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: Потоки в сети
Вариант 4

Студент гр. 8383 _____ Мололкин К.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы

Изучить работу алгоритм Форда-Фалкерсона для нахождения максимального потока в сети и решить поставленную задачу с помощью данного алгоритма.

Постановка задачи

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 w_{ij} - ребро графа с фактической величиной протекающего потока

v_i v_j w_{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).

Вар. 4. Поиск в глубину. Итеративная реализация.

Описание алгоритма

1. В цикле запускаем поиск в глубину, пока можем найти путь в графе.

2. Если нашли путь, пускаем через него максимальный поток.

2.1. Для этого находим в пути ребро с минимальной пропускной способностью – min.

2.2. Затем для каждого ребра пути уменьшаем поток на min, а для противоположного ему ребра увеличиваем на min.

2.3. Так же увеличиваем максимальный поток сети на min.

3. Если пропускная способность ребра в пути равна нулю, удаляем данное ребро из графа.

Поиск в глубину:

1. Создается стек вершин и добавляется в него начальная вершина.

2. Затем в цикле, пока стек не пустой снимем со стека вершину и добавляем все смежные не пройденные с не нулевым весом.

3. Помечаем их пройденными, если среди смежных есть конечная, то заканчиваем поиск.

4. Если не нашли конечную возвращаем false.

Описание структуры данных

Class Edge – используется для хранения ребра

Поля:

1. GraphTop* top – вершина;
2. double weight – вес пути;
3. int factFlow – поток через ребро;
4. bool isStartEdge – флаг показывающий, что ребро входит в исходный граф;

Методы:

1. Pair(GraphTop* top, double weight, bool isStartEdge) – конструктор класса;

Class ResultEdges – используется для хранения и вывода ребер с фактической величиной потока

Поля:

1. char start – начальная вершина;
2. char end – конечная вершина;
3. int flow – максимальный поток через ребро;

Методы:

1. ResultEdges(char start, char end, int flow) – конструктор класса;

Class GraphTop – используется для хранения вершины

Поля:

1. char name – имя вершины;
2. bool wasPassed – флаг для проверки было ли ребро пройдено при поиске в глубину
3. std::vector<Edge*> adjacentEdges – вектор для хранения смежных ребер;

Методы:

1. void addAdjacentEdge(GraphTop* top, double weight, bool isStart) – функция добавляет ребро в вектор для хранения смежных ребер:
   Функция принимает на вход переменные GraphTop* top – смежная вершина и double weight – вес ребра, bool isStart - флаг показывающий, что ребро входит в исходный граф, затем проходится по всем смежным, если данное ребро уже было добавлено с нулевым весом, то обновляем информацию этого ребра, если не было, то добавляем ребро в вектор смежных ребер;
2. GraphTop(char name) – конструктор;

Class EdgesList – используется для хранения графа

Поля:

1. std::vector<GraphTop*> tops – вектор хранящий все вершины графа;
2. std::vector<ResultEdges*> resultEdgesVec – вектор ребер для ответа;
3. int maxFlow – максимальный поток графа;

Методы:

1. void sortTopsForAnswer() – сортировка ребер в векторе resultEdgesVec в лексикографическом порядке сначала по первой, потом по второй вершинам:
2. static bool comp(ResultEdges* a, ResultEdges* b) – компаратор для функции sortTopsForAnswer.
3. void addEdge(char topName, char adjacentTopName, double weight) – добавление ребра в граф:

   Функция принимает на вход char topName – название начальной вершины ребра, char adjacentTopName – название конечной вершины ребра, double weight – вес ребра. Затем для topName и adjacentTopName вызывается функция addOrReturnTop() и результаты записываются в новые переменные, затем для вершины с именем topName добавляет смежное ребро с помощью функции addAdjacentEdge(), передавая параметр isStart = true, также для вершины adjacentTopName добавляем противоположное ребро с параметром isStart = 0 и весом 0;
4. GraphTop* addOrReturnTop (char topName) – проверяет есть ли вершина в графе:

   Функция принимает на вход char topName – название вершины, затем проверяем есть ли заданная вершина в графе, если есть возвращаем указатель на нее, если нет, то создаем вершину и возвращаем указатель на нее;
5. void makeAllTopsNotPassed() – функция проходится по всем ребрам графа и ставит флаг wasPassed = false;
6. bool dfs(char start, char end) – функция поиска в глубину:

Функция принимает на вход char start – начальная вершина, char end – конечная вершина, затем вызывается функция makeAllTopsNotPassed, создается стек и словарь для пути, затем кладем начальную вершину на стек и начинаем цикл пока стек не пуст, снимаем вершину со стека, кладем все ее смежные не пройденные и не с нулевым ребром на стек, если встретили конечную, вызываем функцию changeEdgesWeights() и возвращаем true, если не конечная, то добавляем в словарь пути ребро из смежной в текущую, если цикл закончился, а путь не был найден, возвращаем false.

