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 и сети

Студент гр. 8383 _____ Бессуднов Г. И.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Изучить принцип работы алгоритма Форда-Фалкерсона для нахождения максимального потока в сети на графах. Решить с их помощью задачи

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

Входные данные:

N - количество ориентированных рёбер графа

v0 - 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 с фактической величиной протекающего потока

v_i v_j ω_{ij} - 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).

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

Вар. 5. Поиск не в глубину и не в ширину, а по правилу: каждый раз выполняется переход по дуге, имеющей максимальную остаточную пропускную способность. Если таких дуг несколько, то выбрать ту, которая была обнаружена раньше в текущем поиске пути.

Описание алгоритма.

Строится остаточная сеть, поток в каждом ребре устанавливается в 0. Далее по правилу индивидуализации происходит поиск пути в графе. Если путь был найден, то вычисляется минимальная остаточная пропускная способность ребер в этом пути. Далее это значение прибавляется к значению потока всех

ребер в пути и вычитается из значения потока у обратных ребер. Если же путь найден не был, то алгоритм заканчивает работу.

Описание структур данных.

```
class Edge {
public:
	char start; //начальная вершина
	char end; //конечная вершина
	int flow; //поток через ребро
	int capacity; //пропускная способность
	int residualCapacity; //остаточная пропускная способность
	Colors color; //цвет ребра
	Edge(char start, char end, int capacity);
	void evaluateResidualCapacity();
};
```

Класс необходим для хранения информации о ребре. Содержит функцию `void evaluateResidualCapacity();` которая высчитывает остаточную пропускную способность ребра.

```
class Graph {
private:
	std::map<char, std::vector<Edge*>> graph; //граф
	std::map<char, std::vector<Edge*>>::iterator graphIt; //итератор графа
public:
	//получение ребра по двум точкам
	Edge* operator()(char start, char end);
	Edge* findMaxEdge(char start);
	void pushEdge(Edge* edge);
	int maxFlow(char sink);
	void recolorize();
	void printGraph();
};
```

Класс необходим для хранения графа. Граф хранится в структуре `std::map<char, std::vector<Edge*>> graph;` как список смежности В структуре `map` параметр char отвечает за вершину, а вектор, состоящий из `Edge` - это ребра, смежные с этой вершиной. Содержит функции:

`Edge* findMaxEdge(char start);`- необходима для поиска ребра с максимальной остаточной пропускной способностью из вершины start. Возвращает указатель на ребро, если оно было найдено и nullptr, если оно найдено не было.

`void pushEdge(Edge* edge);` - добавляет ребро в граф.

`int maxFlow(char sink);`-высчитывает максимальный поток, складывая потоки входящие в сток sink. Возвращает величину максимального потока.

`void recolorize();`- перекраска графа в черный.

`void printGraph();`-печатать графа.

```
class PathFinder {
private:
	std::vector<std::pair<char, char>> currentPath; //путь
	std::vector<std::pair<char, char>>::iterator pathIt; //итератор пути
	int minCapacity; //минимальная остаточная пропускная способность пути
	void constructPath(std::stack<Edge*> edgeStack, char start, char end);
	void printPath();
	void setMinCapacity(std::stack<Edge*> edgeStack)
public:
	std::vector<std::pair<char, char>> getPath();
	int getMinCapacity();
	void findPath(Graph &graph, char start, char end);
	bool isPath();
};
```

Класс необходим для поиска путей в графе. Содержит функции:

`std::vector<std::pair<char, char>> getPath();`- возвращает найденный путь.

`int getMinCapacity();` - возвращает минимальную остаточную пропускную способность пути.

`void findPath(Graph &graph, char start, char end);`- нахождение пути в графе `graph` из `start` в `end` с условием индивидуализации.

