Учреждение образования

“Белорусский государственный технологический университет”

Факультет информационных технологий

Кафедра информационных систем и технологий

Курс “Математическое программирование”

Отчёт по лабораторной работе №5

Алгоритмы на графах (алгоритмы поиска в ширину и глубину, топологическая сортировка)

Выполнил:

Студент 2 курса 1 группы

Бузанов Дмитрий Сергеевич

Вариант №8

Проверил: Бракович А.И.

Минск 2017г.

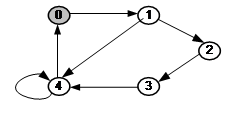
**Цель работы:** Освоить сущность и программную реализацию: а) способов представления графов; б) алгоритмов поиска в ширину и глубину; в) алгоритма топологической сортировки графов.

**Задание 1.** Ориентированный граф G взять в соответствии с вариантом. Осуществить алгоритмы поиска в ширину и глубину, а также алгоритма топологической сортировки аналогично примерам, рассмотренным на лекциях.

|  |  |
| --- | --- |
| Вариант | Граф G |
| 8 |  |

**Алгоритм поиска в ширину**

Исходный граф:

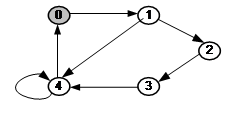


Алгоритм подразумевает, что задана исходная(стартовая) вершина и основан на простом правиле: при выборе очередной вершины предпочтение отдаётся ближайшей. При этом считается, что все дуги графа имеют единичную длину. Сначала посещается стартовая вершина, затем все смежные ей(т.е. находящиеся на расстоянии 1), после чего вершины, находящиеся на расстоянии 2 от старта и т.д.

Обозначения: Q – очередь вершин, C – массив окраски вершин, Р – массив предшествующих вершин, D – массив расстояний. Граф имеет 5 вершин, пронумерованных с нуля. В качестве стартовой выбрана вершина с номером 0.

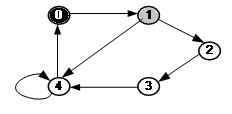
Шаг 1:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Q | 0 |  |  |  |  |
| C | G | W | W | W | W |
| D | 0 | 1 | 1 | 1 | 1 |
| P | 0 | N | N | N | N |



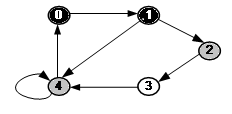
Шаг 2:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Q | 1 |  |  |  |  |
| C | B | G | W | W | W |
| D | 0 | 1 | 1 | 1 | 1 |
| P | 0 | 1 | N | N | N |



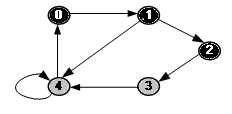
Шаг 3:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Q | 2 | 4 |  |  |  |
| C | B | B | G | W | G |
| D | 0 | 1 | 2 | 1 | 1 |
| P | 0 | 1 | 2 | 4 | N |



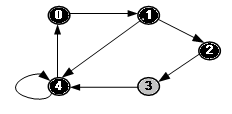
Шаг 4:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Q | 2 |  |  |  |  |
| C | B | B | B | G | G |
| D | 0 | 1 | 2 | 3 | 1 |
| P | 0 | 1 | 2 | 4 | 3 |



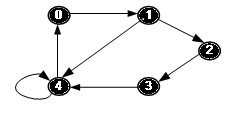
Шаг 5:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Q | 4 |  |  |  |  |
| C | B | B | B | G | B |
| D | 0 | 1 | 2 | 3 | 4 |
| P | 0 | 1 | 2 | 4 | 3 |

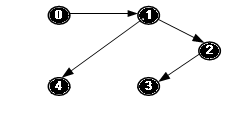


Шаг 6:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Q | 3 |  |  |  |  |
| C | B | B | W | W | W |
| D | 0 | 1 | 2 | 3 | 4 |
| P | 0 | 1 | 2 | 4 | 3 |



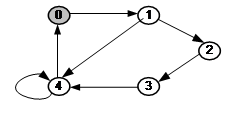
На шаге номер 6 обход дерева был закончен. Все вершины окрашены в черный цвет. Исходя из массива Р имеем BFS – дерево вида:



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| P | 0 | 1 | 2 | 4 | 3 |

**Алгоритм поиска в глубину**

Исходный граф:

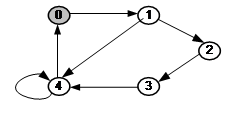


Алгоритм описывается следующим образом: для каждой не пройденной вершины, начиная со стартовой, необходимо найти все смежные вершины и повторить поиск для каждой найденной. Если после завершения алгоритма не все вершины были рассмотрены, то необходимо запустить алгоритм от одной из нерассмотренных вершин.