7. void changeEdgesWeights(std::map<GraphTop*, GraphTop*>& way, GraphTop* startTop, GraphTop* endTop) – функция изменения весов ребер графа из найденного пути:

   Функция принимает на вход std::map<GraphTop*, GraphTop*>& way – ссылка на путь в графе, GraphTop* startTop, GraphTop* endTop – стартовая и конечная вершины в пути. Затем создается два вектора, первый для ребер в пути, а второй для обратных ребер, затем в цикле проходится по всем ребрам пути добавляем их в вектор прямых ребер, а так же добавляем обратные ребра во второй вектор, так же во время добавления прямых ребер ищем ребро минимальный вес ребра в прямом пути - min, затем от каждого прямого ребра отнимем min, и прибавляем min к factFlow, каждого ребра, если вес ребра стал равен нулю, удаляем его из графа, затем проходимся по всем обратным ребрам и увеличиваем из вес на min. В конце прибавляем min к maxflow.

Сложность алгоритма по времени

Поиск в глубину занимает O(E), количество поисков в глубину было равно сумме всех весов ребер, так как минимальный вес ребра равен 1, следовательно максимальное количество поисков в глубину равно сумме весов всех ребер графа, обозначим за n, следовательно сложность алгоритма по времени равна O(E*n).

Сложность алгоритма по памяти

Алгоритм хранит все вершины, ребра, противоположные ребра графа и еще отдельно ребра для ответа, следовательно сложность по памяти O(V + E)

Спецификация программы

Программа написана на языке C++. Программа на вход получает количество ориентированных ребер графа, исток и сток. Затем вводятся ребра графа и их веса. В конце программа печатает максимальный поток в сети.

Тестирование

```
7
a
f
a b 7
a c 6
b d 6
c f 9
d e 3
d f 4
e c 2
Запуск поиск пути
Обрабатываемая вершина: a
        Проверяем смежную вершину: b
        Добавляем на стек
        Проверяем смежную вершину: c
        Добавляем на стек
Обрабатываемая вершина: c
        Проверяем смежную вершину: a
        Пройдена либо вес равен нулю
        Проверяем смежную вершину: f
        Вершина является конечной, заканчиваем
Найденный путь: acf
Поток в пути: 6
Меняем вес ребра: (c,f) 9 -> 3
Меняем вес ребра: (a,c) 6 -> 0
        Вес ребра равен нулю, удаляем его из гр
Меняем вес обратного ребра: (f,c)0 -> 6
Меняем вес обратного ребра: (c,a)0 -> 6
Увеличиваем поток графа на 6
Запуск поиск пути
Обрабатываемая вершина: a
        Проверяем смежную вершину: b
        Добавляем на стек
Обрабатываемая вершина: b
        Проверяем смежную вершину: a
        Пройдена либо вес равен нулю
```

```
        Проверяем смежную вершину: d
        Добавляем на стек
Обрабатываемая вершина: d
        Проверяем смежную вершину: b
        Пройдена либо вес равен нулю
        Проверяем смежную вершину: e
        Добавляем на стек
        Проверяем смежную вершину: f
        Вершина является конечной, заканчиваем поиск
Найденный путь: abdf
Поток в пути: 4
Меняем вес ребра: (d,f) 4 -> 0
        Вес ребра равен нулю, удаляем его из графа
Меняем вес ребра: (b,d) 6 -> 2
Меняем вес ребра: (a,b) 7 -> 3
Меняем вес обратного ребра: (f,d)0 -> 4
Меняем вес обратного ребра: (d,b)0 -> 4
Меняем вес обратного ребра: (b,a)0 -> 4
Увеличиваем поток графа на 4
Запуск поиск пути
Обрабатываемая вершина: a
        Проверяем смежную вершину: b
        Добавляем на стек
Обрабатываемая вершина: b
        Проверяем смежную вершину: a
        Добавляем на стек
        Проверяем смежную вершину: d
        Добавляем на стек
Обрабатываемая вершина: d
        Проверяем смежную вершину: b
        Пройдена либо вес равен нулю
        Проверяем смежную вершину: e
        Добавляем на стек
Обрабатываемая вершина: e
        Проверяем смежную вершину: d
        Пройдена либо вес равен нулю
        Проверяем смежную вершину: c
```

```
        Добавляем на стек
Обрабатываемая вершина: c
        Проверяем смежную вершину: a
        Пройдена либо вес равен нулю
        Проверяем смежную вершину: f
        Вершина является конечной, заканчи
Найденный путь: abdecf
Поток в пути: 2
Меняем вес ребра: (c,f) 3 -> 1
Меняем вес ребра: (e,c) 2 -> 0
        Вес ребра равен нулю, удаляем его
Меняем вес ребра: (d,e) 3 -> 1
Меняем вес ребра: (b,d) 2 -> 0
        Вес ребра равен нулю, удаляем его
Меняем вес ребра: (a,b) 3 -> 1
Меняем вес обратного ребра: (f,c)6 -> 8
Меняем вес обратного ребра: (c,e)0 -> 2
Меняем вес обратного ребра: (e,d)0 -> 2
Меняем вес обратного ребра: (d,b)4 -> 6
Меняем вес обратного ребра: (b,a)4 -> 6
Увеличиваем поток графа на 2
Запуск поиск пути
Обрабатываемая вершина: a
        Проверяем смежную вершину: b
        Добавляем на стек
Обрабатываемая вершина: b
        Проверяем смежную вершину: a
        Добавляем на стек
Обрабатываемая вершина: a
        Проверяем смежную вершину: b
        Пройдена либо вес равен нулю
Путей нет
```

```
Результат работы алгоритма:
12
a b 6
a c 6
b d 6
c f 8
d e 2
d f 4
e c 2
```

