**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: **Алгоритмы поиска максимального потока в графе**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9304 |  |  |
| Преподаватель |  | Шевская Н.В. |

Санкт-Петербург

2021

## Цель работы.

Ознакомиться с алгоритмами поиска максимального в ориентированном графе. Реализовать алгоритм Форда-Фалкерсона поиска максимального потока с выбранной индивидуализацией.

## 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

v0 – исток

vn – сток

- ребро графа

…

Выходные данные:

– величина максимального потока

- 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).

Алгоритм работает согласно алгоритму Форда-Фалкерсона: производится поиск такого пути между необходимыми вершинами, что минимум разности пропускной способности ребра и величины потока среди всех рёбер найденного пути больше 0; затем для каждого ребра найденного пути к значению потока прибавляется найденный минимум. Процедура повторяется до тех пор, пока в сети не останется путей, удовлетворяющих условию.

Сложность реализованного алгоритма заключается в построении процедуры для отбора рёбер и составления путей между стоком и истоком графа.

Алгоритм в бесконечном цикле вызывает функцию, которая находит новые пути в сети, через которые можно пропустить поток, чтобы прибавить минимальное значение разности пропускной способности и потока в ребре ко всем рёбрам пути (или отнимает, в зависимости от направления ребра, см. ниже) и к значению потока через сеть в целом. Выход из бесконечного цикла осуществляется, когда вызываемая функция не находит таких путей.

Функция для нахождения путей в сети в бесконечном цикле вызывает функцию, которая находит новое ребро пути для последнего ребра в частичном пути. Выход из бесконечного цикла осуществляется, когда функция больше не может обнаружить пути из истока. Если функция получает новое ребро, то она добавляет его к основным рёбрам и проверяет, является ли частичный путь полным (соединяет исток и сток сети). Функция может не получить ребро; это означает, что данное ребро не является частью нового пути от истока к стоку, если он существует, и, следовательно, необходимо удалить последнее ребро в частичном пути.

Функция для нахождения ребра для последней вершины в частичном пути определяет новое ребро для текущего частичного пути, проверяя рёбра в порядке, приведённом ниже (*«фильтры»*).

* Содержат вершину последнего ребра сети.
* Не были посещены рёбра в текущей итерации поиска пути в сети.
* Не были посещены вершины в текущей итерации поиска пути в сети.
* Не заполнены (или не пусты для обратного ребра).

Если в конце проверок в контейнере имеется более чем 1 элемент, то выбирается элемент, который соответствует индивидуализации лабораторной работы.

Функция может обращать рёбра, прокладывая путь в сторону, противоположную заданной. Это возможно лишь тогда, когда поток через это ребро не равен нулю. При прохождении по рёбрам сети в противоположную сторону, производится вычитание результирующего потока через сеть по рёбрам найденного пути. Это может означать неоптимальность путей, которые были выбраны на предыдущих итерациях поиска путей в сети.

Функции обращаются к контейнерам, которые хранят все рёбра сети, текущий частичный путь сети, посещённые вершины в текущем частичном пути и возможные рёбра для частичного пути.

## Выполнение работы.

Для построения алгоритма были реализованы следующие классы.

1. Класс *GraphComponent* представляет собой ребро ориентированного графа.

Класс содержит следующие публичные поля.

• *char StartNode, EndNode* − имена вершин, соответствующих началу и концу ребра сети.

• *int flowCapacity, flowCurrent* − пропускная способность ребра и поток через данное ребро (по умолчанию 0).

• *bool isFull, isEmpty, isReversed* − флаги для удобного взаимодействия с объектами данного класса; данным переменным соответствуют максимальной заполненности потока в ребре, отсутствию потока в ребре и противоположное направление ребра.

Класс содержит следующие публичные методы.

• Конструктор класса принимает на вход переменные, соответствующие публичным полям класса *StartNode, EndNode, flowCapacity* и устанавливает их и зависящие от этих величин флаги на на соответствующие значения.

• Деструктор класса, пуст.

• *int getPossibleFlow()* возвращает значение потока, которое можно пропустить или отнять у ребра, в зависимости от его направления в пути.

• *void addFlow()* добавляет или отнимает значение потока у ребра, в зависимости от его направления в пути.

Класс *Graph* хранит все данные о графе и содержит необходимые методы для реализации алгоритма.

Класс содержит следующие публичные поля.

• *char globalStart, globalEnd* − имена вершин, соответствующих истоку и стоку сети.

• *int globalFlow = 0* − значение максимального потока сети.

• *vector<GraphComponent\*> componentVector, visitedComponentVector, WayVector, componentCandidates* − контейнеры (все рёбра сети; все посещённые рёбра сети в пределах одного пути; все рёбра, составляющие

путь; рёбра перед «*фильтром*» функции, проверяющей рёбра), хранящие

указатели на объекты рёбер для удобного взаимодействия в программном коде.

