МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ

по лабораторной работе №3
по дисциплине «Построение и анализ алгоритмов»
Тема: Потоки в сети

Студент гр. 7383 _____ Александров Р.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2019

Цель работы.

Познакомиться с алгоритмом поиска максимального потока в графе, используя алгоритм Форда-Фалкерсона, и его реализацией.

Постановка задачи.

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

1с. Списки смежности. Поиск пути через поиск в ширину.

Реализация задачи.

В ходе работы были написаны классы Runner, Vertex, Edge и Graph.

Класс Runner начинает работу программы, предоставляет консольный ввод и заполнение вершин графа.

Класс Vertex хранит название вершины, её индекс.

Класс Edge хранит ребро между двумя вершинами, обе эти вершины, величины потока и пропускной способности, а также методы:

- Vertex to() возвращает вершину, в которую нужно попасть;
- Vertex other(Vertex v) возвращает обратную вершину по отношению к переданной в аргументе функции;
- int getFlow() возвращает поток через ребро;
- int residualCapacityTo(Vertex v) и void addResidualFlowTo(Vertex v, int delta) реализуют остаточную сеть.

Класс Graph хранит список смежности adjList, являющий собой граф; массив вершин vertexArr, количество вершин totalVerts, стартовую и конечные веришны и их индексы, величину максимального потока, массив ребер.

Метод void defineStartEndIndexes() инициализирует индексы начальной и конечной вершин.

Метод boolean searchBFS(Vertex src, Vertex dest) осуществляет поиск в ширину в графе.

Метод void getMaxFlow() осуществляет основную работу алгоритма, находит кратчайший расширяющий путь в остаточной сети, минимальную пропускную способность в этом пути и расширяет поток вдоль этого пути, продолжая так до исчерпания путей из источника в сток

Метод LinkedList<Edge> getEdge(Vertex v) возвращает список вершин доступных из вершины, переданной в аргументе.

Метод Vertex getVertexByName(String label) возвращает вершину по имени.

Метод void printAnswer(ArrayList<Vertex> closedSet) выводит результат работы алгоритма.

Метод void printGraphFlow() осуществляет вывод ответа, используя вложенный класс Answer.

Метод void addVertex(String label) добавляет вершины в граф.

Метод void addEdge(String start, String end, double weight) добавляет ребра с весом в матрицу смежности графа.

Исследование алгоритма.

Реализация алгоритма нахождения максимального потока Форда-Фалкерсона с помощью кратчайших расширяющих путей (поиска в ширину) в худшем случае выполняется за время, пропорциональное $VE^2 / 2$.

Тестирование.

На табл. 1 представлен результат тестирования алгоритма.

Таблица 1 - Результаты работы алгоритма

Входные данные	Результат
7 a f a b 7	12 a b 6 a c 6 b d 6

a c 6 b d 6 c f 9 d e 3 d f 4 e c 2	c f 8 d e 2 d f 4 e c 2

Выводы.