`bool isPath();` - функция проверки нахождения пути. Возвращает 1, если путь был найден и 0, если нет.

```
class Network {
private:
	PathFinder pathFinder; //объект для нахождения путей
	Graph residualNetwork; //остаточная сеть
	int maxFlow; //максимальный поток
	char source; //исток
	char sink; //сток
public:
	Network();

	void printNetwork();
	void findMaxFlow();
};
```

Класс необходимый для работы с сетью. Содержит функции:

`void printNetwork();`- печать сети.

`void findMaxFlow();` - нахождение максимального потока в сети.

Сложность алгоритма по времени.

В основе алгоритма лежит поиск из варианта индивидуализации, сложность которого можно оценить как $O(|E|)$, так как в худшем случае будут пройдены все ребра. После каждой итерации к значению потоку прибавляется как минимум 1, значит цикл будет длится не больше чем сумма весов ребер, выходящих из истока, обозначим это число f. Итого сложность алгоритма по времени $O(|E| * f)$

Сложность алгоритма по памяти.

Программа хранит только ребра, поэтому сложность $O(|E|)$

Тестирование.

Пример вывода результата работы программы представлен на рис. 1.

```
3
a
c
a b 7
a c 6
b c 4
Current path:
a<--->b
b<--->c
Minimum residual capacity: 4

Residual Network:

   Name   Flow  Capacity  Residual
a<--->b      4         7         3
a<--->c      0         6         6
b<--->c      4         4         0

<--------------------------->


Current path:
a<--->c
Minimum residual capacity: 6

Residual Network:

   Name   Flow  Capacity  Residual
a<--->b      4         7         3
a<--->c      6         6         0
b<--->c      4         4         0

<--------------------------->


<_____________ANSWER_____________>
Max flow: 10

Residual Network:

   Name   Flow  Capacity  Residual
a<--->b      4         7         3
a<--->c      6         6         0
b<--->c      4         4         0

<--------------------------->
```

Рис. 1- Результат работы программы

№	Input	Output
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	17 a b 5 a c 12 b c 5 c d 14 c e 3 d e 14
2	10 a f a b 16 a c 13 c b 4 b c 10 b d 12 c e 14 d c 9 d f 20 e d 7 e f 4	23 a b 12 a c 11 b c 0 b d 12 c b 0 c e 11 d c 0 d f 19 e d 7 e f 4
3	3 a c a b 7 a c 6 b c 4	10 a b 4 a c 6 b c 4

Код программы приведен в приложении А.

Вывод.

В ходе лабораторной работы был реализован на языке C++ алгоритм Форда-Фалкерсона для нахождения максимального потока сети.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include "pch.h"
#include <iostream>
#include <map>
#include <vector>
#include <stack>
#include <algorithm>
#include <iomanip>

//перечисление цветов
enum Colors { White, Black };

//класс ребра
class Edge {
public:
	char start; //начальная вершина
	char end; //конечная вершина
	int flow; //поток через ребро
	int capacity; //пропускная способность
	int residualCapacity; //остаточная пропускная способность
	Colors color; //цвет ребра

	Edge(char start, char end, int capacity) : start(start), end(end),
capacity(capacity), residualCapacity(capacity) {
		color = Colors::Black;
		flow = 0;
	}

	void evaluateResidualCapacity() {
		residualCapacity = capacity - flow;
	}


};

//компоратор для сортировки ребер по остаточной пропускной способности
bool edgeComp(Edge* e1, Edge* e2) {
	return e1->residualCapacity > e2->residualCapacity;
}

//компоратор для сортировки ребер по имени
bool edgeCompOut(Edge* e1, Edge* e2) {
	return e1->end < e2->end;
}

//класс графа
class Graph {
private:
	std::map<char, std::vector<Edge*>> graph; //граф
	std::map<char, std::vector<Edge*>>::iterator graphIt; //итератор графа
public:
	//получение ребра по двум точкам
	Edge* operator()(char start, char end) {
		for (auto& edge : graph[start]) {
			if (edge->end == end) {
				return edge;
```

