МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра САПР

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и Структуры Данных»

Тема: «Потоки в сетях»

Вариант 1

Студент гр. 8301 _____ Забалуев Д.А.

Преподаватель _____ Тутуева А.В.

Санкт-Петербург

2020

1. Цель работы

Реализовать алгоритм Форда-Фалкерсона для нахождения максимального потока в сети. Сеть задана в текстовом файле .txt в виде строк в формате V1 V2 P, где V1,V2 – направленная дуга сети, а P – её пропускная способность. При этом исток обозначается как S, а сток – как T.

2. Описание программы

Идея алгоритма состоит в следующем:

Сначала создается экземпляр класса *Network*. Затем в этот экземпляр через функцию *readFrom* передается путь к файлу, на основе данных которого будет построена сеть.

В функции *readFrom* происходит «конвертация» данных в удобный для дальнейшей работы формат. Файл считывается построчно: каждая строка сохраняется в переменную *line* типа *string*, затем из строки удаляются все пробелы. После этого в массив *string* записываются подряд входные данные из текущей строки: *line[0]* – вершина V1, *line[1]* – вершина V2, все остальные символы – пропускная способность дуги P. Наконец, массив *string* сохраняется в список, и идет чтение следующей строки. В функции реализована проверка консистентности входных данных. Таким образом, если название вершины состоит больше, чем из одного символа; если пропускная способность не является десятичным числом; если обнаружен цикл; если дуга входит в исток, либо выходит из истока; если файл пуст, либо не существует; либо отсутствует исток или сток, то выбрсывается исключение с соответствующим сообщением.

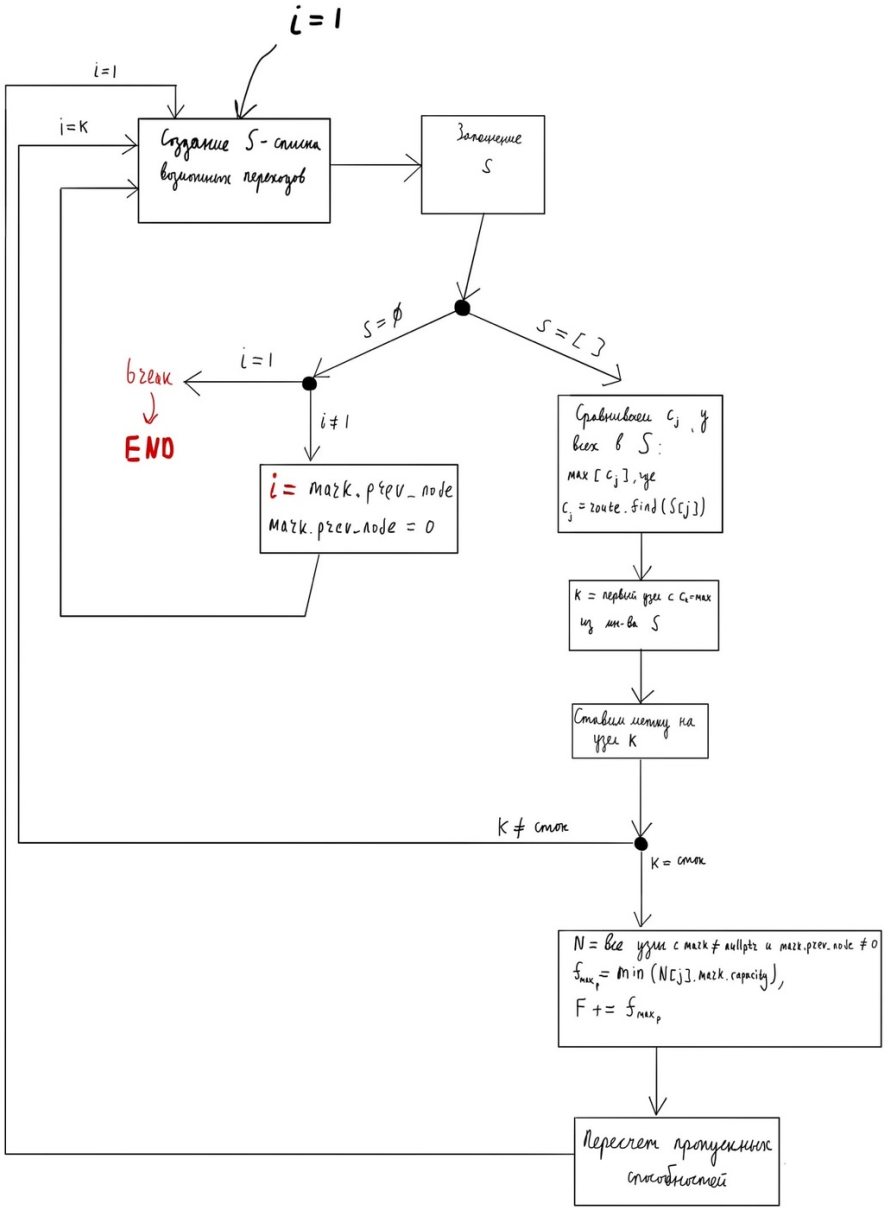
После того, как *readFrom* конвертировала данные из файла в список массивов *string*, вызывается функция *create*, которая (очевидно) строит сеть на основе этих данных.