Класс содержит следующие публичные методы.

• Конструктор устанавливает поля начальной и конечной вершины в соответствии с заданными входными данными.

• Деструктор освобождает всю память, выделенную на хранение рёбер сети.

• *void push\_back(GraphComponent\* component)* добавляет новый компонент в вектор непосещённых рёбер сети. Метод используется при чтении входных данных.

• *bool isUniqueWayNode, isVisitedComponent* проверяют контейнеры с элементами пути и посмотренными компонентами и возвращают *true,* если вершина в текущем пути встречается 1 раз и ребро было посещено соответственно.

• *GraphCompont\* getBestWayComponent (char startNewNode)* производит поиск подходящего ребра для последней вершины частичного пути. Метод возвращает указатель, если подходящее ребро было найдено, и *nullptr,* если ребро подходящее не было найдено. Этот метод реализует индивидуализацию лабораторной работы.

• *bool findWay()* ищет путь, который соответствует условиям поиска пути (см. описание алгоритма). Возвращает *true,* если путь был найден.

• *void addWayFlow()* выполняет операцию по добавлению потока в каждое ребро (с учётом его направления в найденной пути в сети).

• *void findGraphFlow()* вызывает методы для нахождения рёбер, путей и потоков до тех пор, пока в сети существуют пути (см. описание алгоритма).

Для построения алгоритма были реализованы следующие функции.

*• bool compLexicographic(GraphComponent\* a, GraphComponent\* b)*

представляет собой компаратор для сортировки элементов вектора в лексикографическом порядке для корректного вывода ответа. Функция возвращает *true,* если значение слева расположено лексикографически раньше, чем значение справа, и *false* в остальных случаях.

*• bool compSpecial(GraphComponent\* a, GraphComponent\* b)*

представляет собой компаратор для сравнения указателей на класс рёбер сети. Является главной частью реализации индивидуализации лабораторной работы. Функция возвращает *true,* если расстояние между символами вершин (по таблице ASCII) минимально, либо, при равенстве, конечная вершина ребра слева расположена лексикографически раньше, чем аналогичное значени справа, и *false* в остальных случаях.

*• int main* представляет собой главную функцию программы. Функция считывает с входного потока сначала количество рёбер, имя начальной и конечной вершины между которыми требуется найти максимальный поток, а затем наборы данных о рёбрах ориентированного графа. Функция создаёт экземпляр класса *Graph*, находит максимальный поток и выводит данные на экран. В зависимости от состояния директивы «*define*» в начале кода, программа будет выводить промежуточные данные в процессе работы.

Разработанный программный код см. в приложении А.

Результаты тестирования см. в приложении Б.

## Выводы.

Был успешно реализован алгоритм Форда-Фалкерсона для нахождения максимального потока в графе.

# Приложение А Исходный код программы

Название файла: main.cpp.

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

class GraphComponent{

public:

char startNode, endNode;

int flowCapacity, flowCurrent = 0;

bool isFull = false, isEmpty = true, isReversed = false;

GraphComponent(char startNode, char endNode, int flowCapacity) : startNode(startNode), endNode(endNode), flowCapacity(flowCapacity){

if(flowCapacity == 0)

isFull = true;

};

~GraphComponent()= default;;

int getPossibleFlow(){

return isReversed ? flowCurrent : flowCapacity - flowCurrent;

};

void addFlow(int additionalFlow){

if(isReversed)

flowCurrent -= additionalFlow;

else

flowCurrent += additionalFlow;

isFull = (flowCurrent == flowCapacity);

isEmpty = (flowCurrent == 0);

};

};

bool compLexicographic(GraphComponent\* a, GraphComponent\* b){

char aStart = a->startNode, aEnd = a->endNode, bStart = b->startNode, bEnd = b->endNode;

return ( (aStart<bStart) || ((aStart==bStart)&&(aEnd<bEnd)) );

}

bool compSpecial(GraphComponent\* a, GraphComponent\* b){

if(a == nullptr)

return false;

else if(b == nullptr)

return true;

int aStart = int(a->startNode), aEnd = int(a->endNode), bStart = int(b->startNode), bEnd = int(b->endNode);

if(a->isReversed)

swap(aStart, aEnd);

if(b->isReversed)

swap(bStart, bEnd);

return ( (abs(aStart-aEnd)<abs(bStart-bEnd)) || ((abs(aStart-aEnd)==abs(bStart-bEnd))&&(aEnd<bEnd)) );

}

class Graph{

public:

char globalStart, globalEnd;

int globalFlow = 0;

vector<GraphComponent\*> componentVector, visitedComponentVector, wayVector, componentCandidates;

