МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №4

по дисциплине «Построение и анализ алгоритмов»

Тема: Поток в сети

Студент гр. 8382 _____ Щемель Д.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Задание

Найти максимальный поток в сети, а также фактическую величину потока, протекающего через каждое ребро, используя алгоритм Форда-Фалкерсона.

Сеть (ориентированный взвешенный граф) представляется в виде триплета из имён вершин и целого неотрицательного числа - пропускной способности (веса).

Входные данные:
N - количество ориентированных рёбер графа
v_0 - исток
v_n - сток
$v_i\ v_j\ w_{ij}$ - ребро графа
$v_i\ v_j\ w_{ij}$ - ребро графа
...

Выходные данные:
P_{max} - величина максимального потока
$v_i\ v_j\ \omega_{ij}$ - ребро графа с фактической величиной протекающего потока
$v_i\ v_j\ \omega_{ij}$ - ребро графа с фактической величиной протекающего потока ...

В ответе выходные рёбра отсортируйте в лексикографическом порядке по первой вершине, потом по второй (в ответе должны присутствовать все указанные входные рёбра, даже если поток в них равен 0).

Sample Input:
7
a
f
a b 7
a c 6
b d 6
c f 9
d e 3
d f 4
e c 2

Sample Output:

12

a b 6

a c 6

b d 6

c f 8

d e 2

d f 4

e c 2

Номер варианта: 1. Поиск в ширину. Поочерёдная обработка вершин текущего фронта, перебор вершин в алфавитном порядке.

Описание алгоритма

Алгоритм работает следующим образом, чередуя 2 этапа:

Этап 1: Поиск пути.

Изначально имеется очередь из одного элемента - начальной вершины. На этом этапе поиском в ширину с выбором вершин в алфавитном порядке вершины добавляются в очередь на обработку. Вершины добавляются только при выполнение определённых условий:

- Если вершина дочерняя, не находится в очереди и поток не заполнен полностью
- Если вершина родительская, не находится в очереди и поток ненулевой

В добавленную вершину записывается информация, из какой вершины был прололожен путь до добавленной.

После этого как была достигнута конечная вершина(сток) - поиск прекращается.

Этап 2: Изменение потока

На этом этапе в найденном пути ищется максимально допустимый поток: разница между максимальным потоком и текущим для вершин, взятых в прямом направлении и текущий поток для вершин, взятых в обратном направлении.
Найденный максимально возможный поток добавляется к текущему потоку в вершинах, взятых в прямом направлении и вычитается для вершин, взятых в обратном направлении.

После завершения этапа 2 очередь вершин очищается, добавляется в неё начальная вершина (исток) и повторяется этап 1. Программа завершает свою работу тогда, когда невозможно

найти путь в конечную вершину, соблюдая условия из этапа 1. (Это определяется тем, что невозможно добавить вершину, очередь очищается и становится пустой).

Сложности алгоритма:

По памяти: O(V + E), где V - количество вершин, а E - количество рёбер. Для хранения всех вершин и всех рёбер.

По количеству операций: O(E^2 * M), где E - количество рёбер. M - максимальный из потоков на рёбрах. Потому что в худшем случае придётся обойти все дуги. После каждого нахождения пути поток может изменить минимум на единицу. Тогда необходимо будет обходит дуги до тех пор, пока не заполнится поток с максимальным значением.

Описание функций и структур данных

Структуры данных

В программе используются следующие классы:

- Graph - для хранения словаря вершин (соответствие имя -> *Node*) и списка ссылок на рёбра *Edge*

Dictionary<char, Node> _nodes // Вершины

List<Edge> _edges // Рёбра

Node Source { get; set; } // Начальная вершина

Node Sink { get; set; } // Конечная вершина

- *Edge* - для хранения потока, ссылки на вершину, из которой это ребро выходит, ссылки на вершину, куда это ребро входит

FlowEntity Flow { get; set; } // Поток

Node From { get; set; } // Начальная вершина ребра

Node To { get; set; } // Конечная вершина ребра

bool IsFullFlow // Полон ли поток bool IsEmptyFlow // Пустой ли поток

- *Node* - для хранения названия вершины, списка рёбер к дочерним вершинам и списка рёбер от родительских вершин. А так же ребро, которое ведёт к этой вершине во процессе поиска пути (в том числе в обратном направлении)

char Name { get; set; } // Имя вершины

List<Edge> Children { get; } // Дочерние вершины

List<Edge> Parents { get; } // Родительские вершины

Edge CameFrom { get; set; } // Ребро, по которому мы пришли к вершине

Используемые функции (методы)

В программе используются следующие методы классов:

- public void Graph.ReadGraph() читает граф с консоли, сохраняя все необходимые данные

- public int FindMaxFlow() ищет максимальный поток в графе
- private void PatchFlowInPath() - изменяет поток в графе, после нахождения пути из истока в стока.