Назначение и размерность массивов: С – массив окраски вершин, Р – массив предшествующих вершин. В массив D для каждой вершины записывается время обнаружения( шаг окраски в черный цвет) вершины. Кроме того используется переменная t, текущее значение которой – номер шага в алгоритме.

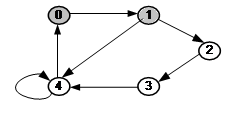
Шаг 1:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| С | G | W | W | W | W |
| D | 1 | 1 | 1 | 1 | 1 |
| P | N | N | N | N | N |
| F | 0 | 0 | 0 | 0 | 0 |



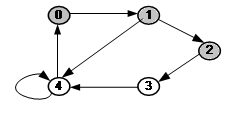
Шаг 2:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| С | G | G | W | W | W |
| D | 1 | 2 | 1 | 1 | 1 |
| P | N | 0 | N | N | N |
| F | 0 | 0 | 0 | 0 | 0 |



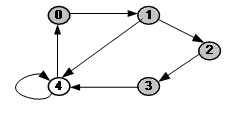
Шаг 3:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| С | G | G | G | W | W |
| D | 1 | 2 | 3 | 1 | 1 |
| P | N | 0 | 1 | N | N |
| F | 0 | 0 | 0 | 0 | 0 |



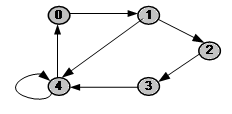
Шаг 4:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| С | G | G | G | G | W |
| D | 1 | 2 | 3 | 4 | 1 |
| P | N | 0 | 1 | 2 | N |
| F | 0 | 0 | 0 | 0 | 0 |



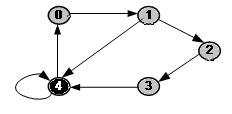
Шаг 5:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| С | G | G | G | G | G |
| D | 1 | 2 | 3 | 4 | 5 |
| P | N | 0 | 1 | 2 | 3 |
| F | 0 | 0 | 0 | 0 | 0 |



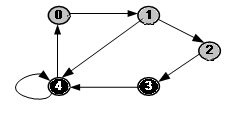
Шаг 6: t=6;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| С | G | G | G | G | B |
| D | 1 | 2 | 3 | 4 | 5 |
| P | N | 0 | 1 | 2 | 3 |
| F | 0 | 0 | 0 | 0 | 6 |



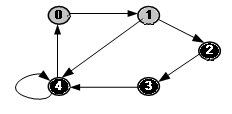
Шаг 7: t=7;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| С | G | G | G | B | B |
| D | 1 | 2 | 3 | 4 | 5 |
| P | N | 0 | 1 | 2 | 3 |
| F | 0 | 0 | 0 | 7 | 6 |



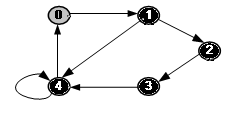
Шаг 8: t=8;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| С | G | G | B | B | B |
| D | 1 | 2 | 3 | 4 | 5 |
| P | N | 0 | 1 | 2 | 3 |
| F | 0 | 0 | 8 | 7 | 6 |



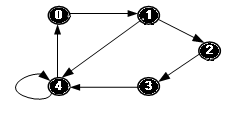
Шаг 9: t=9;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| С | G | B | B | B | B |
| D | 1 | 2 | 3 | 4 | 5 |
| P | N | 0 | 1 | 2 | 3 |
| F | 0 | 9 | 8 | 7 | 6 |

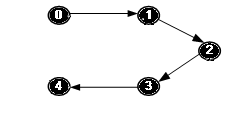


Шаг 10: t=10;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| С | B | B | B | B | B |
| D | 1 | 2 | 3 | 4 | 5 |
| P | N | 0 | 1 | 2 | 3 |
| F | 10 | 9 | 8 | 7 | 6 |

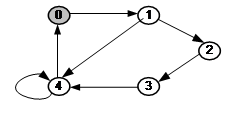


На шаге номер 10 алгоритм поиска в глубину завершён, все вершины графа окрашены в чёрный цвет. По итогу имеем DFS – дерево вида:



**Алгоритм топологической сортировки**

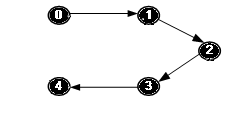
Исходный граф:



Топологическая сортировка – процедура упорядочивания орграфа, не имеющего циклов.

