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

Студент гр. 9382 _____ Савельев И.С.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

На практике ознакомиться с алгоритмом Форда-Фалкерсона поиска максимального потока в 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 ω_{ij} - ребро графа

v_i v_j ω_{ij} - ребро графа

...

Выходные данные:

P_{max} - величина максимального потока

v_i v_j ω_{ij} - ребро графа

v_i v_j ω_{ij} - ребро графа

...

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).

Sample Input:

```
7
a
f
a b 7
a c 6
b d 6
c f 9
d e 3
d f 4
e c 2
```

Sample Output:

```
12
a b 6
a c 6
b d 6
c f 8
d e 2
d f 4
e c 2
```

Описание алгоритма.

Считываются данные пользователя формируется матрица смежности из вершин отсортированных в алфавитном порядке. Все потоки обнуляются, изначально остаточная сеть равна равна исходной. С помощью поиска в глубину в остаточной сети находим путь он стартовой вершины до финальной, если такого пути нет прекращаем алгоритм. В найденном пути находим максимальный возможный поток, перебирая все ребра пути в остаточной сети, в поисках ребра с минимальной пропускной способностью. Пускаем

максимально возможный поток по найденному пути, для каждого ребра на пути увеличиваем поток, а в противоположном направлении уменьшаем. Затем изменяем остаточную сеть, для всех ребер на пути рассчитываем новую пропускную способность.

Описание структур данных представляющих граф.

Граф представляется с помощью двумерного массива vector<vector<int>> graph размера V * V, V - количество вершин в графе, каждой вершине отсортированный в алфавитном порядке соответствует массив пропускной способности ребер исходящих из нее, если вершины не смежны то пропускная способность считается равной 0.

Оценка сложности по памяти и времени.

На каждом шаге алгоритм добавляет поток увеличивающего пути к уже имеющемуся потоку .Следовательно, на каждом шаге алгоритм увеличивает поток по крайней мере на единицу, следовательно, он сойдется не более чем за O(*f*) шагов, где *f* — максимальный поток в графе. Можно выполнить каждый шаг за время O(*E*), где *E* — число ребер в графе, тогда общее время работы алгоритма ограничено O(*Ef*).

Так как программа хранит матрицу смежности графа оценка сложности по памяти будет равна O(V^2), V - количество вершин.

Тестирование.

Таблица 1 результаты работы программы.

Входные данные	Вывод программы
3 b e b c 5 b l 6 l e 2	2 b c 0 b e 2 l e 2

8 a l a b 3 b l 2 a c 4 c d 1 c f 2 f e 3 d e 5 e l 7	5 a b 2 a c 3 b l 2 c d 1 c f 2 d e 1 e l 3 f e 2
4 z y z x 2 x y 1 x z 5 z y 2	3 x y 1 x z 0 z x 1 z y 2

Выводы.

В ходе выполнения данной лабораторной работы был реализован алгоритм Форда Фалкерсона поиска максимального потока в сети на языке программирования C++.