В функции *create* происходит ~~магия~~ создание остаточной сети. Первым делом заполняется мапа *number_name_match*. В этой мапе хранятся соответствия между буквенным названием вершины и её номером. Исток всегда имеет номер 1, а сток всегда имеет последний номер. По окончании заполнения мапы создается список *vertices* для хранения вершин сети. Затем, снова проходя по каждой строке входных данных, идет либо создание вершины и заполнение её данных (если вершины нет в списке *vertices*), либо обновляется мапа путей для этой вершины (если вершина уже есть в списке *vertices*). *Стоит обратить внимание на то, что создание вершины/обновление путей для вершин V1 и V2 из входных данных немного различаются. Так как, фактически, мапа routes хранит смежную вершину и остаточную пропускную способность по данному ребру, то при обработке V2, когда для вершины заполняется/обновляется мапа routes, в соответствие V1 приходится ставить остаточную пропускную способность 0 (см. описание класса Vertex).*

В самом конце происходит вызов функции *ford_fulkerson*, которая ~~шаманит~~ обрабатывает построенную сеть и вычисляет максимальный поток.

Функция *ford_fulkerson* реализована по следующему алгоритму:

Цикл:



Пересчёт пропускных способностей:

cur = vertices.at(getSize-1) – получаем сток.

do

- Получаем номер предыдущего узла по метке
- Апдейт пропуск. способности : $c_{cur/prev} = c + f$
- Получаем пред. узел через его номер
- Апдейт пропуск. способности : $c_{prev/cur} = c - f$
- cur = vertices.at(prev-1)

while cur ≠ 0

В данной программе реализованы и используются следующие структуры данных:

- Структура *Mark*:
  Является оберткой для метки вершин графа. Данная структура используется, для повышения удобочитаемости кода и просто для того чтобы было легче оперировать с данными.

- Класс *Vertex*:
  Является оберткой для объектов – вершин графа. Хранит в себе номер вершины, указатель на метку, и указатель на мапу смежных вершин.
  Вместо самой метки вершина хранит указатель на неё, так как отсутствие у вершины метки легко обозначается как *nullptr*, что упрощает жизнь при написании/чтении кода.
  Для хранения смежных вершин используется ассоциативный массив, так как ассоциативный массив позволяет легко узнать является ли некоторая вершина *V* смежной к данной, а также имеется быстрый доступ к остаточной пропускной способности ребра по направлению от данной вершины к *V*.

- Класс *Network*:
  Является оберткой для объекта – сети. Хранит в себе максимальную пропускную способность, мапу соответствий буквенных имен вершин их номерам, список указателей на вершины сети, и список массивов *string* (входные данные).
  Причины использования именно ассоциативного массива для установки соответствия буквенных имен вершин их номерам очевидны.
  Для хранения вершин сети используется обычный двусвязный список, так как список позволяет легко получить нужный элемент по индексу. При этом вершины кладутся в список так, что на *i*-ом месте списка лежит указатель на *i+1*-ую вершину. Таким образом, можно спокойно получать нужную вершину, просто получая *i-1*-ый элемент списка. *Список заполняется именно так благодаря тому, что он хранит указатели на вершины: после заполнения мапы number_name_match, становится известно количество вершин в сети, соответственно список инициализируется как список размера number_name_match.getSize(), а все элементы списка имеют значение nullptr. В итоге при построении сети делается проверка нужного индекса, если там nullptr, то создается вершина и кладется по этому индексу, если не nullptr, то обновляется мапа путей.*

3. Оценка временной сложности методов

N – количество строк входной информации
V – количество вершин
E – количество ребер

1) ***readFrom*** имеет временную сложность O(*N*);

2) ***getInputData*** имеет временную сложность O(*1*);

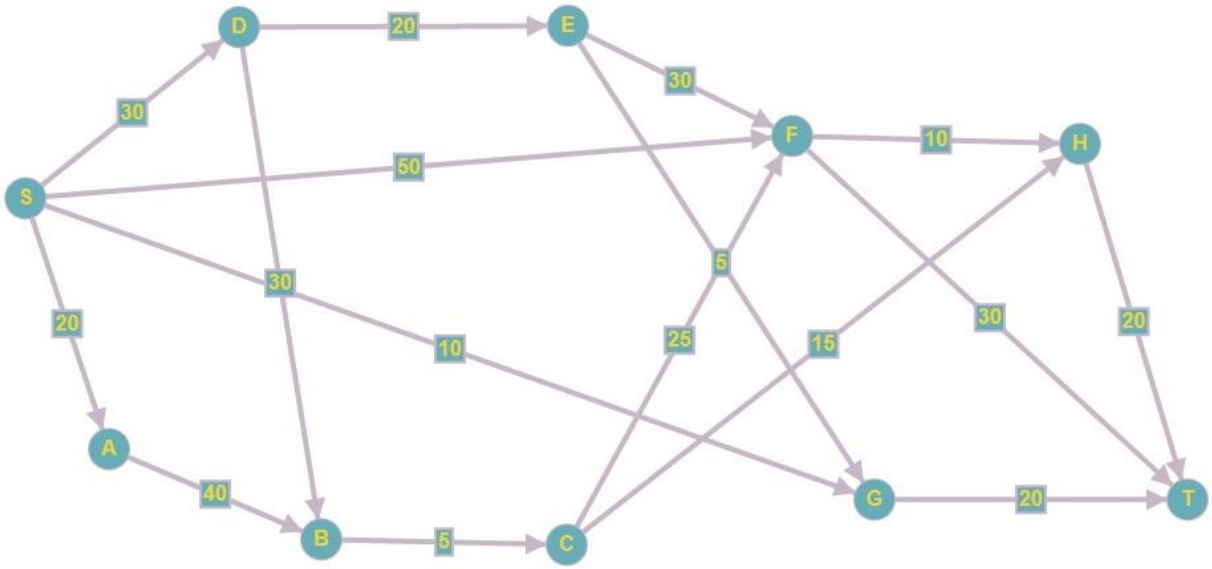
3) ***createVertex*** имеет временную сложность O(*1*);

