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 сети

Студент гр. 9383 _____ Орлов Д.С.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Изучить и реализовать алгоритм Форда-Фалкерсона для поиска максимального потока в сети.

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 имен вершин и целого неотрицательного числа - пропускной способности (веса).

Входные данные:

N- количество ориентированных рёбер графа

*v*0 - исток

vn - сток

vi vj ωij - ребро графа

...

Выходные данные:

Pmax - величина максимального потока

vi vj ωij - 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).

Вариант 6.

Поиск не в глубину и не в ширину, а по правилу: каждый раз

выполняется переход по дуге, соединяющей вершины, имена которых в алфавите ближе всего друг к другу. Если таких дуг несколько, то выбрать ту, имя конца которой в алфавите ближайшее к началу алфавита.

Описание алгоритма.

Алгоритм Форда-Фалкерсона запускает поиск в глубину до тех пор, пока путь возможно найти. После находит ребро с минимальной пропускной способностью и уменьшает пропускную способность всех рёбер, содержащихся в этом пути.

Сложность алгоритма в худшем случае равна:

$O(F*|E|)$, где F - максимальный поток, E - множество дуг

Описание структур и функций.

В программе реализована структура Vertex для хранения информации о вершинах графа, содержащая:
Переменные:
Vertex* parent — указатель на предыдущую вершину в пути, char name — имя вершины, bool visited — посещали ли мы эту вершину, int specification — тип вершины (0 – исток, 2 – сток, 1 – просто вершины), int minC — минимальный вес ребер, встречающихся до данной вершины, std::vector <std::tuple<Vertex*, int, int, bool>> edge — хранит множество ребер в формате: <вершина, куда идет ребро; начальный величина потока в ребре; фактическая величина потока в ребре; флаг, показывающий есть ли данное ребро в исходном графе>

Методы:
void makeEdge(Vertex* v, int cost, bool flag) — добавляет ребро между данной вершиной и вершиной v с величиной потока cost и флагом flag.

В программе были использованы следующие функции:

Vertex* castomFind(std::vector <Vertex*> graph, char v) — для поиска вершины в графе по имени;

Vertex* addVertex (std::vector <Vertex*> &graph, char v) — для добавления вершины в граф.

int getFlag(Vertex* v, int i) — для получения информации о существовании ребра в исходном графе.

int getC(Vertex* v, int i) — для получения исходной величины потока ребра.

int getF(Vertex* v, int i) — для получения фактической величины потока ребра.

Vertex* getNextVertex(Vertex* v, int i) — для получения конечной вершины ребра.

int calcDist(Vertex* v1, Vertex* v2) — для подсчета расстояния между двумя символами в алфавите.

Vertex* findNextVertex(std::vector <Vertex*> Q, Vertex* u) — для выбора следующей вершины при обходе графа.

int findInEdge(Vertex* v, char c) — для поиска определенного ребра.

int getFactCost(Vertex* v, int i) — для подсчета фактической величины потока ребра.

void setTrue(std::vector <Vertex*> &graph) — помечает все вершины графа, как не посещенные.

void killParent(std::vector <Vertex*> &graph) — обнуление пути в графе.

int comp(const void* v1, const void* v2) — компаратор для сортировки вершин в графе.

int comp2(const void* a, const void* b) — компаратор для сортировки списка ребер у вершины.

void printGraph(std::vector <Vertex*> graph) — выводит список ребер графа с фактическими величинами потока.

int FFA(Vertex* u, std::vector <Vertex*> &graph) — реализация алгоритма Форда-Фалкерсона.

Тестирование.

Таблица 1 - результаты тестирования

Тест	Входные данные	результат работы алгоритма
№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	9 a d a b 8 b c 10 c d 10 h c 10 e f 8 g h 11 b e 8 a g 10 f d 8	18 a b 8 a g 10 b c 0 b e 8 c d 10 e f 8 f d 8 g h 10 h c 10
№3	16 a e a b 20 b a 20 a d 10 d a 10 a c 30 c a 30 b c 40	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 c d 0

	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 e 20 d a 0 d c 0 d e 10 e b 0 e c 0 e d 0

Вывод.

В ходе работы было изучено понятие потока, а также алгоритм Форда-Фалкерсона для поиска максимального потока в сети. Алгоритм был рекурсивно реализован с использованием поиска по правилу: каждый раз выполняется переход по дуге, соединяющей вершины, имена которых в алфавите ближе всего друг к другу. Если таких дуг несколько, то выбрать ту, имя конца которой в алфавите ближайшее к началу алфавита. Использование такого поиска никак не повлияло на правильность и скорость работы алгоритма.

