5МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА
ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ ТА ІНФОРМАТИКИ

Кафедра дискретного аналізу та інтелектуальних систем

ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ
ЗВІТ З ЛАБОРАТОРНОЇ РОБОТИ №7
“АЛГОРИТМ ПРИМА”

Роботу виконав:
Сенюк Віталій Васильович
Студент групи ПМІ-33с

Перевірив:
Доц. Пасічник Т.В.
кафедри програмування
Львівського національного
Університету імені Івана Франка

Львів – 2023

Тема: алгоритм Прима.

Мета роботи: Розпаралелити виконання алгоритму Прима.

Завдання

Для зваженого зв'язного неорієнтованого графа G, використовуючи алгоритм Прима, з довільно заданої вершини a побудувати мінімальне кісткове дерево.

Для різної розмірності графів та довільного вузла a порахувати час виконання програми без потоків та при заданих k потоках розпаралелення.

Виконання роботи

1. Створюю клас для роботи з графом у виді матриці.

У даному випадку прийшлось створити вдосконалений клас графу. Тому, щоб показати роботу алогоритму потрібно було створювати великі графи у памяті, що призводило до помилки OUT OF MEMORY.

```
public class Graph
{
    private Dictionary<int, Dictionary<int, int>> adjacencyList;

    1 reference
    public Graph(int numberOfVertices)
    {
        Random random = new Random();
        adjacencyList = new Dictionary<int, Dictionary<int, int>>();

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

    1 reference
    public Dictionary<int, Dictionary<int, int>> GetAdjacencyList()
    {
        return adjacencyList;
    }
```

2. Створюю алгоритм для лінійного обчислення алгоритму Прима.

```
private static Dictionary<int, int> Prim(
    Dictionary<int, Dictionary<int, int>> adjacencyMatrix,
    int source)
{
    Dictionary<int, int> parents = new Dictionary<int, int>();
    Dictionary<int, int> keys = new Dictionary<int, int>();
    HashSet<int> unvisitedNodes = new HashSet<int>();
    foreach (int vertex in adjacencyMatrix.Keys)
    {
        keys[vertex] = int.MaxValue;
        unvisitedNodes.Add(vertex);
    }
    keys[source] = 0;
    while (unvisitedNodes.Count > 0)
    {
        int u = -1;
        foreach (int current in unvisitedNodes)
        {
            if (u == -1 || keys[current] < keys[u])
            {
                u = current;
            }
        }
        unvisitedNodes.Remove(u);
        foreach (var neighbor in adjacencyMatrix[u])
        {
            int v = neighbor.Key;
            int weight = neighbor.Value;

            if (unvisitedNodes.Contains(v) && weight < keys[v])
            {
                parents[v] = u;
                keys[v] = weight;
            }
        }
    }
    return parents;
}
```

Ключові моменти:

1. Ініціалізація: починається з вихідної вершини, позначте всі інші вершини як невідвідані та встановіть для них ваги ребер на нескінченність (int.MaxValue).
2. Вибирається мінімальний ключ: вибирається невідвідана вершина з найменшою вагою.
3. Оновлюються суміжні вершини: після вибору вершини, досліджується її сусіди і оновлюються їхні ваги.
4. Завершення: алгоритм завершуєтсья, коли відвудіє всі вершини, будуючи мінімальне кісткове дерево, яке охоплює всі вершини графа з найменшою вагою.

3. Створюю паралельний алгоритм.

```
private static Dictionary<int, int> ParallelPrim(
    Dictionary<int, Dictionary<int, int>> adjacencyMatrix,
    int source, int threads)
{
    Dictionary<int, int> parents = new Dictionary<int, int>();
    ConcurrentDictionary<int, int> keys = new ConcurrentDictionary<int, int>();
    ConcurrentDictionary<int, byte> unvisitedNodes = new ConcurrentDictionary<int, byte>();
    foreach (int vertex in adjacencyMatrix.Keys)
    {
        keys[vertex] = int.MaxValue;
        unvisitedNodes[vertex] = 0;
    }
    keys[source] = 0;
    while (unvisitedNodes.Count > 0)
    {
        int u = -1;
        int minKey = int.MaxValue;
        object lockObj = new object();

        // Parallelizing the loop to find the next vertex
        Parallel.ForEach(
            unvisitedNodes.Keys,
            new ParallelOptions { MaxDegreeOfParallelism = threads },
            () => (vertex: -1, key: int.MaxValue),
            (vertex, loopState, localState) =>
            {
                int vertexKey = keys[vertex];
                if (vertexKey < localState.key)
                {
                    localState.vertex = vertex;
                    localState.key = vertexKey;
                }
                return localState;
            },
            localState =>
            {
                lock (lockObj)
                {
                    if (localState.key < minKey)
                    {
                        u = localState.vertex;
                        minKey = localState.key;
                    }
                }
            });

        if (u == -1)
        {
            break;
        }
        unvisitedNodes.TryRemove(u, out _);
        // Iterate over neighbors and update the keys
        Parallel.ForEach(
            adjacencyMatrix[u],
            new ParallelOptions { MaxDegreeOfParallelism = threads },
            neighborPair =>
            {
                int v = neighborPair.Key;
                int weight = neighborPair.Value;

                if (unvisitedNodes.ContainsKey(v) && weight < keys[v])
                {
                    lock (lockObj)
                    {
                        if (unvisitedNodes.ContainsKey(v) && weight < keys[v])
                        {
                            parents[v] = u;
                            keys[v] = weight;
                        }
                    }
                }
            });
    }
    return parents;
}
```