4) ***create*** имеет временную сложность O($N^2 + NV + N*log(V)$);
   $log(V) + N*(N + log(V)) + N*(N+ log(V) + V) = N^2 + NV + N*log(V)$

5) ***recountBandwidths*** имеет временную сложность O($V*log(E)+V^3$);
1+ V(V + 2log(E) + 2log(E) + V) = V^2 + V*log(E)*
6) ***dijkstra*** имеет временную сложность O(*log(max[c])*(V^2 +VE + E^2 +E*log(E)+V*log(E))*);
log(max[c])(V + E + EV + E^2 + E*log(E) + V*log(E) + V^2) = log(max[c])*(V^2 +VE + E^2 +E*log(E)+V*log(E))*
прим.: max[c] – максимальная пропускная способность.

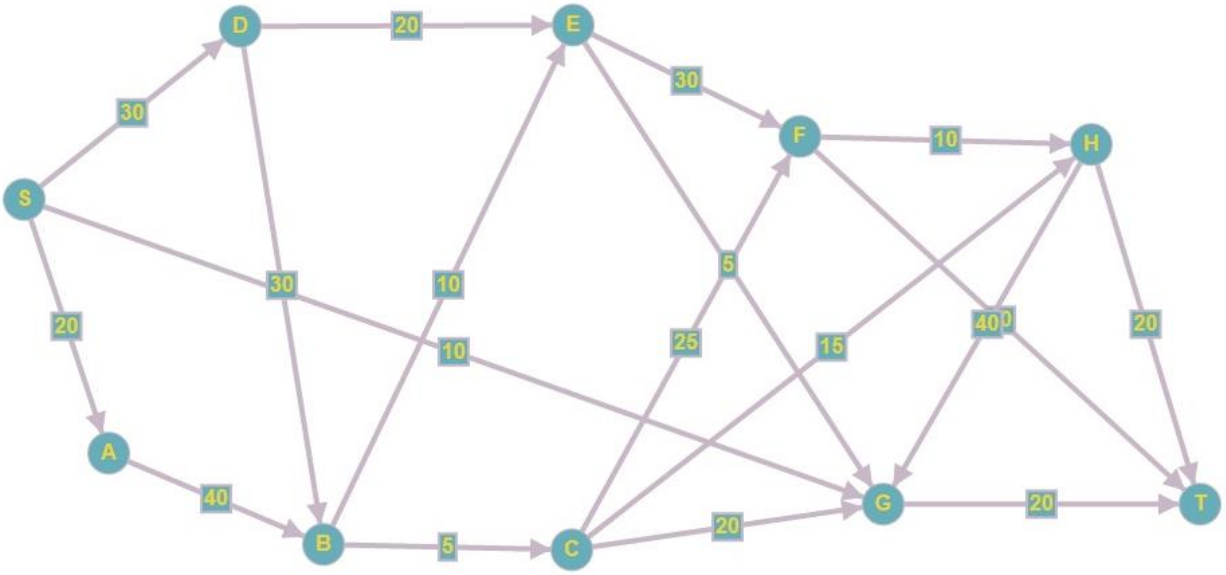
4. Примеры работы

1.



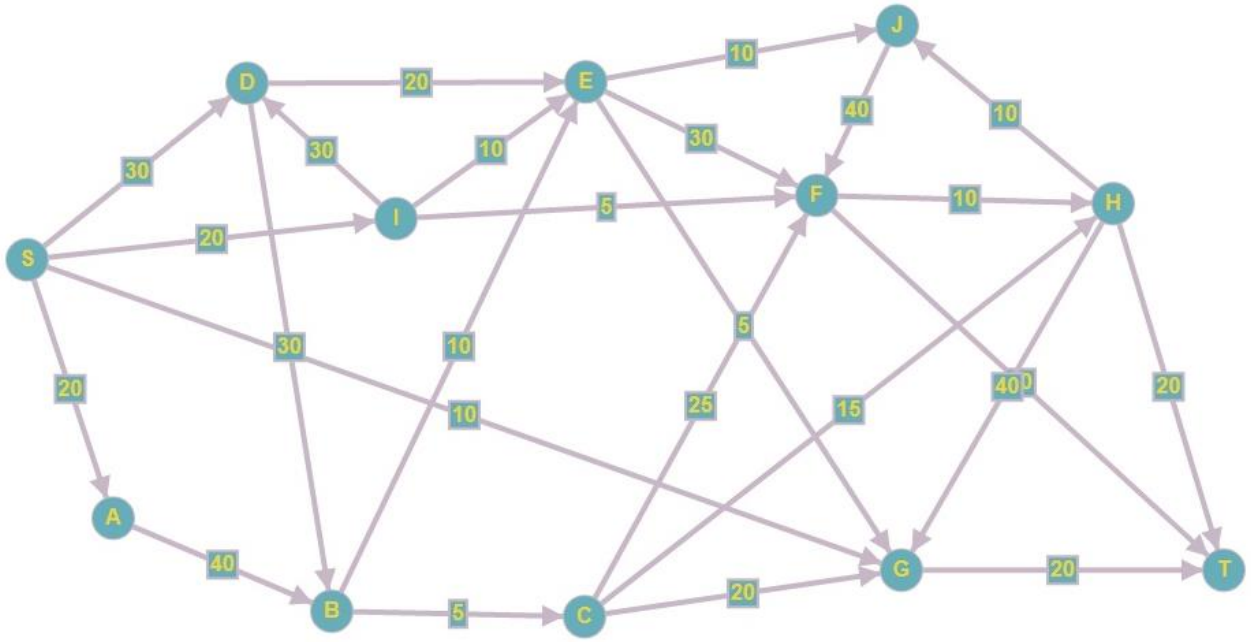

```
Maximal flow for the current network is 60
```

2.



