Учреждение образования

«Белорусский государственный технологический университет»

**Лабораторная работа №6**

Алгоритмы на графах

Выполнила:

Студентка 2 курса 8 группы ИТ

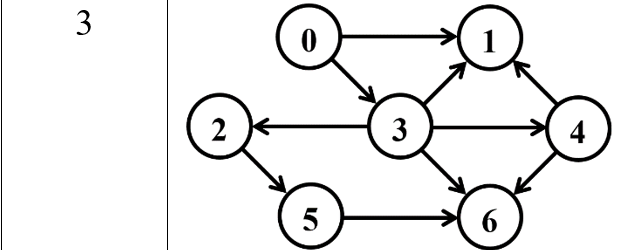
Казакова В.В.

2025 г.

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



**Матрица смежности** — это способ представления графа в виде двумерной матрицы. Каждый элемент матрицы отображает наличие или отсутствие рёбер между вершинами графа.

**Матрица смежности:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Вершина** | **0** | **1** | **2** | **3** | **4** | **5** | **6** |
| **0** | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **2** | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| **3** | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| **4** | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| **5** | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| **6** | 0 | 0 | 0 | 0 | 0 | 0 | 0 |



G = (V, E), V = {0, 1, 2, 3, 4, 5, 6}, E = { <0, 1>0, <0, 3>1, <3, 1>2, <3, 2>3, <3, 4>4, <3, 6>5, <2, 5>6, <5, 6>7, <1, 4>8, <4, 6>9}



7

9

8

5

4

3

2

6

0

1

если ребро исходит от вершины, то 1, если входит, то -1, 0 – если ничего

**Матрица инцидентности:**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Вершина** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| **0** | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **1** | -1 | 0 | -1 | 0 | 0 | 0 | 0 | 0 | -1 | 0 |
| **2** | 0 | 0 | 0 | -1 | 0 | 0 | 1 | 0 | 0 | 0 |
| **3** | 0 | -1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| **4** | 0 | 0 | 0 | 0 | -1 | 0 | 0 | 0 | 1 | 1 |
| **5** | 0 | 0 | 0 | 0 | 0 | 0 | -1 | 1 | 0 | 0 |
| **6** | 0 | 0 | 0 | 0 | 0 | -1 | 0 | -1 | 0 | -1 |

Список смежных вершин - это список всех вершин, с которыми заданная вершина в графе имеет ребра. Другими словами, это список вершин, которые непосредственно соединены с данной вершиной ребрами.

Список смежных вершин:

0 {1,3}

1 ∅

2 {5}

3 {1,2,4,6}

4 {1,6}

5 {6}

6 ∅

***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.

**Алгоритм поиска в ширину (англ. breadth-first search, BFS)** позволяет найти кратчайшие пути из одной вершины невзвешенного графа до всех остальных вершин

Белые вершины не посещены

Серый вершины в очереди

Голубые вершины посещаются

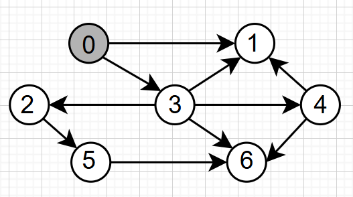
Оранжевый вершина является текущей вершиной

