МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені ІВАНА ФРАНКА

Паралельні та розподілені обчислення

ЛАБОРАТОРНА РОБОТА №7
«Алгоритм Прима»

Виконала:
студентка групи ПМі-31
Дудчак Валентина Юріївна

Львів 2024

Тема: Розпаралелення алгоритму Прима

Мета: Написати програми розв’язування алготму Прима (послідовний та паралельний алгоритми). Для зваженого зв'язного неорієнтованого графа G(V,F), використовуючи алгоритм Прима, з довільно заданої вершини **a** побудувати мінімальне кісткове дерево.

Хід роботи:

Для задачі я подаю граф у вигляді словника, де кожна вершина має словник зі значеннями кожної вершини, з якою вона з’днана, та вагою ребра між ними. Так як граф неорієнтований, вага ребер [i, j] = [j, i]. Вершин, між якими відсутні ребра, та петель немає в словнику.

Послідовний алгоритм

Послідовний алгоритм Дейкстри реалізовано у методі SequentialPrimaAlgorithm.

У ньому знаходиться кісткове дерево від заданої вершини **nodeA** за допомогою алгоритму Прима та зберігається у словник. Використано також об’єкт класу Stopwatch для засікання часу:

```
public static Stopwatch SequentialPrimaAlgorithm(Dictionary<int, Dictionary<int, int>> graph, int nodeA)
{
  Stopwatch stopWatch = new Stopwatch();
  stopWatch.Start();

  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 graph.Keys)
  {
    keys[vertex] = int.MaxValue;
    unvisitedNodes.Add(vertex);
  }
  keys[nodeA] = 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 graph[u])
    {
      int v = neighbor.Key;
      int weight = neighbor.Value;

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

  stopWatch.Stop();
  // return parents;
  return stopWatch;
}
```

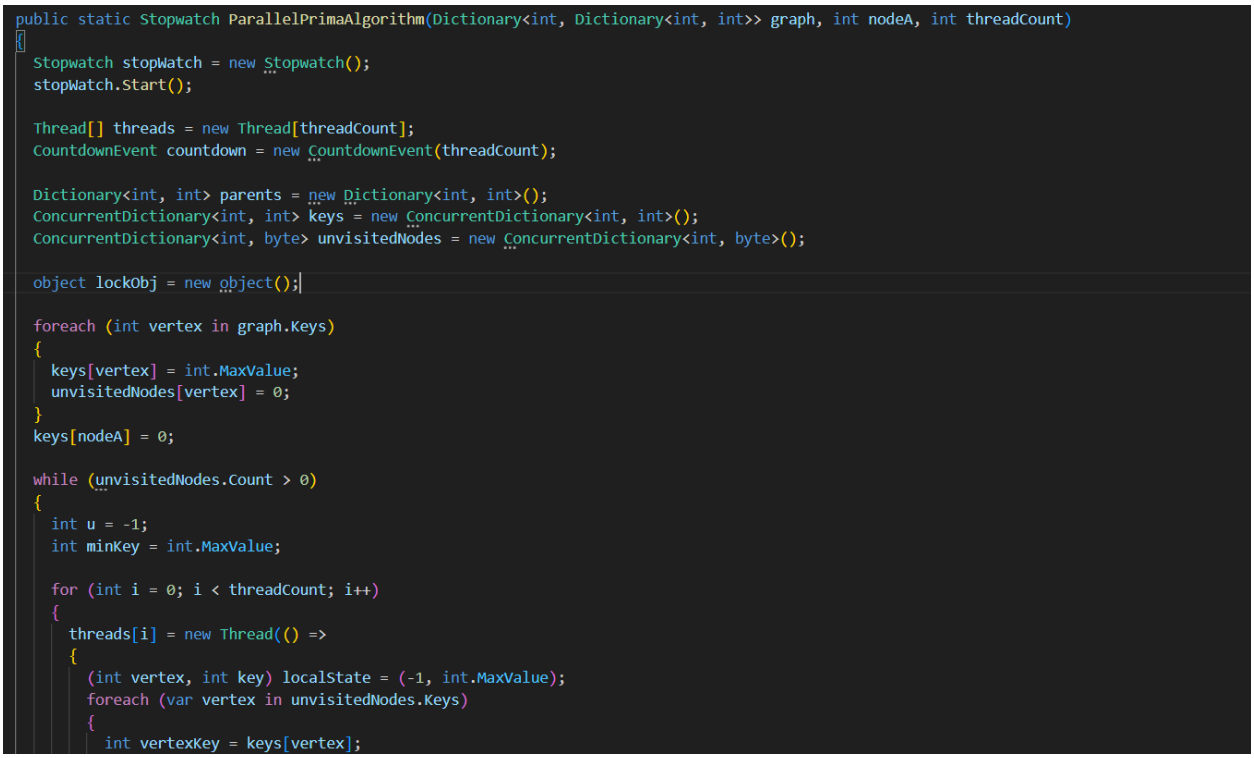
Крім цього, я перевірила, чи результати обчислюються правильно:

```
Vertex 0: (Vertex 3, weight 61) (Vertex 1, weight 34) (Vertex 2, weight 73) (Vertex 4, weight 81)
Vertex 1: (Vertex 4, weight 38) (Vertex 3, weight 22) (Vertex 0, weight 88) (Vertex 2, weight 82)
Vertex 2: (Vertex 3, weight 65) (Vertex 0, weight 23) (Vertex 1, weight 15)
Vertex 3: (Vertex 0, weight 69) (Vertex 4, weight 39) (Vertex 2, weight 19)
Vertex 4: (Vertex 3, weight 87) (Vertex 0, weight 29) (Vertex 2, weight 16) (Vertex 1, weight 88)
n = 5, a = 1
From 4 to 1
From 3 to 1
From 0 to 2
From 2 to 3
Parallel:
From 4 to 1
From 3 to 1
From 0 to 2
From 2 to 3
```

