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. 0382 _____ Бочаров Г.С.

Преподаватель _____ Шевская Н.В.

Санкт-Петербург
2022

Цель работы.

Написать программу, реализовывающую нахождение максимального потока в сети, а также фактическую величину потока, протекающего через каждое ребро, используя алгоритм Форда-Фалкерсона.

Задание.

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

Вариант 4:

Поиск в глубину. Итеративная реализация.

Выполнение работы.

Для хранения графа было решено использовать словарь, где ключом является имя вершина, а значением массив выходяих из нее ребер, массив вершин, чтобы отслеживать их посещение алгоритмом, словарь для обратных ребер и массив всех ребер графа. В массиве ребер в первой его половине хранятся прямые ребра, во второй обратные

В ходе работы были написаны следующие функции и классы:

edge — структура для хранения ребра. Содержит в себе название конечной и начальной вершины (from, to), номер ребра противоположного данному (inv_edge), вес ребра (cost) и поток через это ребро (flow).

read_graph — функция считывает граф из потока ввода всю необходимую для работы алгоритма информацию и заносит ее в соответствующие структуры данных.

path_exists — функция, которая ищет путь от истока в сток. В начале работы алгоритма создается множество, куда будут записаны пройденые вершины (used) и словарь (path) , где ключом является название вершины, а значение номер ребра, по которому мы пришли в данную вершину. Также создается очередь, которая определяет порядок прохождения вершин.

Алгоритм продолжает работу до тех пор, пока не останется вершин для посещения или пока не найдет путь из истока в сток.

На каждом шаге из очереди берется верхняя вершина и ищется ребро из данной вершины в соседние не посещенные, сначала по прямым ребрам потом по обратным. Если такое ребро найдено, то оно записывается в путь (path), а соседи вершины в которую пришел алгоритм заносятся в очередь для посещения.

Если таким образом мы приходим в конечную вершину, значит путь найден и функция вернет истинное значение.

Main — в функции запускаеся функция path_exists до тех пор пока находится путь из истока в сток.

Если алгоритм нашел путь, то поток через соответствующие ребра меняется на величину наименьшего по весу ребра в этом пути, меняется так же вес соответствующих ребер.

Результаты тестирования представлены в приложении Б.

Выводы.