**Поиск в ширину:**

1. Посещенные вершины : {}

Очередь:{}

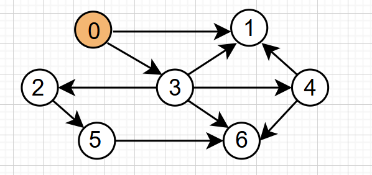
Текущая вершина: -



1. Посещенные вершины : {}

Очередь:{0}

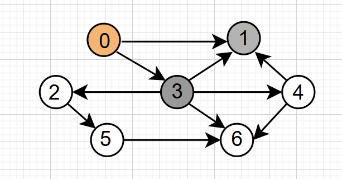
Текущая вершина: -



1. Посещенные вершины : {0}

Очередь:{}

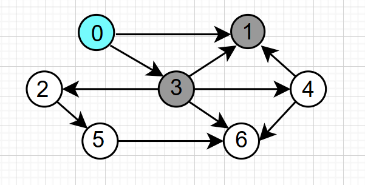
Текущая вершина: 0



1. Посещенные вершины : {0}

Очередь:{1,3}

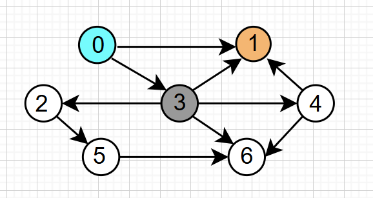
Текущая вершина: 0



1. Посещенные вершины : {0}

Очередь:{1,3}

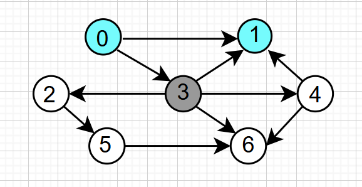
Текущая вершина: -



1. Посещенные вершины : {0,1}

Очередь:{3}

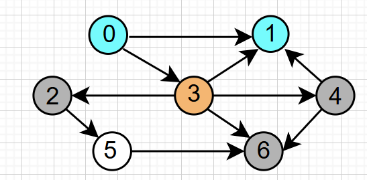
Текущая вершина: 1



1. Посещенные вершины : {0,1}

Очередь:{3}

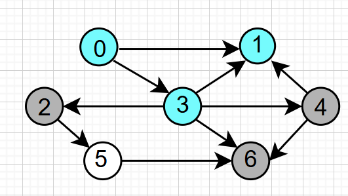
Текущая вершина: -



1. Посещенные вершины : {0,1,3}

Очередь:{2,4,6}

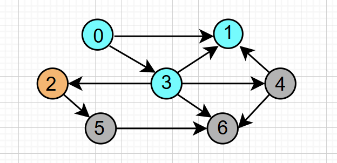
Текущая вершина: 3



1. Посещенные вершины : {0,1,3}

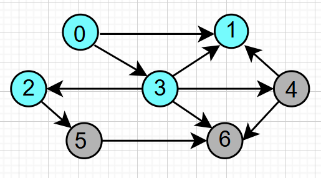
Очередь:{2,4,6}

Текущая вершина: -

1. Посещенные вершины : {0,1,3,2}

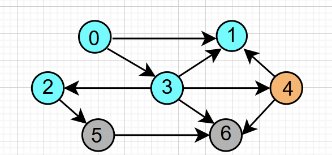
Очередь:{4,6,5}

Текущая вершина: 2

1. Посещенные вершины : {0,1,3,2}

Очередь:{4,6,5}

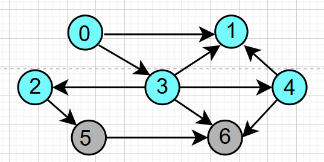
Текущая вершина: -



1. Посещенные вершины : {0,1,3,2,4}

Очередь:{6,5}

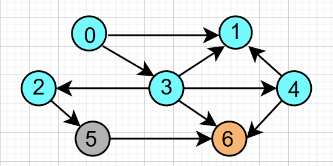
Текущая вершина: 4



1. Посещенные вершины : {0,1,3,2,4}

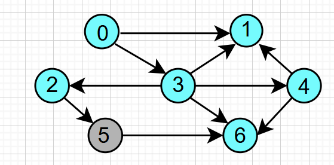
Очередь:{6,5}

Текущая вершина: -

1. Посещенные вершины : {0,1,3,2,4,6}

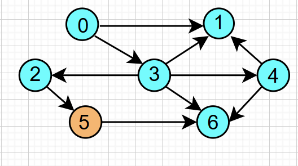
Очередь:{5}

Текущая вершина: 6

1. Посещенные вершины : {0,1,3,2,4,6}

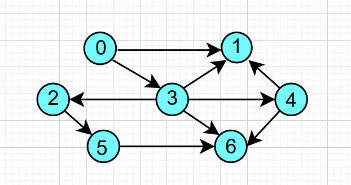
Очередь:{5}

Текущая вершина: -

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

Очередь:{}

Текущая вершина: 5

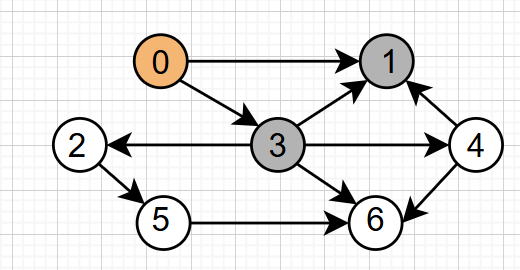
1. Посещенные вершины : {0,1,3,2,4,6,5}

Очередь:{}

Текущая вершина: -

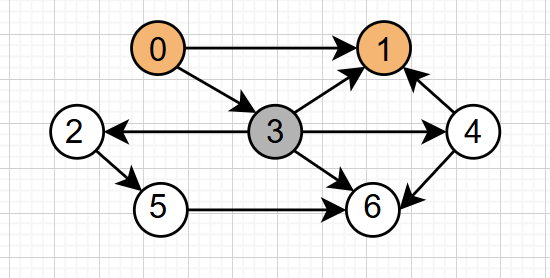
Порядок обхода : 0,1,3,2,4,6,5

**Поиск в глубину**

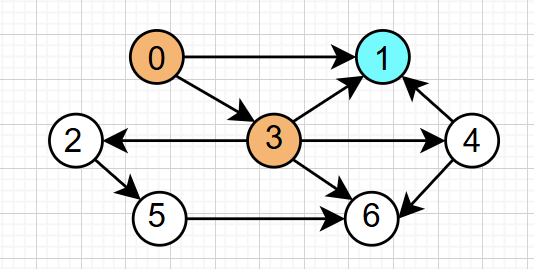
1. ****

Начинаем с первой доступной вершины – в данном случае это вершина 0. Помещаем 0 в стек и помечаем её как посещённую).

