

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

ОТЧЕТ
по практической работе №1
по дисциплине «Построение и Анализ Алгоритмов»
Тема: Поиск с возвратом

Студент гр. 8382

Вербин К.М.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Изучение алгоритма Форда-Фалкерсона для поиска максимальной пропускной способности сети.

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

$v_i v_j \omega_{ij}$ - 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

...

Пример входных данных

7

a

f

a b 7

a c 6

b d 6

c f 9

d e 3

d f 4

e c 2

Соответствующие выходные данные

12

a b 6

a c 6

b d 6

c f 8

d e 2

d f 4

e c 2

Вариант 4

Поиск в глубину. Итеративная реализация.

Описание структур данных

Класс **Edge** является контейнером ребер вершин.

- backEdge – контейнер для хранения имен вершин, из которых идет поток в данную вершину.
- forwardEdge – контейнер для хранения прямых ребер (названия следующей вершины и величин максимального и текущего потоков через рассматриваемое ребро)

Класс ***Graph*** описывает структуру графа и основные методы для работы с ним.

- start и end – исток и сток графа
 - graph – контейнер описывающий структуру графа.
 - way – контейнер для хранения текущего пути в графе
 - backNeighborList – контейнер для хранения множества вершин, которые добавлялись в путь при обратном ходе
- Graph() – конструктор класса, внутри которого задается исток и сток графа, а так же множество ребер и их максимальные пропускные способности.
 - bool dfs() – метод поиска в глубину. Возвращает true, если путь от истока в сток найден.
 - int searchFlow() – метод поиска максимального потока в графе с помощью алгоритма Форда-Фалкерсона. Возвращает значение максимального потока в графе
 - void print() – метод печати максимальной пропускной способности, ребер и потоков, проходящих через них.
 - void printPatch() – метод печати текущего пути в графе.

Описание алгоритма.

Строится изначальная остаточная сеть, в которой каждый поток равен нулю, следовательно, максимальный поток из истока в сток равен нулю.

Алгоритм проверяет с помощью поиска в глубину, есть ли путь из стока в сток. Выбор следующей вершины в пути выбирается в алфавитном порядке, для добавления в путь рассматриваются ребра, у которых прямые потоки не максимальны, а обратные потоки не пусты. Если путь найден, то есть поиск в глубину дошел до стока графа, то в данном пути ищется ребро с наименьшей пропускной способностью. При нахождении значения этого потока, всем прямым ребрам в пути прибавляется этот поток. Если в пути встречается обратное ребро, то из пропускной способности этого ребра вычитается найденная наименьшая пропускная способность. Величина найденного потока прибавляется к текущей пропускной способности всего графа. Далее путь

откатывается назад до вершины, из которой выходит ребро с наименьшей пропускной способностью и запускается новый поиск в глубину. Алгоритм работает до тех пор, пока есть путь из истока в сток.

Сложность алгоритма по памяти

Сложность алгоритма по памяти $O(V+E)$, где V – количество вершин в графе, E – количество ребер в графе

Временная сложность алгоритма

Временная сложность алгоритма $O(Ef)$, где E – количество ребер, f – максимальный поток в графе.

Тестирование

Input	Output
12	10
a	a b 5
e	a c 4
a b 5	a d 1
a c 4	b c 0
a d 2	b f 3
b c 1	b g 2
b g 5	c b 0
b f 3	c f 1
c g 4	c g 3
c f 1	d c 0
d c 2	d f 1
d f 1	f c 0
g e 5	f e 5
f e 8	g c 0

Вывод

В ходе выполнения лабораторной работы были получены знания для работы с алгоритмом Форда-Фалкерсона для поиска максимального потока в графе. Были реализованы классы ребер и графа. Так же в классе графа реализованы методы поиска в глубину и алгоритма Форда-Фалкерсона