№	Input	Output
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
2	16 a e a b 20 b a 20 a d 10 d a 10 a c 30 c a 30	60 a b 20 a c 30 a d 10 b a 0 b c 0 b e 30 c a 0 c b 10

	b c 40 c b 40 c d 10 d c 10 c e 20 e c 20 b e 30 e b 30 d e 10 e d 10	c d 0 c e 20 d a 0 d c 0 d e 10 e b 0 e c 0 e d 0
3	10 a f a b 16 a c 13 c b 4 b c 10 b d 12 c e 14 d c 9 d f 20 e d 7 e f 4	23 a b 12 a c 11 b c 0 b d 12 c b 0 c e 11 d c 0 d f 19 e d 7 e f 4
4	19 a j a b 2 a c 9 a d 3	19 a b 2 a c 9 a d 3 a e 5 b e 0

	a e 5 b e 4 b h 7 c d 8 c f 5 c g 4 d g 6 e g 8 e h 1 e j 6 f g 3 f i 2 g j 4 g i 9 h j 5 i j 8	b h 2 c d 0 c f 5 c g 4 d g 3 e g 0 e h 0 e j 5 f g 3 f i 2 g i 6 g j 4 h j 2 i j 8

Вывод.

В результате выполнения лабораторной работы был изучен алгоритм Форда-Фалкерсона, составлена рабочая программа по поиску максимального потока в сети.

ПРИЛОЖЕНИЕ

КОД ПРОГРАММЫ

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stack>
#include <climits>
#include <map>
#include <utility>
#include <windows.h>

class GraphTop;

class ResultEdges {  // используется для хранения и вывода ребер с
фактической величиной потока
public:
    char start;     //начальная вершина
    char end;       //конечная вершина
    int flow;       //поток через ребро
    ResultEdges(char start, char end, int flow) : start(start),
end(end), flow(flow) {};  //конструктор
};

class Edge { //класс ребро
public:
    GraphTop* top;      // врешина
    int weight;     //вес ребра
    int factFlow;   //поток через ребро
    bool isStartEdge;   //если ребро было стартовым
    Edge(GraphTop* top, int weight, bool isStartEdge) : top(top),
weight(weight), isStartEdge(isStartEdge),factFlow(0) {}; //
конаструктор
};

class GraphTop {
public:
    char name; //имя
    bool wasPassed = false; //было ли пройдено поиском в глубину
    std::vector<Edge*> adjacentEdges; //вектор смежных ребер
    explicit GraphTop(char name) : name(name){}; //конструктор

    void addAdjacentEdge(GraphTop* top, int weight, bool isStart)
{ //добавления смежного ребра
        bool isInAdjacent = false;  //если ребро уже есть в
векторе смежных
        for(auto & adjacentEdge : adjacentEdges){ //проходимся по
всем смежным ребрам
            if(adjacentEdge->top->name == top->name){   //если
ребро уже есть в векторе
                isInAdjacent = true;
```