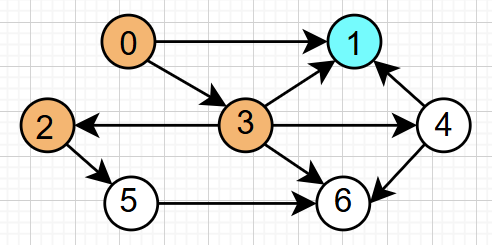
стек: {0}

1. 

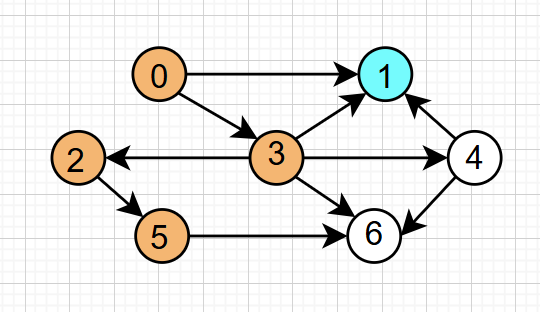
Спуск к вершине 1. Смотрим верхний элемент стека – сейчас это 0. У вершины 0 имеются смежные вершины 1 и 3. Берём первую из них по порядку, вершину 1, которая ещё не посещена. Помещаем 1 в стек (стек: {0, 1}). Вершина 1 не имеет непосещённых соседей (у 1 нет исходящих рёбер), поэтому извлекаем 1 из стека и выводим 1 – это первый элемент в результирующей последовательности вывода.

3) 

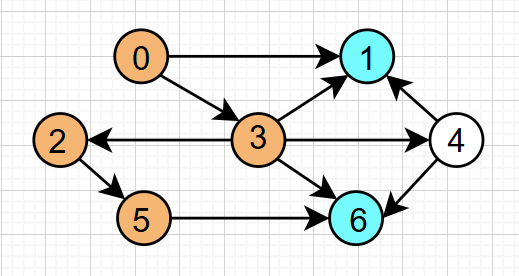
Переход к вершине 3. Теперь верхним элементом стека снова стала 0 (после удаления 1). У вершины 0 больше нет непосещённых соседей, кроме уже рассмотренной 1, поэтому переходим к следующему смежному узлу 0 – вершине 3. Помещаем 3 в стек и отмечаем как посещённую (стек: {0, 3}).

4) 

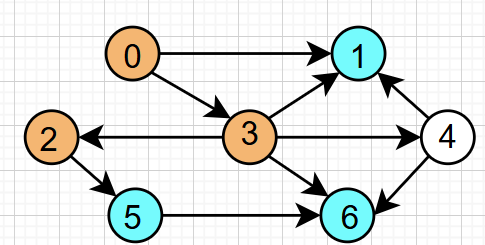
Спуск к вершине 2. Рассматриваем верхний элемент стека – вершину 3. Её смежные вершины (в заданном порядке) – 1, 2, 4, 6. Сосед 1 уже был посещён ранее, поэтому пропускаем его и берём следующий непосещённый сосед, вершину 2. Помещаем 2 в стек (стек: {0, 3, 2}).

5) 

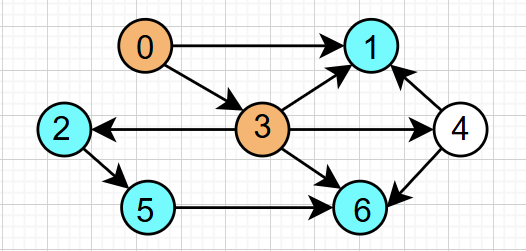
Спуск к вершине 5. Верхний элемент стека теперь 2. У вершины 2 единственный смежный узел – 5, который ещё не посещён. Помещаем вершину 5 в стек (стек: {0, 3, 2, 5}).

6) 

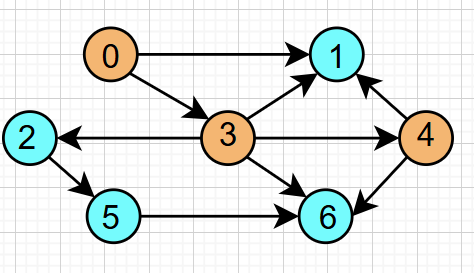
Спуск к вершине 6. Верхушка стека – вершина 5. Её смежная вершина 6 ещё не посещена. Помещаем 6 в стек (стек: {0, 3, 2, 5, 6}). Вершина 6 не имеет исходящих рёбер (смежных вершин нет), поэтому извлекаем 6 из стека и выводим 6 – это следующий элемент в результирующей последовательности.