ПРИЛОЖЕНИЕ

КОД ПРОГРАММЫ

```
#include <iostream>
#include <vector>
#include <map>

#define debug

class Edge
{
public:
    std::vector<char> backEdge;           // -контейнер для хранения
    обратных путей
    std::map<char, std::pair<int, int>> forwardEdge; // -контейнер для
    хранения множества прямых путей и их пропускных способностей
};
class Graph
{
    char start, end;                     // -исток и сток графа
    std::map<char, Edge> graph;           // -контейнер графа
    std::vector<char> way;                // -найденный путь
    std::vector<char> backNeighborList;    // -контейнер для
    хранения множества вершин, которые добавлялись в
public:                                   // путь при обратном проходе
    Graph()
    {
        int edgeNumber;
        std::cin >> edgeNumber;
        std::cin >> start;
        std::cin >> end;
        for(int i = 0; i < edgeNumber; i++)
        {
            int distance;
            char from, where;
            std::cin >> from >> where >> distance;
            graph[from].forwardEdge[where].first = distance;
            graph[from].forwardEdge[where].second = 0;
        }
    }

    bool dfs()                           // -метод поиска в глубину
    {
        std::vector<char> closed;
```

```

#ifdef debug
    std::cout << "-----" << std::endl;
    std::cout << "searching way" << std::endl;
#endif
    while (!way.empty()) {
        char currentTop = way.back();
        if(currentTop == end)                // -если на очередной итерации
        был добавлен сток, то путь найден
            return true;
        bool flag = false;
        bool backNeighborFounded = false;
                                           // ищем прямые пути
#ifdef debug
        std::cout << "forward available ways:" << std::endl;
#endif
        for (const auto &iter : graph[currentTop].forwardEdge) {
            bool inWay = false, inClosed = false;
            for (auto way_iter: way) {        // -проверка на наличии
            рассматриваемой вершины в пути
                if (way_iter == iter.first) {
                    inWay = true;
                    break;
                }
            }
            for(auto closed_iter: closed)      // -проверка на наличии
            рассматриваемой вершины в множестве
            if(closed_iter == iter.first)    // просмотренных вершин
            {
                inClosed = true;
                break;
            }
            if(inClosed || inWay)
                continue;

            if (iter.second.first - iter.second.second) {
#ifdef debug
                std::cout << currentTop << "->" << iter.first << " "
                << iter.second.first << "/" << iter.second.second << std::endl;
#endif
                currentTop = iter.first;
                flag = true;
                break;
            }
        }
    }

```



```

// -рассматриваем множество обратных ребер, которые не входят
в путь
#ifdef debug
    std::cout <<"backward available way"<<std::endl;
#endif
    for (auto backNeighbor: graph[way.back()].backEdge) { // -
аналогично прямым путям, проверяем
        bool inWay = false, inClosed = false;           // есть ли данное
ребро в пути и в множестве
        for (auto way_iter: way) {                       // просмотренных
вершин
            if (way_iter == backNeighbor) {
                inWay = true;
                break;
            }
        }
        for(auto closed_iter: closed)
            if(closed_iter == backNeighbor)
            {
                inClosed = true;
                break;
            }
        if(inClosed || inWay)
            continue;
        if (graph[backNeighbor].forwardEdge[way.back()].second) {
#ifdef debug
            std::cout <<way.back() << "->" <<backNeighbor << " " <<
graph[backNeighbor].forwardEdge[way.back()].second <<std::endl;
#endif
            currentTop = backNeighbor;
            flag = true;
            backNeighborFounded = true;
            break;
        }
    }

    if (flag) {                                           // -Если путь найден, то
заносим его в контейнер и
                                                         // Так же производим проверку,
что эта найденная
                                                         // связана с предыдущей
обратным ребром
#ifdef debug
        std::cout <<"next top: " << currentTop <<std::endl;
#endif
#endif

```

```

way.push_back(currentTop);
if (backNeighborFounded)
{
    bool inList = false;
    for(auto list_iter : backNeighborList)
        if(list_iter == currentTop)
        {
            inList = true;
            break;
        }
    if(!inList)
        backNeighborList.push_back(currentTop);
}
} else {
    closed.push_back(way.back());           // -Если путь не
найден, то помечаем вершину как
    for(auto backNeighbor: backNeighborList)
    {
        if(backNeighbor == way.back())
        {
            backNeighborList.pop_back();
            break;
        }
    }
}

// просмотренную и удаляем ее
из пути
#ifdef debug
    std::cout << "no way from: " << way.back() << std::endl;
    std::cout << "go back to prev top\n" << std::endl;
    std::cout << "-----" << std::endl;
#endif
way.pop_back();
}

}
return false;

}

int searchFlow()
{
    int currentFlow = 0;
    way.push_back(start);

    //пока есть путь до стока и ребра исходящие из истока

