**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №4**

**по дисциплине «Построение и анализ алгоритмов»**

Тема: Алгоритм Форда-Фалкерсона

|  |  |  |
| --- | --- | --- |
| Студент гр. 6304 |  | Корытов П.В. |
| Преподаватель |  | Балтрашевич В.Э. |

Санкт-Петербург

2018

**Цель работы.**

Изучить алгоритм Форда-Фалкерсона и его реализацию на языке C++

**Основные теоретические положения.**

**Сеть -** ориентированный граф , у которого есть взвешенные ребра. У каждого ребра есть сопоставленная ему пропускная способность - capacity. Граф имеет один сток и исток

**Поток -** подграф графа, который имеет пропускную способность, не больше, чем пропускная способность этого графа. Поток в графе - один из возможных путей распространения вещества от истока к стоку.

Формальное определение поток - функция

* Пропускные способности не нарушены:
* Кососимметричность:
* Сохранение потока:

 Задача - найти максимальный поток в графе.

**Метод Форда-Фалкерсона**

* Есть сеть и поток Будем называть **остаточность пропускной способностью**  разность его пропускной способности и пропускной способности потока.
* **Остаточной сетью**, которая порождена потоком, мы будем называть такую сеть, порожденную из тех же вершин, ребра которой будут иметь только остаточную пропускную способность.
* **Дополняющий путь** в сети - обычный путь из истока в сток в остаточной сети.

Если в графе есть хотя бы один дополняющий путь, текущий построенный поток не максимален.

* **Пропускная способность пути** - минимальная пропускная способность ребер пути.
* **Остаточная пропускная способность дополняющего пути** - наибольший поток, который мы можем пропустить по этому пути.
* **Разрез** графа - разделение графа на две части
* **Пропускная способность разреза** - суммарная пропускная способность ребер - мостов, соединяющих разрез.
* **Минимальный разрез** - разрез с минимальной пропускной способностью.

Поток через любой разрез совпадает с величиной самого потока.

Следующие утверждения равносильны:

* Поток максимален
* Остаточная сеть не содержит дополняющих путей
* Существует такой разрез, для которого пропускная способность потока равна пропускной способности разреза.

Сам алгоритм (точнее, его интерпретация, используемая в данной работе) выглядит следующим образом:

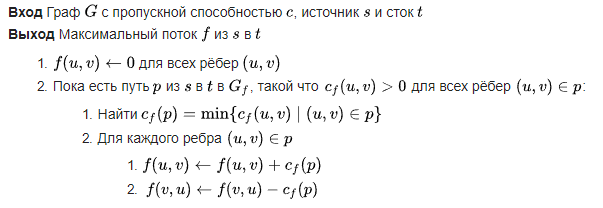


Рисунок 1 – Алгоритм Форда-Фалкерсона

**Ход работы**

1. Основа для работы взята из лабораторной №3 – в ней уже реализованы возможности построения графа и работы с ним. Существенные изменения, перечислены ниже:
   * Добавлен поиск истока ребра по указателю на ребро
   * Расширены возможности отображения пути на графе и его хранения
   * Изменено отображение весов – теперь вместе с весами отображается остаточная пропускная способность
   * Добавлена анимация для путей
   * Добавлены функции, необходимые для работы алгоритма
2. Как и прежде, алгоритм изменен – за один заход в процедуру он делает один шаг.

Процедура инициализации алгоритма выглядит следующим образом:

void Graph::**FordFalkersonInit**()

{

if (!sink || !source){

QMessageBox box;

box.setText("Не задан сток или исток");

box.*exec*();

return;

}

if (!weights){

QMessageBox box;

box.setText("Граф не взвешенный");

box.*exec*();

return;

}

max\_iter = 15;

}

Фактически, это всего лишь проверка на корректность введенных параметров.

