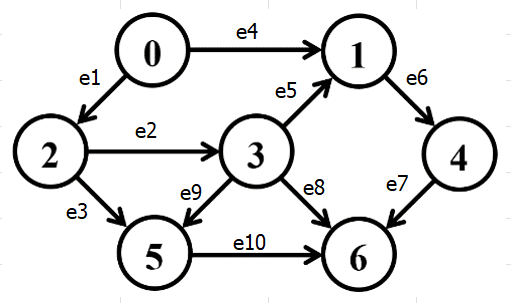
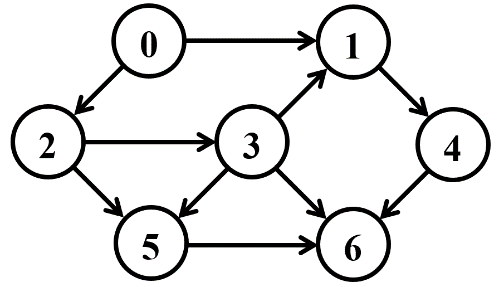
**Лабораторная работа №6. Алгоритмы на графах**

**Вариант 9**

**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

Решение:

Граф моего варианта выглядит так.



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

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

Матрица смежности

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | е1 | е2 | е3 | е4 | е5 | е6 | е7 | е8 | е9 | е10 |
| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | -1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | -1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | -1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 4 | 0 | 0 | 0 | 0 | -1 | -1 | 1 | 0 | 0 | 0 |
| 5 | 0 | 0 | -1 | 0 | 0 | 0 | 0 | 0 | -1 | 1 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | -1 | -1 | 0 | -1 |

