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 №6

Алгоритм Дейкстри

Роботу виконала:

Студентка ПМІ-33

Багінська Маргарита

Прийняв:

доц. Пасічник Т.В.

Львів 2023

Тема: розв'язання задачі про найкоротший шлях в орієнтованому зваженому графі з додатними вагами ребер за допомогою алгоритму Дейкстри.

Мета: Реалізувати послідовний та паралельний алгоритм розв'язання задачі знаходження найкоротшого шляху в орієнтованому зваженому графі з додатними вагами ребер за допомогою алгоритму Дейкстри.

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;
  }
```

Цей GenerateGraph метод створює словник graph, який представляє список суміжності графа. Ключі представляють вихідні вершини, а значення є словниками, що представляють ребра та їх ваги для вершин.

Для кожної вершини створюється і додається новий словник. Генерується випадкове число numberOfEdges від 1 до 10 (включно), щоб визначити кількість вихідних ребер для вершини. Потім вкладений цикл генерує ребра.

Для кожного numberOfEdges: Вершина призначення randomVertex вибирається випадковим чином із доступних вершин. Якщо обрана вершина randomVertexне збігається з вихідною вершиною та ще не з’єднана з ребром, створюється ребро.

Послідовний алгоритм

```
private static TimeSpan RunSequentialDijkstra(Dictionary<int, Dictionary<int, int>> graph, int
sourceNode)
{
    Stopwatch stopWatch = new Stopwatch();
    stopWatch.Start();

    var nodeDistances = new Dictionary<int, int>();
    var nodeVisited = new Dictionary<int, bool>();
    int vertexCount = graph.Count;
    for (int i = 0; i < vertexCount; i++)
    {
        nodeDistances[i] = int.MaxValue;
        nodeVisited[i] = false;
    }
    nodeDistances[sourceNode] = 0;

    for (int i = 0; i < vertexCount - 1; i++)
    {
        int minimalDistance = int.MaxValue;
        int currentNode = -1;

        foreach (var node in nodeDistances)
        {
            if (!nodeVisited[node.Key] && node.Value <= minimalDistance)
            {
                minimalDistance = node.Value;
                currentNode = node.Key;
            }
        }
        if (currentNode != -1)
        {
            nodeVisited[currentNode] = true;
            foreach (var edge in graph[currentNode])
            {
                int updatedDistance = nodeDistances[currentNode] + edge.Value;
                if (updatedDistance < nodeDistances[edge.Key])
                {
                    nodeDistances[edge.Key] = updatedDistance;
                }
            }
        }
    }

    stopWatch.Stop();

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

Опис:

Два словники, nodeDistances і nodeVisited, ініціалізуються для зберігання найкоротших відстаней шляху від sourceNode до кожної вершини та стану відвідування вершин, відповідно. Відстань для кожної вершини спочатку встановлено int.MaxValue за винятком sourceNode, який встановлено на 0.

Алгоритм проходить через усі вершини (з vertexCount – 1 ітераціями). У кожній ітерації він виконує такі кроки:

a. Встановлює змінній minimalDistance int.MaxValue як початкове значення, а змінній поточної вершини (currentNode) встановлює значення -1.

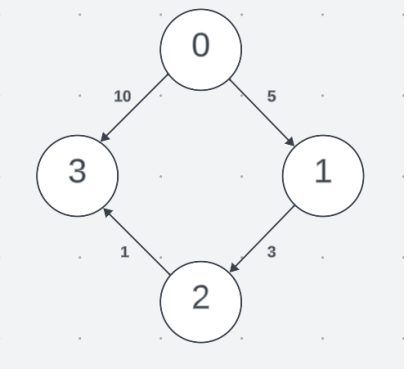
b. Для кожної невідвіданої вершини в nodeDistances словнику, якщо її відстань менша або дорівнює minimalDistance, оновлює minimalDistance і встановлює currentNode в цю вершину.

в. Якщо currentNode знайдено (тобто не дорівнює -1), позначає його як відвідану, встановивши відповідний запис у словнику nodeVisited на true.

d. Для кожного суміжного ребра в graph[currentNode] встановлює нову відстань для з’єднаних вершин, додавши відстань поточного вузла та вагу ребра. Якщо оновлена відстань менша за існуючу у nodeDistances, оновлює значення відстані.

Переконаємось у правильності обчислень.

Граф має вигляд:




```
Shortest path distances from source vertex (0):
Vertex 0: Distance 0
Vertex 1: Distance 5
Vertex 2: Distance 8
Vertex 3: Distance 9
Sequential time ~ 00:00:00.0005484
```

Алгоритм працює коректно.