```
                }
            }

            return nullptr;
        }

        //нахождение ребра с максимальной остаточной пропускной способностью
        Edge* findMaxEdge(char start);

        //функция добавления ребра в граф
        void pushEdge(Edge* edge) {
            graph[edge->start].push_back(edge);
            graph[edge->end];
        }

        //высчитывание максимального потока
        int maxFlow(char sink) {
            int maxFlow = 0;
            for (graphIt = graph.begin(); graphIt != graph.end(); graphIt++) {
                for (auto &edge : graphIt->second) {
                    if (edge->end == sink) {
                        maxFlow += edge->flow;
                    }
                }
            }

            return maxFlow;
        }

        //перекрасить граф в черный цвет
        void recolorize() {
            for (graphIt = graph.begin(); graphIt != graph.end(); graphIt++) {
                for (auto &edge : graphIt->second) {
                    edge->color = Colors::Black;
                }
            }
        }

        //печать графа
        void printGraph() {
            std::cout << "Residual Network:\n" << std::endl;
            std::cout << std::setw(7) << "Name" << std::setw(7) << "Flow" <<
std::setw(10)
                << "Capacity" << std::setw(10) << "Residual" << std::endl;
            for (graphIt = graph.begin(); graphIt != graph.end(); graphIt++) {
                std::sort(graphIt->second.begin(), graphIt->second.end(),
edgeCompOut);
                for (auto &edge : graphIt->second) {
                    if (edge->capacity > 0) {
                        std::cout << edge->start << "<--->" << edge->end;
                        std::cout << std::setw(7) << ((edge->flow > 0) ? edge-
>flow : 0);
                        std::cout << std::setw(10) << edge->capacity;
                        std::cout << std::setw(10) << edge->residualCapacity <<
std::endl;
                    }
                }
            }
            std::cout << "\n<--------------------------->\n\n" << std::endl;
        }
};
```

```
Edge* Graph::findMaxEdge(char start) {
	if (graph[start].empty()) {
		return nullptr;
	}

	//сортировка ребер, исходящих из данной вершины
	std::sort(graph[start].begin(), graph[start].end(), edgeComp);

	for (auto &edge : graph[start]) {
		if (edge->color == Colors::Black && edge->residualCapacity > 0) {
			return edge;
		}
	}

	return nullptr;
}

//класс для нахождения пути
class PathFinder {
private:
	std::vector<std::pair<char, char>> currentPath; //путь
	std::vector<std::pair<char, char>>::iterator pathIt; //итератор пути
	int minCapacity; //минимальная остаточная пропускная способность пути

	//воссоздание пути
	void constructPath(std::stack<Edge*> edgeStack, char start, char end);
	void printPath();
	//нахождение минимальной остаточной пропускной способности в пути
	void setMinCapacity(std::stack<Edge*> edgeStack) {
		minCapacity = edgeStack.top()->residualCapacity;
		while (!edgeStack.empty()) {
			if (edgeStack.top()->residualCapacity < minCapacity) {
				minCapacity = edgeStack.top()->residualCapacity;
			}

			edgeStack.pop();
		}
	}

public:
	std::vector<std::pair<char, char>> getPath() {
		return currentPath;
	}

	int getMinCapacity() {
		return minCapacity;
	}

	//функция нахождения пути
	void findPath(Graph &graph, char start, char end);

	bool isPath() {
		return !currentPath.empty();
	}
};

void PathFinder::printPath() {
	std::cout << "Current path:" << std::endl;
	for (pathIt = currentPath.end() -1; pathIt != currentPath.begin(); pathIt--) {
		std::cout << pathIt->first << "<--->" << pathIt->second << std::endl;
```