Матрица инцидентности

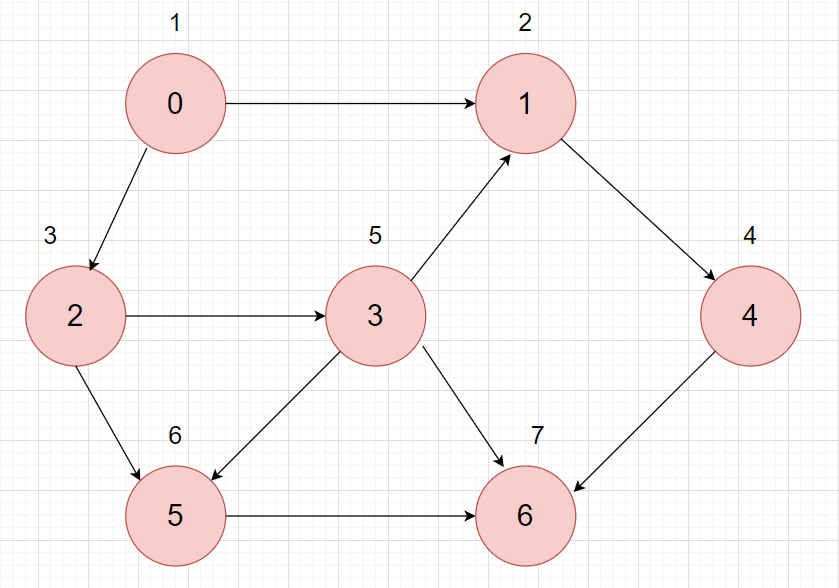
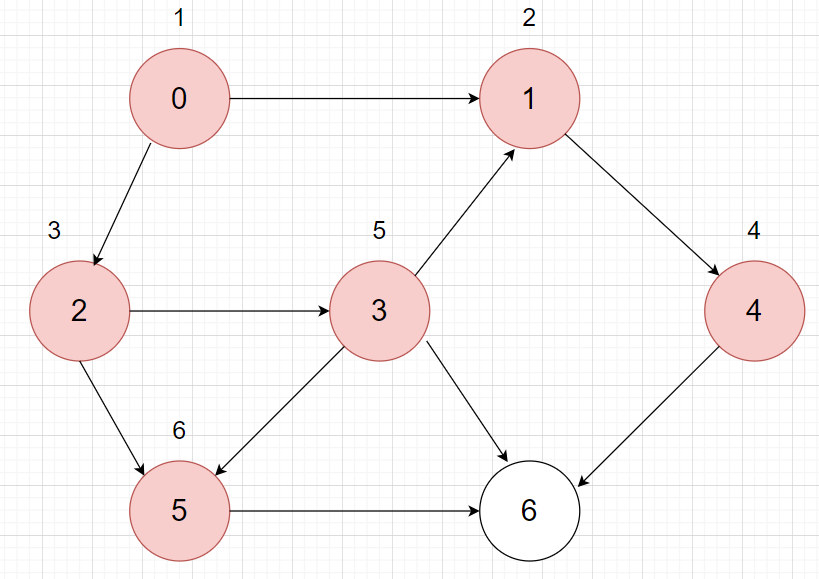
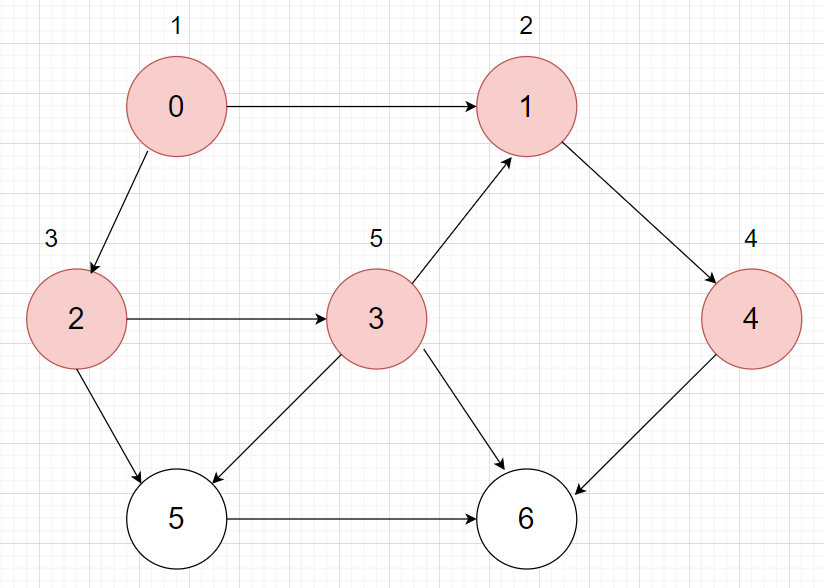
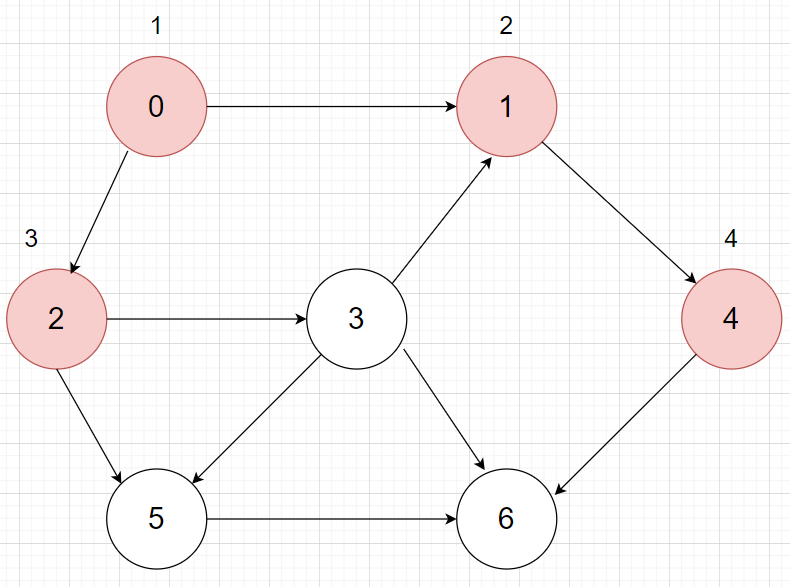
