**Лабораторная работа 6. АЛГОРИТМЫ НА ГРАФАХ**

**ЦЕЛЬ РАБОТЫ:** Освоить сущность и программную реализацию: а) способов представления графов; б) алгоритмов поиска в ширину и глубину; в) алгоритма топологической сортировки графов. Разобрать алгоритм Прима и алгоритм Крускала

**ЗАДАНИЕ ДЛЯ ВЫПОЛНЕНИЯ:**

***Задание 1.*** Ориентированный граф **G** взять в соответствии с вариантом. Представить его в отчете в виде матрицы смежности, матрицы инцидентности, списка смежных вершин.

***Задание 2.*** Осуществить алгоритмы поиска в ширину и глубину, а также алгоритма топологической сортировки аналогично примерам, рассмотренным на лекциях. Оформить отчет, включив в него **каждый** шаг выполнения алгоритмов.

***Задание 3.*** Осуществить программную реализацию алгоритмов на C++. Разработать структуры **AMatrix** и **АList**  для представления ориентированного графа матричным и списковым способом. Разработать функции преобразования из одного способа представления в другой. Разработать функцию **BFS** обхода вершин графа, используя метод поиска в ширину. Продемонстрировать работу функции. Копии экрана вставить в отчет.

***Задание 4.*** Разработать функцию **DFS**  обхода вершин графа, используя метод поиска глубину. Продемонстрировать работу функции. Копии экрана вставить в отчет.

***Задание 5.*** Доработайте функцию **DFS**,для выполнения топологической сортировки графа. Продемонстрировать работу функции. Копии экрана вставить в отчет.

***Задание 6.*** По графу, соответствующему варианту составить минимальное остовное дерево по алгоритму Прима. Шаги построения отразить в отчете.

Веса ребер принять:

W:

W(e0,1)=8; W(e1,0)=5;

W(e0,2)=1; W(e2,0)=3;

W(e0,3)=2; W(e3,0)=8;

W(e1,3)=11; W(e3,1)=4;

W(e1,4)=5; W(e4,1)=3;

W(e2,3)=7; W(e3,2)=9;

W(e2,5)=11; W(e5,2)=10;

W(e4,3)=4; W(e3,4)=1;

W(e4,6)=10; W(e6,4)=2;

W(e5,6)=2; W(e6,5)=6;

W(e5,3)=3; W(e3,5)=6;

W(e6,3)=7; W(e3,6)=9;

***Задание 7.*** По графу, соответствующему варианту составить минимальное остовное дерево по алгоритму Крускала. Шаги построения отразить в отчете.

|  |  |  |  |
| --- | --- | --- | --- |
| Вариант | Граф G | Вариант | Граф G |
| 1 |  | 9 |  |
| 2 |  | 10 |  |
| 3 |  | 11 |  |
| 4 |  | 12 |  |
| 5 |  | 13 |  |
| 6 |  | 14 |  |
| 7 |  | 15 |  |
| 8 |  |  | |

**ПРИМЕР ПРОГРАММНОГО КОДА:**

// --- Graph.h

//

#pragma once

#include <list>

#include "Graph.h"

namespace graph

{

struct AList;

struct AMatrix // матрица смежности

{

int n\_vertex; // количество вершин

int \*mr; // матрица

AMatrix(int n); // создать нулевую матрицу n\*n

AMatrix(int n, int mr[]); // создать матрицу n\*n и

AMatrix(const AMatrix& am); // создать подобную матрицу

AMatrix(const AList& al); // создать матрицу из спискового

void set(int i, int j, int r); // записать mr[i,j] = r

int get(int i, int j)const; // элемент mr[i,j]

};

struct AList // списки смежности

{

int n\_vertex; // количество вершин

std::list<int> \*mr; // массив списков

void create(int n); // создать массив пустых списков

AList(int n); // создать массив пустых списков

AList(int n, int mr[]); // создать списковое представление

AList(const AMatrix& am); // создать списковое представление

AList(const AList& al); // создать подобную структуру

void add(int i, int j); // добавить в i-ый список

int size(int i) const; // размер i-го списка

int get(int i, int j)const; // j-ый элемент i-го списка

};

};

// --- Graph.cpp

//

#include "stdafx.h"

#include "Graph.h"

namespace graph

{

AMatrix::AMatrix(int n)

{

this->n\_vertex = n;

this->mr = new int [this->n\_vertex\*this->n\_vertex];

for(int i = 0; i < n\*n; i++)mr[i] = 0;

};

AMatrix::AMatrix(int n, int mr[])

{

this->n\_vertex = n;

this->mr = mr;

};

AMatrix::AMatrix(const AMatrix& am)

{

this->n\_vertex = am.n\_vertex;

this->mr = new int [this->n\_vertex\*this->n\_vertex];

for (int i = 0; i < this->n\_vertex; i++)

for (int j = 0; j < this->n\_vertex; j++)

this->set(i,j, am.get(i,j));

};

AMatrix::AMatrix(const AList& al)