7) 

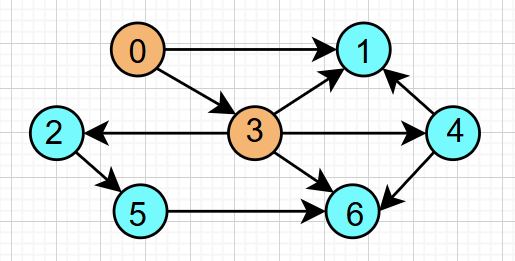
Возврат к вершине 5. После удаления 6 верхним элементом стека снова стала 5. У вершины 5 больше нет непосещённых соседей (единственный сосед 6 уже обработан), поэтому извлекаем 5 и выводим 5.

8) 

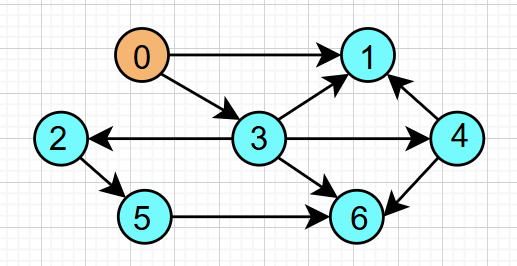
Возврат к вершине 2. Теперь верхушка стека – вершина 2. У 2 не осталось непосещённых соседей (она уже обработала 5), поэтому извлекаем 2 и выводим 2.

9) 

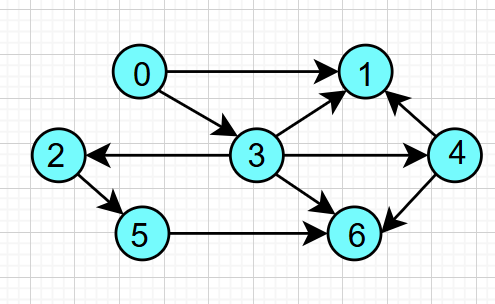
Возврат к вершине 3 и переход к 4. Верхним элементом стека снова стала 3. Мы уже посетили соседей 1 и 2 для вершины 3; следующий смежный узел по порядку – вершина 4 (непосещённая). Помещаем 4 в стек (стек: {0, 3, 4}).

10) 

Вершина 4 не имеет новых непосещённых соседей (её смежные 1 и 6 уже были посещены ранее), поэтому сразу извлекаем 4 и выводим 4.

11) 

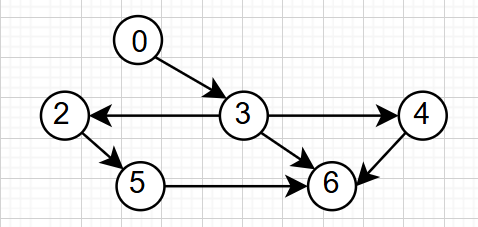
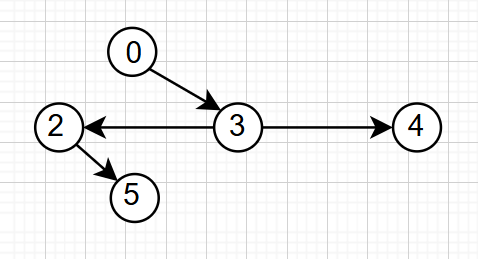
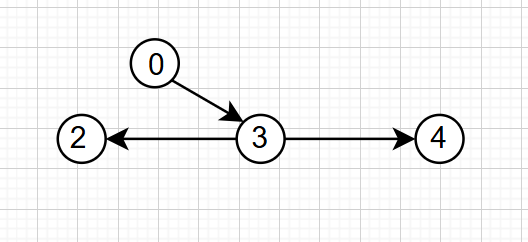
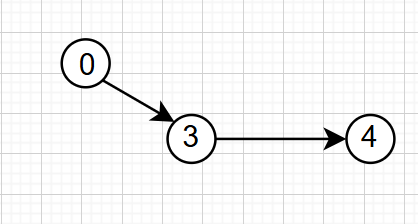
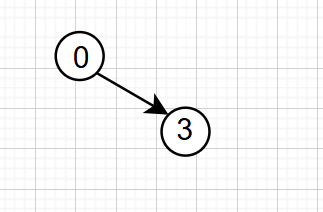
Возврат к вершине 3. После удаления 4 верхним элементом стека снова является 3. У вершины 3 больше нет непосещённых соседей (оставшийся сосед 6 уже был посещён). Поэтому извлекаем 3 из стека и выводим 3.

12) 

Завершение обхода на вершине 0. Верхушка стека вновь стала 0. У вершины 0 не осталось непосещённых соседей (оба её соседа, 1 и 3, уже были обработаны), поэтому извлекаем 0 и выводим 0.