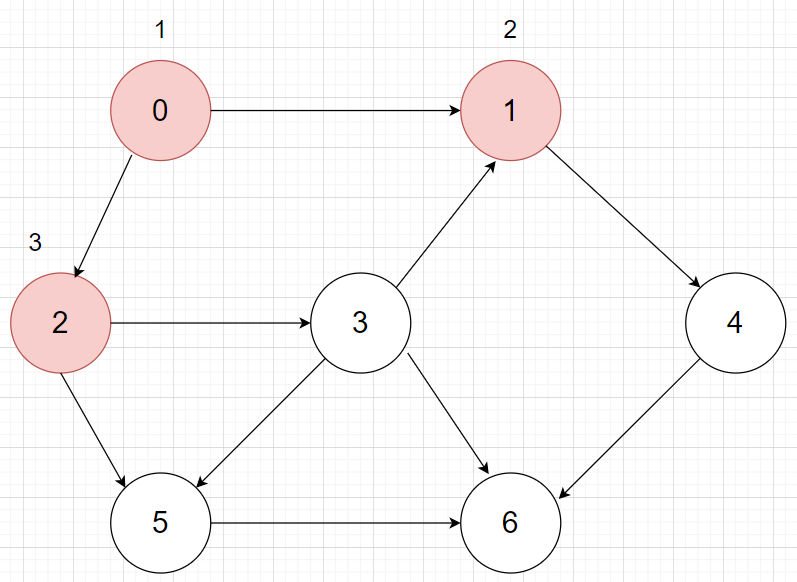
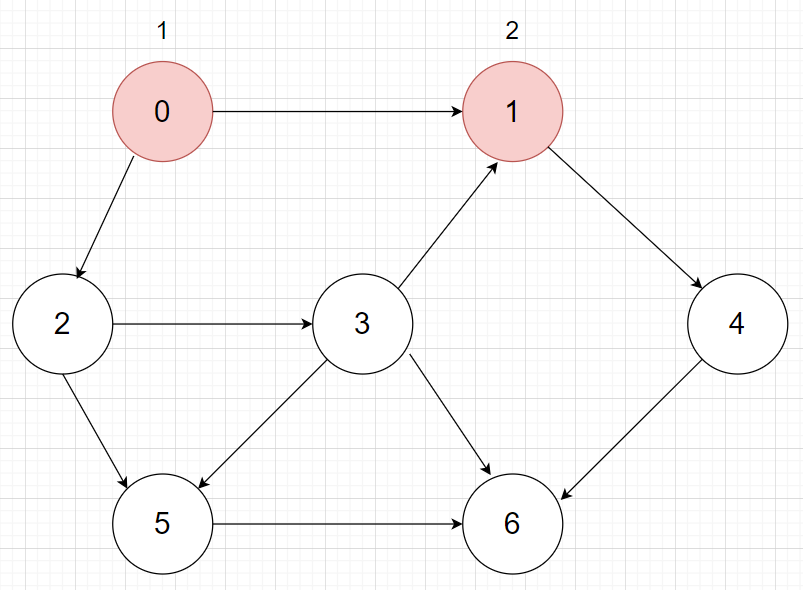
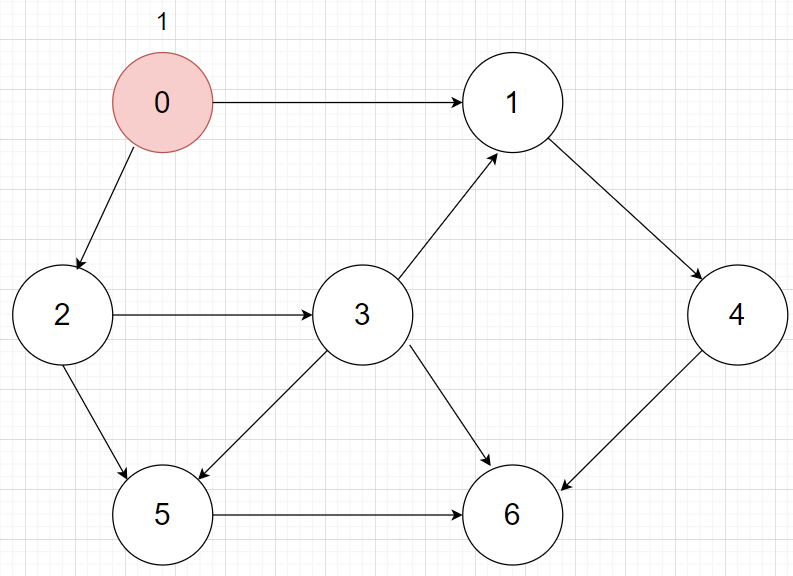
|  |  |
| --- | --- |
| 0 | 2,1 |
| 1 | 4 |
| 2 | 3,5 |
| 3 | 1,5,6 |
| 4 | 6 |
| 5 | 6 |
| 6 | - |

