***Министерство образования Республики Беларусь***

***Учреждение образования***

***«Брестский государственный технический университет»***

***Кафедра ИИТ***

**Лабораторная работа №2**

**По дисциплине МОИС за III семестр**

**Тема: «Нахождение эйлеровых и гамильтоновых циклов**

**в неориентированном графе»**

**Выполнил:**

Студент группы ИИ-15 (1)

2-го курса

Волк И. А.

**Проверил:**

Шуть В. Н.

Брест 2018

Цель: научиться находить эйлеровы и гамильтоновы циклы.

***Задание.***

1. Написать программу нахождения эйлерова цикла в графе. Результатом программы является последовательность вершин в цикле. Предусмотреть проверку критерия на наличия эйлерова цикла в графе.
2. Написать программу нахождения гамильтонова цикла в графе. Результат программы – последовательность вершин цикла.
3. Варианты заданий указаны в таблице. Графы заданы списком ребер, *n*– количество вершин, -- проверка графа на эйлеров цикл,  - на гамильтонов цикл.
4. Изобразить графы и циклы в них.

Вариант №2

G1 = (1,2),(1,4),(2,3),(2,5).(2,6),(3,4), (3,5),(3,6),(4,5),(4,6),(5,6)

n1 – 6

G2 = (a,b),(a,c),(a,d),(b,c),(b,d),(b,e),(c,d),(c,g),(d,e),(d,f),(d,g),(e,f),(f,g)

n2 – 7

Код программы:

**// Application.cpp**

#include <iostream>

#include "Graphs.h" // graph.h eulhamgr.h eul.h ham.h grtrav.h

int main()

{

const GraphNumberOfVertices n1 = 6;

const GraphNumberOfVertices n2 = 7;

Graph G1;

G1.addVer(n1);

G1.addE(1, 2); G1.addE(1, 4);

G1.addE(2, 3); G1.addE(2, 5); G1.addE(2, 6);

G1.addE(3, 4); G1.addE(3, 5); G1.addE(3, 6);

G1.addE(4, 5); G1.addE(4, 6);

G1.addE(5, 6);

Eul eulG1(G1);

Ham hamG1(G1);

std::cout << "G1: " << std::endl;

eulG1.showCir();

eulG1.showPath();

hamG1.showCir();

hamG1.showPath();

Graph G2;

G2.addVer(n2);

G2.addE(1, 2); G2.addE(1, 3); G2.addE(1, 4);

G2.addE(2, 3); G2.addE(2, 4); G2.addE(2, 5);

G2.addE(3, 4); G2.addE(3, 7);

G2.addE(4, 5); G2.addE(4, 6); G2.addE(4, 7);

G2.addE(5, 6);

G2.addE(6, 7);

Eul eulG2(G2);

Ham hamG2(G2);

std::cout << "G2: " << std::endl;

eulG2.showCir();

eulG2.showPath();

hamG2.showCir();

hamG2.showPath();

std::cin.get();

}

**// graph.h**

#pragma once

#include "List\List.h" // for List

#define GraphVertex unsigned int

#define GraphEdgeWeight double

#define GraphVertexDegree unsigned int

#define GraphNumberOfVertices unsigned int

#define GraphNumberOfEdges unsigned int

enum GraphResponse

{

GRAPH\_NOT\_OK = 0,

GRAPH\_OK = 1

};

struct Edge

{

GraphEdgeWeight weight;

GraphVertex fVertex;

GraphVertex secVertex;

Edge();

Edge(const GraphVertex a, const GraphVertex b);

Edge(const Edge& other);

Edge(const GraphVertex a, const GraphVertex b, const GraphEdgeWeight weight);

GraphVertex& operator[](const int id);

};

class Graph

{

protected:

GraphNumberOfVertices numOfVer;

GraphNumberOfEdges numOfE;

List<GraphVertexDegree> deg;

List<Edge> e;

void findDeg();

public:

Graph();

Graph(const Graph& obj);

