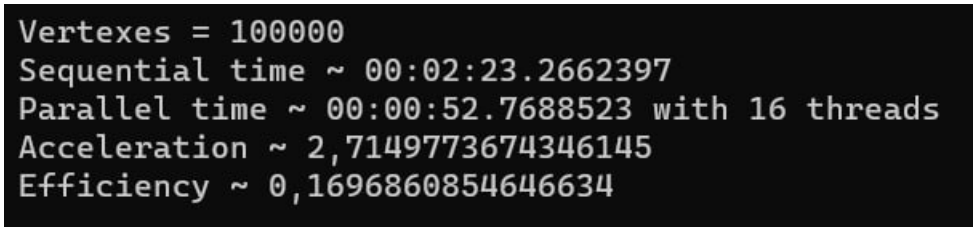
```
Vertexes = 10000
Sequential time ~ 00:00:01.3944027
Parallel time ~ 00:00:01.9882038 with 3 threads
Acceleration ~ 0,7013379111336574
Efficiency ~ 0,23377930371121913
```

Зі збільшенням розмірності паралельність ефективніша.





Можемо спостерігати ефективність паралельного алгоритму при великих об’ємах та розумній кількості потоків.



Висновок: У результаті виконання лабораторної роботи було реалізовано послідовний та паралельний алгоритм побудови мінімального кісткового дерево в зв’язному зваженому неорієнтованому графі за допомогою алгоритму Прима мовою програмування C# та за допомогою методу Parallel.For із простору імен System.Threading.Tasks.