МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Программное обеспечение информационных технологий

Дисциплина Математическое программирование

**Отчет по лабораторным работам по дисциплине**

**“Математическое программирование”**

Выполнил: студент 2курса 5 группы специальности “ПОИТ” Хатченок Д.Н.

(Ф.И.О)

Минск 2024

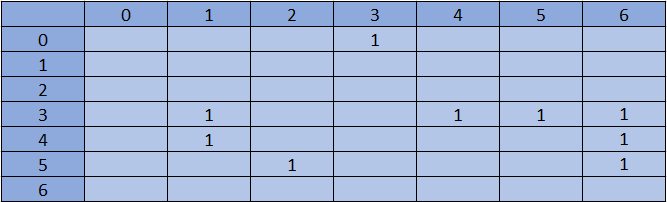
**Лабораторная работа 6. АЛГОРИТМЫ НА ГРАФАХ**

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



Матрица смежности:

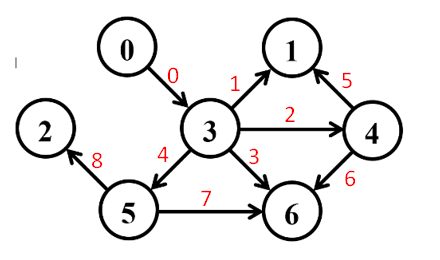


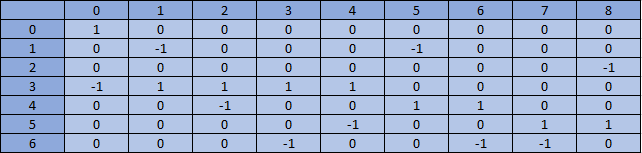
Список смежных вершин:

0 {3} 1 {-} 2 {-} 3 {1, 4, 5, 6} 4 {1, 6}

5 {2, 6}

Матрица инцидентности





***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.

***1.Поиск в ширину***



1.Посещённые вершины: {}

Очередь: {0}

Текущая вершина: {}

2.Посещённые вершины: {0}

Очередь: {}

Текущая вершина: {0}

3.Посещённые вершины: {0}

Очередь: {3}

Текущая вершина: {0}

4.Посещённые вершины: {0, 3}

Очередь: {1, 4, 5, 6}

Текущая вершина: {3}

5.Посещённые вершины: {0, 3, 1}

Очередь: {4, 5, 6}

Текущая вершина: {1}

6.Посещённые вершины: {0, 3, 1, 4}

Очередь: {5, 6}

Текущая вершина: {4}

7.Посещённые вершины: {0, 3, 1, 4, 5}

Очередь: {6, 2}

Текущая вершина: {5}

8.Посещённые вершины: {0, 3, 1, 4, 5, 6}

Очередь: {2}

Текущая вершина: {6}

9.Посещённые вершины: {0, 3, 1, 4, 5, 6, 2}

Очередь: {}

Текущая вершина: {2}

***2.Поиск в глубину***



1.Посещённые вершины: {}

Стек: {0}

Текущая вершина: {}

2.Посещённые вершины: {0}

Стек: {3}

Текущая вершина: {0}

3.Посещённые вершины: {0, 3}

Стек: {6, 5, 4, 1}

Текущая вершина: {3}

4.Посещённые вершины: {0, 3, 6}

Стек: {5, 4, 1}

Текущая вершина: {6}

5.Посещённые вершины: {0, 3, 6, 5}

Стек: {2, 4, 1}

Текущая вершина: {5}

6.Посещённые вершины: {0, 3, 6, 5, 2}

Стек: {4, 1}

Текущая вершина: {2}

7.Посещённые вершины: {0, 3, 6, 5, 2, 4}

Стек: {1}

Текущая вершина: {4}

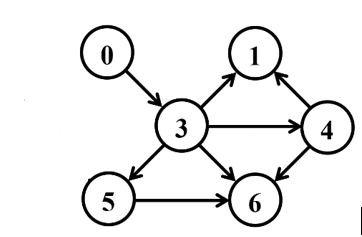
8.Посещённые вершины: {0, 3, 6, 5, 2, 4, 1}