Паралельний алгоритм

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

            var nodeDistances = new ConcurrentDictionary<int, int>();
            var nodeVisited = new ConcurrentDictionary<int, bool>();

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

            int completedNodesCount = 0;

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

                Parallel.ForEach(Partitioner.Create(0, graph.Count, graph.Count / maxParallelism),
range =>
                {
                    for (int i = range.Item1; i < range.Item2; i++)
                    {
                        if (!nodeVisited[i] && nodeDistances[i] <= minimalDistance)
                        {
                            lock (nodeVisited)
                            {
                                if (!nodeVisited[i] && nodeDistances[i] <= minimalDistance)
                                {
                                    minimalDistance = nodeDistances[i];
                                    currentNode = i;
                                }
                            }
                        }
                    }
                });

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

                    var currentEdges = graph[currentNode];

                    Parallel.ForEach(Partitioner.Create(0, currentEdges.Count), range =>
                    {
                        int localIndex = 0;
                        foreach (var edge in currentEdges)
                        {
                            if (localIndex >= range.Item1 && localIndex < range.Item2)
                            {
                                int updatedDistance = nodeDistances[currentNode] + edge.Value;
                                if (updatedDistance < nodeDistances[edge.Key])
                                {
                                    lock (nodeDistances)
                                    {
                                        if (updatedDistance < nodeDistances[edge.Key])
                                        {
                                            nodeDistances[edge.Key] = updatedDistance;
                                        }
                                    }
                                }
                            }
                            localIndex++;
                        }
                    });
                }
            }

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

Опис:

Два словники, nodeDistances (ConcurrentDictionary) і nodeVisited (ConcurrentDictionary), ініціалізуються для зберігання найкоротших відстаней шляху від вихідного вузла до кожної вершини та стану відвідування вершин відповідно. Відстань для кожної вершини спочатку встановлено на int.MaxValue, а для всіх вузлів nodeVisited – на false. Відстань вихідного вузла встановлено на 0.

Змінна completedNodesCount визначена для відстеження кількості оброблених вузлів.

Поки completedNodesCount менше загальної кількості вузлів у графі - 1:

a. Визначається minimalDistance і currentNode змінні з початковими значеннями int.MaxValuei -1 відповідно.

b. Використовується Parallel.ForEach метод із Partitioner.Create діапазоном для проходження всіх вузлів паралельно. У кожному фрагменті, якщо вузол не відвідуваний і має меншу або рівну відстань порівняно з поточним minimalDistance, оновлює його значення та встановлює currentNode для цього вузла.

в. Якщо a currentNode знайдено, позначає його як відвідану, збільшує completedNodesCount та оброблює його суміжні краї.

d. Сусідні краї оброблює іншим циклом Parallel.ForEach. Розділює набір ребер на паралельні фрагменти та оновлює відстань для кожного вузла. Блокує nodeDistances словник під час оновлення, щоб забезпечити коректний одночасний доступ.

Переконаємось у правильності обчислень для графу згаданого вище:

```
Shortest path distances from source vertex (0):
Vertex 0: Distance 0
Vertex 1: Distance 5
Vertex 2: Distance 8
Vertex 3: Distance 9
Parallel time ~ 00:00:00.0159231 with 2 threads
```

Паралельний алгоритм також працює коректно.

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. Загалом алгоритм Дейкстри не найкращий вибір для розпаралелення, адже має спільні ресурси.

```
Vertexes = 1000
Sequential time ~ 00:00:00.0210501
Parallel time ~ 00:00:00.1552543 with 2 threads
Acceleration ~ 0,13558465047344903
Efficiency ~ 0,06779232523672452
```

Зі збільшенням розмірності паралельність не сильно, але ефективніша.

```
Vertexes = 25000
Sequential time ~ 00:00:09.8759057
Parallel time ~ 00:00:08.9941414 with 2 threads
Acceleration ~ 1,0980376292505252
Efficiency ~ 0,5490188146252626
```

```
Vertexes = 30000
Sequential time ~ 00:00:14.2231420
Parallel time ~ 00:00:10.1005648 with 5 threads
Acceleration ~ 1,4081531361493764
Efficiency ~ 0,2816306272298753
```

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

```
Vertexes = 35000
Sequential time ~ 00:00:21.2156860
Parallel time ~ 00:00:13.9146269 with 4 threads
Acceleration ~ 1,524703907080685
Efficiency ~ 0,38117597677017123
```

Висновок: У результаті виконання лабораторної роботи було реалізовано послідовний та паралельний алгоритм розв'язання задачі знаходження найкоротшого шляху в орієнтованому зваженому графі з додатними вагами ребер за допомогою алгоритму Дейкстри мовою програмування C# та за допомогою методу Parallel.For із простору імен System.Threading.Tasks. Переконались у ефективності розпаралелення процесу, але слід зазначити, що даний алгоритм для цього не дуже підходить з тієї причини, що використовує спільні ресурси.