Список 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.

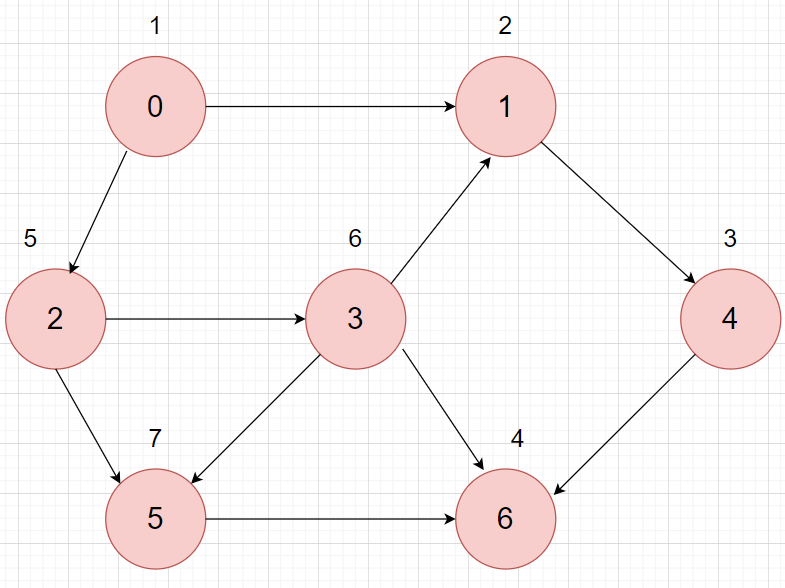
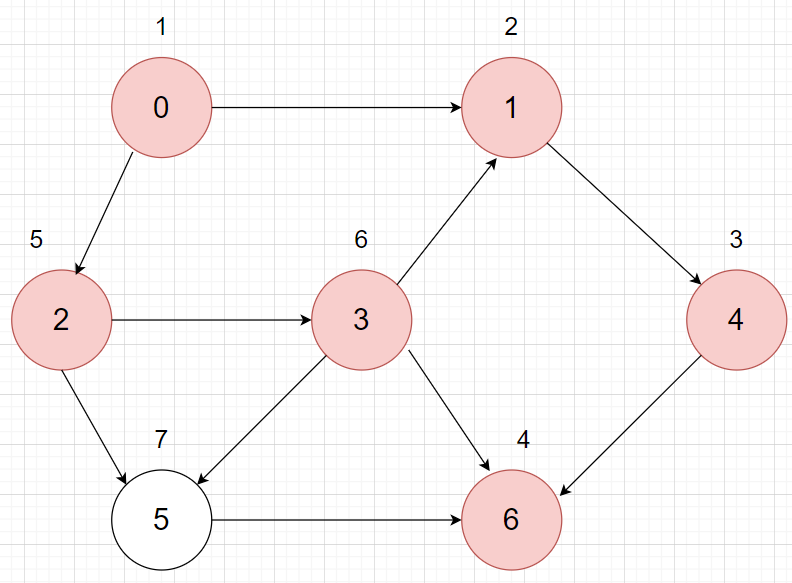
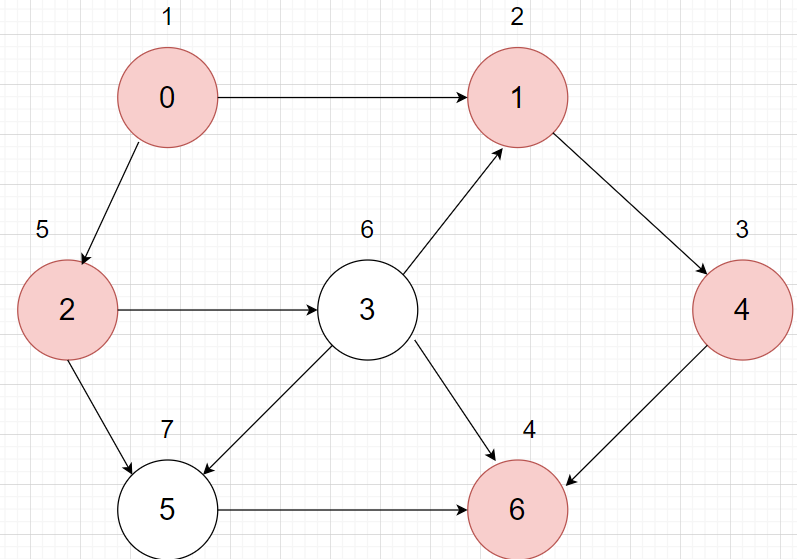