Приложение А. Исходный код программы.

```
#include <iostream>
#include <vector>
#include <limits.h>
#include <algorithm>
#include <stack>

using namespace std;

// компраратор для сортировки по второй вершине
bool compare(pair <pair <int, int>, int> a, pair <pair <int, int>, int>
b){
      if (a.first.first == b.first.first) {
       if (a.first.second < b.first.second) {
            return true;
      }
     }
      return false;
}

// Выводит ответ
void print_answer(string node, vector<vector<int>>& graph, int N, string
from, string to, vector<int> edge_weight) {
      std::vector<pair<pair<int, int>, int> > result;
      for (int q = 0; q < node.length(); q++) { // перебор вершин
      vector <int> edge_mass; // сколько ребер выходит из node[q] и их
индексы
      for (int j = 0; j < N; j++) {
            if (from[j] == node[q]) {
                  edge_mass.push_back(j);
            }
      }
      vector <int> mass_to; // в какую верш идет ребро исходящие из
node[q] отсортир в алф
      // ищем вершины в которые входят ребра node[q]
      for (int i = 0; i < edge_mass.size(); i++) {
            for (int j = 0; j < node.length(); j++) {
                  if (node[j] == to[edge_mass[i]]) {
                  mass_to.push_back(j);
                  }
            }
      }
      // заносим результат в массив
      for (int i = 0; i < edge_mass.size(); i++) {
            if (graph[q][mass_to[i]] < 0) {
                  result.push_back(make_pair(make_pair(q,      mass_to[i]),
0));
            }
            else {
```

```
                result.push_back(make_pair(make_pair(q,      mass_to[i]),
graph[q][mass_to[i]]));
           }
      }
      }
      sort(result.begin(), result.end(), compare); // сортируем
      // выводим результа
      for (auto i: result) {
      std::cout << node[i.first.first] << ' ';
      std::cout << node[i.first.second] << ' ';
      std::cout << i.second << '\n';
      }
}

// Поиск в глубину
bool    dfs(vector<vector<int>>    seti,    int    start,    int    finish,
vector<int>&previus_node, string node){
      // массив посещенных вершин
      vector<bool> vis_mass(node.length(), 0);
      // стек
      stack <int> node_stack;
      // кладем исходную вершину в стек
      node_stack.push(start);
      // отмечаем стартовую вершину как посещенную
      vis_mass[start] = true;
      previus_node[start] = -1;
      // пока стек не пуст и не найден финиш
      std::cout << '\n';
   while (!node_stack.empty() && (vis_mass[finish] == false)) {
       int i = node_stack.top();
       node_stack.pop();
      std::cout << "Рассматриваемая вершина: " << node[i] << '\n';
      // проверяем все смежные с  node[i] не посещенные вершины
      for(int j = 0 ; j < node.length(); j++){
           if(seti[i][j] > 0 && vis_mass[j] == false) {
                node_stack.push(j); // добавляем в стек
                previus_node[j] = i; // хранит своего предка
                vis_mass[j] = true; // отмечаем как посещенную
                std::cout << "\tВершина смежная с рассматриваемой по
которой можно пустить поток: " << node[j] << '\n';
           }
      }
      }
      // если финиш был посещен
      if(vis_mass[finish] == true) {
      std::cout << "Найденный путь: " << '\n';
      string Way;
      for (int i = finish; i != start; i = previus_node[i]) {
           Way = node[i] + Way;
      }
      Way = node[start] + Way;
```

```
      std::cout << Way << "\n\n";
      return true;
      }
      // если финиш не был достигнут
      std::cout << "\tНет вершин смежных с рассматриваемой по которым можно пустить поток" << "\n";
      std::cout << "Путь не найден" << "\n\n";
      return false;
}

// Заполняем начальную сеть
void init_graph(string node, vector<vector<int>>& graph, int N, string from, string to, vector<int> edge_weight) {
      // поиск всех ребер, ведущих из вершины node[q]
    for (int q = 0; q < node.length(); q++) { // перебор вершин
       vector <int> edge_mass; // сколько ребер выходит из node[q] и их индексы
       for (int j = 0; j < N; j++) {
            if (from[j] == node[q]) {
                  edge_mass.push_back(j);
            }
       }
      //поиск в строке node[q] вершины, в которую ведут ребра из вектора edge_mass
       vector <int> mass_to; // в какую верш идет ребро исходящие из node[q] отсортир в алф
      // номер верш в отсортированном node
      // ищем вершины в которые входят ребра node[q]
       for (int i = 0; i < edge_mass.size(); i++) {
            for (int j = 0; j < node.length(); j++) {
                  if (node[j] == to[edge_mass[i]]) {
                        mass_to.push_back(j);
                  }
            }
       }
      // заполняем список смежности отсортированный в алф порядке
       for (int i = 0; i < edge_mass.size(); i++) {
            graph[q][mass_to[i]] = edge_weight[edge_mass[i]];
       }

    }
      // на выходе двумерный массив отсортированный в алф порядке где каждой вершине соответствует вектор ребер с проходимостью
}

// Алгоритм Форда Фалкерсона
int f_f(vector<vector<int>>& graph, vector<vector<int>>& seti, int start, int finish, string node) {
    for (int u = 0; u < node.length(); u++)
      for (int v = 0; v < node.length(); v++) {
            seti[u][v] = graph[u][v];
```

```
            graph[u][v] = 0;
      }

    int max_flow = 0;
     // массив для хранения пути
    vector <int> previus_node(node.length(), 0);
     // увеличивается поток, пока есть путь от истока к стоку
    while (dfs(seti, start, finish, previus_node, node)) {
      int path_flow = INT_MAX;
     // находим поток через найденный путь
      for (int v = finish; v != start; v = previus_node[v]) {
            path_flow = min(path_flow, seti[previus_node[v]][v]); //
ребро между родителем и потомком
          std::cout << "Максимальный поток который можно пустить между
вершинами " << node[previus_node[v]] << " и " << node[v] << ": " <<
path_flow <<'\n';
      }
      std::cout << "Максимальный поток по данному пути : " << path_flow
<< "\n";
      // обновление пропускной способности каждого ребра
      for (int v = finish; v != start; v = previus_node[v]) {
          // graph[v][u] - вектор пропускной способности + от старка к
фин с - от фин к старту
            seti[previus_node[v]][v] -= path_flow; // ребро между
родителем и сыном
            seti[v][previus_node[v]] += path_flow; // ребро между
родителем и сыном в обратном направлении
            graph[previus_node[v]][v] += path_flow; // ребро между
родителем и сыном
            graph[v][previus_node[v]] -= path_flow; // ребро между
родителем и сыном в обратном направлении
      }
      max_flow += path_flow;
    }

    return max_flow;
}

int main() {
     char start;  //исток
    char finish; //сток
    int N = 0; //количество ориентированных рёбер графа
    string from; // вершины из которых выходят ребра
    string to; // вершины в которые входят ребра
    string node; // все вершины графа
     cin >> N >> start >> finish; // Кол-во, исток, сток
    vector <int> edge_weight; // веса ребер
    node = node + start;
     char node_from;
    char node_to;
     int weigth;
```

```
      // Ввод пользователя
    for (int i = 0; i < N; i++) {
       cin >> node_from; // откуда
       cin >> node_to; // куда
       cin >> weigth; // пропускная способность
       from = from + node_from;
       to = to + node_to;
       edge_weight.push_back(weigth);
      // если вершины нет в node
       if (node.find(node_to) == string::npos) {
            node = node + node_to;
      }
    }
      // отсортировали в алфавитном порядке
      sort(node.begin(), node.end());

      vector<vector<int>> graph(node.length(), vector<int>(node.length(),
0));
      init_graph(node, graph, N, from, to, edge_weight);

    int start_ind = 0;
    int finish_ind = 0;

    for (int i = 0; i < node.length(); i++) {
      // находим индекс старта в отсортированном массиве
       if (node[i] == start) { // равен истору
            start_ind = i;
      }
      // находим индекс финиша в отсортированном массиве
       else if (node[i] == finish) { // равен стоку
            finish_ind = i;
      }
    }

      vector<vector<int> > seti(node.length(), vector<int>(node.length(),
0)); // двумерный массив
      // алгоритм Форда Фалкерсона
    int max_flow = f_f(graph, seti, start_ind, finish_ind, node);

      std::cout <<"Максимальный поток в сети: " << max_flow << '\n';
      print_answer(node, graph, N, from, to, edge_weight);
    return 0;
}
```