Алгоритм выглядит так:

int Graph::**FordFalkerson**()

{

if (max\_iter == 0)

FordFalkersonInit(); //Запуск инициализации, если надо

if (max\_iter == 0)

return 1; //Если не инициализировалось

it\_num++;

ResetFindPath(); //Сброс поиска в ширину

Elem\* res = FindPath(); //Поиск в ширину

if (res == nullptr)

return 1; //Если путь не найден

RestorePath(sink); //Восстановление пути

SetMinPath(); //Установка минимального потока

int i = 0;

for (auto it : path.way){ //Итерация по всем ребрам пути

it->flow = it->flow + path.min\_flow;

Elem\* temp = path.way\_el.at(i++);

List\* templist = GetEdge(it->node, temp);

if (templist)

templist->flow = it->flow;

}

return 0;

}

После проверки на корректность инициализации алгоритм с помощью поиска в ширину ищет кратчайший поток от истока к стоку. Если путь найден, то для всех ребер пути устанавливает поток, равный минимальному весу ребра всего пути.

Поиск в ширину выглядит так:

Elem \*Graph::**FindPath**()

{

SAVEITS;

QQueue<Elem\*> queue; //Очередь для поиска

queue.enqueue(source); //Загрузка стока в очередь

source->searched = 1; //Пометка

List\* ls;

while(!queue.isEmpty()){ //Цикл, пока очередь не пуста

Elem\* el2 = queue.dequeue(); //Взять один элемент

if (el2 == sink){

RESTOREITS;

return el2 ;//Путь найден, конец

}

while (ls = it(el2)){ //Итерация по всем ребрам

if ((!ls->node->searched) && ((ls->weight - ls->flow) != 0)){

ls->node->searched = 1; //Добавление и пометка всех

ls->mark2 = 1; //Непомеченных соседей

queue.enqueue(ls->node);

}

}

}

RESTOREITS;

return nullptr; //Путь не найден

}

И восстановление пути после поиска:

void Graph::**RestorePath**(Elem\* el)

{

SAVEITS;

List\* ls;

int pos = rand() % 100; //Настройка анимации для пути

path.way\_el.push\_front(el); //Записать текущий элемент в конец пути

if (el!=source){ //Если это не исток

linpos = 0; //Сброк итератора через входящие ребра

while (ls = itin(el)){ //Итератор через входящие ребра

if (ls->mark2){ //Если ребро помечено, значит через него был поиск

path.way.push\_front(ls); //Записать в путь

ls->mark = it\_num; //Пометить

ls->edge->AddAnimation(pos); //Установка анимации

ls->edge->update(); //Обновление

RestorePath(FindElem(ls)); //Рекурсия для следующего элемента

RESTOREITS;

return;

}

}

}

else{ //Исток найден

Elem\* el2;

while ((el2 = it())!=nullptr){ //Итерация через все ребра

while ((ls = it(el2))!=nullptr){

ls->mark2 = 0; //Сброс пометок

}

}

}

RESTOREITS;

}

1. Для работы с анимацией в класс Edge добавлена структура, хранящая параметры для объекта анимации:

typedef struct MovingElliplse{

QGraphicsEllipseItem\* Ellipse;

int position = 0;

}MovingElliplse;

После чего создана процедура, добавляющая анимированный круг на ребро:

void Edge::**AddAnimation**(int pos)

{

QColor col = Qt::black;

if (list->mark!=0){

double hue = (list->mark - 1);

hue = hue\*360;

hue = hue/(graph->gr->max\_iter);

col.setHsv(hue, 255, 255);

}

QGraphicsEllipseItem\* a = this->scene()->addEllipse(0, 0, 10, 10);

a->setParentItem(this);

a->setPos(sourcePoint);

a->setPen(QPen(col, 0));

a->setBrush(QBrush(col));

a->update();

MovingElliplse str;

str.Ellipse = a; str.position = pos;

Anim.push\_back(str);

}

Затем процедура, удаляющая все анимированные части с ребра:

void Edge::**RemoveAnimation**()

{

for (int e = 0; e < Anim.size(); e++){

delete Anim[e].Ellipse;

}

Anim.clear();

}

Деструктор QGraphicEllipseItem корректно освобождается от родителя, т.к. родитель явно задан в AddAnimation. Эта связь работает и в обратную сторону – при уничтожении ребра в анимированные объекты пропадут.