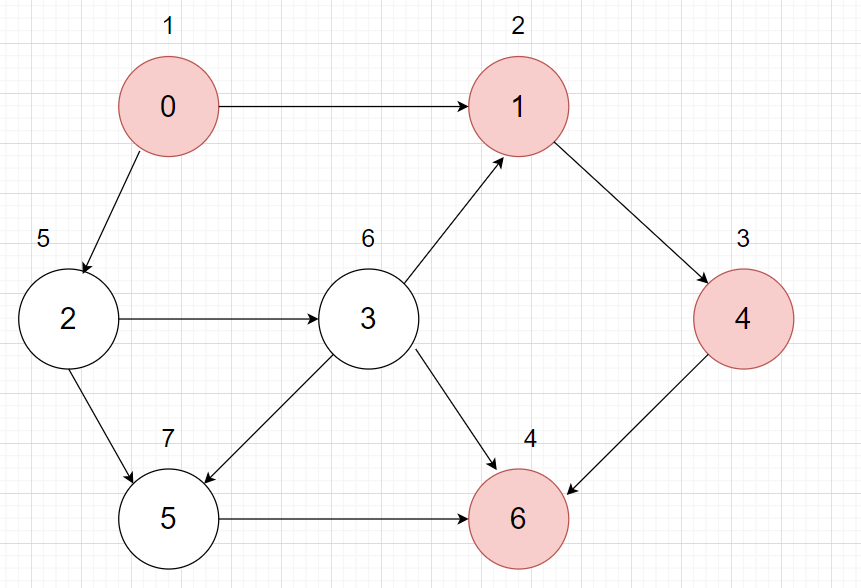
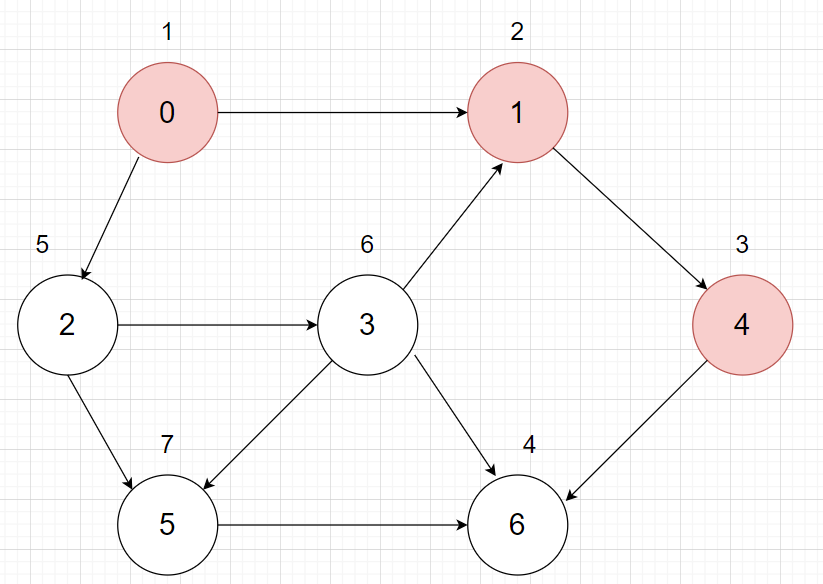
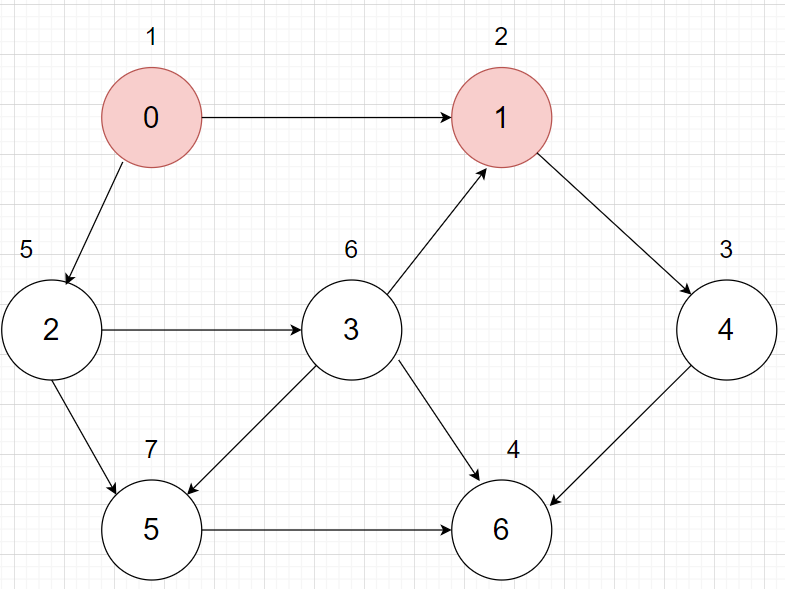
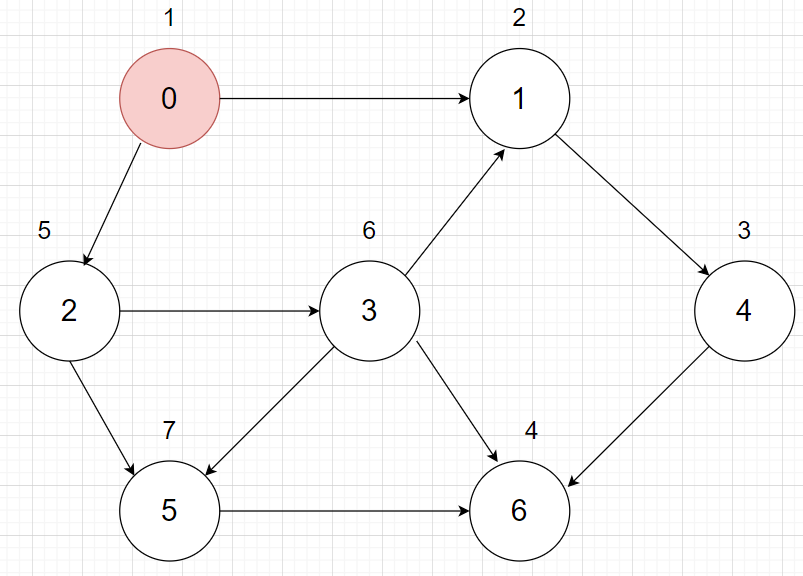
BFS