```
	}
	std::cout << pathIt->first << "<--->" << pathIt->second << std::endl;
	std::cout << "Minimum residual capacity: " << minCapacity << "\n\n" << std::endl;
}

void PathFinder::constructPath(std::stack<Edge*> edgeStack, char start, char end) {
	char symb;
	symb = end;
	Edge *currentEdge;
	while (!edgeStack.empty()) {
		currentEdge = edgeStack.top();
		edgeStack.pop();
		currentPath.push_back(std::pair<char, char>(currentEdge->start,
currentEdge->end));
	}
	printPath();
}

//клас сети
class Network {
private:
	PathFinder pathFinder; //объект для нахождения путей
	Graph residualNetwork; //остаточная сеть
	int maxFlow; //максимальный поток
	char source; //исток
	char sink; //сток

public:
	Network() : maxFlow(0) {
		int edgesAmount;
		std::cin >> edgesAmount;
		std::cin >> source;
		std::cin >> sink;

		char v1;
		char v2;
		int capacity;

		for (int i = 0; i < edgesAmount; i++) {
			std::cin >> v1 >> v2 >> capacity;
			residualNetwork.pushEdge(new Edge(v1, v2, capacity));
			residualNetwork.pushEdge(new Edge(v2, v1, 0));
		}
	}

	void printNetwork();
	//функция нахождения максимального потока
	void findMaxFlow();
};

void Network::printNetwork() {
	maxFlow = residualNetwork.maxFlow(sink);
	std::cout << "<_____________ANSWER_____________>" << std::endl;
	std::cout << "Max flow: " << maxFlow << "\n" << std::endl;
	residualNetwork.printGraph();
}

//нахождение пути в графе
void PathFinder::findPath(Graph &graph, char start, char end) {
	//очистка предыдущего пути и перекраска графа
	currentPath.clear();
```

```
	graph.recolorize();

	//стэк пути
	std::stack<Edge*> edgeStack;

	//кладем в стэк начальное значение для запуска алгоритма
	Edge *currentEdge = graph.findMaxEdge(start);
	if (currentEdge == nullptr) {
		currentPath.clear();
		return;
	}

	graph(currentEdge->start, currentEdge->end)->color = Colors::White;
	edgeStack.push(currentEdge);

	//если попложенное ребро является концом, то выходим
	if (currentEdge->end == end) {
		setMinCapacity(edgeStack);
		constructPath(edgeStack, start, end);
		return;
	}

	for (;;) {
		currentEdge = graph.findMaxEdge(edgeStack.top()->end);
		//если не смогли найти ребро из данной вершины
		if (currentEdge == nullptr) {
			edgeStack.pop();
			//если стэк опустел, то пути нет
			if (edgeStack.empty()) {
				currentPath.clear();
				return;
			}
		} else {
			//ребро было найдено
			graph(currentEdge->start, currentEdge->end)->color = Colors::White;

			//если дошли до конца, то выходим
			if (currentEdge->end == end) {
				edgeStack.push(currentEdge);
				setMinCapacity(edgeStack);
				constructPath(edgeStack, start, end);
				return;
			}

			edgeStack.push(currentEdge);
		}
	}

}

//поиск максимального потока
void Network::findMaxFlow() {
	for (;;) {
		//ищем путь
		pathFinder.findPath(residualNetwork, source, sink);

		//если путь не был найден, то цикл заканчивается
		if (!pathFinder.isPath()) {
			printNetwork();
			return;
		}
```

```
            //если путь был найден, то изменяем поток и остаточную пропускную
способность у ребер в пути
            for (auto& edge : pathFinder.getPath()) {
                  residualNetwork(edge.first, edge.second)->flow +=
pathFinder.getMinCapacity();
                  residualNetwork(edge.first, edge.second)-
>evaluateResidualCapacity();

                  residualNetwork(edge.second, edge.first)->flow -=
pathFinder.getMinCapacity();
                  residualNetwork(edge.second, edge.first)-
>evaluateResidualCapacity();

            }

            residualNetwork.printGraph();
      }


}

int main() {
      Network network;
      network.findMaxFlow();
}
```