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. 9383 _____ Арутюнян С.Н.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Изучить алгоритмы нахождения потока в сети.

Основные теоретические положения.

Сетью называется ориентированный граф, в котором каждое ребро имеет положительный вес. Каждая сеть имеет начальную вершину (исток - S) и конечную (сток - F).

Потоком f в графе G является вещественной функцией f: V x V → R такой, что она удовлетворяет условиям:

1) f(u, v) = -f(v, u)

2) f(u, v) <= c(u, v)

3) Сумма по всем f(u, v), кроме S и F, равна нулю.

Задание.

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

Входные данные:

N — количество ориентированных ребер графа

v0 — исток

vn — сток

vi vj wij — ребро графа с весом wij, соединяющее вершины vi и vj.

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.

Ход работы:

1. Была взята структура данных Graph из второй лабораторной работы.

2. Был разработан алгоритм Форда-Фалкерсона, который работает так:

1) Ищется путь от стока к истоку методом обхода в ширину, который выбирает следующую вершину в обратном алфавитном порядке по минимальному ребру. Вместе с самим путем в процессе обхода накапливается информация о потоке текущего пути. Если путь не был найден, алгоритм завершается.

2) К общей сумме потока прибавляется поток найденного пути.

3) Весс всех ребер, участвующих в пути, уменьшаются на поток всего пути, а вес всех ребер в пути, но в обратном направлении, увеличивается на тот же самый поток.

4) Переход к шагу 1.

Описание функций и структур данных:

1. Класс Graph. Он хранит двумерный массив такой, что array[i][j] равен весу ребра между вершиной i и j. Он имеет следующие методы:

1) void AddEdge(char u, char v, float weight) – добавляет вершину в граф.

2) void ChangeWeight(char u, char v, float weight) – изменяет вес уже существующего ребра.

3) template <typename Func> std::deque<char> GetNeighbours(char u, Func predicate) – возвращает всех соседей вершины u, удовлетворяющих условию predicate.

4) float GetEdgeWeight(char u, char v) – возвращает вес ребра между u и v.

5) bool HasEdge(char u, char v) – возвращает true, если между u и v существует ребро.

2. Функция std::pair<Graph, float> FordFalkerson(const Graph& graph, char source, char drain) – возвращает величину максимального потока графа graph.

3. Функция WebPath DFSFindPath(const Graph& graph, char source, char drain) – возвращает информацию о пути в графе и величине его потока.

4. Структура WebPath. Она хранит в себе вектор вершин, представляющий собой путь в графе, а также величину потока в этом пути.

Тесты

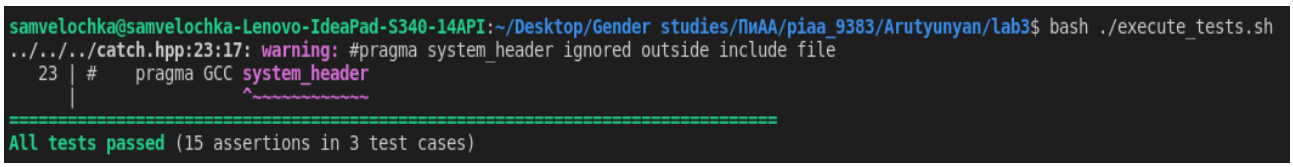
Были написаны следующие тесты:

1. Тест, проверяющий корректность добавления ребер в граф.

2. Тест, проверяющий корректность работы метода GetNeighbours с разными условиями.

3. Тест, проверяющий корректность работы алгоритма Форма-Фалкерсона.

Доказательство успешного прохождения тестов:



Примеры работы программы

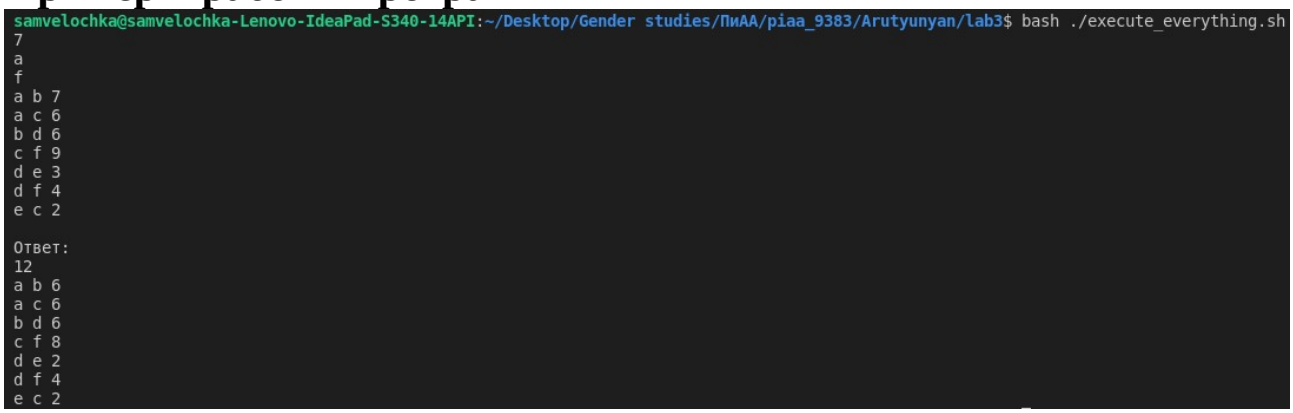


Рисунок 1. Пример работы программы (1)