BFS - это алгоритм поиска в дереве или графе, который начинается с корневой вершины и посещает все вершины на одном уровне, прежде чем переходить к следующему уровню. BFS использует очередь для хранения вершин, которые нужно посетить. Каждый раз, когда BFS посещает вершину, он добавляет все ее непосещенные соседи в очередь и продолжает поиск. BFS останавливается, когда очередь пуста или когда найдена целевая вершина.



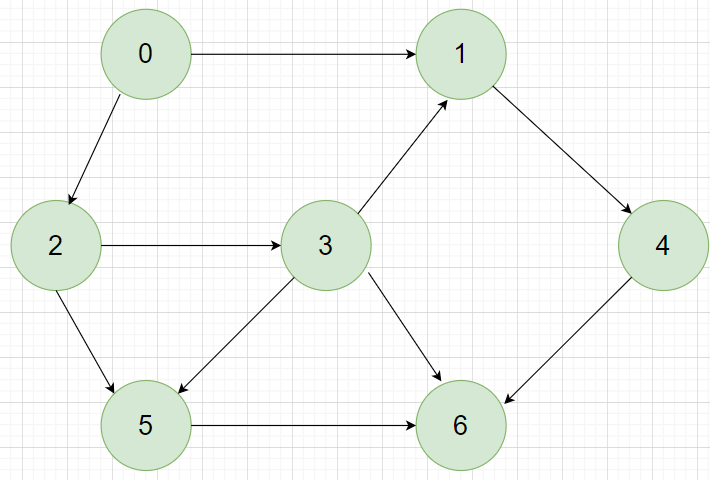