Итоговая последовательность вывода получается: 1 6 5 2 4 3 0

**Топологическая сортирвка**

1. ** 1**
2. ** 6**
3. ** 5**
4. ** 2**
5. **4**

Результат: 1 6 5 2 4 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.

|  |
| --- |
| #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;  } |

Листинг 3.1 — файл BFS.cpp

|  |
| --- |
| #pragma once  #include "Graph.h"  #include <queue>  struct BFS  {  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  }; |

Листинг 3.2 — файл BFS.h

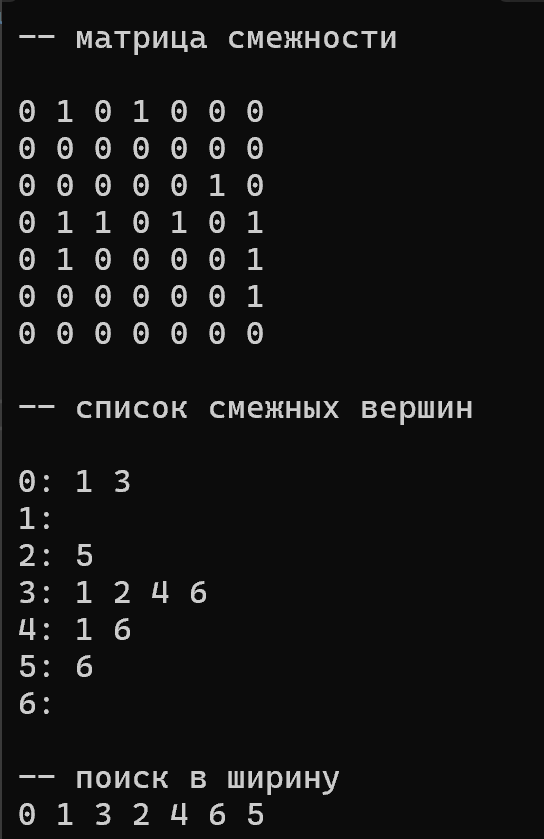


Рисунок 3.1 – результат программы

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

|  |
| --- |
| #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++)  {  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 4.1 — файл DFS.cpp

|  |
| --- |
| #pragma once  #include "Graph.h"  #include <vector>  struct DFS  {  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  }; |

Листинг 4.2 — файл DFS.h

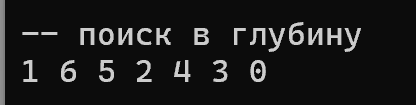


Рисунок 4.1 – результат программы

***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 5.1.

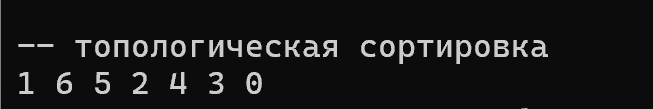


Рисунок 5.1 - результат программы

***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;

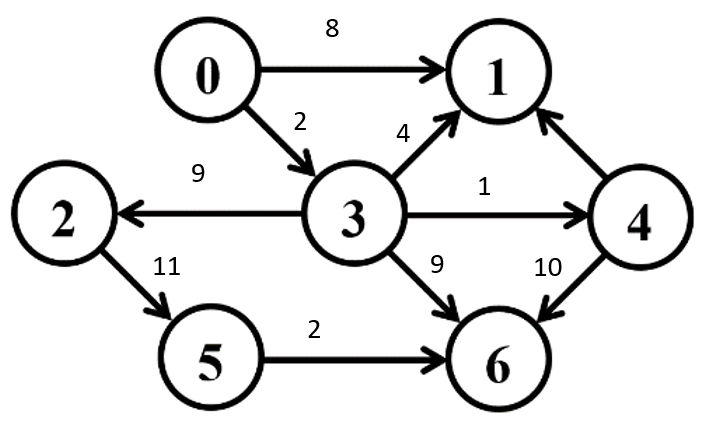
W(e4,3)=4; W(e3,4)=1;

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

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

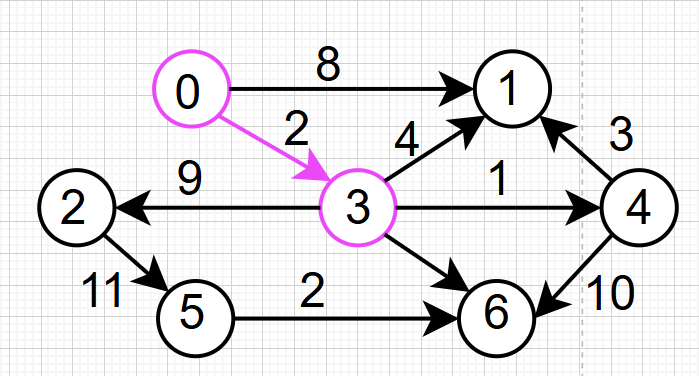
W(e5,3)=3; W(e3,5)=6;

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



Алгоритм Прима строит минимальное остовное дерево, начиная с произвольной вершины и постепенно расширяя дерево, выбирая минимальные рёбра, которые соединяют дерево с вершинами, ещё не входящими в дерево.

