

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

$v_i v_j \omega_{ij}$ - ребро графа

$v_i v_j \omega_{ij}$ - 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).

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

std::vector<Edge*> neighbourEdges – исходящие ребра вершины

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(std::vector<Edge> vec, std::vector<Edge> blocked) – нахождение дуги максимальной пропускной способности, vec – вектор, в котором ищется дуга; blocked – заблокированные дуги, которые игнорируются при поиске. Возвращаемое значение – найденная дуга.

6) std::deque<Edge> findPath(Graph graph) – поиск пути. graph – граф, в котором ищется путь. Возвращаемое значение - очередь из дуг, входящих в 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 соответственно хранятся дуги, которые можно выбрать для продолжения пути на данном шаге, имеющиеся в пути дуги и заблокированные дуги, которые уже были обработанные.

2) На первом шаге в вектор возможных кладутся дуги, исходящие из начальной вершины.

3) Затем из возможных дуг выбирается дуга с максимальной пропускной способностью.

4) Она помещается в вектор пути и вектор заблокированных дуг, т.е. вновь ее выбрать уже будет нельзя.

5) Каждая дуга, исходящая из конечной вершины(а также входящая в нее, т.е. обратная) найденной дуги добавляется в вектор возможных дуг для продолжения пути.

6) Шаги 3 – 5 повторяются пока, вновь найденная дуга ведет к конечной вершине или когда продолжить путь не удастся.

Сложность по времени - $O(E^2)$, по памяти – $O(E)$, 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.

Сложность по времени можно оценить $O(E \cdot f)$, по памяти – $O(E + V)$, где E – количество ребер в графе, V – количество вершин, f – максимальный поток.

Исходный код см. в приложении А.

Тестирование.

Результаты тестирования 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 5 5 4 2 5 6 1	
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 6 1 4 33 1 5 57 2 3 0 2 5 20 2 8 12 3 4 0 3 6 6 4 5 24 4 6 9 5 7 20 5 8 90 6 5 9 6 7 6 7 8 26	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 1 b c 0 b d 0 b f 8 c b 0 c d 0 c g 10 d g 0 e t 3 f d 0 f t 9 g f 0 g t 10 s a 4 s b 8 s c 10	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;
    std::vector<Edge*> neighbourEdges;
    void addNeighbour(Edge* e) {
        neighbourEdges.push_back(e);
    }
    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;
}

```

```

//rpaф
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);
        }
    }
};

```

```

        (*this)(v1)->addNeighbour(addedEdge);
    }
}

char root, goal;

//ввод
bool input() {
    char verName1;
    char verName2;
    int capacity;

    //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(std::vector<Edge> vec, std::vector<Edge>
blocked) {

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

    for (const auto& it: vec) {
#ifdef INFO
        std::cout << "Дура ";
        if (it.isReverse) std::cout << "\"обратная\" ";
        std::cout << it;
#endif
    }
}

```

```

        if (max.capacity < it.capacity && it.capacity > 0){

            //если ребро не содержится среди посещенных
            bool has = isInVector(blocked, it);
            if (!has) {
                max = it;
            }
            else {

#ifdef INFO
                std::cout << "(уже обработана)";
#endif

            }
        }

#ifdef INFO
        std::cout << "\n";
#endif

    };
    //-----
    return max;
}

std::deque<Edge> findPath(Graph graph){

    std::vector<Edge> path;
    std::vector<Edge> reserve;
    std::vector<Edge> blocked;
    for (const auto& it : graph(graph.root)->neighbourEdges){
        if (!isInVector(reserve, *it))
            reserve.push_back(*graph(it->v1.name, it->v2.name));
    }
    while (1){
#ifdef INFO
        std::cout << "\nСписок возможных дуг для построения пути\n";
#endif

        //берется дуга с максимальной пропускной способностью
        Edge nxt = findMaxCapacityEdge(reserve, blocked);

        //если путь найти не удастся
        if (nxt.capacity == -200){
#ifdef INFO
            std::cout << "\nНет доступных путей. Завершение
алгоритма\n";
#endif
            return std::deque<Edge>();
        }
        else{
#ifdef INFO
            std::cout << "\nВыбранная дуга - " << nxt << "\n";
#endif

```

```

        //восстановление пути, если вновь взятая дуга ведет к
конечной вершине
        if (nxt.v2 == graph.goal) {
            std::deque<Edge> actualPath;
            actualPath.push_front(nxt);
            Edge previous;
            while (nxt.v1.name != graph.root) {
                for (auto it : path) {
                    if (it.v2 == nxt.v1) {
                        previous = it;
                        break;
                    }
                }
                nxt = previous;
                actualPath.push_front(nxt);
            }

#ifdef INFO
            std::cout << "\nНайденный путь:";
            std::cout << actualPath[0].v1.name;
            for (auto it : actualPath)
                std::cout << it.v2.name;
            std::cout << "\n";
#endif

            return actualPath;
        }
        //-----

        //добавление найденной дуги в список заблокированных и
список пути
        blocked.push_back(nxt);
        path.push_back(nxt);

        //добавление соседей и их обратные дуги в список
возможных для продолжения пути
        for (auto it : graph(nxt.v2.name)->neighbourEdges){
            if (!isInVector(reserve, *it) && !isInVector(blocked,
*it)){

                it = graph(nxt.v2.name, it->v2.name);
                reserve.push_back(*graph(it->v1.name, it->v2.name));
                for (auto itr : graph.edgeVector){
                    Edge r = *itr.reverseEdge;
                    if (r.v1 == it->v1 && !isInVector(reserve, r)){
                        reserve.push_back(r);
                        reserve.back().isReverse = true;
                    }

#ifdef INFO
                    std::cout << "\"Обратная\" дуга " << r << "
добавлена в список возможных дуг\n";
#endif
                }
            }
        }
    }
}

```

```

    }

#ifdef INFO
        std::cout <<"Дуга " << *it << " добавлена в список
возможных дуг\n";
#endif
    }
}

}

}

void findMaxFlow(Graph graph) {
    Graph rGraph(graph); //остаточная сеть
    int maxflow = 0;
    Edge* currEdge;
    std::deque<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 способностей ребер,
участвующих в пути
            for (const 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);
    }
#ifdef INFO
    std::cout << "\nПоиск максимального потока. "<<graph.root << " - источник, " << graph.goal << " - сток\n";
#endif
    findMaxFlow(graph);
    return 0;
}

```