Изучен алгоритм Форда – Фалкерсона, найден с его помощью максимальный поток в сети, а также фактическая величина потока, протекающего через каждое ребро.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: surce.cpp

```
#include <iostream>
#include <set>
#include <vector>
#include <map>
#include <list>
#include <queue>
#include <sstream>
#include <algorithm>
class Grapth;
#define MAX_INT  2147483647;



struct edge {
     char from, to;
     int inv_edge;
     int cost;
     int flow;
};

void print_edge_1(edge e)
{
     std::cout << " from : " << e.from << " to : " << e.to << "
cost : " << e.cost << " flow : " << e.flow << std::endl;

}
void print_edge(edge e)
{
     std::cout  <<  e.from  <<  "  "  <<  e.to  <<  "  "  <<  e.flow  <<
std::endl;

}

void print_edges_1(std::vector<edge>& edges)
{
     for (auto i : edges)
     {
          print_edge_1(i);
     }
}
void print_edges(std::vector<edge>& edges)
{
     for (int i = 0 ; i <edges.size(); i++)
     {
          print_edge(edges[i]);
     }
}
```

```
void print_path(std::map<char, int> path, std::vector<edge>&
edges)
{
     for (auto i : path)
     {
          //std::cout << "from : " << i.first << " to : " <<
i.second << std::endl;
          if(i.second >= 0)
               std::cout << "to : " << i.first << " from : " <<
edges[i.second].from << " flow : " << edges[i.second].flow <<
std::endl;
          else
               std::cout << "to : " << i.first << " from : " <<
i.second << std::endl;
     }
}


void print_augmenting_path(std::vector<int>& augmenting_path,
std::vector<edge>& edges)
{
     for (auto i : augmenting_path)
     {

          std::cout<< " from : "<< edges[i].from << " to : " <<
edges[i].to << " cost : " << edges[i].cost << " flow : " <<
edges[i].flow<< std::endl;
     }
}


struct cmp {
     bool operator()(char a, char b) const
     {
          return a > b;
     }
} ;



void print_a(std::map<char, std::vector< int > >& a,
std::vector<edge>& edges)
{
     for (auto i : a)
     {
          std::cout << std::endl;
          std::cout << "v : " << i.first << " vert : ";
          for(auto j : i.second)
               std::cout << edges[j].to << "";
     }
}
```

```
bool comp(char c1, char c2) {
     return c1 < c2;
}

void read_graph(char goal, std::map<char, std::vector< int > > &a,
std::map<char, std::vector< int > >& ia, std::vector<edge>& edges,
std::vector<int>& stock_edges, int m)
{
     char from;
     char to;
     int cap;

     for (int i = 0; i < m; i++)
     {
          std::cin >> from >> to >> cap;

          if (a.count(from))
               a[from].push_back(i); // исходящие ребра
          else
               a.insert({ from, {i} });


          if (ia.count(to))
               ia[to].push_back(i + m); // входящие ребра
          else
               ia.insert({ to, {i + m} });



          // a в ребрах - вся информация
          // от 0 до m-1 - прямые ребра, от m до 2m-1 - обратные
          //if (from == goal)
               //cap = 0;
          edges[i].from = from;
          edges[i].to = to;
          edges[i].inv_edge = m + i;
          edges[i].cost = cap;
          edges[m + i].from = to;
          edges[m + i].to = from;
          edges[m + i].inv_edge = i;
          // запоминаем ребра, выходящие из стока
          if (from == goal) {
               stock_edges.push_back(i);
          }
     }


     for (auto &i : a)
     {
          for (int j = 0; j < i.second.size(); j++)
               for (int k = 0; k < i.second.size(); k++)
               {
                    if (edges[j].to > edges[k].to)
```

```
                    {
                         //std::cout                                   <<
"ffgggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggg
gggggggggggggg";
                         auto t = i.second[k];
                         i.second[k] = i.second[j];
                         i.second[j] = t;
                    }
               }
          //std::sort(i.second.begin(), i.second.end(), comp);
     }



     for (auto& i : ia)
     {
          for (int j = 0; j < i.second.size(); j++)
               for (int k = 0; k < i.second.size(); k++)
               {
                    if (edges[j].to > edges[k].to)
                    {
                         //std::cout                                   <<
"ffgggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggggg
gggggggggggggg";
                         auto t = i.second[k];
                         i.second[k] = i.second[j];
                         i.second[j] = t;
                    }
               }
          //std::sort(i.second.begin(), i.second.end(), comp);
     }

}



bool path_exists(char s, char t, std::map<char, std::vector< int >
