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

Студентгр. 8304 _____ Чешуин Д.И.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург
2020

Цель работы.

Ознакомиться с алгоритмом Форда-Фалкерсона, научиться оценивать временную сложность алгоритма и применять его для решения задач.

Постановка задачи.

Поиск 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.

Изначальный поток равен нулю. На каждой итерации алгоритма ищется путь из истока в сток по ребрам с немаксимальным текущим потоком. Поиск происходит соответственно варианту. Далее находится ребро с минимальной оставшейся пропускной способностью, к потокам задействованных в пути ребер прибавляется эта минимальная величина потока. Алгоритм заканчивает свою работу, если больше нет возможных путей.

Величина максимального потока равна сумме потоков ребер, инцидентных истоку.

Анализ алгоритма.

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 сойдётся не более чем за $O(f)$ шагов, где f — максимальный поток в графе. Можно выполнить каждый шаг за время $O(E)$, где E — число рёбер в графе, тогда общее время работы алгоритма ограничено $O(Ef)$.

Требуемая память: $O(V^2)$, так как для хранения связей используется матрица смежности.

Описание функций и СД.

Для решения задачи был реализован класс Graph, класс FordFulkerson и структура отдельного ребра графа – Edge. UML диаграмму использованных структур данных смотри на рисунке 1.

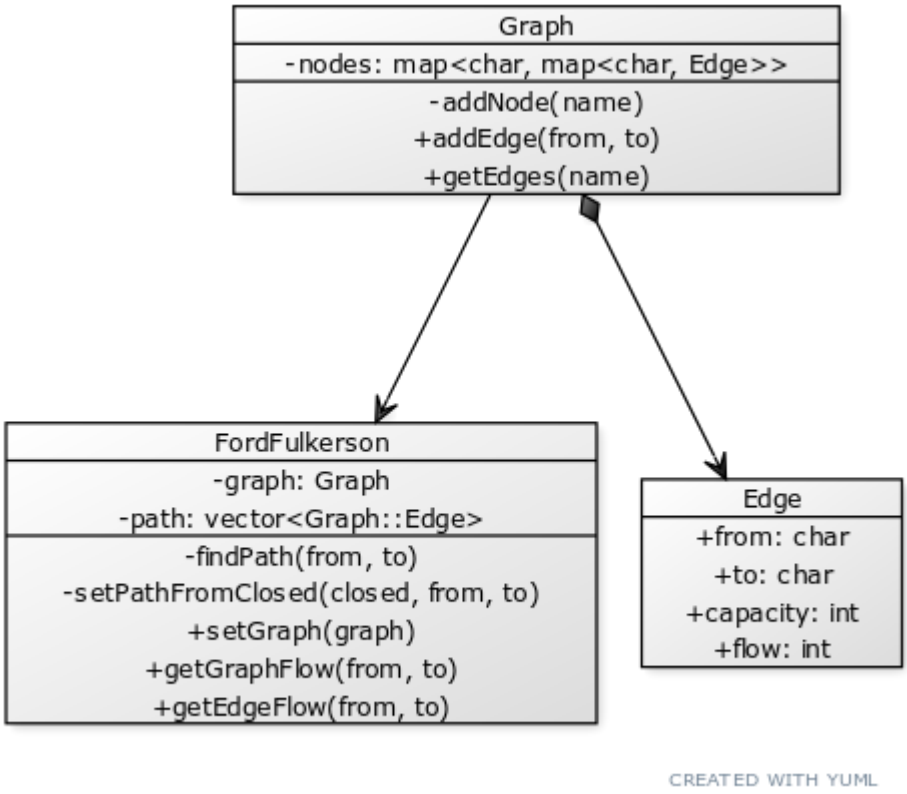

Рисунок 1 – UML диаграмма использованных структур данных

Метод поиска пути:

bool findPath(char from, char to);

Принимает начальной и конечной вершины, возвращает true, если путь найден и false в противном случае.

void setPathFromClosed(vector<Graph::Edge>& closed, char from, char to);

Метод восстанавливает путь по закрытым рёбрам и устанавливает его в качестве текущего.

void setGraph(Graph graph);

Метод устанавливает переданный граф в качестве рабочего.

int getGraphFlow(char from, char to);

Метод ринимает исток и сток и возвращает максимальный поток между данными вершинами.

int getEdgeFlow(char from, char to);

Метод принимает начальную и конечную вершины ребра и возвращает текущий поток через данное ребро.

Спецификация программы.

Программа предназначена для нахождения максимального потока в графе. Программа написана на языке C++. Входными данными является число N рёбер, начальная и конечная вершины, перечень рёбер графа, а выходными – максимальный поток в графе и перечень потоков в рёбрах графа.

Тестирование.