При реализации топологической сортировки с помощью алгоритма поиска в глубину используется массив вершин с помощью которого моделируется удаление вершин из графа и сохраняются новые номера вершин.

Для топологической сортировки используем найденное нами DFS – дерево с помощью алгоритма поиска в глубину:



Результат проведенной нами топологической сортировки:

**Программный код:**

// --- 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}

};

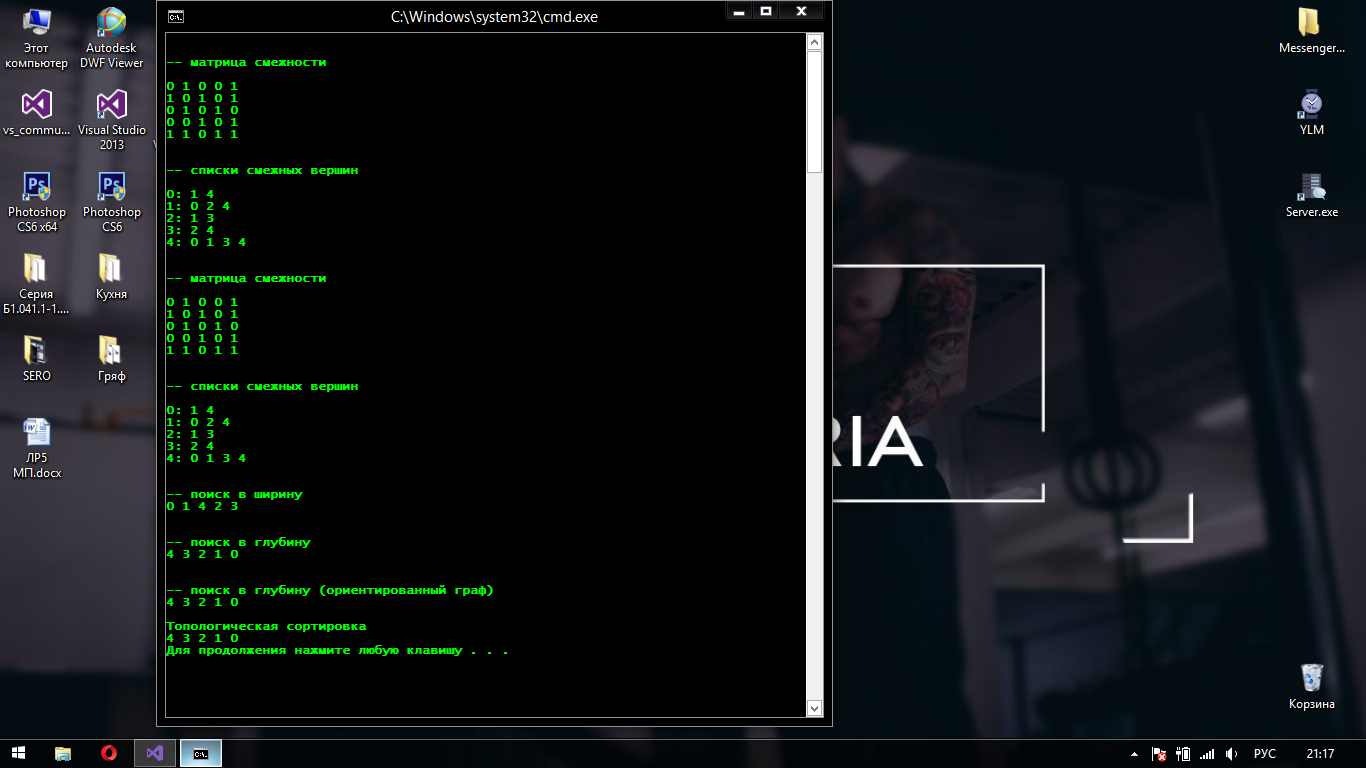
graph::AList g5(5, (int\*)m3);

DFS b3(g5);

std::cout<<std::endl;

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

<< std::endl;



**Вывод:** В данной лабораторной работе я освоил сущность и программную реализацию: способов представления графов, алгоритмов поиска в ширину и глубину, алгоритма топологической сортировки графов. Поработал с этими алгоритмами на практике на заданном графе.