```

```

        //находим ребро
        //добавляем его в путь. Пока не дошли до стока, ищем новое
        наибольшее ребро, которое не входит в путь.
        // Дойдя до стока, ищем наименьшее значение потока в текущем
        пути.
        // Откатываемся назад, до вершины, которая находится перед
        ребром с наименьшей пропускной способностью
        while(dfs())
        {

#ifdef debug
            std::cout << "founded ";
            printPatch();
            std::cout << "start search bottle neck " << std::endl;
#endif
            int minFlow = INT32_MAX;
            char minTop = end;

            for(auto way_iter = way.begin(); way_iter != way.end() - 1;
way_iter++) // -поиск минимального потока
            {
                bool inList = false;
                for(auto list_iter : backNeighborList)
                    if(list_iter == *(way_iter+1) )
                    {
                        inList = true;
                        break;
                    }
                if(!inList)
                {
                    for(const auto& next_iter : graph[*way_iter].forwardEdge)
                    {

                        if(next_iter.first == *(way_iter + 1) && minFlow >
next_iter.second.first - next_iter.second.second)
                        {
                            minFlow = next_iter.second.first - next_iter.second.second;
#ifdef debug
                                std::cout << "bottle neck: " << minFlow << std::endl;
#endif
                        }
                    }
                }
            }
            else
            {

```

```

        for(const auto& next_iter : graph[*way_iter +
1)].forwardEdge)
        {
            if(next_iter.first == *way_iter && minFlow >
next_iter.second.second)
            {
                minFlow = next_iter.second.second;
#ifdef debug
                std::cout << "bottle neck: " << minFlow << std::endl;
#endif
            }
        }

        }
        currentFlow += minFlow;
#ifdef debug
        std::cout << "current flow: " << currentFlow << std::endl;
#endif
        bool flag = true;
        bool backTop = true;
        for(auto way_iter = way.begin(); way_iter != way.end() - 1;
way_iter++) // -заполняем путь потоком
        {
            bool inList = false;
            for(auto list_iter : backNeighborList)
                if(list_iter == *(way_iter+1) )
                {
                    inList = true;
                    break;
                }
            if(!inList)
                to прибавляем поток // -если путь прямой,
                {
                    graph[*way_iter].forwardEdge[*way_iter + 1].second +=
minFlow;
#ifdef debug
                    std::cout << "flow of " << *way_iter<< "->" << *(way_iter+1)
<< " = " << graph[*way_iter].forwardEdge[*way_iter + 1].second << std::endl;
#endif
                    bool hasThatNeighbor = false;
                    for(auto& backNeighbor: graph[*way_iter + 1].backEdge)
                        // -проверка на наличие

```

```

        {
        множество обратных путей
            if(*way_iter == backNeighbor)
            //следующей вершины
            {
                hasThatNeighbor = true;
                break;
            }
        }
        if(!hasThatNeighbor)
            graph[*(way_iter + 1)].backEdge.push_back(*way_iter);

        if(flag && backTop &&
graph[*way_iter].forwardEdge[*(way_iter + 1)].second ==
graph[*way_iter].forwardEdge[*(way_iter + 1)].first)
        {
            #ifdef debug
                std::cout << "bottle neck " << *way_iter<< "->" <<
*(way_iter+1) << std::endl;
            #endif

            minTop = *way_iter;
            flag = false;
        }
        } else
        //из обратного ребра
        ВЫЧИТАЕМ
        {
            //МИНИМАЛЬНЫЙ ПОТОК
            graph[*(way_iter + 1)].forwardEdge[*way_iter].second -=
minFlow;

            if(flag && backTop /*&& graph[*(way_iter +
1)].forwardEdge[*way_iter].second*/)
            {
                minTop = *way_iter;
                backTop = false;

                #ifdef debug
                    std::cout << "bottle neck " << *(way_iter+1)<< "->" <<
*(way_iter) << std::endl;
                #endif
            }

            #ifdef debug
                std::cout << "flow of " << *(way_iter+1)<< "->" << *way_iter
<< " = " << graph[*(way_iter + 1)].forwardEdge[*way_iter].second << std::endl;
            #endif
        }
    }
}
#endif debug

```

```

        std::cout << "go back" << std::endl;
    #endif
        while(way.back() != minTop)                //-откатываемся
назад до вершины,
        {
наименьший поток                                // из которой выходит
        #ifdef debug
            std::cout << "pop " << way.back() << std::endl;
        #endif
            if(!backNeighborList.empty() && way.back() ==
backNeighborList.back())
                backNeighborList.pop_back();
            way.pop_back();
        }

    }
    return currentFlow;
}

void print()                // -метод печати максимального потока и
ребер графа
{
    for(const auto& iter: graph) {
        for (const auto& next_iter: iter.second.forwardEdge)
            std::cout << iter.first << " " << next_iter.first << " " <<
next_iter.second.second << std::endl;

    }
}

void printPatch()           // -метод печати пути
{
    std::cout << "way: ";
    for(auto iter:way)
        std::cout << iter;
    std::cout << std::endl;
}

};

int main() {
    Graph graph;
    std::cout << graph.searchFlow() << std::endl;
    graph.print();
    return 0;
}

```