1. Выбираем вершину 0



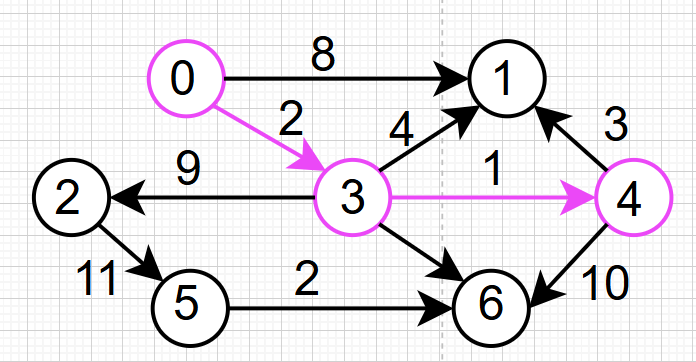
Начинаем с вершины 0. Из вершины 0 есть рёбра:

- 0→1 с весом 8

- 0→3 с весом 2

Выбираем минимальное ребро: 0→3 с весом 2.

**Посещённые вершины**: {0, 3}

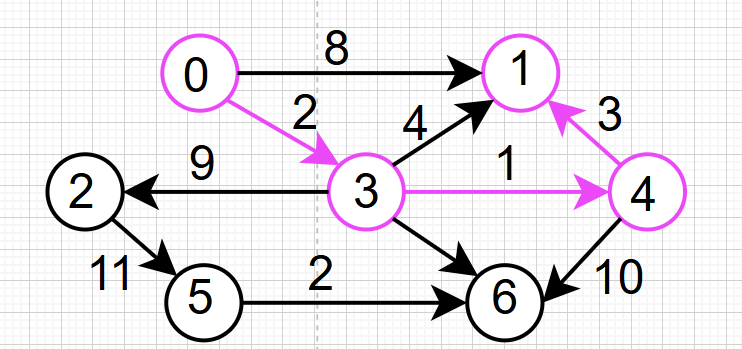
2) 

**Обработка вершины 3**: Из вершины 3 есть рёбра:

* 3→1 с весом 4
* 3→2 с весом 9
* 3→4 весом 1
* 3→6 с весом 9

Выбираем минимальное ребро: 3→4 с весом 1.

**Посещённые вершины**: {0, 3, 4}

**3)** ****

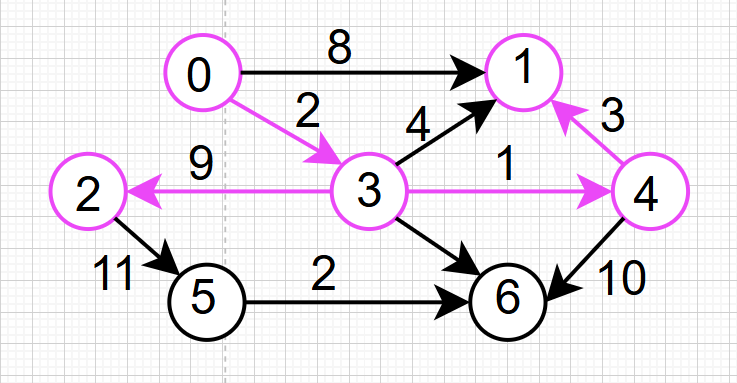
**Обработка вершины 4**: Из вершины 444 есть рёбра:

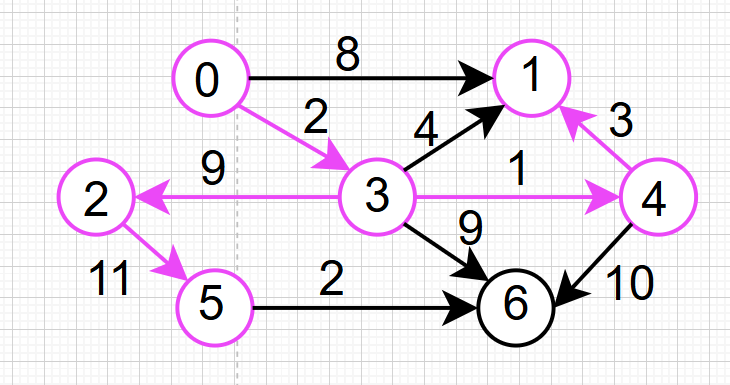
* 4→1 с весом 3
* 4→6 с весом 10

Выбираем минимальное ребро: 4→1 с весом 3.

**Посещённые вершины**: {0, 3, 4, 1}

4) Из вершины 1 больше нет исходящих рёбер (она не соединена с другими вершинами), поэтому переходим к следующему шагу.