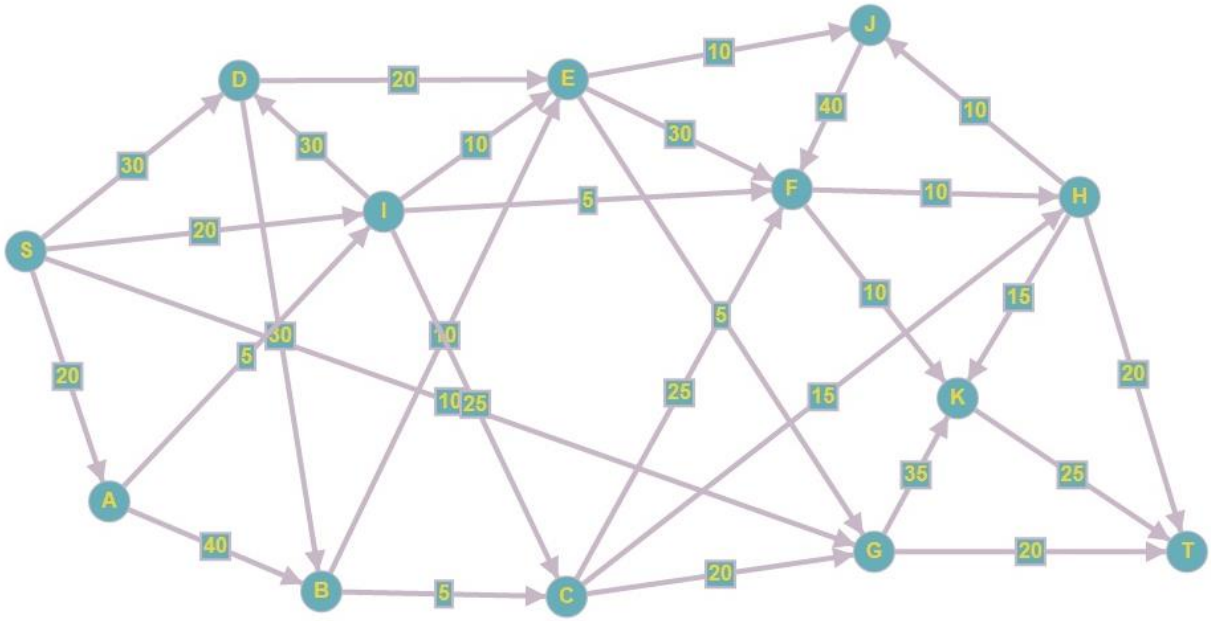

```
Maximal flow for the current network is 45
```

3.



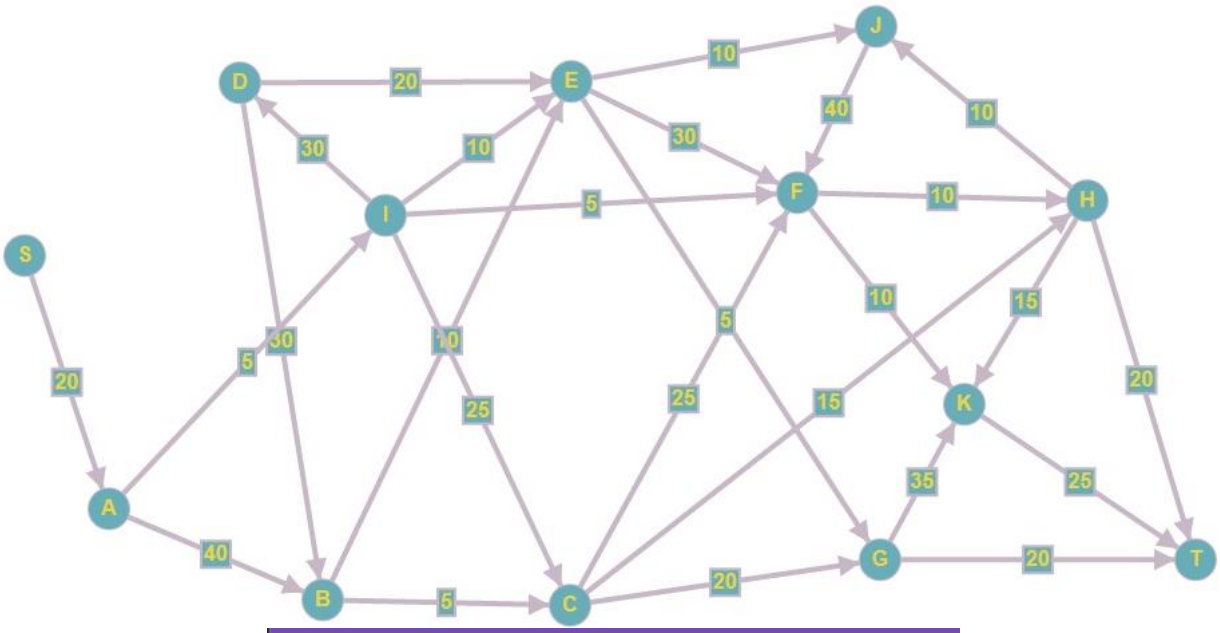

```
Maximal flow for the current network is 60
```