Тестирование

Тест 1:

Исходные данные:

7

a

f

a b 7

a c 6

b d 6

c f 9

d e 3

d f 4

e c 2

Выходные данные:

12

a b 6

a c 6

b d 6

c f 8

d e 2

d f 4

e c 2

Тест 2:

Исходные данные:

7

a

f

a b 3

a c 2

b d 13

c f 4

d e 2

d f 7

e c 2

Выходные данные:

5

a b 3 a c 2 b d 3 c f 2 d e 0 d f 3 e c 0

Тест 3:

Исходные данные:

8 a f a b 3 a c 4 b d 5 c f 4 d e 6 d f 7 e c 2 d b 7

Выходные данные:

7 a b 3 a c 4 b d 3 c f 4 d b 0 d e 0 d f 3 e c 0

Тест 4:

Исходные данные:

5 a e a b 10 a c 10 b d 6 c f 9 d e 8

Выходные данные:

Current node => a Current node => b Current node => c Current node => d Current node => f Current node => e Start patch Min flow = 6 Flow from d to e = 6 (max = 8) Flow from b to d = 6 (max = 6) Edge from b to d is full now Flow from a to b = 6 (max = 10) Current node => a Current node => b Current node => c Current node => f 6 a b 6 a c 0 b d 6 c f 0 d e 6

Вывод

В результате выполнения лабораторной работы была написана программа для нахождения максимального потока в сети. А так же были получены практические навыки реализации алгоритма Форда-Фалкерсона.

Приложение А. Исходный код

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace lab3
{
    /// <summary>
    /// Class wrapper for printing
    /// </summary>
    public static class Logger
    {
        public enum LogLevel
        {
            Debug,
            Info
        }

        /// <summary>
        /// Log message with specify log level
        /// </summary>
        /// <param name="text"></param>
        /// <param name="level"></param>
        public static void Log(object text, LogLevel level = LogLevel.Info)
        {
            int logLevel;
            var hasVariable = int.TryParse(Environment.GetEnvironmentVariable("LAB3
            logLevel = hasVariable ? logLevel : (int)LogLevel.Info;

            if ((int)level >= logLevel)
            {
                Console.WriteLine(text.ToString());
            }
        }
```

```
}

/// <summary>
/// Extensions for <see cref="Queue{T}"/>
/// </summary>
public static class QueueExtension
{
    /// <summary>
    /// Add element to Queue and return Queue
    /// </summary>
    /// <param name="queue"><see cref="Queue{T}"/></param>
    /// <param name="elem">First element to add</param>
    /// <typeparam name="T">Type in Queue</typeparam>
    /// <returns>New queue</returns>
    public static Queue<T> Add<T>(this Queue<T> queue, T elem)
    {
        queue.Enqueue(elem);
        return queue;
    }
}

/// <summary>
/// Class for storing start and end points of graph
/// </summary>
public class Graph
{
    /// <summary>
    /// Map of nodes in graph by their names
    /// </summary>
    private readonly Dictionary<char, Node> _nodes = new Dictionary<char, Node>

    private readonly List<Edge> _edges = new List<Edge>();

    /// <summary>
    /// Print all edges in graph as {From} {To} {Flow}
```

```
/// </summary>
public void PrintEdges()
{
    _edges.OrderBy(x => x.From.Name)
        .ThenBy(x => x.To.Name)
        .ToList()
        .ForEach(x => Logger.Log($"{x.From.Name} {x.To.Name} {x.Flow.Curren
}

/// <summary>
/// Read graph from stdin
/// </summary>
/// <returns>Graph</returns>
public void ReadGraph()
{
    var countEdges = int.Parse(Console.ReadLine()?.Split(' ').First());
    var start = Console.ReadLine().Split(' ').First().First();
    var end = Console.ReadLine().Split(' ').First().First();

    var sourceName = start;
    var sinkName = end;
    Source = new Node { Name = sourceName };
    Sink = new Node { Name = sinkName };
    _nodes.Add(sourceName, Source);
    _nodes.Add(sinkName, Sink);

    for (var i = 0; i < countEdges; i++)
    {
        var input = Console.ReadLine()?.Split(' ');

        var newNodeStart = new Node() { Name = (input[0]).First() };
        var newNodeEnd = new Node() { Name = input[1].First() };
        var maxFlowForNode = int.Parse(input[2]);

        Node fromNode = null;
```

```
            if (!_nodes.TryGetValue(newNodeStart.Name, out fromNode))
            {
                _nodes.Add(newNodeStart.Name, newNodeStart);
                fromNode = newNodeStart;
            }

            Node toNode = null;
            if (!_nodes.TryGetValue(newNodeEnd.Name, out toNode))
            {
                _nodes.Add(newNodeEnd.Name, newNodeEnd);
                toNode = newNodeEnd;
            }

            /// Add just one node for child and parent
            var edge = new Edge { From = fromNode, To = toNode, Flow = new Edge
            fromNode.Children.Add(edge);
            toNode.Parents.Add(edge);
            _edges.Add(edge);
        }
    }