Паралельний алгоритм

Паралельний алгоритм Прима реалізовано у методі ParallelPrimAlgorithm.

Для розпаралелення я використовую об’єкти класу Thread та CountdownEvent для сигналу про завершення роботи потоків. Так як потоки залежать одне від одного, при оновленні відстані між вершинами використовується блокування (lock)



```
public static Stopwatch ParallelPrimaAlgorithm(Dictionary<int, Dictionary<int, int>> graph, int nodeA, int threadCount)
{
  Stopwatch stopWatch = new Stopwatch();
  stopWatch.Start();

  Thread[] threads = new Thread[threadCount];
  CountdownEvent countdown = new CountdownEvent(threadCount);

  Dictionary<int, int> parents = new Dictionary<int, int>();
  ConcurrentDictionary<int, int> keys = new ConcurrentDictionary<int, int>();
  ConcurrentDictionary<int, byte> unvisitedNodes = new ConcurrentDictionary<int, byte>();

  object lockObj = new object();

  foreach (int vertex in graph.Keys)
  {
    keys[vertex] = int.MaxValue;
    unvisitedNodes[vertex] = 0;
  }
  keys[nodeA] = 0;

  while (unvisitedNodes.Count > 0)
  {
    int u = -1;
    int minKey = int.MaxValue;

    for (int i = 0; i < threadCount; i++)
    {
      threads[i] = new Thread(() =>
      {
        (int vertex, int key) localState = (-1, int.MaxValue);
        foreach (var vertex in unvisitedNodes.Keys)
        {
          int vertexKey = keys[vertex];
```

```
            if (vertexKey < localState.key)
            {
              localState.vertex = vertex;
              localState.key = vertexKey;
            }
          }

          lock (lockObj)
          {
            if (localState.key < minKey)
            {
              u = localState.vertex;
              minKey = localState.key;
            }
          }
          countdown.Signal();
        });
        threads[i].Start();
      }

      countdown.Wait();
      countdown.Reset();

      if (u == -1)
      {
        break;
      }
      unvisitedNodes.TryRemove(u, out _);

      for (int i = 0; i < threadCount; i++)
      {
        threads[i] = new Thread(() =>
        {
          foreach (var neighborPair in graph[u])
          {
            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;
                }
              }
            }
          }

          countdown.Signal();
        });
        threads[i].Start();
      }

      countdown.Wait();
      countdown.Reset();
    }

    stopWatch.Stop();
    return stopWatch;
    // return parents;
  }
```

Метод також працює правильно (видно у результатах вище)

Аналіз результатів

На малих розмірах графа розпаралелення зовсім не ефективне:

```
n = 100, a = 1
Sequential time: 0 ms
Parallel time: 204 ms, threads: 10
Acceleration: 0.0016291821329934037
Efficiency: 0.00016291821329934036
```

```
n = 1000, a = 1
Sequential time: 15 ms
Parallel time: 1746 ms, threads: 10
Acceleration: 0.008670454201249473
Efficiency: 0.0008670454201249473
```

```
n = 10000, a = 1
Sequential time: 1075 ms
Parallel time: 9353 ms, threads: 5
Acceleration: 0.11500333379858084
Efficiency: 0.023000666759716168
```

На більших розмірностях та при меншій кількості потоків розпаралелення ефективніше, але все ще гірше за послідовний алгоритм:

```
n = 30000, a = 1
Sequential time: 8821 ms
Parallel time: 21116 ms, threads: 2
Acceleration: 0.41775409543773434
Efficiency: 0.20887704771886717
```

```
n = 60000, a = 1
Sequential time: 34308 ms
Parallel time: 48539 ms, threads: 1
Acceleration: 0.7068080186829054
Efficiency: 0.7068080186829054
```

Алгоритм Прима розпаралелювати дуже неефективно, тому бачимо, що навіть при великій розмірності даних та розумній кількості потоків розпаралелення дає поганий результат:

```
n = 80000, a = 1
Sequential time: 60985 ms
Parallel time: 200095 ms, threads: 5
Acceleration: 0.304780933489616
Efficiency: 0.0609561866979232
```

Висновок:

Виконуючи лабораторну роботу, я реалізувала послідовний та паралельний варіанти алгоритму Прима для побудови кістякового дерева у зваженому неорієнтованому графі. Для розпаралелення використала класи Thread та CountdownEvent. Порівняння часу виконання показало, що паралельний алгоритм не є ефективним для цієї задачі.