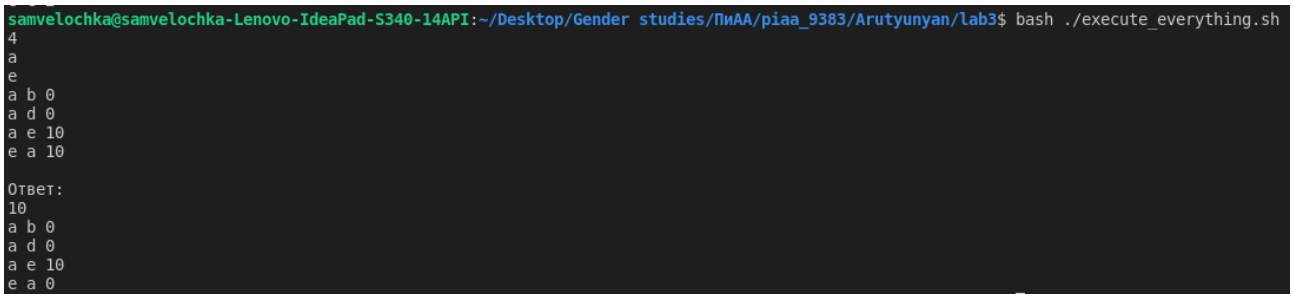


Рисунок 2. Пример работы программы (2)

Рисунок 3. Пример работы программы (3)

Выводы.

В выполненной лабораторной работе был изучен и применен на практике алгоритм Форда-Фалкерсона для нахождения максимального потока в сети.

ПРИЛОЖЕНИЕ А

GRAPH.HPP

```
#pragma once

#include <vector>
#include <deque>
#include <functional>
#include <algorithm>


class Graph {
public:
   explicit Graph(size_t vertexes_amount) {
      incident_matrix.resize(vertexes_amount, std::vector<float>(vertexes_amount, -1));
   }

   void SetOffset(int offset_) {
      offset = offset_;
   }

   int GetOffset() const {
      return offset;
   }

   void AddEdge(char u, char v, float weight) {
      incident_matrix[u - offset][v - offset] = weight;
   }

   void ChangeWeight(char u, char v, float weight) {
      incident_matrix[u - offset][v - offset] = weight;
   }

   template <typename Func>
   std::deque<char> GetNeighbours(char u, Func predicate) const
   {
      std::deque<char> neighbours;
      for (char i = (offset == 97 ? 'a' : '1'); i <= (offset == 97 ? 'z' : '9'); ++i) {
         if (incident_matrix[u - offset][i - offset] > 0 && predicate(i))
            neighbours.push_back(i);
      }
      return neighbours;
   }

   std::deque<char> GetNeighbours(char u) const
   {
      std::deque<char> neighbours;
      for (char i = (offset == 97 ? 'a' : '1'); i <= (offset == 97 ? 'z' : '9'); ++i) {
         if (incident_matrix[u - offset][i - offset] > 0)
            neighbours.push_back(i);
      }
      return neighbours;
   }

   float GetEdgeWeight(char u, char v) const {
      return incident_matrix[u - offset][v - offset];
```

```
    }

    bool HasEdge(char u, char v) const {
        return incident_matrix[u - offset][v - offset] != -1;
    }

    size_t Size() const { return incident_matrix.size(); }

private:
    std::vector<std::vector<float>> incident_matrix;
    int offset = 97;
};
```

MAIN.CPP

```
#include <iostream>
#include <fstream>
#include <algorithm>
#include "graph.hpp"
#include "ford_falkerson.hpp"

int main() {
    std::ifstream in("input.txt");

    Graph graph(26);
    auto comparator = [](const auto& p1, const auto& p2) {
        return std::tie(p1.first, p1.second) < std::tie(p2.first, p2.second);
    };
    std::set<std::pair<char, char>, decltype(comparator)> to_check(comparator);

    int n;
    char source, drain;
    std::cin >> n >> source >> drain;

    if (std::isdigit(source))
        graph.SetOffset(48);

    for (int i = 0; i < n; ++i) {
        char u, v;
        float weight;
```

```
        std::cin >> u >> v >> weight;

        graph.AddEdge(u, v, weight);
        to_check.insert({u, v});
    }


    auto flow_info = FordFalkerson(graph, source, drain);

    std::cout << "\nОтвет:" << std::endl;
    std::cout << flow_info.second << std::endl;
    for (const auto &edge : to_check) {
        float weight = flow_info.first.GetEdgeWeight(edge.first, edge.second);
        if (weight != -1)
            std::cout << edge.first << " " << edge.second << " " << weight << std::endl;
        else
            std::cout << edge.first << " " << edge.second << " " << 0 << std::endl;
    }

    return 0;
}
```

FORD-FALKERSON.HPP