Наконец, добавлена процедура, обрабатывающая анимацию:

void Edge::**ProcessAnimation**()

{

for (int e = 0; e < Anim.size(); e++){

if (Anim[e].position == 100)

Anim[e].position = 0;

Anim[e].position++;

qreal coef = 1 - (qreal)(Anim[e].position)/100;

int xs = sourcePoint.x() - 5;

int ys = sourcePoint.y() - 5;

int xd = destPoint.x() - 5;

int yd = destPoint.y() - 5;

qreal newx = xs\*coef + xd \* (1-coef);

qreal newy = ys\*coef + yd \* (1-coef);

Anim[e].Ellipse->setX(newx);

Anim[e].Ellipse->setY(newy);

Anim[e].Ellipse->update();

}

}

В классе GraphWidget добавлен слот, обрабатывающий все анимации ребёр:

void GraphWidget::**Animation**()

{

foreach (QGraphicsItem \*item, scene()->items()) {

if (Edge \*edge = qgraphicsitem\_cast<Edge\*>(item))

edge->ProcessAnimation();

}

}

И уже этот слот подсоединен к сигналу таймера

AnimationTimer = new QTimer(this);

AnimationTimer->start(10);

connect(AnimationTimer, SIGNAL(timeout()), this, SLOT(Animation()));

1. Проведено тестирование программы на простом примере

|  |  |  |
| --- | --- | --- |
| Шаг | Результат | Вид графа |
|  |
| 0 |  |  |
| 1 | 2: 0 1 3 |  |
| 2 | 2: 0 1 3  4: 0 2 3 |  |

Как видно, алгоритм успешно завершил работу. Получился суммарный поток длины 6.

1. Взят пример из курса дискретной математики (файл Chukhnov.txt):

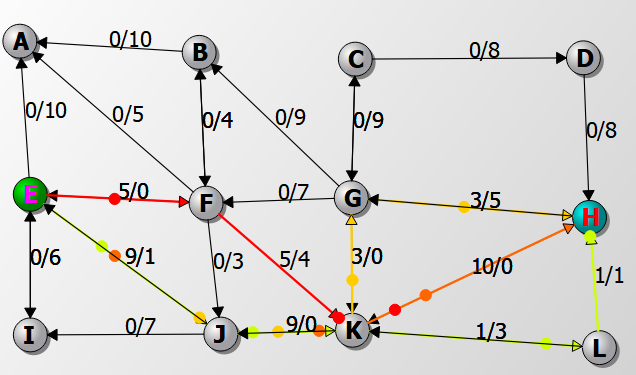


Рисунок 2 – пример из курса ДМ

При поиске потока из вершины E в вершину H построен следующий поток:

5: E F K H

5: E J K H

3: E J K G H

1: E J K L H

1. Взят пример из книги Либского:

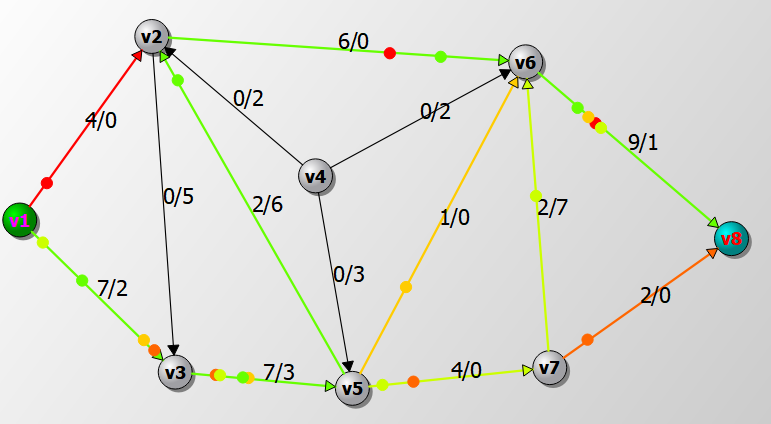


Рисунок 3 – пример из книги Либского

Построен следующий поток:

4: v1 v2 v6 v8

2: v1 v3 v5 v7 v8

1: v1 v3 v5 v6 v8

2: v1 v3 v5 v7 v6 v8

2: v1 v3 v5 v2 v6 v8

Суммарная величина потока – 11. Это совпадает с результатом, полученным в книге Либского

**Выводы.**

При выполнении данной работы изучен алгоритм Форда-Фалкерсона и его реализация на C++.

Изучена работа с простыми анимациями в Qt. Исследовано разрушение объектов в иерархии классов Qt.