4.




```
Maximal flow for the current network is 65
```

5.




```
Maximal flow for the current network is 20
```

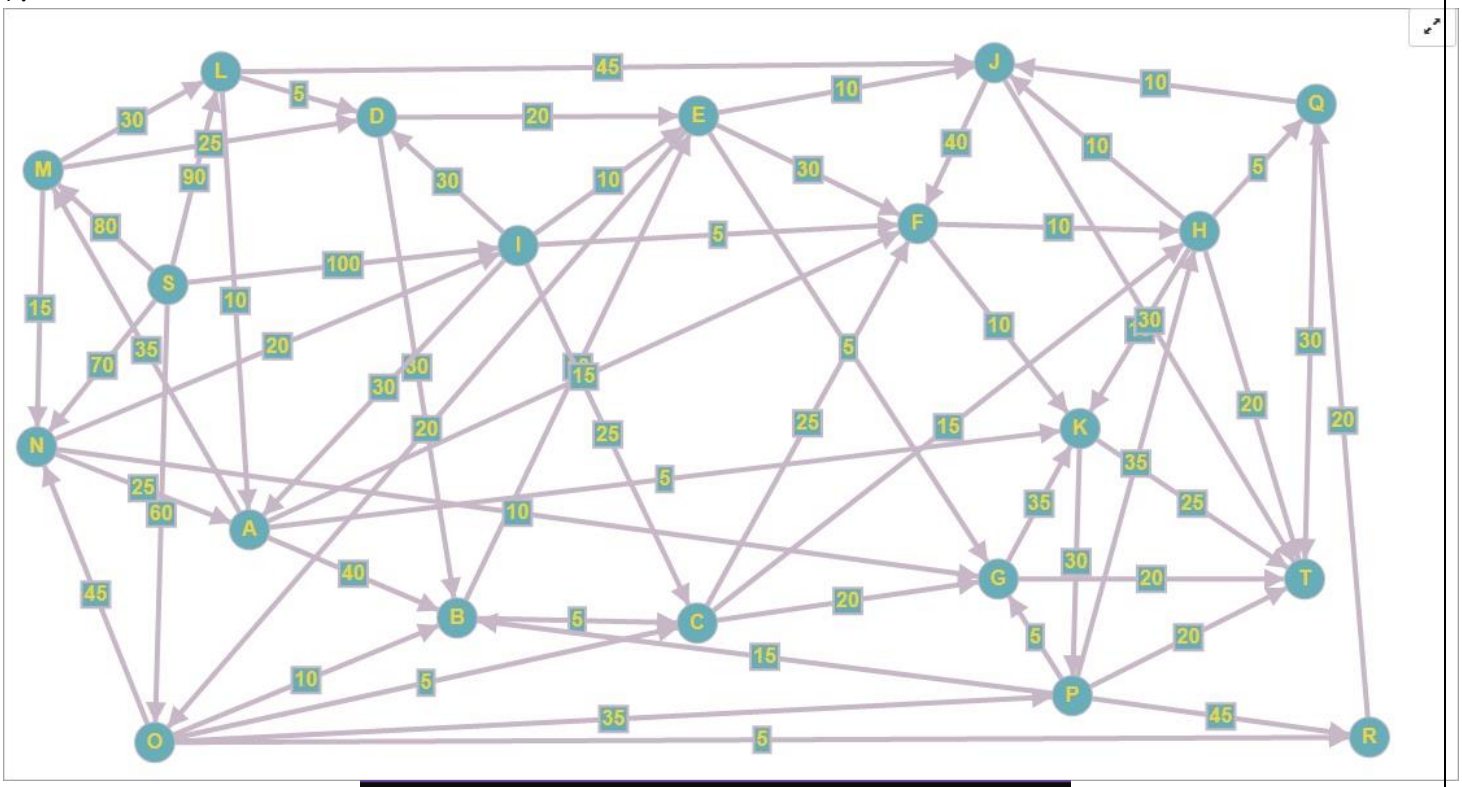
6.

Здесь я забыл сделать скриншот сети, поэтому вместо него будут данные из файла

```
S I 100
A B 40
A F 15
A K 5
B C 5
B E 10
C F 25
C H 15
C G 20
D B 30
D E 20
E G 5
E F 30
E J 10
F H 10
F K 10
G T 20
G K 35
H T 20
H J 10
H K 15
I A 30
I D 30
I E 10
I F 5
I C 25
J F 40
J T 30
K T 25
```

```
Maximal flow for the current network is 70
```

7.



5. Листинг

MAIN.CPP

```
#include <iostream>
#include "Network.h"

using std::cin;
using std::cout;
using std::endl;

int main()
{
	std::string filepath = "../TEST/test7.txt";
	Network network;
	network.readFrom(filepath);
	network.create();

	try
	{
		uint64_t maximal_flow = network.ford_fulkerson();
		cout << "Maximal flow for the current network is " << maximal_flow << endl << endl;
	}
	catch (std::exception& e)
	{
		cout << e.what() << endl << endl;
	}

	return 0;
}
```