```
                if(adjacentEdge->weight == 0) adjacentEdge = new
Edge(top, weight, isStart);  //если вес не нулевой, то обновляем
            }
        }
        if(!isInAdjacent) adjacentEdges.push_back(new Edge(top,
weight, isStart)); //если нет ребра
    };
};

class EdgesList {
public:
    std::vector<GraphTop*> tops;    //все вершины графа
    std::vector<ResultEdges*> resultEdgesVec;  //ребра для ответа
    int maxFlow = 0; //максимальный поток через граф

    void sortTopsForAnswer() {  //сортировка ребер для ответа
        for (auto & top : tops) {
            for (int j = 0; j < top->adjacentEdges.size(); ++j) {
                if(top->adjacentEdges[j]->isStartEdge)
resultEdgesVec.push_back(new ResultEdges(top->name, top-
>adjacentEdges[j]->top->name, top->adjacentEdges[j]->factFlow));
            }
        }
        std::sort(resultEdgesVec.begin(), resultEdgesVec.end(),
comp);
    };

    static bool comp(ResultEdges* a, ResultEdges* b) {
//компаратор
        if(a->start == b->start) return a->end < b->end;
        return a->start < b->start;
    };


    void printResult() { //вывод ребер для ответа
        for (auto & i : resultEdgesVec) {
            std::cout << i->start << " " << i->end << " " << i-
>flow << std::endl;
        }
    }

    void addEdge(char topName, char adjacentTopName, int weight) {
//добавление ребра
        GraphTop* top = addOrReturnTop(topName); //ищем вершину по
имени
        GraphTop* adjacentTop = addOrReturnTop(adjacentTopName);
//ищем смежную вершину по имени
        top->addAdjacentEdge(adjacentTop, weight, true);
//вызываем функцию добавление смежного ребра
        adjacentTop->addAdjacentEdge(top, 0, false);  //вызываем
добавление обратного смежного ребра

    };
```

```
    GraphTop* addOrReturnTop(char topName) { //функция проверки
наличия ребра либо ее создания
        for(auto & top : tops){ //ищем переданную вершину в графе
            if(top->name == topName) return top;  //если нашли то
возвращаем
        }
        auto* top = new GraphTop(topName); //если не нашли то
создаем, добавляем в вектор вершин и возвращаем
        tops.push_back(top);    //добавляем вершину в вектор
        return top;
    };

    void makeAllTopsNotPassed(){  //делаем все вершины не
пройденными
        for(auto & top : tops){
            top->wasPassed = false;
        }
    };

    void changeEdgesWeights(std::map<GraphTop*, GraphTop*>& way,
GraphTop* startTop, GraphTop* endTop){  //изменение веса вершин
пути
        std::vector<std::pair<char, Edge*>*> edges;  //ребра пути
        std::vector<std::pair<char, Edge*>*> reversedEdges;
//обратные ребра пути
        int min = INT_MAX;  //мин поток через путь
        std::string foundedWay;
        foundedWay += endTop->name;
        GraphTop* top1 = endTop;  //вершина 1
        GraphTop* top2 = way[endTop]; //вершина 2
        while(top1 != startTop) {
            for (int i = 0; i < top2->adjacentEdges.size(); i++) {
//проходимся по всем смежным для 2 вершины
                if (top2->adjacentEdges[i]->top == top1) {  //
если нашли ребро из 1 вершины во 2

                    edges.push_back(new std::pair<char,
Edge*>(top2->name, top2->adjacentEdges[i]));    //добавляем ребро
в вектор ребер пути
                    for(int j = 0; j < top1->adjacentEdges.size();
j++){  //поиск и добавление обратных ребер в вектор
                        if(top2->name == top1->adjacentEdges[j]-
>top->name)
                            reversedEdges.push_back(new
std::pair<char, Edge*>(top1->name, top1->adjacentEdges[j]));  //д
                    }
                    if (top2->adjacentEdges[i]->weight < min) {
//сравниваем вес ребра с минимальным в пути
                        min = top2->adjacentEdges[i]->weight;
//если нашли меньшее
                    }
                }
```