5) 

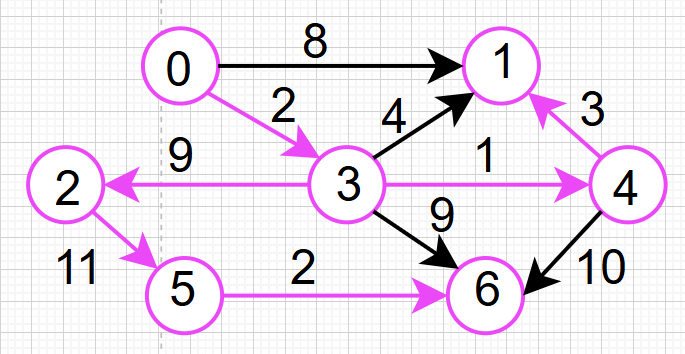
6) 

Из вершины 2 есть рёбра:

* 2→5 с весом 11

Выбираем минимальное ребро: 2→5 с весом 11.

**Посещённые вершины**: {0, 3, 4, 1, 2, 5}

**7) **

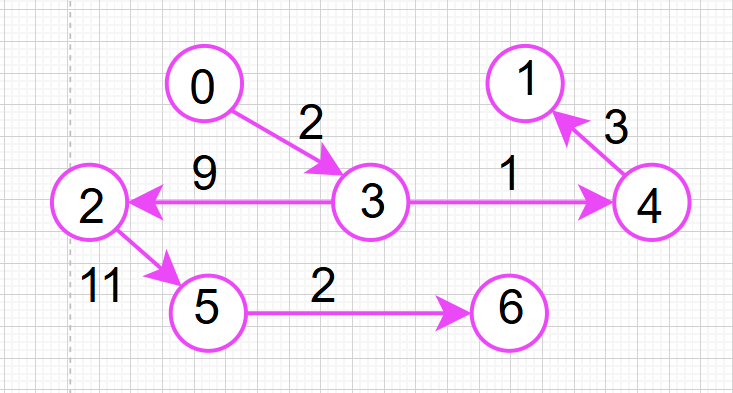
Из вершины 5 есть рёбра:

* 5→6 с весом 2

Выбираем минимальное ребро: 5→6 с весом 2.

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

**Результат** :

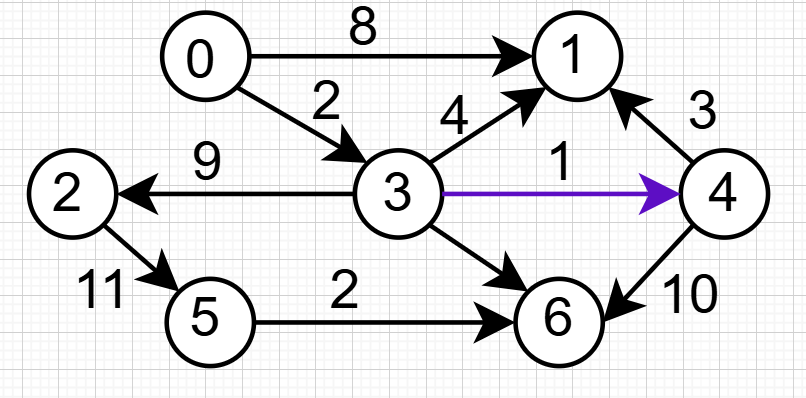


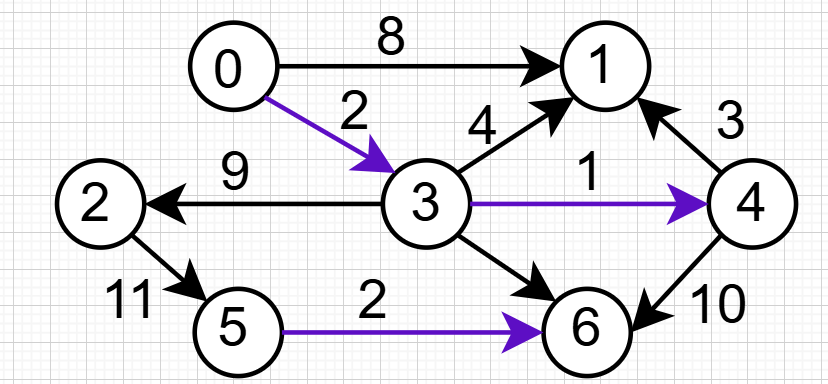
Вес минимального остовного дерева: (0→3),(3→4),(4→1),(3→2),(2→5) ,(5→6) = 2+1+3+9+11+2=28

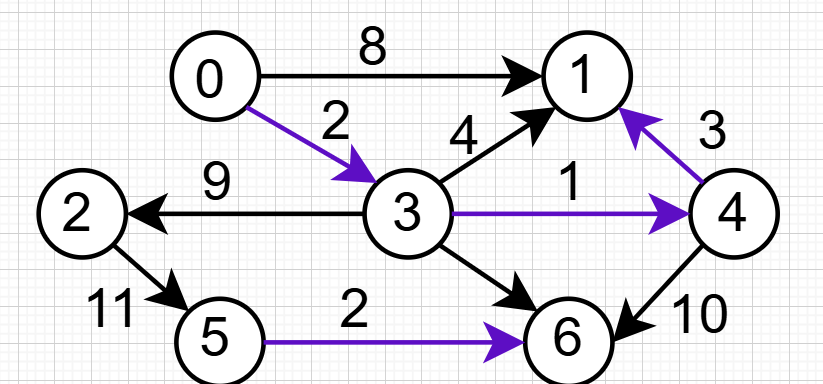
***Задание 7.*** По графу, соответствующему варианту составить минимальное остовное дерево по алгоритму Крускала. Шаги построения отразить в отчете.

