**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №2**

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

Тема: Поиск Эйлерового цикла

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

Санкт-Петербург

2018

**Цель работы.**

Изучить алгоритм поиска Эйлерового цикла в графе и его реализацию на языке C++.

**Основные теоретические положения.**

*Эйлеров путь* – путь, проходящий по всем ребрам графа ровно один раз. Если начало пути совпадает с концом, это *эйлеров цикл*.

Согласно теореме, чтобы в графе имелся эйлеров цикл, необходимо и достаточно, чтобы вершины графа имели четную степень.

Алгоритм поиска эйлерового цикла на псевдокоде в книге Либского выглядит следующим образом:

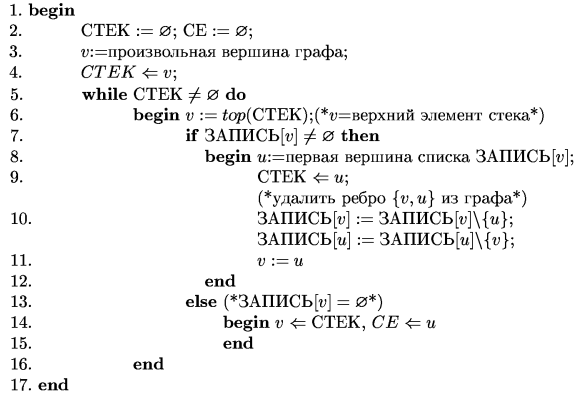


Рисунок 1 – алгоритм поиска эйлерового цикла на псевдокоде

Алгоритм начинает строить путь с определенной вершины. Вершины помещаются во временный стек СТЕК, а ребра удаляются из графа. Действия продолжаются до тех пор, пока список инцидентности текущей вершины не опустеет.

Для того, чтобы начальная вершина была равна конечной, степень вершины очевидно должна быть четной. Кроме того, после завершения текущего шага все вершины в графе остаются четными.

В результате этих шагов из графа был удален цикл, а его вершины были записаны во временный стек. Текущая вершина записывается во второй стек СЕ, и очередной вершиной становится верхний элемент стека.

Процесс повторяется, и в стеке снова оказывается цикл, содержащий данную вершину. Все продолжается до того момента, пока стек не опустеет.

Очевидно, что вершина записывается во второй стек только после того, как все инцидентные ей ребра были удалены из стека. Значит, по окончании работы второй стек содержит эйлеров цикл.

**Ход работы**

1. Написана основа для работа с графами – класс Graph.

Сам граф хранится с помощью двух структур:

typedef struct Elem{ //Элементы графа

Elem() = default;

char name[Numb];

Elem\* next=nullptr;

List\* childs=nullptr; //Дети элемента

Node\* node;

}Elem;

typedef struct List{

List() = default;

char name[Numb];

bool mark = false;

Elem\* node = nullptr; //Сам элемент

List\* next = nullptr;

Edge\* edge;

}List;

В первой структуре хранятся элементы графа, а во второй – его сыновья. Каждая структура содержит указатель на своего представителя в графической сцене. Ребра содержат не только указатель на второй элемент, но и его имя. Это связано с тем, что при вводе из файла может быть дана ссылка на вершину, которой ещё нет в основном списке. В таком случае записывается только имя, и зависимости разрешаются позднее.

Сам класс выглядит следующим образом:

class Graph

{

public:

Graph();

~Graph();

void **Clear**(); //Очистка

Elem\* **it**(); //Итератор через вершины

List\* **it**(Elem\* el, bool marked = 1); //Итератор через сыновей

void **ResetIts**();

void **ReadFile**(QString fileName); //Считать из файла

void **SaveFile**(QString fileName); //Сохранить в файл

char\* **GetMinStupidName**(); //Получение минимального числового имени

char\* **GetLastStupidName**(); //Получение последнего такого

void **AddElem**(char\* name); //Добавить элемент

void **AddEdge**(Elem\* el1, Elem\* el2); //Добавить линию

void **AddEdge**(Elem\* el1, char\* name); //Добавить "заготовку" для линии

bool **Solve**(); //Разрешить зависимости "заготовок" для линий

void **RemoveElem**(char\* name); //Удалить элемент

void **RemoveEdge**(Elem\* el1, Elem\* el2); //Удалить линию

void **RemoveEdges**(Elem\* el); //Удалить связи элемента

void **RenameElem**(char\* oldname, char\* newname); //Переименовать элемент

void **Desorientate**(); //Снять ориентацию графа

Elem\* **FindElem**(char\* name); //Найти элемент

int **CountChildren**(Elem\* el, bool marked = 1); //Сколько детей

int **CountElems**(); //Сколько элементов

bool **Is\_Egde**(Elem\* el1, Elem\* el2); //Есть ли связь от 1 к 2

List\* **GetEdge**(Elem\* el1, Elem\* el2); //Обратить связь

int **Max\_Width**(); //Максимальная длина имени в графе

void **Inc\_Matr**(QTextStream& os); //Матрица инцидентности

bool **Euler**(); //Один шаг поиска эйлерового цикла

void **ResetEuler**(); //Сброс эйлеровых циклов

void **ClearMarks**(); //Очистка меток исключений на графе

Elem\* **operator**[](int i); //Доступ к элементам по индексу

GraphWidget\* widget;