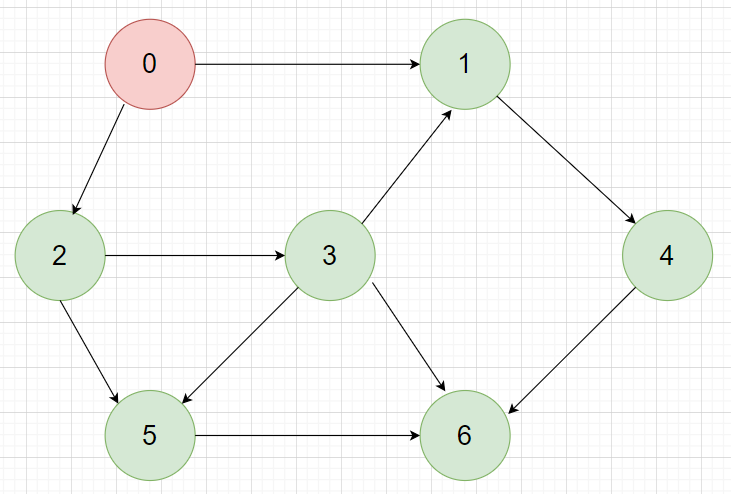
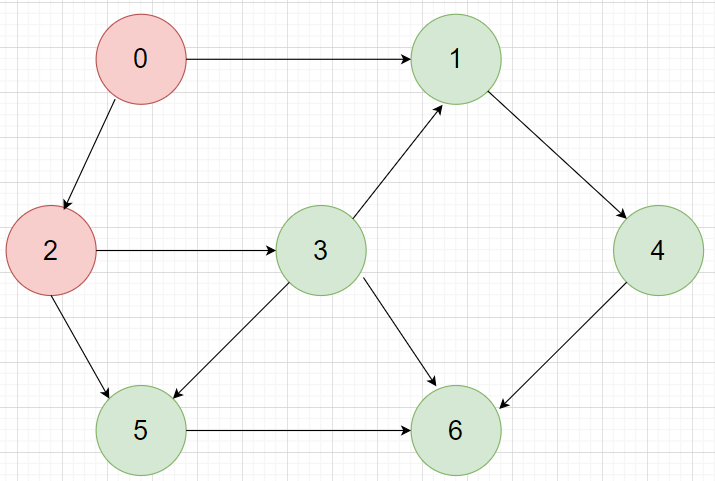
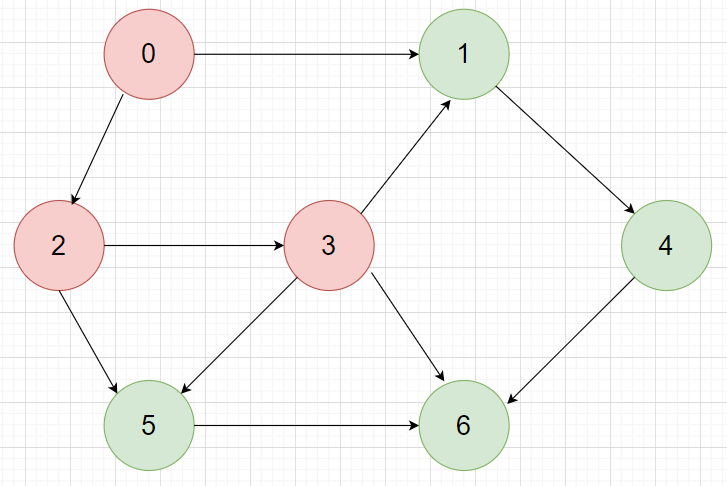
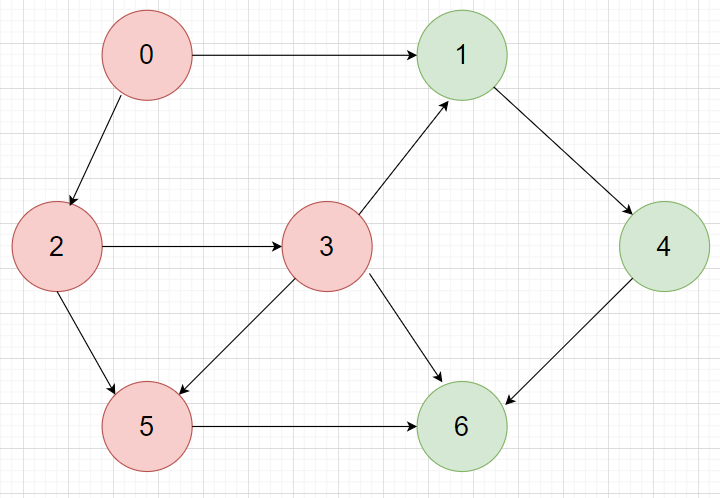
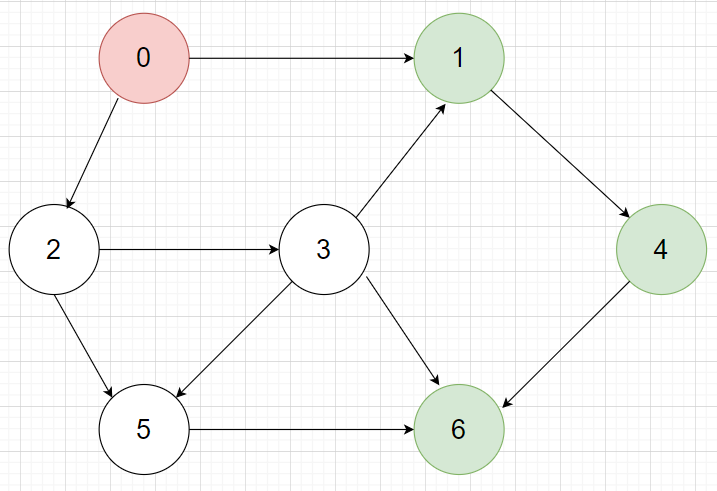
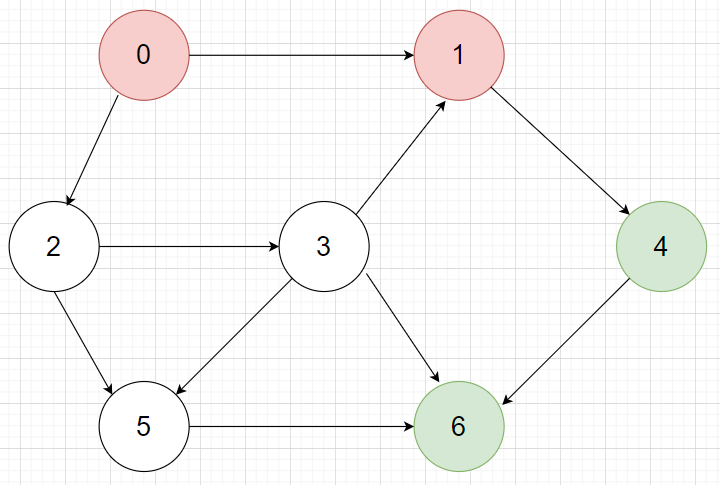
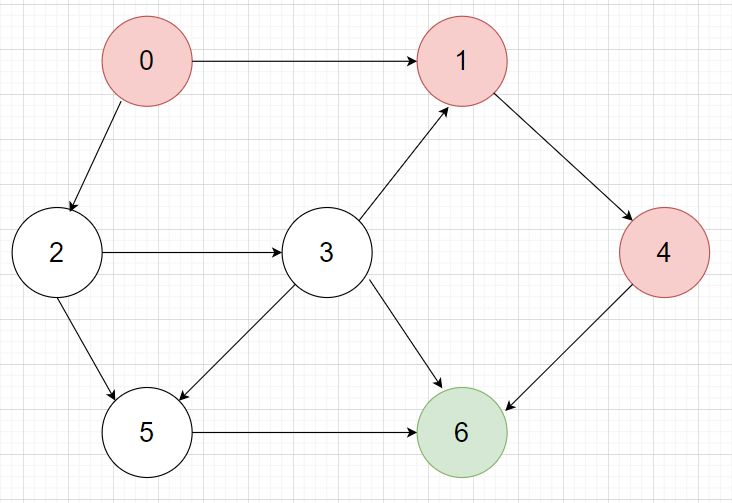
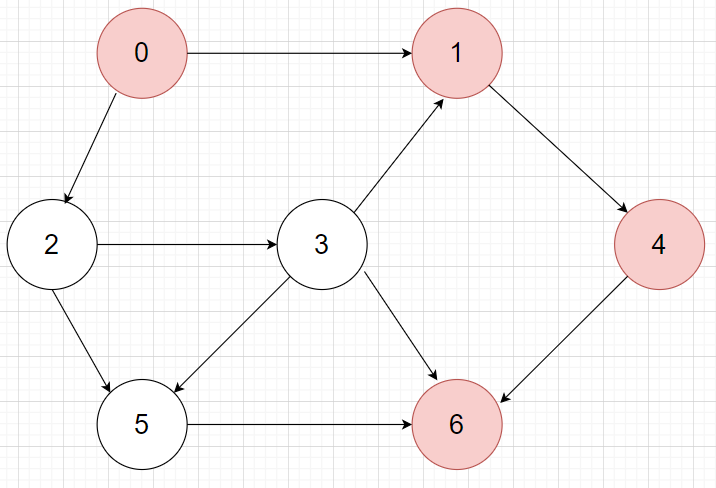
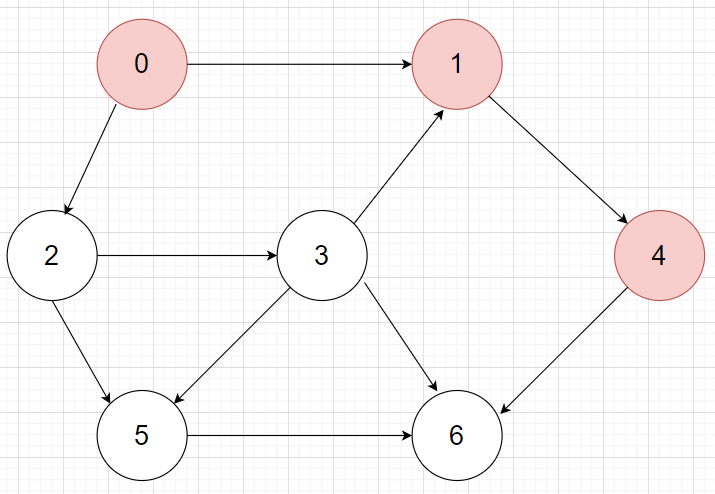
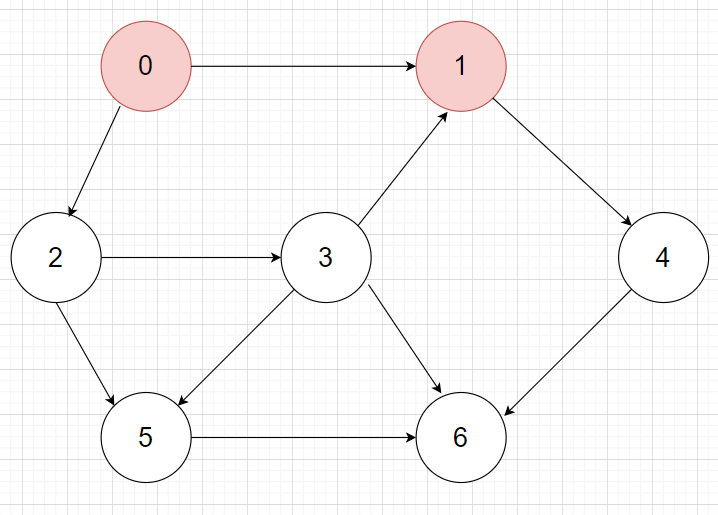
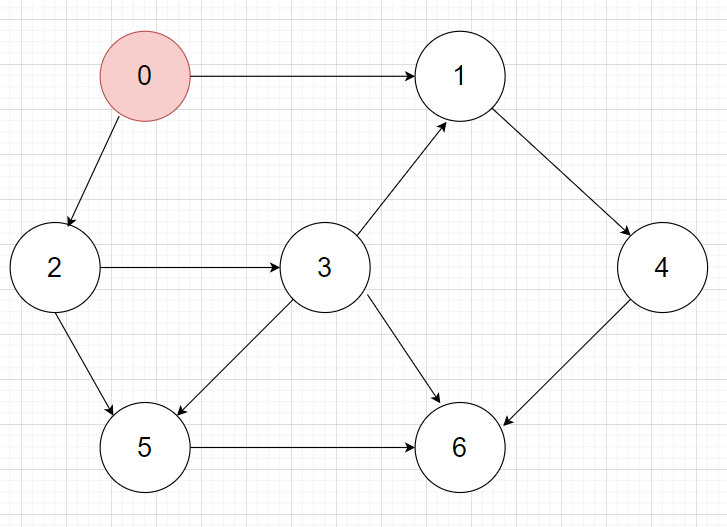
DFS

