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

ОТЧЕТ

по лабораторной работе №3
по дисциплине «Построение и анализ алгоритмов»
Тема: Максимальный поток

Студент гр. 9382 _____ Кузьмин Д. И.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург
2021

Цель работы.

Изучить алгоритм Форда-Фалкерсона для нахождения максимального потока в сети. Освоить навыки разработки программ, реализующих этот алгоритм.

Основные теоретические положения.

Задача о максимальном потоке заключается в нахождении такого потока по транспортной сети, что сумма потоков из истока, или, что то же самое, сумма потоков в сток максимальна.

Задание.

Вар. 5. Поиск не в глубину и не в ширину, а по правилу: каждый раз выполняется переход по дуге, имеющей максимальную остаточную пропускную способность. Если таких дуг несколько, то выбрать ту, которая была обнаружена раньше в текущем поиске пути.

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 ω_{ij} - ребро графа
v_i v_j ω_{ij} - ребро графа
...

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

Описание функций и структур данных.

1) Для описания вершина графа использовался класс Vertex, имеющий поля:

char name – имя вершины;

bool operator== (Vertex v2) – оператор для сравнения вершин, v2 – сравниваемая с данной вершиной

2) Для описания ребра графа использовался класс Edge, имеющий поля:

Vertex v1 –начальная вершина

Vertex v2 –конечная вершина

int flow – поток, который идет по ребру

int capacity – остаточная пропускная способность

int originalCapacity – изначальная пропускная способность

Edge* reverseEdge – указатель на обратное ребро

bool isReverse – показывает, является ли ребро обратным

bool operator==(Edge e) – оператор для сравнения с ребром E

void addReverseEdge(Edge* e) – добавление обратной дуги к ребру

3) Для представления графа используется класс Graph, реализованный в виде списка ребер на std::vector<Edge>

4) В этом классе реализованы функции:

Vertex* operator()(char verName1, char verName2) – переопределенный оператор () для получения указателя на ребро. verName1 – имя начальной вершины ребра; verName2 – имя конечной вершины ребра. Возвращаемое значение – указатель на ребро, если оно есть в графе и nullptr, если нет

void addEdge(char v1, char v2, int cap, int flow = 0) – добавление ребра в граф. v1, v2 - имена начальной и конечной вершины ребра, cap – пропускная способность ребра, flow – поток ребра.

bool input() – считывание графа через список ребер. Возвращает true, если граф удалось считать и false – если ввод некорректен.

char root – начальная вершина при поиске пути

char goal– конечная вершина при поиске пути

5) Edge findMaxCapacityEdge(Graph graph, Vertex v1, std::vector<Vertex> blocked) – нахождение ребра максимальной пропускной способности, исходящей из вершины. v1 – вершина, из которой ищется ребро; graph – граф, в котором ищется ребро; blocked – заблокированные вершины, путь ребра до которых не ищутся.

6) Edge findPath(Graph graph) – поиск пути. graph – граф, в котором ищется путь.

7) void findMaxFlow(Graph graph) – нахождение максимального потока. graph – граф, в котором ищется поток

Описание алгоритма (поиск пути)

1) Для алгоритма используется 3 вектора, в которых хранятся посещенные вершины, имеющиеся в пути ребра и заблокированные вершины, в которые нельзя проложить путь на данном шаге.

2) Первым шагом в посещенные кладется начальная вершина и выбирается дуга с максимальной пропускной способностью(рассматриваются как прямые дуги, так и обратные), ведущая к еще непосещенной вершине.

3) Далее, если такая дуга существует, то осуществляется переход по ней и она добавляется в вектор пути.

4) Если не существует, то осуществляется возврат к последней вершине и выбирается другая дуга. Вершина из которой сделан возврат добавляется в вектор заблокированных, т.е. из нее путь построить не удастся.

5) Алгоритм завершается, когда конец обнаруженной дуги – конечная вершина пути. Либо, когда вектор посещенных вершин – пуст, такое происходит, когда путь найти не удается.

6) Сложность по времени - $O(E \cdot \mathrm{V})$, по памяти – $O(E + V)$, где E – ребра в графе, V – вершины

Описание алгоритма (нахождение максимального потока) .

1) Вводится понятие остаточной сети, в которой остаточная пропускная способность каждой дуги равна разности изначальной пропускной способности и потока по ней.

2) Алгоритм итеративный и повторяется до тех пор, пока можно найти путь из источника в сток.