{

this->n\_vertex = al.n\_vertex;

this->mr = new int [this->n\_vertex \* this->n\_vertex];

for(int k = 0; k < this->n\_vertex\*this->n\_vertex; k++)mr[k] = 0;

for (int i = 0; i < this->n\_vertex; i++)

for (int j = 0; j < al.size(i); j++) this->set(i,al.get(i,j),1);

};

void AMatrix::set(int i, int j, int r) {this->mr[i\*this->n\_vertex+j] = r;};

int AMatrix::get(int i, int j)const

{return this->mr[i\*this->n\_vertex+j];};

void AList::create(int n)

{this->mr = new std::list<int>[this->n\_vertex = n];};

AList::AList(int n) {create(n);}

AList::AList(const AMatrix& am)

{

this->create(am.n\_vertex);

for (int i = 0; i < this->n\_vertex; i++)

for (int j = 0; j < this->n\_vertex; j++)

if (am.get(i,j)!=0) this->add(i,j);

};

AList::AList(const AList& al)

{

this->create(al.n\_vertex);

for (int i = 0; i < this->n\_vertex; i++)

for (int j = 0; j < al.size(i); j++) this->add(i, al.get(i,j));

};

AList::AList(int n, int mr[])

{

this->create(n);

for (int i = 0; i < this->n\_vertex; i++)

for (int j = 0; j < this->n\_vertex; j++)

if (mr[i\*this->n\_vertex+j]!= 0) this->add(i,j);

};

void AList::add(int i, int j){this->mr[i].push\_back(j);};

int AList::size(int i) const {return (int)this->mr[i].size();};

int AList::get(int i, int j)const

{

std::list<int>::iterator rc = this->mr[i].begin();

for (int k = 0; k < j; k++) rc++;

return (int)\*rc;

};

};

// ---BFS.h

//

#pragma once

#include "Graph.h"

#include <queue>

struct BFS // breadth-first search поиск в ширину (связный граф)

{

const static int INF = 0x7fffffff;

const static int NIL = -1;

enum Color {WHITE, GRAY, BLACK}; //

const graph::AList \*al; // исходный граф

Color \*c; // цвет вершины

int \*d; // расстояние до вершины

int \*p; // предшествующая вершина

std::queue<int> q; // очередь

BFS(const graph::AList& al, int s);

BFS(const graph::AMatrix& am, int s);

void init(const graph::AList& al, int s);

int get(); // получить следующую вершину

};

// ---BFS.cpp

//

#include "stdafx.h"

#include "BFS.h"

void BFS::init(const graph::AList& al, int s)

{

this->al = &al;

this->c = new Color[this->al->n\_vertex];

this->d = new int[this->al->n\_vertex];

this->p = new int[this->al->n\_vertex];

for (int i = 0; i < this->al->n\_vertex; i++)

{

this->c[i] = WHITE;

this->d[i] = INF;

this->p[i] = NIL;

};

this->c[s] = GRAY;

this->q.push(s);

};

BFS::BFS(const graph::AList& al, int s){this->init(al,s);};

BFS::BFS(const graph::AMatrix& am, int s)

{

this->init(\*(new graph::AList(am)),s);

};

int BFS::get()

{

int rc = NIL, v = NIL;

if (!this->q.empty())

{

rc = this->q.front();

for (int j = 0; j < this->al->size(rc); j++)

if (this->c[v = this->al->get(rc,j)] == WHITE)

{

this->c[v] = GRAY;

this->d[v] = this->d[rc]+1;

this->p[v] = rc;

this->q.push(v);

};

this->q.pop();

this->c[rc] = BLACK;

};

return rc;

}

// ---DFS.h

//

#pragma once

#include "Graph.h"

#include <vector>

struct DFS // depth-first search поиск в глубину

{

const static int NIL = -1;

enum Color {WHITE, GRAY, BLACK}; //

const graph::AList \*al; // исходный граф

Color \*c; // цвет вершины

int \*d; // время обнаружения

int \*f; // время завершения обработки

int \*p; // предшествующая вершина

int t; // текущее время

DFS(const graph::AList& al);

DFS(const graph::AMatrix& am);

std::vector <int> topological\_sort; //результат топологической сортировки

void visit(int v);

void init(const graph::AList& al);

int get(int i); // получить вершину

};

// ---DFS.cpp

//

#include "stdafx.h"

#include "DFS.h"

#define NINF 0x80000000

#define INF 0x7fffffff

void DFS::init(const graph::AList& al)

{

this->al = &al;

this->c = new Color[this->al->n\_vertex];

this->d = new int[this->al->n\_vertex];

this->f = new int[this->al->n\_vertex];

this->p = new int[this->al->n\_vertex];

this->t = 0;

for (int i = 0; i < this->al->n\_vertex; i++)

{

this->c[i] = WHITE;

this->d[i] = this->f[i] = 0;

this->p[i] = NIL;

};

for (int i = 0; i < this->al->n\_vertex; i++)

if (this->c[i] == WHITE)

{this->visit(i);

this->topological\_sort.push\_back (i);}

};

DFS::DFS(const graph::AList& al){this->init(al);};