NETWORK.H

```
#pragma once
#include <fstream>
#include <iostream>
#include <stdexcept>
#include <string>
#include "List.h"
#include "Queue.h"
#include "Map.h"
constexpr short INPUT_DATA_AMOUNT = 3;
constexpr uint64_t SOURCE = 1;
constexpr uint64_t INF = UINT64_MAX;

class Network
{
private:
	enum {V1, V2, C}; //start vertex, end vertex, flow capacity

	//represents node's marking for the ford-fulkerson algorithm
	struct Mark
	{
		uint64_t capacity;
		uint64_t prev_node;

		Mark(uint64_t capacity, uint64_t prev_node);

		~Mark();
	};

	//represents each vertex of the network
	class Vertex
	{
	public:
		uint64_t number; //vertex number
		Mark* mark;
		Map<uint64_t, uint64_t>* routes; //represents links with other nodes and remnant bandwidth
on the edge between them

		Vertex(uint64_t number, Map<uint64_t, uint64_t>* routes, Mark* mark = nullptr);

		~Vertex();
	};

	uint64_t maximal_flow; //value of maximal flow in the network
	Map<std::string, uint64_t> number_name_match; //represents match between node's name and number
	List<Vertex*> vertices; //stores all network's vertices; vertex number is always equals (list
index + 1)
	List<std::string*> input; //store's input lines

	Vertex* createVertex(uint64_t number, uint64_t neighbor, uint64_t capacity);
	void recountBandwidths(uint64_t f);

public:

	Network();

	Network(std::string& filepath);

	void readFrom(std::string& filepath); //reads file and interpets it as queue of char arrays

	List<std::string*> getInputData(); //returns input data converted to List of string Arrays

	void create(); //builds the network from input data

	uint64_t ford_fulkerson(); //counts maximal flow in the network using Ford-Fulkerson's algorithm
};
```

NETWORK.CPP

```
#include "Network.h"

/// FOR MARK ///
Network::Mark::~Mark() = default;

Network::Mark::Mark(uint64_t capacity, uint64_t prev_node)
	: capacity(capacity), prev_node(prev_node) { }


/// FOR VERTEX ///
Network::Vertex::~Vertex()
{
	delete routes;
	delete mark;
}

Network::Vertex::Vertex(uint64_t number, Map<uint64_t, uint64_t>* routes, Mark* mark)
	: number(number), mark(mark), routes(routes) { }


/// FOR PUBLIC IN NETWORK ///
Network::Network()
	: maximal_flow(0) { }


Network::Network(std::string& filepath)
	: maximal_flow(0)
{
	readFrom(filepath);
	create();
}


void Network::readFrom(std::string& filepath)
{
	bool start_exists = false, finish_exists = false; //for checking if network has source and runoff
	std::string line; //for storing one file line
	std::ifstream f(filepath);

	if (f.is_open())
	{
		while (std::getline(f, line))
		{
			auto arr = new std::string[INPUT_DATA_AMOUNT];

			if (line[1] != ' ' || line[3] != ' ') throw std::invalid_argument("wrong input
format: vertex name");

			for (unsigned i = 0; i < line.length(); ++i)
			{
				if (line[i] == ' ')
					line.erase(i, 1);
			}

			arr[V1] = line[V1];
			arr[V2] = line[V2];
			arr[C] = line.erase(0,2);

			try
			{
				std::stoi(arr[C]);
			}
			catch (std::invalid_argument& e)
			{
				throw std::invalid_argument("wrong input format: bandwidth");
			}
```

```
                        if (arr[V1] == arr[V2]) throw std::logic_error("loops are not allowed");
                        if (arr[V2] == "S") throw std::logic_error("ways to the source can not exist");
                        if (arr[V1] == "T") throw std::logic_error("ways from the runoff can not exist");

                        if (!start_exists)
                                if (arr[V1] == "S") start_exists = true;

                        if (!finish_exists)
                                if (arr[V2] == "T") finish_exists = true;

                        input.push_back(arr);
                }

                if (input.getSize() == 0) throw std::invalid_argument("file was empty");

                if (!start_exists) throw std::logic_error("source is missing");

                if (!finish_exists) throw std::logic_error("runoff is missing");
        }
        else
                throw std::exception("could not open the file");

        f.close();
}


List<std::string*> Network::getInputData()
{
        return input;
}


void Network::create()
{
        uint64_t last_added = 1; //number of last added node

        /* creating matches between nodes names and numbers */
        number_name_match.insert("S", 1);

        for (size_t i = 0; i < input.getSize(); ++i)
        {
                const auto line = input.at(i);

                for (auto j = 0; j < 2; ++j)
                {
                        auto cur = line[j]; //current symbol of input

                        if(!number_name_match.contains(cur))
                        {
                                if (cur != "T" && cur != "S")
                                {
                                        number_name_match.insert(cur, last_added + 1);
                                        last_added++;
                                }
                        }
                }
        }

        number_name_match.insert("T", last_added + 1);

        vertices = List<Vertex*>(number_name_match.getSize(), nullptr);

        /* creating network */
        for (size_t i = 0; i < input.getSize(); ++i)
        {
                const auto line = input.at(i);
```

```
			uint64_t first_vertex_num = number_name_match.find(line[V1]);
			uint64_t second_vertex_num = number_name_match.find(line[V2]);
			uint64_t capacity = std::stoi(line[C]);

			/* if vertex is not in the list yet */
			if (vertices.at(first_vertex_num - 1) == nullptr)
				vertices.set(first_vertex_num - 1, createVertex(first_vertex_num, second_vertex_num,
capacity));
			else
			{
				Vertex* vertex = vertices.at(first_vertex_num - 1);
				if (vertex->routes->contains(second_vertex_num))
					vertex->routes->update(second_vertex_num, capacity);
				else
					vertex->routes->insert(second_vertex_num, capacity);
			}

			if (vertices.at(second_vertex_num - 1) == nullptr)
				vertices.set(second_vertex_num - 1, createVertex(second_vertex_num,
first_vertex_num, 0));
			else
				vertices.at(second_vertex_num - 1)->routes->insert(first_vertex_num, 0);

		}
}

uint64_t Network::ford_fulkerson()
{
		uint64_t i = 1; //current vertex number (step 1)

		while(true)
		{
			auto current_vertex = vertices.at(i - 1);

			List<Vertex*> S; //list of possible transitions
			auto neighbors = current_vertex->routes->get_keys(); //list of all current's neighbors

			//filling the list of possible transitions (step 2)
			for (size_t j = 0; j < neighbors.getSize(); ++j)
			{
				auto neighbour_vertex = vertices.at(neighbors.at(j) - 1);

				/* if neighbor is not marked and remnant bandwidth is not zero */
				if (neighbour_vertex->mark == nullptr && current_vertex->routes-
>find(neighbour_vertex->number) != 0)
					S.push_back(neighbour_vertex);

			}

			/* counting next vertex (step 3) */
			if (!S.isEmpty())
			{
				uint64_t maximal_capacity = 0;
				Vertex* k = S.at(0); //vertex with maximal capacity in S

				for (size_t j = 0; j < S.getSize(); ++j)
				{
					auto current_capacity = current_vertex->routes->find(S.at(j)->number);

					if (current_capacity > maximal_capacity)
					{
						maximal_capacity = current_capacity;
						k = S.at(j);
					}
				}

				/* marking the vertex and reassign number of current vertex*/
				k->mark = new Mark(maximal_capacity, i);
```

```
                        i = k->number;

                        /* if i is not the runoff */
                        if (i != vertices.getSize())
                                continue; //goto step 2

                }
                else if (S.isEmpty()) // backup or finishing (step 4)
                {
                        /*i f there is no way out of the source - maximal flow is found */
                        if (i == SOURCE)
                                break;

                        /* back upping to the previous vertex to change other way*/
                        i = current_vertex->mark->prev_node;
                        current_vertex->mark->prev_node = 0;

                        continue; //goto step 2
                }

                /*if i is runoff (step 5)*/

                List<Vertex*> N; // list of all vertices on the path (all marked vertices excluding source
and back upped vertices)

                /* filling N */
                for (size_t j = 0; j < vertices.getSize(); ++j)
                        if (vertices.at(j)->mark != nullptr && vertices.at(j)->mark->prev_node != 0)
                                N.push_back(vertices.at(j));

                auto current_max_f = INF;

                /* finding minimal capacity */
                for (size_t j = 0; j < N.getSize(); ++j)
                        if (N.at(j)->mark->capacity < current_max_f)
                                current_max_f = N.at(j)->mark->capacity;

                maximal_flow += current_max_f;

                /* recounting remnant bandwidths*/
                recountBandwidths(current_max_f);

                for (size_t j = 1; j < vertices.getSize(); ++j)
                {
                        delete vertices.at(j)->mark;
                        vertices.at(j)->mark = nullptr;
                }

                i = 1;
        }

        return maximal_flow;
}


/// FOR PRIVATE IN NETWORK ///

Network::Vertex* Network::createVertex(uint64_t number, uint64_t neighbor, uint64_t capacity)
{
        auto routes = new Map<uint64_t, uint64_t>(neighbor, capacity);
        Mark* mark = nullptr;

        /* the source is being created with [INF; 0] marking*/
        if (number == SOURCE)
                mark = new Mark(INF, 0);

        return new Vertex(number, routes, mark);
}
```

```
void Network::recountBandwidths(uint64_t f)
{
	Vertex* current = vertices.at(vertices.getSize() - 1); // getting the runoff

	do
	{
		Vertex* prev = vertices.at(current->mark->prev_node - 1);
		current->routes->update(prev->number, current->routes->find(prev->number) + f);

		prev->routes->update(current->number, prev->routes->find(current->number) - f);

		current = vertices.at(prev->number - 1);
	}
	while (current->number != SOURCE);
}
```