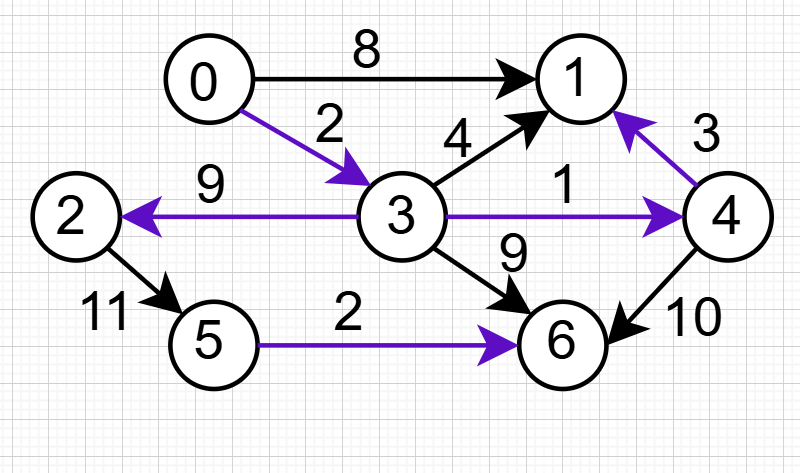
Алгоритм **Крцскала** — это жадный алгоритм для нахождения минимального остовного дерева в графе. Он строит остовное дерево, начиная с рёбер минимального веса и добавляя их в дерево, если они не образуют цикл.

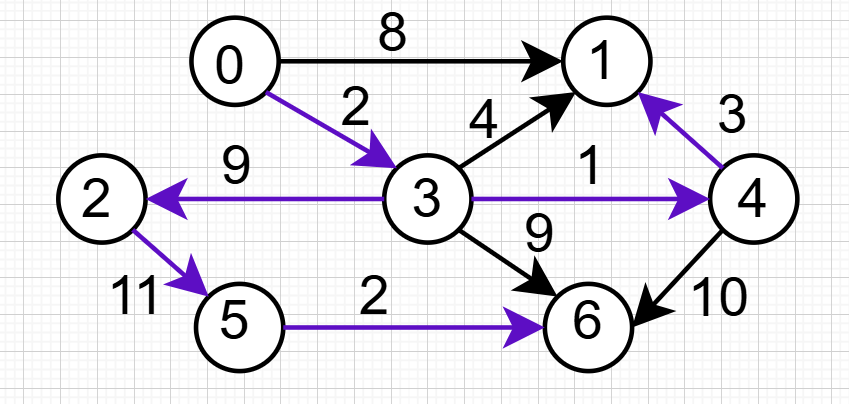
Из всех ребер выбираем ребро с минимальной стоимостью (весом) и включаем это ребро в остовное дерево. Повторяем так далее, однако смотрим на то, чтобы следующее ребро соединяло вершины из разных компонент (чтобы не было циклов)

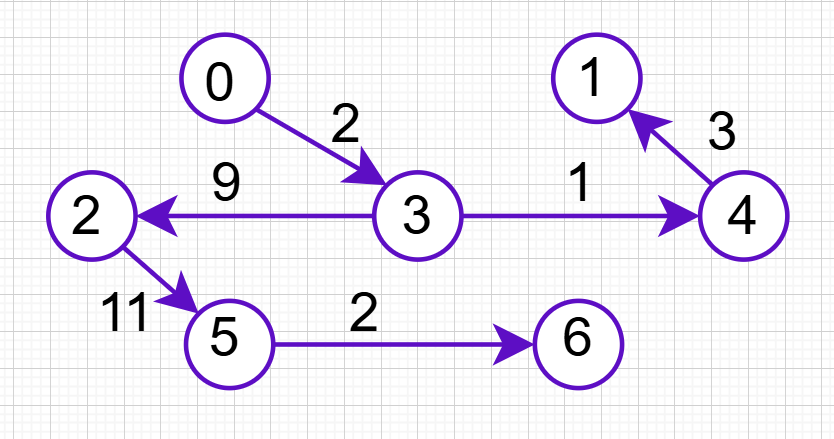
1) 

2)-3) 

4) 

5) 

6) 



2+1+3+9+11+2=28

**Вывод:** была освоена сущность и программная реализация: способов представления графов; алгоритмов поиска в ширину и глубину; алгоритма топологической сортировки графов. Был разобран алгоритм Прима и алгоритм Крускала.