DFS::DFS(const graph::AMatrix& am)

{

this->init(\*(new graph::AList(am)));

};

void DFS::visit(int u)

{

int v = NIL;

this->c[u] = GRAY;

this->d[u]= ++(this->t);

for (int j = 0; j < this->al->size(u); j++)

if (this->c[v = this->al->get(u,j)] == WHITE)

{

this->p[v] = u;

this->visit(v);

this->topological\_sort.push\_back (v);

}

this->c[u] = BLACK;

this->f[u]= ++(this->t);

};

int DFS::get(int i)

{

int j = 0, min1 = INF , min2 = NINF, ntx = NIL;

for (int j = 0; j <= i; j++) // iая статистика

{

for (int k = 0; k < this->al->n\_vertex; k++)

if (this->f[k] < min1 && this->f[k] > min2)

{min1 = this->f[k]; ntx = k;};

min2 = min1; min1 = INF;

};

return ntx;

};

// Graphs\_Optimization.cpp: определяет точку входа для консольного приложения.

// --- main

// алгоритмы BFS и DFS

#include "stdafx.h"

#include <iostream>

#include "Graph.h"

#include "BFS.h"

#include "DFS.h"

int \_tmain(int argc, \_TCHAR\* argv[])

{

int m[5][5] ={

{0, 1, 0, 1, 1},

{1, 0, 1, 1, 0},

{0, 1, 0, 1, 0},

{1, 1, 1, 0, 1},

{1, 0, 0, 1, 1}

};

setlocale(LC\_ALL, "rus");

graph::AMatrix g1(5,(int\*) m);

std::cout<<std::endl;

std::cout<<std::endl<< "-- матрица смежности "<< std::endl;

for (int i = 0; i < g1.n\_vertex; i++)

{

std::cout<<std::endl;

for (int j = 0; j < g1.n\_vertex; j++) std::cout<< g1.get(i,j) << " ";

};

std::cout<<std::endl;

graph::AList g2(g1);

std::cout<<std::endl;

std::cout<<std::endl<< "-- списки смежных вершин "<< std::endl;

for (int i = 0; i < g1.n\_vertex; i++)

{

std::cout<<std::endl<< i << ": ";

for (int j = 0; j < g2.size(i); j++) std::cout<<g2.get(i,j)<< " ";

}

std::cout<<std::endl;

graph::AMatrix g3(g1);

std::cout<<std::endl;

std::cout<<std::endl<< "-- матрица смежности "<< std::endl;

for (int i = 0; i < g3.n\_vertex; i++)

{

std::cout<<std::endl;

for (int j = 0; j < g3.n\_vertex; j++) std::cout<< g3.get(i,j) << " ";

};

std::cout<<std::endl;

graph::AList g4(5,(int\*) m);

std::cout<<std::endl;

std::cout<<std::endl<< "-- списки смежных вершин "<< std::endl;

for (int i = 0; i < g4.n\_vertex; i++)

{

std::cout<<std::endl<< i << ": ";

for (int j = 0; j < g4.size(i); j++) std::cout<<g4.get(i,j)<< " ";

}

std::cout<<std::endl;

BFS b1(g2,0);

std::cout<<std::endl;

std::cout<<std::endl<< "-- поиск в ширину "<< std::endl;

int k1;

while ((k1 = b1.get())!= BFS::NIL) std::cout<< k1 << " ";

std::cout<<std::endl;

DFS b2(g2);

std::cout<<std::endl;

std::cout<<std::endl<< "-- поиск в глубину "<< std::endl;

for (int i = 0; i < g2.n\_vertex; i++) std::cout<< b2.get(i) << " ";

std::cout<<std::endl;

int m3[5][5] = {

{0, 1, 0, 0, 0},

{0, 0, 0, 1, 0},

{0, 0, 0, 1, 0},

{1, 0, 0, 0, 1},

{1, 0, 0, 0, 1}

};

graph::AList g5(5, (int\*)m3);

DFS b3(g5);

std::cout<<std::endl;

std::cout<<std::endl<< "-- поиск в глубину (ориентированный граф) "

<< std::endl;

for (int i = 0; i < g5.n\_vertex; i++) std::cout<< b3.get(i) << " ";

std::cout<<std::endl;

std::cout << std::endl << "Топологическая сортировка" << std::endl;

for (std::vector <int>::iterator i (b3.topological\_sort.begin ()); i != b3.topological\_sort.end (); ++i) std::cout << \*i << ' ';

std::cout << std::endl;

system("pause");

return 0;

}

**Вопросы для защиты**

1. Какие представления графов Вы знаете?
2. В чем заключается поиск в ширину? Где рационально его использовать?
3. В чем заключается поиск в глубину? В каких ситуациях рационально его использовать?
4. В чем смысл топологической сортировки? Для чего она применяется?
5. Что такое минимальное остовное дерево?
6. В чем заключается стандартный алгоритм построения минимального остовного дерева?
7. К какой категории алгоритмов относятся алгоритмы Прима и Крускала?
8. Опишите один шаг алгоритма Крускала? Когда алгоритм прекращает свою работу?