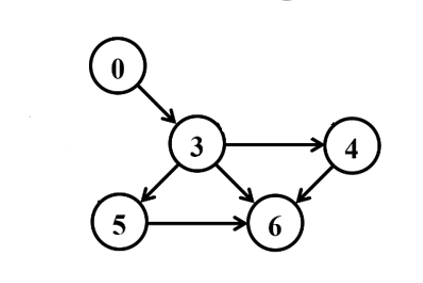
Стек: {}

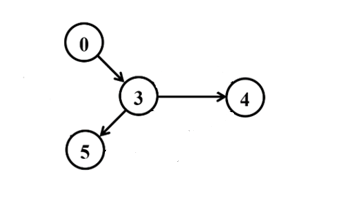
Текущая вершина: {1}

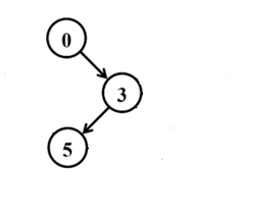
***3.Топологическая сортировка:***

2

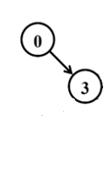
1

6

4



5



3

0

**Результат: 2-1-6-4-5-3-0**

***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.

Заголовочный файл для структур представления матричным и списковым способом представлен на рисунке 1:

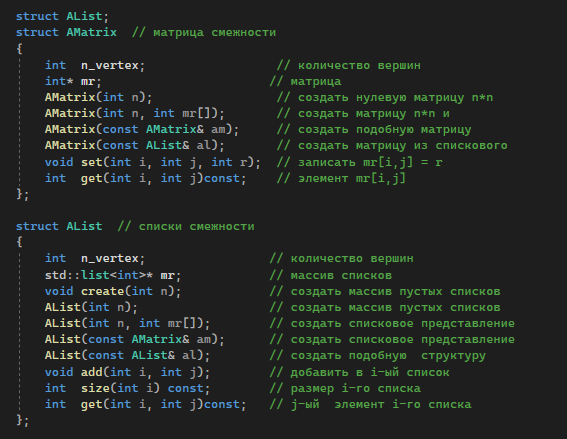


Рисунок 1 – struct of AList and AMatrix

В листинге 1 демонстрируется реализация структур AMatrix и AList:

#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;

};

};

Листинг 1 – Graph.cpp

В листинге 2 показана реализация функции BFS:

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;

}

Листинг 2 – BFS.cpp

Результат выполнения программы представлен на рисунке 2:

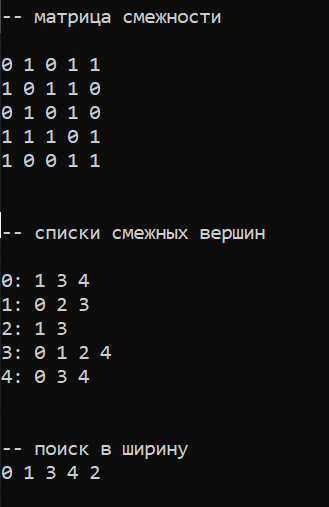


Рисунок 2 – результат выполнения программы

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

Заголовочный файл функции DFC.h:

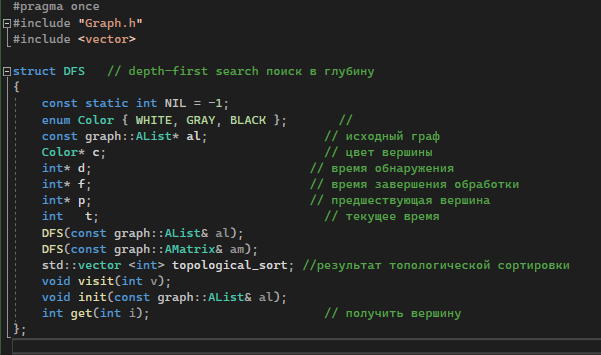


Рисунок 3 – DFC.h

Реализация заголовка представлена в листинге 3:

#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;

};

Листинг 3 - DFC.cpp

Результат выполнения программы представлен на рисунке 4:

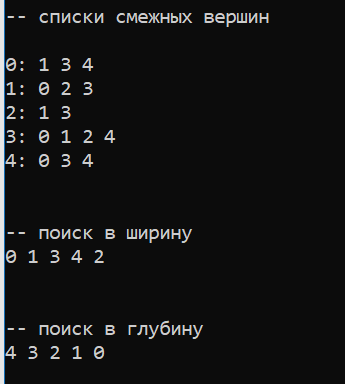


Рисунок 4 – результат выполнения программы

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

Реализация кода представлена в предыдущем задании, а результат выполнения – на рисунке 6.