    /// <summary>
    /// Find max flow in stream
    /// </summary>
    /// <returns>Value of flow</returns>
    public int FindMaxFlow()
    {
        var nodesToVisit = new Queue<Node>().Add(Source);
        var visitedNodes = new HashSet<Node>();

        while (nodesToVisit.Count != 0)
        {
            var tmp = nodesToVisit.First();
```

```
Logger.Log($"Current node => {tmp.Name}", Logger.LogLevel.Debug);

if (tmp == Sink)
{
    Logger.Log("Start patch", Logger.LogLevel.Debug);
    PatchFlowInPath(); // Change glow in path
    nodesToVisit = new Queue<Node>().Add(Source);
    visitedNodes.Clear();
    continue;
}

/// Search path to sink
foreach (var edge in tmp.Children
    .OrderBy(x => x.To.Name)
    .Where(x => !x.IsFullFlow) // In direct direction
    .Concat(tmp.Parents
        .OrderBy(x => x.From.Name)
        .Where(x => !x.IsEmptyFlow)) // In reverse direction
    )
{
    var from = edge.From;
    var to = edge.To;
    if (tmp == from && !visitedNodes.Contains(to)) // Came from dir
    {
        to.CameFrom = edge;
        nodesToVisit.Enqueue(to);
    }
    else if (tmp == edge.To && !visitedNodes.Contains(from)) // Cam
    {
        from.CameFrom = edge;
        nodesToVisit.Enqueue(from);
    }
}

visitedNodes.Add(tmp);
```

```
            nodesToVisit.Dequeue();
        }

        return Sink.Parents.Sum(x => x.Flow.Current);
    }

    /// <summary>
    /// Find max available flow and fill path with it
    /// </summary>
    private void PatchFlowInPath()
    {
        var tmpNode = Sink;
        var tmpEdge = Sink.CameFrom;
        var minFlow = tmpEdge.Flow.Max;

        while (tmpNode != Source)
        {
            var isDirectWay = tmpEdge.To == tmpNode;
            var tmpFlow = isDirectWay ? tmpEdge.Flow.Max - tmpEdge.Flow.Current
            if (tmpFlow < minFlow)
            {
                minFlow = tmpFlow;
            }

            tmpNode = isDirectWay ? tmpEdge.From : tmpEdge.To;
            tmpEdge = tmpNode.CameFrom;
        }

        Logger.Log($"Min flow = {minFlow}", Logger.LogLevel.Debug);

        tmpNode = Sink;
        tmpEdge = Sink.CameFrom;

        while (tmpNode != Source)
        {
```

```
            var isDirectWay = tmpEdge.To == tmpNode;
            if (isDirectWay)
            {
                tmpEdge.Flow.Current += minFlow;
            }
            else
            {
                tmpEdge.Flow.Current -= minFlow;
            }

            Logger.Log($"Flow from {tmpEdge.From.Name} to {tmpEdge.To.Name} = {

            if (tmpEdge.IsFullFlow)
            {
                Logger.Log($"Edge from {tmpEdge.From.Name} to {tmpEdge.To.Name}
            }

            tmpNode = isDirectWay ? tmpEdge.From : tmpEdge.To;
            tmpEdge = tmpNode.CameFrom;
        }
    }

    /// <summary>
    /// Edge between two nodes
    /// </summary>
    private class Edge
    {
        public class FlowEntity
        {
            /// <summary>
            /// Max flow
            /// </summary>
            public int Max { get; set; }

            /// <summary>
```

```
    /// Current flow
    /// </summary>
    public int Current { get; set; }
}

/// <summary>
/// <see cref="Flow"/>
/// </summary>
public FlowEntity Flow { get; set; }

/// <summary>
/// Start <see cref="Node"/>
/// </summary>
public Node From { get; set; }

/// <summary>
/// Target <see cref="Node"/>
/// </summary>
public Node To { get; set; }

/// <summary>
/// Check that <see cref="FlowEntity"/> has max value
/// </summary>
public bool IsFullFlow
{
    get { return Flow.Current == Flow.Max; }
}

/// <summary>
/// Check that <see cref="FlowEntity"/> has zero value
/// </summary>
public bool IsEmptyFlow
{
    get { return Flow.Current == 0; }
}
```

```
}

/// <summary>
/// Node in graph
/// </summary>
private class Node
{
    /// <summary>
    /// Name of node
    /// </summary>
    public char Name { get; set; }

    /// <summary>
    /// Children to children nodes
    /// </summary>
    public List<Edge> Children { get; } = new List<Edge>();

    /// <summary>
    /// Parents of children nodes
    /// </summary>
    public List<Edge> Parents { get; } = new List<Edge>();

    /// <summary>
    /// <see cref="Node"/> parent node in path
    /// </summary>
    public Edge CameFrom { get; set; }

}

/// <summary>
/// Start point
/// </summary>
private Node Source { get; set; }

/// <summary>
```

```
        /// End point
        /// </summary>
        private Node Sink { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var graph = new Graph();
            graph.ReadGraph();
            Logger.Log(graph.FindMaxFlow());
            graph.PrintEdges();
        }
    }
}
```