Graph(const GraphNumberOfVertices numOfVer, const List<Edge>& e,

const GraphNumberOfEdges numOfE);

~Graph();

GraphNumberOfVertices getNumOfVer() const;

GraphNumberOfEdges getNumOfE() const;

void addE(const Edge& newEdge);

void addE(const GraphVertex fVertex, const GraphVertex secVertex);

GraphNumberOfVertices addVer();

GraphNumberOfVertices addVer(const int num);

int remE(const GraphVertex a, const GraphVertex b);

int remE(const int id);

GraphEdgeWeight weight(const GraphVertex a, const GraphVertex b) const;

GraphVertexDegree getDeg(const int id) const;

Edge & operator[](const int id);

GraphNumberOfVertices operator++();

Graph& operator=(const Graph& obj);

};

**// Path.h**

#pragma once

#include "List\List.h"

#include "graph.h"

class Path

{

List<Edge> path;

public:

Path();

Path(const Edge& initEdge);

GraphResponse add(const Edge& nEdge);

GraphEdgeWeight length();

void viewPath();

void removeEdge(const int id);

GraphResponse changeEdge(const int id, double nWeight);

Edge& operator[](const int id);

Path& operator=(const Path& obj);

friend Path operator+(const Path& p, const Edge& e);

GraphNumberOfEdges numOfE() const;

};

**// euhamgr.h**

#pragma once

#include "grtrav.h"

#include "graph.h"

#include "Path.h"

class EuHamGr

{

protected:

Graph graph;

Path pathE;

Path cirE;

public:

// Constructors

EuHamGr();

EuHamGr(const EuHamGr& obj);

EuHamGr(const GraphNumberOfVertices numOfVer, const List<Edge>& e, const GraphNumberOfEdges numOfE);

EuHamGr(const Graph& graph);

~EuHamGr();

// Is semi-Eulerian/semi-Hamiltonian

virtual bool isSemi() = 0;

// Is Eulerian/Hamiltonian

virtual bool is() = 0;

virtual void showCir() = 0;

virtual void showPath() = 0;

};

**// eul.h**

#pragma once

#include "euhamgr.h"

#include "grtrav.h"

#include "graph.h"

namespace Eulerian

{

enum mode

{

circuit = 0,

path

};

}

// Eulerian graph

class Eul : public EuHamGr

{

protected:

void find(GrTrav& bf, Eulerian::mode mode);

bool isSemi(GrTrav& tr);

bool is(GrTrav& tr);

public:

// Is semi-Eulerian

bool isSemi();

// Is Eulerian

bool is();

Eul(const Graph& graph);

Eul(const GraphNumberOfVertices numOfVer, const List<Edge>& e, const GraphNumberOfEdges numOfE);

Eul(const Eul& obj);

~Eul();

void showCir();

void showPath();

};

**//eul.cpp**

#include "stdafx.h"

#include "eul.h"

#include <iostream>

Eul::Eul(const Graph& graph)

: EuHamGr(graph)

{

BreadthFirst bf(this->graph);

if (is(bf))

{

find(bf, Eulerian::mode::circuit);

pathE = cirE;

return;

}

if (isSemi(bf))

find(bf, Eulerian::mode::path);

}

Eul::Eul(const GraphNumberOfVertices numOfVer, const List<Edge>& e, const GraphNumberOfEdges numOfE)

: EuHamGr(numOfVer, e, numOfE)

{

BreadthFirst bf(this->graph);

if (is(bf))

{

find(bf, Eulerian::mode::circuit);

pathE = cirE;

}

if (isSemi(bf))

find(bf, Eulerian::mode::path);

}

Eul::Eul(const Eul& obj)

:EuHamGr(obj) {}

Eul::~Eul() {}

// is semi-Eulerian

bool Eul::isSemi()

{

return pathE.numOfE();

}

bool Eul::isSemi(GrTrav& tr)

{

// checking if there is a eulerian path

int numOfEven = 0, numOfOdd = 0;