Пример работы программы вывода для графа “a b 7, a c 6, b d 6, c f 9, d e 3, d f 4, e c 2”, начальной вершины a и конечной - f .

```
Edge: d e capacity - 3 flow - 0
Edge: b d capacity - 6 flow - 4
Edge: a b capacity - 7 flow - 4
Minimal remaining capacity: 2
-----------------------------------------------------
Trying to find path from: a to f
-----------------------------------------------------
Current edge - from: a to: b
Checking node:
Has capacity.
Good edge, adding new edges check:
Edge: b a capacity - 0 flow - -6
Edge: b d capacity - 6 flow - 6
-----------------------------------------------------
Current edge - from: a to: c
Checking node:
Has capacity.
To-node already checked. Ignored
-----------------------------------------------------
Current edge - from: a to: c
Checking node:
Remaining capacity = 0.Ignored
-----------------------------------------------------
Current edge - from: b to: d
Checking node:
Remaining capacity = 0.Ignored
-----------------------------------------------------
Max flow in graph:12
Edge flow:
a b 6
Edge flow:
a c 6
Edge flow:
b d 6
Edge flow:
c f 8
Edge flow:
d e 2
Edge flow:
d f 4
Edge flow:
e c 2
```

Рисунок 2 – Пример вывода для простого графа

Выводы.

В ходе выполнения лабораторной работы был реализован алгоритм Форда-Фалкерсона, дана оценка времени работы алгоритма, а также были получены навыки решения задач с помощью алгоритма Форда-Фалкерсона