```
            }
            top1 = top2; //переходи к след. ребру пути
            top2 = way[top1];
            foundedWay += top1->name;
        }
        std::reverse(foundedWay.begin(), foundedWay.end());
        std::cout <<"Найденный путь: " << foundedWay << std::endl;
        std::cout <<"Поток в пути: " << min << std::endl;
        for (auto & edge : edges) {  //уменьшаем веса ребер пути
            std::cout << "Меняем вес ребра: (" << edge->first <<
"," << edge->second->top->name << ") " << edge->second->weight <<
" -> ";
            edge->second->weight -= min;
            std::cout << edge->second->weight << std::endl;
            edge->second->factFlow += min;  // увеличиваем
максимальный поток через это ребро графа
            if (edge->second->weight == 0){  //если вес ребра
равен нулю, то удаляем
                for (auto & top : tops) {
                    if(top->name == edge->first) {
                        for (auto j = top->adjacentEdges.begin();
j < top->adjacentEdges.end(); j++) {
                            if(edge->second == *j) {
                                resultEdgesVec.push_back(new
ResultEdges(edge->first, edge->second->top->name, edge->second-
>factFlow)); //добавляем удаленное в ответ
                                top->adjacentEdges.erase(j);
                                std::cout << "\tВес ребра равен
нулю, удаляем его из графа" << std::endl;
                            }
                        }
                    }
                }
            }
        }
        for (auto & edge : reversedEdges) {  //увеличиваем вес
обратного ребра
            std::cout << "Меняем вес обратного ребра: (" << edge-
>first << "," << edge->second->top->name << ") " << edge->second-
>weight << " -> ";
            edge->second->weight += min;
            std::cout << edge->second->weight << std::endl;
        }
        maxFlow += min; //увеличиваем поток через граф
        std::cout << "Увеличиваем поток графа на " << min <<
std::endl;

    }

    bool dfs(char start, char end) {
        std::cout << "Запуск поиск пути" << std::endl;
        std::map<GraphTop*, GraphTop*> way; //путь после поиска в
глубину
```

```
        makeAllTopsNotPassed(); //делаем все вершины не
пройденными
        std::stack<GraphTop*> stackForDfs; //стек для поиска в
глубину
        GraphTop* startTop = addOrReturnTop(start);  //кладем
стартовую на стек
        stackForDfs.push(startTop);
        GraphTop* endTop = addOrReturnTop(end);

        while(!stackForDfs.empty()){ //пока стек не пустой
            GraphTop* curTop = stackForDfs.top();  //делаем
текущую вершиной стека
            std::cout << "Обрабатываемая вершина: " << curTop ->
name<< std::endl;
            stackForDfs.pop(); //снимаем вершину со стека
            for (int i = 0; i < curTop->adjacentEdges.size(); ++i)
{ //проходим по всем смежным для текущей
                std::cout<< "\t" << "Проверяем смежную вершину: "
<< curTop->adjacentEdges[i]->top->name << std::endl;
                if(curTop->adjacentEdges[i] -> top == endTop &&
curTop->adjacentEdges[i]->weight != 0) {  //если путь до конечной
не нулевой
                    std::cout << "\t" << "Вершина является
конечной, заканчиваем поиск" << std::endl;
                    endTop->wasPassed = true; //делаем кон
пройденной
                    stackForDfs.push(endTop); //пушим на стек
                    way[endTop] = curTop; //добавляем ее в путь
                    changeEdgesWeights(way, startTop, endTop);
//делаем перерасчет весов
                    return true; //возвращаем что путь был найден
                }
                if(curTop->adjacentEdges[i]->weight != 0 &&
!curTop->adjacentEdges[i]->top->wasPassed) {  //если не пройдена
вершина и вес пути не нулевой
                    std::cout << "\tДобавляем на стек" <<
std::endl;
                    curTop->adjacentEdges[i]->top->wasPassed =
true;  //делаем пройденной
                    stackForDfs.push(curTop->adjacentEdges[i]-
>top); //кладем на стек

                    way[curTop->adjacentEdges[i] -> top] = curTop;
//добавляем в путь
                }
                else std::cout << "\tПройдена либо вес равен нулю"
<< std::endl;
            }
        }
        std::cout << "Путей нет" << std::endl;
        return false; //если не нашил вершину
    };
};
```

```
int main() {
    SetConsoleOutputCP(CP_UTF8);
    EdgesList edgesList;  //создаем граф
    char source, stock, curStart, curEnd;
    int numOrientedEdges, curWeight;
    std::cin >> numOrientedEdges >> source >> stock; //считываем
начальную вершину, конечную и кол-во ребер
    for (int i = 0; i < numOrientedEdges; ++i) {
        std::cin >> curStart >> curEnd >> curWeight;
//счиываем ребро
        edgesList.addEdge(curStart, curEnd, curWeight);
//добавляем в вектор
    }
    while(edgesList.dfs(source, stock)){} //производим поиск пока
есть пути
    std::cout <<
"_____________________________________________________" << std::endl;
    std::cout << "Результат работы алгоритма: " << std::endl;
    std::cout <<edgesList.maxFlow << std::endl; //вывод
максимального потока
    edgesList.sortTopsForAnswer(); //сортируем и печатаем
результат
    edgesList.printResult();
    return 0;
}
```