3) После нахождения пути, пропускная способность дуг, входящих в него уменьшается на величину, равную минимальной пропускной способности дуги в этом пути. Поток по всем дугам увеличивается на эту величину. Остаточная пропускная способность обратных дуг уменьшается на эту величину.

4) Сложность по времение можно оценить $O(E^2 \cdot V)$, по памяти – $O(E + V)$, где E – количество ребер в графе, V– количество вершин.

Исходный код см. в приложении А.

Тестирование.

Результаты тестирование представлены в табл. 1.

Таблица 1 — результаты тестирования.

№ п/п	Входные данные	Выходные данные	Комментарий
1	7 a f a b 7 a c 6 b d 6 c f 9 d e 3 d f 4 e c 2	12 a b 6 a c 6 b d 6 c f 8 d e 2 d f 4 e c 2	Получен максимальный поток.
2	10	6	Такой поток существует.

	2 6 1 2 1 1 3 7 2 3 1 2 4 3 2 5 2 3 5 4 4 5 1 4 6 6 5 6 2 5 4 5	1 2 0 1 3 0 2 3 1 2 4 3 2 5 2 3 5 1 4 5 0 4 6 6 5 4 3 5 6 0	
3	16 1 8 1 2 32 1 3 95 1 4 75 1 5 57 2 8 16 2 3 5 2 5 23 3 4 18 3 6 6 4 6 9 4 5 24 5 7 20 5 8 94 6 5 11 6 7 7 7 8 81	128 1 2 32 1 3 24 1 4 15 1 5 57 2 3 0 2 5 16 2 8 16 3 4 18 3 6 6 4 5 24 4 6 9 5 7 18 5 8 87 6 5 8 6 7 7 7 8 25	Найден корректный поток.
4	-9 s t s a 7	Некорректный ввод	Число ребер не может быть отрицательным

	a b 5 b t 8 s d 4 d c 2 c t 6 c b 3 a c 3 d a 3		
5	18 s t s a 12 s b 8 s c 11 a b 5 a e 7 a f 6 b c 2 b f 9 b d 4 c b 13 c d 6 c g 12 d g 7 f d 15 f t 9 g f 8 g t 10 e t 3	22 a b 0 a e 3 a f 6 b c 0 b d 0 b f 7 c b 1 c d 0 c g 6 d g 7 e t 3 f d 7 f t 9 g f 3 g t 10 s a 9 s b 6 s c 7	Поток найден корректно, т.к. для каждого ребра сумма исходящих потоков равна сумме входящих.

Выводы.

Был изучен принцип алгоритма поиска максимального потока в сети. Получены навыки разработки программ, реализующих этот алгоритм.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <stack>
#include <algorithm>
#include <deque>
#include <iostream>
#include <vector>
#include <chrono>
#include <fstream>
#include <functional>
#include <queue>
#include <string>
#include <map>

#define INFO


class Edge;
//вершина графа
class Vertex {
public:

    char name;
    Vertex(char name) :name(name) {}
    Vertex() {}

    bool operator==(Vertex v2) {
        return this->name == v2.name;
    }

    friend bool operator<(Vertex v1, Vertex v2) {
        return v1.name < v2.name;
    }
};


//ребро графа
class Edge {
public:
    Vertex v1;
    Vertex v2;
    int flow = 0;
    int capacity;
    int originalCapacity;
    Edge* reverseEdge;
    bool hasReverse = false;
    bool isReverse = false;
    void addReverseEdge(Edge* e) {
        this->reverseEdge = e;
        this->hasReverse = true;
        if (e != nullptr) {
            e->reverseEdge = this;
            e->hasReverse = true;
        }
    }
    Edge() {}
```

```
    Edge(Vertex v1, Vertex v2, int capacity, int flow = 0) :
        v1(v1), v2(v2), capacity(capacity), flow(flow) {
        originalCapacity = capacity;

    }
    friend std::ostream& operator<< (std::ostream& out, Edge e) {
        out << "(" << e.v1.name << "," << e.v2.name << "," << e.capacity
<< ") ";
        return out;
    }
    friend bool operator< (Edge e1, Edge e2) {
        if (e1.v1.name < e2.v1.name)
            return true;

        else if (e1.v1.name == e2.v1.name)
            return (e1.v2.name < e2.v2.name);

        return false;
    }
    bool operator==(Edge e) {
        return this->v1.name == e.v1.name && this->v2.name == e.v2.name;
    }