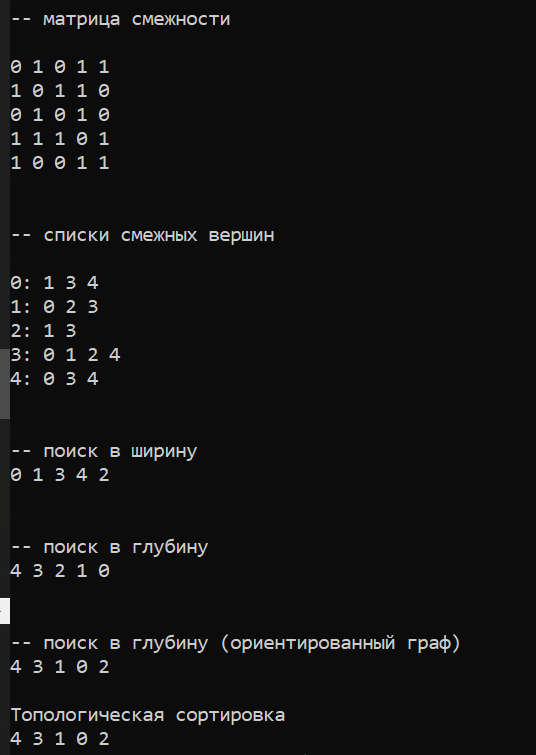
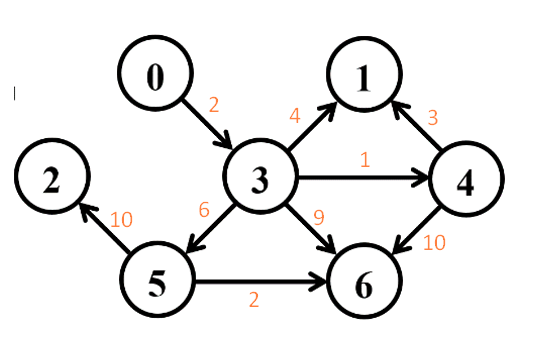
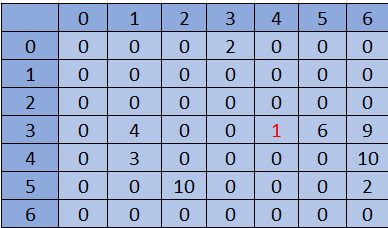


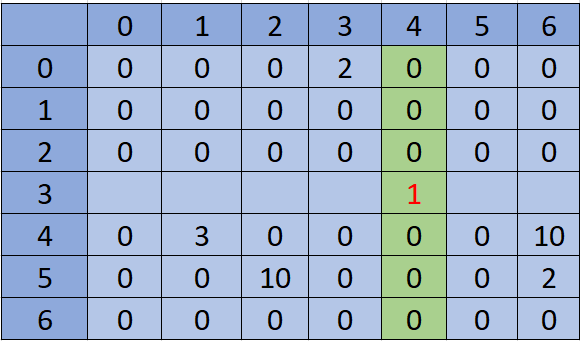
Рисунок 6 – результат выполнения программы

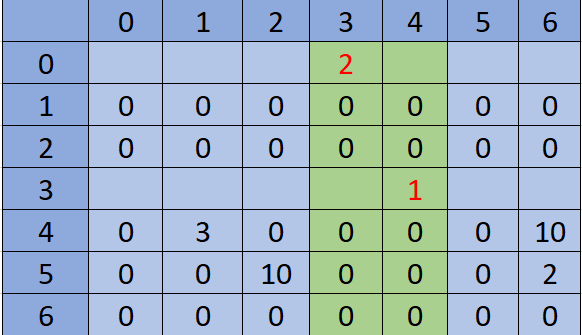
***Задание 6.*** По графу, соответствующему варианту составить минимальное остовное дерево по алгоритму Прима. Шаги построения отразить в отчете.

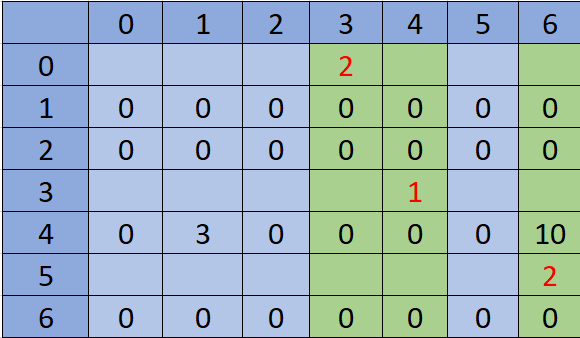


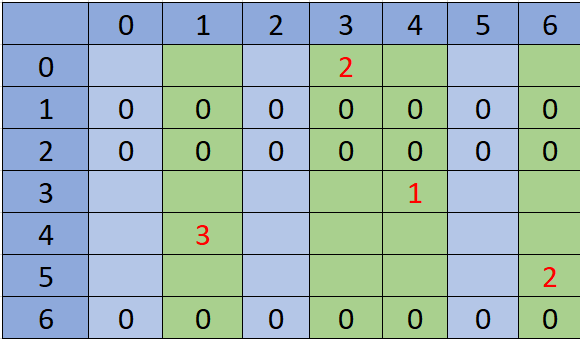
(Можно решать матрицей, вычёркивая строки и выделяя столбцы, можно через граф).