QStack<Elem\*> SE;

QStack<Elem\*> Stack;

Elem\* v0 = nullptr;

private:

int stupidnames = 0;

Elem\* gr;

Elem\* pos;

List\* lpos;

void **Clear**(List\* ls, Elem \*el);

void **Clear**(Elem\* gr);

Elem\* **KeepItE**();

List\* **KeepItL**();

void **RestoreItE**(Elem\* t\_pos);

void **RestoreItL**(List\* t\_lpos);

};

Пояснения ко всем публичным методам даны в комментариях.

Помимо всего прочего, класс включает в себя макросы для сохранения и восстановления итераторов:

#define SAVEITS List\* t\_lpos = KeepItL(); Elem\* t\_pos = KeepItE();

#define RESTOREITS RestoreItL(t\_lpos); RestoreItE(t\_pos);

Необходимость в них связана с тем, что запуск методов класса не должен ломать итераторы, запущенные на уровень выше.

1. Для обеспечения наглядной работы данный алгоритм изменен – так, цикл while заменен на if, чтобы обеспечит пошаговое выполнение. Кроме того, ребра не удаляются из графа, а «помечаются». Это необходимо для возможности повторной прогонки алгоритма. Также необходимо периодическое обновление элементов графической сцены – текущий элемент и «помеченные» ребра отображаются особым образом.

В остальном алгоритм остался идентичен данному на рисунке 1.

bool Graph::**Euler**()

{

SAVEITS; //Сохранение и сброс итераторов

Elem\* v;

List\* u;

if (Stack.isEmpty()){ //Первый запуск

ResetEuler();

Stack.push(gr);

}

if (!Stack.isEmpty()){

v = Stack.top();

if (CountChildren(v, 0)){

if (!v0)

v0 = v;

u = it(v, 0); //Итератор через непомеченные ребра вершины

Stack.push(u->node);

v->node->update(); //Обновление картинки в графе

u->mark = 1; //Пометка ребра в одну сторону

if (Is\_Egde(u->node, v))

GetEdge(u->node, v)->mark = 1; //Пометка ребра в другую сторону

u->edge->update(); //Обновление картинки

v = u->node;

}

else{

if ((v0!=v) && (v0!=nullptr)){

QMessageBox msg; //Если зашли в тупик

msg.setText("В графе тупик. Эйлерова цикла нет");

msg.*exec*();

return 1;

}

else{

v0 = nullptr;

v = Stack.pop();

if (v)

v->node->update(); //Обновление картинки

SE.push(v);

}

}

if (!Stack.isEmpty())

Stack.top()->node->update();

}

RESTOREITS; //Восстановление итераторов

return 0;

}

1. Реализовано отображение графа на QGraphicsWidget с применением информации из примера Elastic Nodes.

Оригинальные методы изменены и расширены. Существенные изменения перечислены ниже:

* Класс GraphWidget теперь полностью взаимосвязан с классом Graph. Работа с элементами графа в основном осуществляется через последний – так, изменение в классе (с помощью публичных методов) Graph неизменно влечет за собой изменение в GraphWidget, но и обратное тоже верно.
* Расширены возможности форматирования графа – ребра теперь могут быть направлены в одну сторону, на вершинах размещается текст, величина которого настраивается динамически так, чтобы он помещался. Вершины можно перекрасить – цвет шрифта также подстраивается автоматически.



Рисунок 2 – новые вершины

* Добавлена возможность выключения взаимодействия между вершинами – «гравитации».

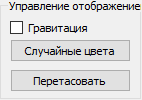


Рисунок 3 – некоторые элементы управления сценой

* Добавлена возможность выделения вершин при помощи клавиши Ctrl. Выделенные вершины перемещаются вместе.
* Выделенные вершины можно удалить кнопкой Delete.
* Добавлено добавление вершины двойным шелчком. Вершине присваивается числовое имя.
* Добавлено контекстное меню для каждой вершины с важными действиями.

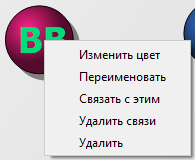


Рисунок 4 – контекстное меню.

**Выводы.**

Приложение А

Код