В ходе лабораторной работы был изучен и реализован алгоритм поиска максимального потока в графе, используя алгоритм Форда-Фалкерсона.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД

```
public class Runner {

    public void start() {
        Scanner sc = new Scanner(System.in);
        String line;
        int counter = sc.nextInt();
        ArrayList<String> allVertices = new ArrayList<>();
        String startVertex = sc.next();
        String endVertex = sc.next();
        ArrayList<Integer> weights = new ArrayList<>();
        HashSet<String> notRepeatVertex = new HashSet<>();
        sc.skip("\n");
        for (int i = 0; i < counter; ) {
            line = sc.nextLine();
            String[] inits = line.split(" ");
            String start = inits[0];
            String end = inits[1];
            int weight = Integer.parseInt(inits[2]);
            allVertices.add(start);
            allVertices.add(end);
            weights.add(weight);
            notRepeatVertex.add(start);
            notRepeatVertex.add(end);
            i++;
        }

        Graph graph = new Graph(notRepeatVertex.size());
        for (String s : notRepeatVertex) {
            graph.addVertex(s);
        }

        graph.initStartVertex(startVertex);
        graph.initEndVertex(endVertex);

        for (int i = 0, j = 0; i < weights.size(); i++, j++) {
            graph.addEdge(allVertices.get(j),  allVertices.get(j  +
1), weights.get(i));
            j += 1;
        }

        graph.getMaxFlow();
```

```
        graph.printGraphFlow();
    }


    public static void main(String[] args) {
        new Runner().start();
    }
}

public class Graph {
    private Vertex[] vertexArr;
    private LinkedList<Edge>[] adjList;
    private int totalVerts;
    private Vertex startVertex;
    private Vertex endVertex;
    private int startVertexIndex;
    private int endVertexIndex;
    private int maxFlow;
    private Edge[] edgeTo;
    public Graph(int initSize) {
        vertexArr = new Vertex[initSize];
        totalVerts = 0;
        adjList = new LinkedList[initSize];
        for (int i = 0; i < initSize; i++) {
            adjList[i] = new LinkedList<>();
        }
    }

    private boolean searchBFS(Vertex src, Vertex dest) {
        edgeTo = new Edge[totalVerts];
        boolean[] marked = new boolean[totalVerts];
        Queue<Vertex> queue = new LinkedList<>();
        queue.add(src);
        marked[src.getIndex()] = true;
        while (!queue.isEmpty()) {
            Vertex curV = queue.poll();
            for (Edge e : getEdge(curV)) {
                Vertex destCurv = e.other(curV);
                if         (!marked[destCurv.getIndex()]         &&
(e.residualCapacityTo(destCurv) > 0)) {
                    edgeTo[destCurv.getIndex()] = e;
                    marked[destCurv.getIndex()] = true;
                    queue.add(destCurv);
                }
            }
        }
```

```
        return marked[dest.getIndex()];
    }

    public void getMaxFlow() {
        defineStartEndIndexes();
        int flow = 0;
        startVertex = vertexArr[startVertexIndex];
        endVertex =  vertexArr[endVertexIndex];
        while (searchBFS(startVertex, endVertex)) {
            int pathFlow = Integer.MAX_VALUE;
            for (Vertex v = endVertex; v.getIndex() !=
startVertex.getIndex(); v = edgeTo[v.getIndex()].other(v)) {
                pathFlow = Math.min(pathFlow,
edgeTo[v.getIndex()].residualCapacityTo(v));
            }
            for (Vertex v = endVertex; v.getIndex() !=
startVertex.getIndex(); v = edgeTo[v.getIndex()].other(v)) {
                edgeTo[v.getIndex()].addResidualFlowTo(v,
pathFlow);
            }
            flow += pathFlow;
        }
        maxFlow = flow;
    }


    private void defineStartEndIndexes() {
        for (int i = 0; i < vertexArr.length; i++) {
            if
(vertexArr[i].getLabel().equals(startVertex.getLabel())) {
                startVertexIndex = i;
            }
        }
        for (int i = 0; i < vertexArr.length; i++) {
            if
(vertexArr[i].getLabel().equals(endVertex.getLabel())) {
                endVertexIndex = i;
            }
        }
    }

    public void addVertex(String label) {
        vertexArr[totalVerts] = new Vertex(label);
        vertexArr[totalVerts].setIndex(totalVerts);
        totalVerts++;
    }
```

```
public void addEdge(String start, String end, int weight) {
    Vertex startV = getVertexByName(start);
    Vertex endV = getVertexByName(end);
    Edge edge = new Edge(startV, endV, weight);
    adjList[startV.getIndex()].add(edge);
    adjList[endV.getIndex()].add(edge);
}

private LinkedList<Edge> getEdge(Vertex v) {
    return adjList[v.getIndex()];
}

private Vertex getVertexByName(String label) {
    for (int i = 0; i < totalVerts; i++) {
        if (vertexArr[i].getLabel().equalsIgnoreCase(label)) {
            return vertexArr[i];
        }
    }
    return vertexArr[startVertexIndex];
}

public void initStartVertex(String startVertex) {
    this.startVertex = new Vertex(startVertex);
}

public void initEndVertex(String endVertex) {
    this.endVertex = new Vertex(endVertex);
    defineStartEndIndexes();
}

private class Answer {
    Vertex first;
    Vertex second;
    int flow;
    Answer(Vertex first, Vertex second, int flow) {
        this.first = first;
        this.second = second;
        this.flow = flow;
    }
}

public void printGraphFlow() {
    ArrayList<Answer> answers = new ArrayList<>();
    for (int i = 0; i < totalVerts; i++) {
        LinkedList<Edge> list = adjList[i];
```

```
            for (int j = 0; j < list.size(); j++) {
                if (i == list.get(j).to().getIndex()) continue;
                Answer answer = new Answer(vertexArr[i],
list.get(j).to(), list.get(j).getFlow());
                answers.add(answer);
            }
        }

        Collections.sort(answers, Comparator.comparing((Answer o) -
> o.first.getLabel()).thenComparing(o -> o.second.getLabel()));
        System.out.println(maxFlow);
        for (int i = 0; i < answers.size(); i++) {
            System.out.println(answers.get(i).first.getLabel() + "
" + answers.get(i).second.getLabel() + " " + answers.get(i).flow);
        }
    }
}
public class Vertex {
    private String label;
    private int index;

    public Vertex(String label) {
        this.label = label;
    }

    public void setLabel(String label) {
        this.label = label;
    }

    public int getIndex() {
        return index;
    }

    public void setIndex(int index) {
        this.index = index;
    }

    public String getLabel() {
        return label;
    }
}
public class Edge {
    private Vertex srcVert;
    private Vertex destVert;
    private int capacity;
    private int flow;
```

```
    public Edge(Vertex srcVert, Vertex destVert, int capacity) {
        this.srcVert = srcVert;
        this.destVert = destVert;
        this.capacity = capacity;
        this.flow = 0;
    }

    public Vertex to() {
        return destVert;
    }

    public int getFlow() {
        return flow;
    }

    public Vertex other(Vertex v) {
        if (v.getIndex() == srcVert.getIndex()) {
            return destVert;
        } else if (v.getIndex() == destVert.getIndex()) {
            return srcVert;
        }
        else throw new IllegalArgumentException();
    }

    public int residualCapacityTo(Vertex v)
    {
        if (v.getIndex() == srcVert.getIndex()) return flow;
        else  if  (v.getIndex()  ==  destVert.getIndex())  return
capacity - flow;
        else throw new IllegalArgumentException();
    }

    public void addResidualFlowTo(Vertex v, int delta)
    {
        if (v.getIndex() == srcVert.getIndex()) flow -= delta;
        else if (v.getIndex() == destVert.getIndex()) flow += delta;
    }

}
```