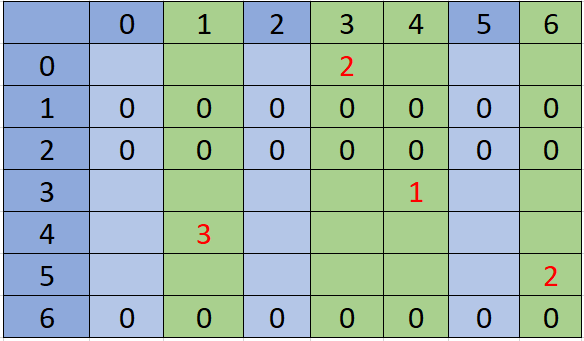












Т.е. 0-1-3-4-6

Теперь графически решим:

Для решения по алгоритму Прима необходимо выбрать произвольную начальную вершину и добавлять к остовному дереву ребра с наименьшей стоимостью, пока все вершины не будут включены в дерево.

Можно начать с вершины 0. Перечислим все ребра, связанные с этой вершиной и их стоимость:

* 0->3 (стоимость 2)

Добавляем это ребро к остовному дереву.

Теперь мы имеем две вершины в остовном дереве: 0 и 3. Необходимо выбрать следующее ребро с наименьшей стоимостью, которое связывает вершину из дерева и вершину, не принадлежащую дереву. Перечислим все такие ребра и их стоимость:

* 3->4 (стоимость 1)
* 3->5 (стоимость 6)
* 3->6 (стоимость 9)
* 3->1 (стоимость 4)

Выбираем ребро 3->4, так как оно имеет наименьшую стоимость. Добавляем вершину 4 в остовное дерево.

Теперь в дереве находятся вершины 0, 3 и 4. Необходимо выбрать следующее ребро с наименьшей стоимостью, которое связывает вершину из дерева и вершину, не принадлежащую дереву. Перечислим все такие ребра и их стоимость:

* 4->1 (стоимость 3)
* 4->6 (стоимость 10)
* 3->5 (стоимость 6)
* 3->6 (стоимость 9)
* 5->6 (стоимость 2)

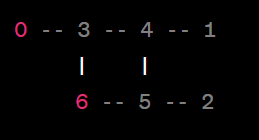
Выбираем ребро 5->6, так как оно имеет наименьшую стоимость. Добавляем вершину 6 в остовное дерево.

Теперь в дереве находятся вершины 0, 3, 4 и 6. Необходимо выбрать следующее ребро с наименьшей стоимостью, которое связывает вершину из дерева и вершину, не принадлежащую дереву. Перечислим все такие ребра и их стоимость:

* 5->2 (стоимость 10)
* 3->1 (стоимость 4)

Выбираем ребро 3->1, так как оно имеет наименьшую стоимость. Теперь в дереве находятся все вершины: 0, 3, 4, 6 и 1. Нет больше ребер, связывающих вершину из дерева и вершину, не принадлежащую дереву, поэтому алгоритм завершен.

Таким образом, остовное дерево, построенное по алгоритму Прима, имеет следующий вид:



Общая стоимость дерева равна 16.

\*Остовное дерево - это подграф, содержащий все вершины исходного графа и являющийся деревом (т.е. связным графом без циклов).

Однако, если вам нужно пройти все вершины графа, то минимальное остовное дерево может не включать все вершины. Это происходит, когда исходный граф не является связным, то есть имеет несколько компонент связности. В таком случае остовное дерево будет содержать только вершины из одной компоненты связности, а вершины из других компонент связности будут исключены из дерева.

Если нужно пройти все вершины графа, вам необходимо обходить все компоненты связности графа. Для этого можно использовать алгоритм поиска в глубину или алгоритм поиска в ширину, который позволит пройти все вершины графа.

***Задание 7.*** По графу, соответствующему варианту составить минимальное остовное дерево по алгоритму Крускала.