Graph(char globalStart, char globalEnd) : globalStart(globalStart), globalEnd(globalEnd){};

~Graph(){

for(auto & i : componentVector)

delete i;

};

void push\_back(GraphComponent\* component){

componentVector.push\_back(component);

};

void printAnswer(){

printf("%d\n", globalFlow);

for(auto & i : componentVector)

printf("%c %c %d\n", i->startNode, i->endNode, i->flowCurrent);

};

bool isUniqueWayNode(GraphComponent\* component){

char checkNode = component->isReversed ? component->startNode : component->endNode;

if(wayVector.empty()){

if(globalStart == checkNode)

return false;

return true;

} else {

for(auto & i : wayVector)

if( ((i->startNode == checkNode) && (!i->isReversed)) || ((i->endNode == checkNode) && (i->isReversed)) )

return false;

return true;

}

};

bool isVisitedComponent(GraphComponent\* component){

for(auto & i : visitedComponentVector)

if(i == component)

return true;

return false;

};

GraphComponent\* getBestWayComponent(char startNewNode){

GraphComponent\* componentReturn = nullptr;

componentCandidates.clear();

for(auto & i : componentVector)

if((i->startNode == startNewNode)||(i->endNode == startNewNode))

componentCandidates.push\_back(i);

for(int i = 0; i < componentCandidates.size(); i++)

if(isVisitedComponent(componentCandidates[i])){

componentCandidates.erase(componentCandidates.begin() + i);

i--;

}

for(int i = 0; i < componentCandidates.size(); i++){

if(componentCandidates[i]->endNode == startNewNode)

componentCandidates[i]->isReversed = true;

if(!isUniqueWayNode(componentCandidates[i])){

componentCandidates[i]->isReversed = false;

visitedComponentVector.push\_back(componentCandidates[i]);

componentCandidates.erase(componentCandidates.begin() + i);

i--;

}

}

for(int i = 0; i < componentCandidates.size(); i++)

if(((componentCandidates[i]->isReversed) && (componentCandidates[i]->isEmpty)) || (!(componentCandidates[i]->isReversed) && (componentCandidates[i]->isFull))){

componentCandidates[i]->isReversed = false;

componentCandidates.erase(componentCandidates.begin() + i);

i--;

}

for(int i = 0; i < componentCandidates.size(); i++)

if(compSpecial(componentCandidates[i], componentReturn)){

if(componentReturn != nullptr)

componentReturn->isReversed = false;

componentReturn = componentCandidates[i];

} else {

componentCandidates[i]->isReversed = false;

}

if(componentReturn != nullptr)

visitedComponentVector.push\_back(componentReturn);

return componentReturn;

};

bool findWay(){

GraphComponent\* newWayComponent;

visitedComponentVector = wayVector;

while(true){

newWayComponent = getBestWayComponent(

wayVector.size() == 0 ? globalStart : (

wayVector.back()->isReversed ? wayVector.back()->startNode : wayVector.back()->endNode

));

if(newWayComponent == nullptr){

if(wayVector.empty()){

return false;

} else {

wayVector.pop\_back();

}

} else {

wayVector.push\_back(newWayComponent);

if(wayVector.back()->endNode == globalEnd)

return true;

}

}

};

void addWayFlow(){

int wayFlow = wayVector[0]->getPossibleFlow(), firstComponentIndexDrop = 0;

for(int i = 1; i < wayVector.size(); i++)

if(wayFlow > wayVector[i]->getPossibleFlow()){

wayFlow = wayVector[i]->getPossibleFlow();

firstComponentIndexDrop = i;

}

globalFlow += wayFlow;

for(auto & i : wayVector){

i->addFlow(wayFlow);

}

for(int i = wayVector.size()-1; i >= firstComponentIndexDrop; i--){

wayVector.back()->isReversed = false;

wayVector.pop\_back();

}

visitedComponentVector = wayVector;

}

void findGraphFlow(){

sort(componentVector.begin(), componentVector.end(), compLexicographic);

while(findWay()){

addWayFlow();

}

};

};

int main(){

char globalStartNodeChar, globalEndNodeChar;

char startNodeChar, endNodeChar;

int componentCount, flowCapacityValue;

cin >> componentCount >> globalStartNodeChar >> globalEndNodeChar;

Graph graph(globalStartNodeChar, globalEndNodeChar);

for(int i = 0; i < componentCount; i++){ // adding all componetns

cin >> startNodeChar >> endNodeChar >> flowCapacityValue;

graph.push\_back(new GraphComponent(startNodeChar, endNodeChar, flowCapacityValue));

}

graph.findGraphFlow(); // Ford-Fulkerson

graph.printAnswer(); // output result

return 0;

}