};

template<typename T>
void print_queue(T q) {
    while (!q.empty()) {
        std::cout << *q.top();
        q.pop();
    }
    std::cout << '\n';
}


template <typename T, typename B>
bool isInVector(T vec, B e) {
    for (auto it : vec)
        if (it == e)return true;
    return false;
}

//граф
class Graph {
public:
    std::vector<Vertex> vertexVector;
    std::vector<Edge> edgeVector;

    bool isInGraph(char verName) {
        for (auto it : vertexVector)
            if (it.name == verName)
                return true;
        return false;
    }

    Vertex* operator()(char verName) {
        for (auto& it : vertexVector) {
            if (it.name == verName)
```

```
                return &it;
            }
            return nullptr;
        }
        Edge* operator()(char verName1, char verName2) {
            for (auto& it : edgeVector) {
                if (it.v1.name == verName1 && it.v2.name == verName2)
                    return &it;
            }
            return nullptr;
        }

            //вывод ребер
            void printEdges() {
                std::sort(edgeVector.begin(), edgeVector.end());
                for (auto it : edgeVector) {

                    if (it.flow < 0)it.flow = 0;
                    std::cout << it.v1.name << " " << it.v2.name << " " <<
it.flow << "\n";
                }
            }

            //добавление вершины
            void addVertexByName(char verName) {
                if (!isInGraph(verName))
                    vertexVector.push_back(Vertex(verName));
            }

            //добавление ребра
            void addEdge(char v1, char v2, int cap, int flow = 0) {

                Edge* addedEdge = new Edge(v1, v2, cap);
                if (!isInVector(edgeVector, Edge(v1, v2, cap))) {

                    edgeVector.push_back(*addedEdge);

                }
            }

            char root, goal;

            //ввод
            bool input() {
                char verName1;
                char verName2;
                int capacity;
#ifdef FILEINPUT
                std::cout << "Ввод:\n";

                while (std::getline(file, tmp)) {
                    std::cout << tmp << "\n";
                }
                std::cout << "\n";
                file.close();
#endif

                //считывание списка ребер
```

```
            int n;
            bool inputSuccess = true;
            std::cin >> n >> root >> goal;
            if (n < 0) inputSuccess = false;

            for (int i = 0; i < n; i++) {
                std::cin >> verName1 >> verName2 >> capacity;
                if (capacity < 0) inputSuccess = false;
                addVertexByName(verName1);
                addVertexByName(verName2);
                addEdge(verName1, verName2, capacity);
            }

            //добавление обратных ребер
            for (auto& it : edgeVector) {
                it.addReverseEdge((*this)(it.v2.name, it.v1.name));
                if (it.reverseEdge == nullptr) {
                    it.addReverseEdge(new Edge(it.v2.name, it.v1.name, 0,
it.capacity));
                }
            }
            return inputSuccess;
        }
    };

    Edge findMaxCapacityEdge(Graph graph, Vertex v1, std::vector<Vertex>
blocked) {

        //ребро, которое возвращается, если пути нет
        Edge max = Edge('_', '-', -200);
        bool reverse = false;

        //лямбда функция проверки, подходит ли ребро для продолжения пути
        auto lambda = [&](Edge it, bool reverse) {
            bool found = false;
#ifdef INFO
            if (it.v1 == v1) {
                std::cout << "Вершина " << it.v2.name << " дуга ";
                if (reverse) std::cout << "\"обратная\" ";
                std::cout << it;
            }
#endif
            if (it.v1 == v1 && max.capacity < it.capacity && it.capacity
> 0) {

                bool has = isInVector(blocked, it.v2); //если вершина не
содержится среди посещенных
                if (!has) {
                    max = it;
                    max.isReverse = reverse;
                }
                else {
#ifdef INFO
                    std::cout << "(уже посещена)";
#endif
                }
            }
#ifdef INFO
```

```
            if (it.v1 == v1) std::cout << "\n";
#endif
        };
        //-------

        //проверка для обычных ребер
        for (const auto& it : graph.edgeVector) {
            lambda(it, 0);
        }

        //для "обратных"
        for (const auto& it : graph.edgeVector) {
            auto it2 = *graph(it.v1.name, it.v2.name)->reverseEdge;
            lambda(it2, 1);
        }
        return max;
    }
    std::vector<Edge> findPath(Graph graph) {


        std::vector<Edge> path;
        std::vector<Vertex> block;
        std::vector<Vertex> pathV;
        pathV.push_back(*graph(graph.root));

#ifdef INFO
        std::cout << "\n\nПоиск пути \n";
#endif
        Vertex current = *graph(graph.root);