// finding a number of odd and even degrees

for (int i = 0; i < graph.getNumOfVer(); i++)

if (graph.getDeg(i) % 2)

numOfOdd++;

else

numOfEven++;

// if there aren't odd degrees

// then eulerian path is equal to eulerian circuit

if (numOfOdd == 0)

{

pathE = cirE;

return true;

}

// if number of odd degrees don't equal 2

// then there is no eulerian path

if (numOfOdd != 2)

{

return false;

}

if (tr.getConnComp() > 1)

{

return false;

}

return true;

}

bool Eul::is()

{

return cirE.numOfE();

}

bool Eul::is(GrTrav& tr)

{

// checking if there is a eulerian circuit

// if there is an odd degree of vertex then there is no eulerian circuit

for (int i = 0; i < graph.getNumOfVer(); i++)

{

if (graph.getDeg(i) % 2)

{

return false;

}

}

if (tr.getConnComp() > 1)

{

return false;

}

return true;

}

void Eul::find(GrTrav& bf, Eulerian::mode mode)

{

//List<Edge> remEdges;

Graph& trGraph = bf.getGraph();

int initVer = 1;

if(mode == Eulerian::mode::path)

for (int i = 0; i < trGraph.getNumOfVer(); i++)

if (trGraph.getDeg(i) % 2)

{

initVer = i + 1;

break;

}

// current vertex

int cur\_ver = initVer;

while (true)

{

// id of next vertex when number of connected component has not been changed during deleting of edge

int next\_good = 0;

Edge remE\_good;

// id of next vertex when number of connected component has been changed during deleting of edge

int next\_bad = 0;

Edge remE\_bad;

// position in array of the edge

int j;

for (int i = 0; i < trGraph.getNumOfE(); i++)

{

if (trGraph[i][0] == cur\_ver || trGraph[i][1] == cur\_ver)

{

if (trGraph[i][0] == cur\_ver)

j = 1;

else

j = 0;

int tempConnComp = bf.getConnComp();

Edge remEdge = trGraph[i];

trGraph.remE(i);

bf.trav();

if (bf.getConnComp() > tempConnComp)

{

trGraph.addE(remEdge);

next\_bad = remEdge[j];

remE\_bad = remEdge;

bf.trav();

}

else if (bf.getConnComp() == tempConnComp)

{

trGraph.addE(remEdge);

next\_good = remEdge[j];

remE\_good = remEdge;

bf.trav();

break;

}

}

}

if (next\_good == 0 && next\_bad == 0)

break;

if (next\_good != 0)

{

if (mode == Eulerian::mode::circuit)

{

cirE.add(remE\_good);

}

else if (mode == Eulerian::mode::path)

{

pathE.add(remE\_good);

}

trGraph.remE(remE\_good.fVertex, remE\_good.secVertex);

bf.trav();

cur\_ver = next\_good;

}

else if(next\_bad != 0)

{

if (mode == Eulerian::mode::circuit)

{

cirE.add(remE\_bad);

}

else if (mode == Eulerian::mode::path)

{

pathE.add(remE\_bad);

}

trGraph.remE(remE\_bad.fVertex, remE\_bad.secVertex);

bf.trav();

cur\_ver = next\_bad;

}

}

}

void Eul::showCir()

{

if (!is())

return;

std::cout << "Eularian circuit: " << std::endl;

for (int i = 0; i < cirE.numOfE(); i++)

{

std::cout << cirE[i][0] << " -> ";

}

std::cout << cirE[cirE.numOfE() - 1][1] << std::endl;

}

void Eul::showPath()

{

if (!isSemi())

return;

std::cout << "Eularian path: " << std::endl;

for (int i = 0; i < pathE.numOfE(); i++)

std::cout << pathE[i][0] << " -> ";

std::cout << pathE[pathE.numOfE() - 1][1] << std::endl;

}

**// ham.h**

#pragma once

#include "euhamgr.h"

class Ham : public EuHamGr

{

protected:

void find();

bool cir(int \* ver);

bool path(int \* ver);

public:

Ham(const Graph& graph);

Ham(const GraphNumberOfVertices numOfVer, const List<Edge>& e, const GraphNumberOfEdges numOfE);

Ham(const Ham& obj);

~Ham();

bool isSemi();

bool is();

void showCir();

void showPath();

};

**// ham.cpp**

#include "stdafx.h"

#include "ham.h"

#include "permut.h"

#include "common.h"

Ham::Ham(const Graph& graph)

: EuHamGr(graph)

{

find();

}

Ham::Ham(const GraphNumberOfVertices numOfVer, const List<Edge>& e,

const GraphNumberOfEdges numOfE)

: EuHamGr(numOfVer, e, numOfE)

{

find();

}

Ham::Ham(const Ham& obj)

: EuHamGr(obj) {}

Ham::~Ham(){}

bool Ham::isSemi()

{

return pathE.numOfE();

}

bool Ham::is()

{

return cirE.numOfE();

}

void Ham::find()

{

int size = graph.getNumOfVer();

int \* arr = new int[size];

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

arr[i] = i + 1;

// potential path

Permut<int> p(arr, size);

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

{

delete[] arr;

p.get(i, size, arr);

if(cirE.numOfE() == 0)

cir(arr);

if(pathE.numOfE() == 0)

path(arr);

}

}

bool Ham::cir(int \* ver)

{

for (int v = 0; v < graph.getNumOfVer(); v++)

{

bool isEdge = false;

for (int edge = 0; edge < graph.getNumOfE(); edge++)

{

if (graph[edge][0] == ver[v] && graph[edge][1] == ver[(v + 1)

% graph.getNumOfVer()])

{

cirE.add(graph[edge]);

isEdge = true;

}

else if (graph[edge][1] == ver[v] && graph[edge][0] == ver[(v + 1)

% graph.getNumOfVer()])

{

cirE.add(graph[edge]);

isEdge = true;

}

}

if (!isEdge)

{

cirE = Path();

return false;

}

}

return true;

}

bool Ham::path(int \* ver)

{

for (int v = 1; v < graph.getNumOfVer(); v++)

{

bool isEdge = false;

for (int edge = 0; edge < graph.getNumOfE(); edge++)

{

if (graph[edge][0] == ver[v] && graph[edge][1] == ver[v - 1])

{

pathE.add(graph[edge]);

isEdge = true;

}

else if (graph[edge][1] == ver[v] && graph[edge][0] == ver[v - 1])

{

pathE.add(graph[edge]);

isEdge = true;

}

}

if (!isEdge)

{

pathE = Path();

return false;

}

}

return true;

}

void Ham::showCir()

{

if (!is())

return;

std::cout << "Hamiltonian circuit: " << std::endl;

for (int i = 0; i < cirE.numOfE(); i++)

{

std::cout << cirE[i][0] << " -> ";

}

std::cout << cirE[cirE.numOfE() - 1][1] << std::endl;

}

void Ham::showPath()

{

if (!isSemi())

return;

std::cout << "Hamiltonian path: " << std::endl;

for (int i = 0; i < pathE.numOfE(); i++)

{

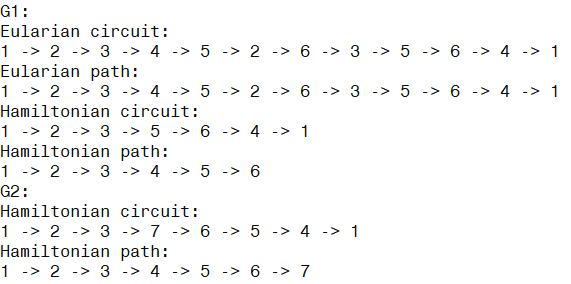
std::cout << pathE[i][0] << " -> ";

}

std::cout << pathE[pathE.numOfE() - 1][1] << std::endl;

}

Листинг результата:



Вывод: научился находить эйлеров и гамильтонов цикл.