Даний паралельний алгоритм неможливо ефективно реалізувати паралельно. У даному випадку я добавив два Parallel.ForEach, але вони двоє маю спільний ресурс, для якого потрібен синхронізатор, що унеможливлює якісне розпаралелення.

4. Правильність роботи алгоритму.

```
GraphSize 4
Prim's algorithm 'a' (0),
From 1 to 0
From 3 to 0
From 2 to 1
Parallel t=2 'a' (0) time:
From 3 to 0
From 1 to 0
From 2 to 1
Parallel t=4 'a' (0) time:
From 1 to 0
From 3 to 0
From 2 to 1
Parallel t=8 'a' (0) time:
From 1 to 0
From 3 to 0
From 2 to 1
```

Як можемо бачити, два алгоритми демонструють однаковий результат.

5. Час та параметри виконання алгоритмів.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
GraphSize 100
Prim's algorithm 'a' (0), time:00:00:00.0022037
Parallel t=2 'a' (0) time: 00:00:00.0267986 acceleration: 0,08223190763696611 efficiency: 0,041115953818483056
Parallel t=4 'a' (0) time: 00:00:00.0029370 acceleration: 0,7503234593122233 efficiency: 0,18758086482805583
Parallel t=8 'a' (0) time: 00:00:00.0043039 acceleration: 0,5120239782522829 efficiency: 0,06400299728153536
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
GraphSize 500
Prim's algorithm 'a' (0), time:00:00:00.0033267
Parallel t=2 'a' (0) time: 00:00:00.0182663 acceleration: 0,18212226887765995 efficiency: 0,09106113443882997
Parallel t=4 'a' (0) time: 00:00:00.0177256 acceleration: 0,18767770907613845 efficiency: 0,0469194272690346l4
Parallel t=8 'a' (0) time: 00:00:00.0201347 acceleration: 0,16522222829245034 efficiency: 0,020652778536556293
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
GraphSize 2500
Prim's algorithm 'a' (0), time:00:00:00.0861888
Parallel t=2 'a' (0) time: 00:00:00.2289218 acceleration: 0,376498874288075666 efficiency: 0,1882494371440378 3
Parallel t=4 'a' (0) time: 00:00:00.2000312 acceleration: 0,4308767832218174 efficiency: 0,10771919580545435
Parallel t=8 'a' (0) time: 00:00:00.1764488 acceleration: 0,48846350896123975 efficiency: 0,06105793862015497
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
GraphSize 12500
Prim's algorithm 'a' (0), time:00:00:01.8242364
Parallel t=2 'a' (0) time: 00:00:01.8233285 acceleration: 1,0004979355064103 efficiency: 0,5002489677532052
Parallel t=4 'a' (0) time: 00:00:01.7009763 acceleration: 1,0724643253406883 efficiency: 0,26811608133517206
Parallel t=8 'a' (0) time: 00:00:01.9164267 acceleration: 0,9518946902586987 efficiency: 0,11898683628233733
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
GraphSize 62500
Prim's algorithm 'a' (0), time:00:00:45.4794000
Parallel t=2 'a' (0) time: 00:00:41.5378069 acceleration: 1,0948916997348745 efficiency: 0,54744584986743 73
Parallel t=4 'a' (0) time: 00:00:33.6247126 acceleration: 1,3525587725023411 efficiency: 0,3381396931255853
Parallel t=8 'a' (0) time: 00:00:32.3314641 acceleration: 1,4066607023837192 efficiency: 0,1758325877979649
```

Як бачимо, алгоритм починає себе проявляти на великих даних, і вже на графі розміром у 62500, можемо бачити що прискорення досягає 1.4, ефективність зрозуміло, що низька.

Висновки

У результаті виконання роботи:

1. Спромігся розпаралелити алгоритм Прима, хоча даний алгоритм не підходить для цього, так як має спільний ресурс, який потрібно синхронізувати між потокам, тому частка розпаралелення низька.

Список використаних джерел

1. Вільний простір інтернету.