Юнит-тесты

```
#include "pch.h"
#include "CppUnitTest.h"
#include "../Ford-Fulkerson algorithm/Network.h"
#include "../Ford-Fulkerson algorithm/Network.cpp"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace FordFalkersontest
{
	TEST_CLASS(FordFalkersontest)
	{
	public:

		TEST_METHOD(readFrom_success_test_1)
		{
			Network network;
			std::string filepath = "../UNIT_TEST/test1.txt";
			network.readFrom(filepath);
			auto result = network.getInputData();
			List<std::string*> expected;

			auto str = new std::string[INPUT_DATA_AMOUNT] {"S", "A", "2"};
			expected.push_back(str);

			str = new std::string[INPUT_DATA_AMOUNT]{ "S", "B", "3" };
			expected.push_back(str);

			str = new std::string[INPUT_DATA_AMOUNT]{ "A", "T", "4" };
			expected.push_back(str);

			str = new std::string[INPUT_DATA_AMOUNT]{ "B", "T", "6" };
			expected.push_back(str);

			for (size_t i = 0; i < result.getSize(); ++i)
				for (int j = 0; j < INPUT_DATA_AMOUNT; ++j)
					Assert::AreEqual(expected.at(i)[j], result.at(i)[j]);
		}

		TEST_METHOD(readFrom_exception_test_1)
		{
			Network network;
			std::string filepath = "../UNIT_TEST/test2.txt";
			try
			{
				network.readFrom(filepath);
			}
			catch(std::logic_error& e)
			{
				Assert::AreEqual(e.what(), "loops are not allowed");
```

```
	}
}

TEST_METHOD(readFrom_exception_test_2)
{
	Network network;
	std::string filepath = "../UNIT_TEST/test3.txt";
	try
	{
		network.readFrom(filepath);
	}
	catch (std::logic_error& e)
	{
		Assert::AreEqual(e.what(), "ways to the source can not exist");
	}
}

TEST_METHOD(readFrom_exception_test_3)
{
	Network network;
	std::string filepath = "../UNIT_TEST/test4.txt";
	try
	{
		network.readFrom(filepath);
	}
	catch (std::logic_error& e)
	{
		Assert::AreEqual(e.what(), "ways from the runoff can not exist");
	}
}

TEST_METHOD(readFrom_exception_test_4)
{
	Network network;
	std::string filepath = "../UNIT_TEST/test5.txt";
	try
	{
		network.readFrom(filepath);
	}
	catch (std::logic_error& e)
	{
		Assert::AreEqual(e.what(), "source is missing");
	}
}

TEST_METHOD(readFrom_exception_test_5)
{
	Network network;
	std::string filepath = "../UNIT_TEST/test6.txt";
	try
	{
		network.readFrom(filepath);
	}
	catch (std::logic_error& e)
	{
		Assert::AreEqual(e.what(), "runoff is missing");
	}
}

TEST_METHOD(readFrom_exception_test_6)
{
	Network network;
	std::string filepath = "../UNIT_TEST/test7.txt";
	try
	{
		network.readFrom(filepath);
	}
	catch (std::invalid_argument& e)
```

```
		{
			Assert::AreEqual(e.what(), "file was empty");
		}
	}

	TEST_METHOD(readFrom_exception_test_7)
	{
		Network network;
		std::string filepath = "../UNIT_TEST/testEMPTY.txt";
		try
		{
			network.readFrom(filepath);
		}
		catch (std::exception& e)
		{
			Assert::AreEqual(e.what(), "could not open the file");
		}
	}

	TEST_METHOD(readFrom_exception_test_8)
	{
		Network network;
		std::string filepath = "../UNIT_TEST/test9.txt";
		try
		{
			network.readFrom(filepath);
		}
		catch (std::invalid_argument& e)
		{
			Assert::AreEqual(e.what(), "wrong input format: vertex name");
		}
	}

	TEST_METHOD(readFrom_exception_test_9)
	{
		Network network;
		std::string filepath = "../UNIT_TEST/test10.txt";
		try
		{
			network.readFrom(filepath);
		}
		catch (std::invalid_argument& e)
		{
			Assert::AreEqual(e.what(), "wrong input format: bandwidth");
		}
	}

	TEST_METHOD(ford_fulkerson_success_test_1)
	{
		Network network;
		std::string filepath = "../UNIT_TEST/test8.txt";
		network.readFrom(filepath);
		network.create();
		uint64_t f = network.ford_fulkerson();
		Assert::AreEqual(uint64_t(60), f);
	}

	TEST_METHOD(ford_fulkerson_success_test_2)
	{
		std::string filepath = "../UNIT_TEST/test8.txt";
		Network network(filepath);
		uint64_t f = network.ford_fulkerson();
		Assert::AreEqual(uint64_t(60), f);
	}
};
}
```