>& a, std::map<char, std::vector< int > >& ia, std::vector<edge>&
edges, std::vector<int>& augmenting_path,int* cmin)
{
     //std::cout << "dd" << std::endl;
     std::set<char> used;
     // тут сохраняем номер ребра, по которому мы пришли в вершину
     std::map<char, int> path;

     std::list<char> q;
     q.push_back(s);

     used.insert(s);
     path.insert({s, -1}); // в первую не приходим ниоткуда

     bool found = false;
     char from;
```

```
    edge e;
    while (!q.empty() && !found) {

        from = q.back();
        q.pop_back();

        for (int j = 0; j < a[from].size() && !found; ++j) {

            e = edges[a[from][j]];
            if (!used.count(e.to) && (e.cost - e.flow) > 0) {
                q.push_back(e.to);

                used.insert(e.to);


                if(path.count(e.to))
                    path[e.to] = a[from][j];
                else
                    path.insert({ e.to , a[from][j] });

                // если достигли сток, то можно выходить
                found = e.to == t;
                //std::cout  <<  "Findddd  ->  :  "  <<  e.to  <<
std::endl;
            }
        }

        for (size_t j = 0; j < ia[from].size() && !found; ++j) {

            e = edges[ia[from][j]];

            if (!used.count(e.to) && (e.cost - e.flow) > 0) {
                q.push_back(e.to);
                used.insert(e.to);
                if (path.count(e.to))
                    path[e.to] = ia[from][j];
                else
                    path.insert({ e.to , ia[from][j] });
                // если достигли сток, то можно выходить
                found = e.to == t;
                //std::cout  <<  "Findddd  <-  :  "  <<  e.to  <<
std::endl;
            }
        }
    }

    if (!found) {
        return false;
    }
    //std::cout << "Findddd : " << e.to << std::endl;
    //std::cout << "path = " << path.size() << std::endl;
    //print_path(path, edges);
    augmenting_path.clear();
```

```
	// начинаем с конца
	*cmin = MAX_INT;
	//пока не достигли начала

	char cur = t;
	while (cur != s)
	{
		int i = path[cur];
		augmenting_path.push_back(i);   //   заносим   ребро,   по
которому пришли сюда
		e = edges[i];
		//  минимальный  добавляемый  поток  -  самый  слабый  из
потоков в пути
		if (*cmin > e.cost - e.flow) {
			*cmin = e.cost - e.flow;
		}
		cur = edges[i].from;
	}

	return true;


}

struct {
	bool operator()(edge a, edge b) const
	{
		if(a.from == b.from)
			return a.to < b.to;
		return a.from < b.from;
	}
} customLess;



int main()
{
	std::map<char, std::vector< int> > a;
	std::map<char, std::vector< int> > ia;
	std::vector<int> stock_edges;
	std::vector<edge> edges;

	std::vector<int> augmenting_path;


	int size;
	char start;
	char end;



	std::cin >> size;
	std::cin >> start >> end;
```

```
    std::cin.ignore();

    edges.resize(size*2);

    read_graph(end, a, ia, edges, stock_edges, size);

    int t = a.size() + ia.size();
    augmenting_path.reserve(t);



    //print_edges(edges);


    int flow = 0;
    int cmin;
    while (path_exists(start, end, a, ia, edges, augmenting_path,
&cmin)) {
        flow += cmin;
        //print_augmenting_path(augmenting_path, edges);
        //std::cout << "cmin = " << cmin << std::endl;
        //std::cout << std::endl;
        // вдоль всего увеличивающего пути увеличиваем поток
        for (size_t i = 0; i < augmenting_path.size(); ++i) {
            int edge_num = augmenting_path[i];
            edges[edge_num].flow += cmin;
            edges[edges[edge_num].inv_edge].flow       =       -
edges[edge_num].flow;
        }
        //print_edges_1(edges);
        //std::cout << "------------" << std::endl;
        cmin = 0;
    }

    for (size_t i = 0; i < stock_edges.size(); i++) {
        flow = flow + edges[stock_edges[i]].flow;
    }
    std::cout << flow << std::endl;

    //print_a(a, edges);
    std::vector<edge> new_edges(size);

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

    std::sort(new_edges.begin(), new_edges.end(), customLess);


    print_edges(new_edges);
```

```
    return 0;
}
```

ПРИЛОЖЕНИЕ В
ТЕСТИРОВАНИЕ

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
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 f a b 7 a c 7 c b 5 b d 4 b e 3 c e 10 d f 8 e f 4 e d 11	12 a b 7 a c 5 b d 4 b e 3 c b 0 c e 5 d f 8 e d 4 e f 4	
3.	5 a d a b 7 a c 6 b d 6 c f 9	6 a b 6 a c 0 b d 6 c f 0 d e 0	

	d e 3		
4.	1 a f a b 7	0 a b 0	
5.	1 a b a b 7	7 a b 7	Разбивается как 3x3
6.	6 a e a b 0 a c 0 c b 0 b d 0 b e 0 c e 0	0 a b 0 a c 0 b d 0 b e 0 c b 0 c e 0	Разбивается как 5x5
7.	5 a f a b 7 c b 5 b e 3 c e 10 d f 8 e d 11	0 a b 0 b e 0 c b 0 c e 0 d f 0	
8.	6 a f a b 7 c b 5 b e 3 c e 10	3 a b 3 b e 3 c b 0 c e 0 d f 0 e f 3	

	e f 100 d f 8 e d 11		