ПРИЛОЖЕНИЕ А
Исходный код программы

```
#include <iostream>
#include <vector>
#include <string>
#include <tuple>

#define INF 1000000

struct Vertex
{
public:
    Vertex* parent;
    char name;
    bool visited;
    int specification, minC;
    std::vector <std::tuple<Vertex*, int, int, bool>> edge;

    Vertex(char nameV, int spec = 1)
    {
        name = nameV;
        specification = spec;
        visited = false;
        parent = nullptr;
        minC = 0;
    }

    void makeEdge(Vertex* v, int cost, bool flag)
    {
        edge.push_back(std::make_tuple(v, cost, cost, flag));
    }
};

Vertex* castomFind(std::vector <Vertex*> graph, char v)
{
```

```
    for(int i = 0; i < graph.size(); i++)
    {
        if(graph[i]->name == v)
        {
            return graph[i];
        }
    }
    return nullptr;
}

Vertex* addVertex (std::vector <Vertex*> &graph, char v)
{
    Vertex* ptr = castomFind(graph, v);
    Vertex* ver;
    if(ptr == nullptr)
    {
        ver = new Vertex(v);
        graph.push_back(ver);
        return graph.back();
    }
    return ptr;
}

int getFlag(Vertex* v, int i)
{
    return std::get<3>(v->edge[i]);
}

int getC(Vertex* v, int i)
{
    return std::get<1>(v->edge[i]);
}

int getF(Vertex* v, int i)
{
```

```
    return std::get<2>(v->edge[i]);
}

Vertex* getNextVertex(Vertex* v, int i)
{
    return std::get<0>(v->edge[i]);
}

int calcDist(Vertex* v1, Vertex* v2)
{
    return abs(v1->name - v2->name);
}

Vertex* findNextVertex(std::vector <Vertex*> Q, Vertex* u)
{
    int dist = INF, index = -1;

    for(int i = 0; i < Q.size(); i++)
    {
        if(calcDist(u, Q[i]) <= dist)
        {
            if(calcDist(u, Q[i]) == dist)
            {
                if(index == -1)
                {
                    index = i;
                }
                else
                {
                    if(Q[i]->name < Q[index]->name)
                    {
                        index = i;
                    }
                    else
                    {
```

```
                        continue;
                    }
                }
            }
            else
            {
                index = i;
                dist = calcDist(u, Q[i]);
            }
        }
    }
    return Q[index];
}

int findInEdge(Vertex* v, char c)
{
    for(int i = 0; i < v->edge.size(); i++)
    {
        if(getNextVertex(v,i)->name == c)
        {
            return i;
        }
    }
    return -1;
}

int getFactCost(Vertex* v, int i)
{
    int a = getC(v,i) - getF(v,i);
    Vertex* ptr = getNextVertex(v,i);
    if(a > 0)
    {
        return a;
    }
    return 0;
```

```
}


void setTrue(std::vector <Vertex*> &graph)
{
    for(auto i : graph)
    {
        i->visited = false;
    }
}

void killParent(std::vector <Vertex*> &graph)
{
    for(auto i : graph)
    {
        i->parent = nullptr;
        i->minC = 0;
    }
}

int comp(const void* v1, const void* v2)
{
    return ((*(Vertex**)v1)->name - (*(Vertex**)v2)->name);
}

int comp2(const void* a, const void* b)
{
    return (std::get<0>(*(std::tuple<Vertex*, int, int,
bool>*)a)->name - std::get<0>(*(std::tuple<Vertex*, int, int,
bool>*)b)->name);
}

void printGraph(std::vector <Vertex*> graph)
{
    for(auto i : graph)
    {
```

```
        if(i->edge.size() != 0)
        {
            std::qsort(&(i->edge[0]), i->edge.size(),
sizeof(std::tuple<Vertex*, int, int, bool>), comp2);
            for(int j = 0; j < i->edge.size(); j++)
            {
                if(getFlag(i,j) == true)
                {
                    std::cout<<i->name<<" "<<getNextVertex(i,
j)->name<<" "<<getFactCost(i, j)<<"\n";
                }
            }
        }

    }
}

int FFA(Vertex* u, std::vector <Vertex*> &graph)
{
    std::vector <Vertex*> Q;
    int delta = INF;

    while(u->specification != 2)
    {
        u->visited = true;
        u->minC = delta;

        for(int i = 0; i < u->edge.size(); i++)
        {
            if(getNextVertex(u,i)->visited == false && getF(u,i) > 0)
            {
                Q.push_back(getNextVertex(u,i));
            }
        }
```

```
    if(Q.size() == 0)
    {
        if(u->specification == 0)
        {
            return 0;
        }
        else
        {
            u = u->parent;
            delta = u->minC;
            continue;
        }
    }

    Vertex* ptr = findNextVertex(Q, u);

    ptr->parent = u;

    int c = getF(u, findInEdge(u, ptr->name));

    if(c < delta)
    {
        delta = c;
    }

    u = ptr;
    Q.clear();
}

char c;
while(u->specification != 0)
{
    c = u->parent->name;
    std::get<2>(u->edge[findInEdge(u,c)]) += delta;
    c = u->name;
```

```
        u = u->parent;
        std::get<2>(u->edge[findInEdge(u,c)]) -= delta;
    }

    return delta;
}

int main()
{
    int n, cost;
    char source, sink, first, second;
    std::vector <Vertex*> graph;
    std::cin>>n;
    std::cin>>source>>sink;
    int index;
    for(int i = 0; i < n; i++)
    {
        std::cin>>first>>second>>cost;

        Vertex* ptr1 = addVertex(graph, first);
        Vertex* ptr2 = addVertex(graph, second);

        if((index = findInEdge(ptr1, ptr2->name)) == -1)
        {
            ptr1->makeEdge(ptr2, cost, true);
        }
        else
        {
            std::get<1>(ptr1->edge[index]) = cost;
            std::get<2>(ptr1->edge[index]) = cost;
            std::get<3>(ptr1->edge[index]) = true;
        }

        if((index = findInEdge(ptr2, ptr1->name)) == -1)
        {
```

```
            ptr2->makeEdge(ptr1, 0, false);
        }

    }

    Vertex* ptr = castomFind(graph, source);
    ptr->specification = 0;

    ptr = castomFind(graph, sink);
    ptr->specification = 2;

    Vertex* start = castomFind(graph, source);
    int maxFlow = 0;
    int result;

    while((result = FFA(start,graph)) > 0)
    {
        maxFlow += result;
        setTrue(graph);
        killParent(graph);
    }

    std::cout<<maxFlow<<"\n";


    std::qsort(&graph[0], graph.size(), sizeof(Vertex*), comp);

    printGraph(graph);

    for(int i = 0; i < graph.size(); i++)
    {
        delete graph[i];
    }

    return 0;
```

```
}
```