        //основной цикл
        while (!pathV.empty()) {

#ifdef INFO
            std::cout << "\nСоседи вершины " << current.name << ":\n";

#endif
            //нахождение макисмальной дуги
            Edge nextEdge = findMaxCapacityEdge(graph, current, block);

            //если нашлась продолжение пути
            if (nextEdge.capacity != -200) {
#ifdef INFO
                std::cout << "Переход по дуге " << nextEdge << "\n";
#endif
                current = *graph(nextEdge.v2.name);
                path.push_back(nextEdge);
                pathV.push_back(current);
                if (nextEdge.v2.name == graph.goal) break;
                block.push_back(current);
            }
            //если нет, возвращение к последней посещенной вершине
            else {
#ifdef INFO
                std::cout << "Нет доступных путей из вершины " <<
current.name << "\nВозврат к предыдущей вершине\n";
#endif
                block.push_back(pathV.back());
```

```
                pathV.pop_back();

                if (!path.empty()) path.pop_back();

                if (pathV.empty()) {
#ifdef INFO
                    std::cout << "\nПути не существует\nЗавершение
алгоритма\n\n";
#endif
                    return path;
                }
                else current = *graph(pathV.back().name); //когда ребер в
пути нет, но есть одна вершина - начальная
            }
            //----------
        }

        //вывод
#ifdef INFO
        std::cout << "\nНайденный путь: ";
        for (auto it : pathV)
            std::cout << it.name;
        std::cout << "\n";
#endif

        return path;
    }
    void findMaxFlow(Graph graph) {

        Graph rGraph(graph); //остаточная сеть
        int maxflow = 0;
        Edge* currEdge;
        std::vector<Edge> currentPath = findPath(rGraph);

        while (currentPath.size() != 0) {
            int min = currentPath.front().capacity;
            //вычисление минимальной пропускной способности дуги в
найденном пути
            for (const auto& pathEdge : currentPath) {

                int tmpCap = pathEdge.capacity;
                min = (tmpCap < min) ? tmpCap : min;
            }
            //------
#ifdef INFO
            std::cout << "Минимальная пропускная способность: " << min <<
"\n";
#endif
            //обновление потоков и пропускных спобностей ребер,
участвуюших в пути
            for (auto pathEdge : currentPath) {

                //если ребро в пути не является обратным
                currEdge = rGraph(pathEdge.v1.name, pathEdge.v2.name);

                //если является, то его надо взять как обратное от ребра
с противоположными вершинами
                if (currEdge == nullptr) {
```

```
                    currEdge = rGraph(pathEdge.v2.name,
pathEdge.v1.name)->reverseEdge;
                }

#ifdef INFO
                std::cout << "\nПропускная способность: " << *currEdge <<
" изменена с " << currEdge->capacity;
                std::cout << " на " << currEdge->capacity - min << "\n";
                std::cout << "Пропускная способность \"обратного\" ребра:
" << *currEdge->reverseEdge << " изменена с " << currEdge->reverseEdge-
>capacity;
                std::cout << " на " << currEdge->reverseEdge->capacity +
min << "\n";

                std::cout << "Поток по дуге: " << *currEdge << " изменен
с " << currEdge->flow;
                std::cout << " на " << currEdge->flow + min << "\n";
                std::cout << "Поток по \"обратной\" дуге: " << *currEdge-
>reverseEdge << " изменен с " << currEdge->reverseEdge->flow;
                std::cout << " на " << currEdge->reverseEdge->flow - min
<< "\n";
#endif

                //обновление потоков и пропускных способностей
                currEdge->flow += min;
                currEdge->capacity -= min;
                Edge* reverseOfCurr = rGraph(currEdge->v2.name, currEdge-
>v1.name);

                if (reverseOfCurr != nullptr) {
                    reverseOfCurr->reverseEdge = currEdge;
                    currEdge->reverseEdge = reverseOfCurr;
                }

                currEdge->reverseEdge->flow -= min;
                currEdge->reverseEdge->capacity += min;
            }
            maxflow += min;
            currentPath = findPath(rGraph);
        }
        std::cout << maxflow << "\n";
        rGraph.printEdges();
    }

    int main()
    {
        Graph graph;
        setlocale(LC_ALL, "rus");
        bool input = graph.input();
        if (!input) {
            std::cout << "Некорректный ввод\n";
            exit(-1);
        }
        std::cout << "\nПоиск максимального потока. "<<graph.root << " -
источник, " << graph.goal << " - сток";
        findMaxFlow(graph);
        return 0;
    }
```