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД ПРОГРАММЫ

main.cpp.

```
#include <iostream>
#include <algorithm>
#include <map>
#include <queue>
#include <cstring>
#include <fstream>
#include <cmath>
#include <vector>
#include <string>

using namespace std;

class IOManager
{
private:
    static istream* input;
    static ostream* output;
public:
    static void setStreamsFromArgs(int argc, char** argv)
    {
        if(argc > 1)
        {
            for(int i = 1; i < argc; i++)
            {
                if(strcmp(argv[i], "-infile") == 0)
                {
                    if(i + 1 < argc)
                    {
                        input = new ifstream(argv[i + 1]);
                        i += 1;
                    }
                }
                if(strcmp(argv[i], "-outfile") == 0)
                {
                    if(i + 1 < argc)
                    {
                        output = new ofstream(argv[i + 1]);
                        i += 1;
                    }
                }
            }
        }
    }
    static istream& getIS()
    {
        return *input;
    }
    static ostream& getOS()
    {
        return *output;
    }
    static void resetStreams()
    {
```

```
        if(input != & cin)
        {
            delete input;
            input = &cin;
        }
        if(output != & cout)
        {
            delete output;
            output = &cout;

        }
    }
};

class Graph
{
public:
    struct Edge
    {
        char from;
        char to;
        int capacity = 0;
        int flow = 0;
    };
private:
    map<char, map<char, Edge>> nodes;
    bool addNode(char name);
public:
    bool addEdge(char from, char to, unsigned capacity);
    map<char, Edge>& getEdges(char name);
};

class FordFulkerson
{
private:
    Graph graph;
    vector<Graph::Edge> path;
private:
    bool findPath(char from, char to);
    void setPathFromClosed(vector<Graph::Edge>& closed, char from, char to);
public:
    void setGraph(Graph graph);
    int getGraphFlow(char from, char to);
    int getEdgeFlow(char from, char to);
};

int main(int argc, char** argv)
{
    IOManager::setStreamsFromArgs(argc, argv);
    Graph graph;
    FordFulkerson alg;

    int edgesCount = 0;
    string fromFlow, toFlow;
    vector<string> edges;

    IOManager::getOS() << "Enter edges count:" << endl;
    IOManager::getIS() >> edgesCount;
    IOManager::getOS() << "Enter start and finish nodes:" << endl;
```

```
    IOManager::getIS() >> fromFlow >> toFlow;
    IOManager::getOS() << "Enter edges:" << endl;
    for(int i = 0; i < edgesCount; i++)
    {
        string from, to;
        int capacity;

        IOManager::getIS() >> from >> to >> capacity;

        graph.addEdge(from[0], to[0], capacity);

        edges.push_back(from + to);
    }

    sort(edges.begin(), edges.end());

    alg.setGraph(graph);
    int flow = alg.getGraphFlow(fromFlow[0], toFlow[0]);
    IOManager::getOS() << "Max flow in graph:" << flow << endl;

    for(auto edge : edges)
    {
        IOManager::getOS() << "Edge flow:" << endl;
        IOManager::getOS() << edge[0] << " " << edge[1] << " " << alg.getEdgeFlow(edge[0],
edge[1]) << endl;
    }

    return 0;
}

bool Graph::addNode(char name)
{
    map<char, Edge> emptyEdges;
    return nodes.emplace(name, emptyEdges).second;
}

bool Graph::addEdge(char fromName, char toName, unsigned capacity)
{
    auto fromNode = nodes.find(fromName);
    auto toNode = nodes.find(toName);
    if(fromNode == nodes.end())
    {
        if(!addNode(fromName))
        {
            return false;
        }
    }
    if(toNode == nodes.end())
    {
        if(!addNode(toName))
        {
            return false;
        }
    }

    auto node = nodes.find(fromName);
    auto edge = node->second.find(toName);
    if(edge == node->second.end())
    {
```

```
        Edge newEdge;
        newEdge.to = toName;
        newEdge.from = fromName;
        newEdge.capacity = capacity;
        node->second.emplace(toName, newEdge);
    }
    else
    {
        edge->second.capacity = capacity;
    }

    auto reverseEdge = nodes.find(toName)->second.find(fromName);

    if(reverseEdge == nodes.find(toName)->second.end())
    {
        addEdge(toName, fromName, 0);
    }

    return true;
}

map<char, Graph::Edge>& Graph::getEdges(char name)
{
    return nodes.find(name)->second;
}

void FordFulkerson::setGraph(Graph graph)
{
    this->graph = graph;
}

int FordFulkerson::getGraphFlow(char from, char to)
{
    unsigned flow = 0;

    while(findPath(from, to))
    {
        IOManager::getOS() << "Path finded: ";
        IOManager::getOS() << "----------------------------------------------------------------------
--" << endl;
        for(int i = path.size(); i >= 0; i--)
        {
            IOManager::getOS() << path[i].from;
        }

        IOManager::getOS() << endl;

        int minCapacity = path.front().capacity - path.front().flow;
        for(auto edge : path)
        {
            IOManager::getOS() << "Edge: " << edge.from << " " << edge.to
                               << " capacity - " << edge.capacity << " flow - " << edge.flow <<
endl;
            if(edge.capacity - edge.flow < minCapacity)
            {
                minCapacity = edge.capacity - edge.flow;
            }
        }
```

```
        IOManager::getOS() << "Minimal remaining capacity: " << minCapacity << endl;

       for(auto edge : path)
       {
           auto directEdge = graph.getEdges(edge.from).find(edge.to);
           auto reverseEdge = graph.getEdges(edge.to).find(edge.from);

           directEdge->second.flow += minCapacity;
           reverseEdge->second.flow -= minCapacity;
       }

        path.clear();
    }

    for(auto edge : graph.getEdges(from))
    {
        flow += edge.second.flow;
    }

    return flow;
}


int FordFulkerson::getEdgeFlow(char from, char to)
{
    int flow = graph.getEdges(from).find(to)->second.flow;
    return flow > 0 ? flow : 0;
}

bool FordFulkerson::findPath(char from, char to)
{
    IOManager::getOS()  <<  "----------------------------------------------------------------------"
<< endl;
    IOManager::getOS() << "Trying to find path from: " << from <<  " to " << to << endl;
    IOManager::getOS()  <<  "----------------------------------------------------------------------"
<< endl;

    vector<Graph::Edge> open;
    vector<Graph::Edge> closed;
    vector<char> closedNodes;

    closedNodes.push_back(from);
    for(auto pair : graph.getEdges(from))
    {
        open.push_back(pair.second);
    }

    while(!open.empty())
    {
        Graph::Edge curEdge = open.front();
        IOManager::getOS() << "Current edge - from: " << curEdge.from <<  " to: " << curEdge.to <<
endl;

        auto minEdge = open.begin();
        for(auto edge = open.begin(); edge < open.end(); edge++)
        {
            if(abs(edge->to - edge->from) < abs(minEdge->to - minEdge->from))
            {
                minEdge = edge;
```

```
            }
            else if(abs(edge->to - edge->from) == abs(minEdge->to - minEdge->from))
            {
                if(edge->to < minEdge->to)
                {
                    minEdge = edge;
                }
            }
        }

        curEdge = *minEdge;
        open.erase(minEdge);

        IOManager::getOS() << "Checking node: " << endl;

        bool isGoodEdge = true;

        if(curEdge.capacity - curEdge.flow == 0)
        {
            IOManager::getOS() << "Remaining capacity = 0." << "Ignored" << endl;
            isGoodEdge = false;
        }

        if(isGoodEdge)
        {
            IOManager::getOS() << "Has capacity." << endl;
            for (auto node : closedNodes)
            {
                if(node == curEdge.to)
                {
                    isGoodEdge = false;

                    IOManager::getOS() << "To-node already checked. Ignored" << endl;

                    break;
                }
            }
        }

        if(isGoodEdge)
        {
            IOManager::getOS() << "Good edge, adding new edges check:" << endl;

            for(auto pair : graph.getEdges(curEdge.to))
            {
                IOManager::getOS() << "Edge: " << pair.second.from << " " << pair.second.to
                                   << " capacity - " << pair.second.capacity << " flow - " <<
pair.second.flow << endl;
                open.push_back(pair.second);
            }

            closed.push_back(curEdge);
            closedNodes.push_back(curEdge.to);

            if(curEdge.to == to )
            {
                setPathFromClosed(closed, from, to);

                return true;
```

```
            }
        }

        IOManager::getOS() << "-----------------------------------------------------------------
--" << endl;
    }

    return  false;
}

void FordFulkerson::setPathFromClosed(vector<Graph::Edge>& closed, char from, char to)
{
    char curNode = to;
    while(curNode != from)
    {
        for(auto edge : closed)
        {
            if(edge.to == curNode)
            {
                path.push_back(edge);

                curNode = edge.from;

                break;
            }
        }
    }
}

istream* IOManager::input = &cin;
ostream* IOManager::output = &cout;
```