6. Входные данные для примеров

1.	2.	3.	4.	5.	6.	7.
S A 20 S D 30 S G 10 S F 50 A B 40 B C 5 C F 25 C H 15 D B 30 D E 20 E G 5 E F 30 F H 10 F T 30 G T 20 H T 20	S A 20 S D 30 S G 10 A B 40 B C 5 B E 10 C F 25 C H 15 C G 20 D B 30 D E 20 E G 5 E F 30 F H 10 F T 30 G T 20 H G 40 H T 20	S A 20 S D 30 S G 10 S I 20 A B 40 B C 5 B E 10 C F 25 C H 15 C G 20 D B 30 D E 20 E G 5 E F 30 E J 10 F H 10 F T 30 G T 20 H G 40 H T 20 H J 10 I D 30 I E 10 I F 5 J F 40	S A 20 S D 30 S G 10 S I 20 A B 40 A I 5 B C 5 B E 10 C F 25 C H 15 C G 20 D B 30 D E 20 E G 5 E F 30 E J 10 F H 10 F K 10 G T 20 G K 35 H T 20 H J 10 H K 15 I D 30 I E 10 I F 5 I C 25 J F 40 K T 25	S A 20 A B 40 A I 5 B C 5 B E 10 C F 25 C H 15 C G 20 D B 30 D E 20 E G 5 E F 30 E J 10 F H 10 F K 10 G T 20 G K 35 H T 20 H J 10 H K 15 I D 30 I E 10 I F 5 I C 25 J F 40 K T 25	S I 100 A B 40 A F 15 A K 5 B C 5 B E 10 C F 25 C H 15 C G 20 D B 30 D E 20 E G 5 E F 30 E J 10 F H 10 F K 10 G T 20 G K 35 H T 20 H J 10 H K 15 I A 30 I D 30 I E 10 I F 5 I C 25 J F 40 J T 30 K T 25	S I 100 S L 90 S M 80 S N 70 S O 60 A B 40 A F 15 A K 5 A M 35 B C 5 B E 10 C F 25 C H 15 C G 20 D B 30 D E 20 E G 5 E F 30 E J 10 E O 20 F H 10 F K 10 G T 20 G K 35 H T 20 H J 10 H K 15 H Q 5 I A 30 I D 30