МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Построение и анализ алгоритмов»

Тема: Потоки в сети

Студент гр.8304 _____ Сани З. Б.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2020

Задание.

Вариант 4
Поиск в глубину. Итеративный метод.

Цель работы.

Разработать программу, которая находит максимальный поток в сети, а также фактическую величину потока, протекающего через каждое ребро, используя алгоритм Форда-Фалкерсона.

Описание алгоритма.

1) Обнуляем все потоки. Остаточная сеть изначально совпадает с исходной сетью.
2) В остаточной сети находим любой путь из источника в сток. Если такого пути нет, останавливаемся.
3) Пускаем через найденный путь (он называется ***увеличивающим путём*** или ***увеличивающей цепью***) максимально возможный поток:
    1) На найденном пути в остаточной сети ищем ребро с минимальной пропускной способностью .
    2) Для каждого ребра на найденном пути увеличиваем поток на , а в противоположном ему — уменьшаем на .
    3) Модифицируем остаточную сеть. Для всех рёбер на найденном пути, а также для противоположных им рёбер, вычисляем новую пропускную способность. Если она стала ненулевой, добавляем ребро к остаточной сети, а если обнулилась, стираем его.
4) Возвращаемся на шаг 2.

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

Описание функций и структур данных.

Граф хранится в словаре `std::map<char, Vertex>`, где `Vertex`– структура, где хранится информация о предыдущем элементе и о том, присутствует ли

данный элемент в пути(Нужно для поиска в глубину итеративным способом) Также в ней хранится информация о вершинах в которые можно попасть из данной и пропускная способность и поток этих путей.

```
struct Vertex {
    std::vector<std::pair<char, VectorInfo>> destinations;
    char parentVertex;
    bool visited;
};

struct VertexInfo{
    int capacity;
    int flow;
};
```

Была определена функция итеративного поиска в глубину `std::string dfsAugmentedPath(std::map<char, Vertex> graph, char source, char sink)`, возвращающая путь, если он был найден и строку, состоящую из символа source, если путь не был найден.

Функция возвращающая минимальную пропускную способность на пути:

```
int minCapacity(std::string augPath, std::map<char, Vertex> residualgraph)
```

Функция изменения потоков и модификации пропускных способностей:

```
void updateResidualGraph(std::string augmentedPath, std::map<char, Vector>&
residualGraph)
```

Тестирование

Таблица 1 – результаты работы

Input	Output
7 a f a b 7 a c 6 b d 6 c f 9 d e 3 d f 4	12 a b 6 a c 6 b d 6 c f 8 d e 2 d f 4 e c 2

e c 2	
5 a d a b 20 b c 20 c d 20 a c 1 b d 1	21 a b 20 a c 1 b c 19 b d 1 c d 20

Выводы.

В ходе выполнения данной работы была написана программа, которая находит максимальный поток в сети, а также фактическую величину потока, протекающего через каждое ребро, используя алгоритм Форда-Фалкерсона.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД

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


struct VertexInfo{
    int capacity;
    int flow;
};

struct Vertex {
    std::vector<std::pair<char, VertexInfo>> destinations;
    bool visited;
    char parentVertex;
};


char selectAvailabevertex(std::vector<std::pair<char, VertexInfo>> const&
destinations,std::map<char, Vertex> graph){
    char availableVertex = 0;

    for(auto i : destinations){
        if(!graph[i.first].visited && i.second.capacity > 0) {
            availableVertex = i.first;
        }
    }
    return availableVertex;
}

std::string dfsAugmentedPath(std::map<char, Vertex> graph, char source, char
sink){
    char currentVertex = source;
    std::string path = {source};

    char selectedVertex =
selectAvailabevertex(graph[currentVertex].destinations, graph);

    //iterative DFS
    while(currentVertex != sink && !(currentVertex == source &&
selectedVertex == 0)){

        graph[currentVertex].visited = true;

        if(selectedVertex != 0) {
            char parentVertex = currentVertex;
            currentVertex = selectedVertex;
            graph[currentVertex].parentVertex = parentVertex;
            path += currentVertex;
        }
        else {
            path.erase(path.end()-1);
            currentVertex = graph[currentVertex].parentVertex;
        }

        selectedVertex =
selectAvailabevertex(graph[currentVertex].destinations, graph);
    }
```

```
    return path;
}


int minCapacity(std::string augPath, std::map<char, Vertex> residualGraph){
    int min = INT_MAX;

    while(augPath.size() > 1){
        for(auto vertex : residualGraph[augPath[0]].destinations)
            if(vertex.first == augPath[1] && vertex.second.capacity < min) {
                min = vertex.second.capacity;
            }
        augPath.erase(augPath.begin());
    }

    return min;
}

void updateResidualGraph(std::string augPath, std::map<char, Vertex>&
residualGraph){
    int min = minCapacity(augPath,residualGraph);

    while(augPath.size() > 1){
        for(auto& vertex : residualGraph[augPath[0]].destinations)
            if(vertex.first == augPath[1]) {
                vertex.second.flow += min;
                vertex.second.capacity -= vertex.second.flow;

                for(auto& j:residualGraph[augPath[1]].destinations) {
                    if(j.first == augPath[0]) {
                        j.second.flow -= min;
                        j.second.capacity -= j.second.flow;
                    }
                }
            }
        augPath.erase(augPath.begin());
    }

}

bool cmp(std::pair<char, VertexInfo> const& a, std::pair<char, VertexInfo>
const& b) {
    return a.first < b.first;
}

void maxFlow(std::map<char, Vertex> graph, char source, char sink){

    std::string augPath = dfsAugmentedPath(graph,source,sink);

    while (augPath != std::string(1,source)) {
        updateResidualGraph(augPath,graph);
        augPath = dfsAugmentedPath(graph,source,sink);
    }

    //max flow going out from source = the flow coming to sink
    int maxFlowOut = 0;
    for(auto i : graph[source].destinations){
        maxFlowOut += i.second.flow;
    }
    std::cout << maxFlowOut <<std::endl;

    for(auto vertex : graph) {
```

```
std::sort(vertex.second.destinations.begin(),vertex.second.destinations.end()
,cmp);
        for(auto dest : vertex.second.destinations)
            std::cout << vertex.first << " " << dest.first << " " <<
std::max(0,dest.second.flow) << std::endl;
     }

}


int main(){

    int N;
    char source,sink;
    std::map<char, Vertex> residualGraph;


    std::cin >> N;
    std::cin >>source;
    std::cin>> sink;
    char from,to;
    int capacity;
    for(int i = 0; i < N; ++i) {
        int flow = 0;
        std::cin >> from >> to >> capacity;
        residualGraph[from].destinations.push_back({to,{capacity,flow}});
    }

    maxFlow(residualGraph,source,sink);


    return 0;
}
```