DFS - это алгоритм обхода или поиска в дереве или графе, который начинается с корневой вершины (или любой произвольной вершины в случае графа) и идет вглубь графа, посещая каждую ветвь до конца, прежде чем возвращаться к предыдущей вершине. DFS использует стек для хранения вершин, которые нужно посетить. Каждый раз, когда DFS посещает вершину, он добавляет одного из ее непосещенных соседей в стек и продолжает обход. DFS останавливается, когда стек пуст или когда найдена целевая вершина.

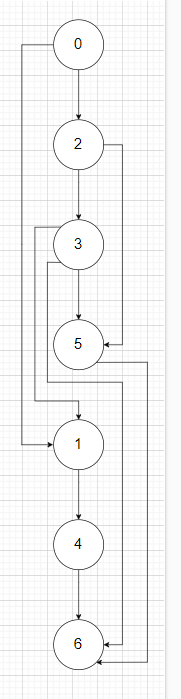


Топологическая сортировка

Топологическая сортировка - это упорядочивание вершин бесконтурного ориентированного графа согласно частичному порядку, заданному ребрами орграфа на множестве его вершин. Это означает, что если в графе есть ребро от вершины A к вершине B, то в топологическом порядке A должна быть перед B. Топологическая сортировка существует только для ациклических графов, то есть графов без циклов. Топологическая сортировка может быть неоднозначной, то есть существовать несколько способов упорядочить вершины графа. Топологическая сортировка может быть выполнена с помощью алгоритма DFS или алгоритма Кана.

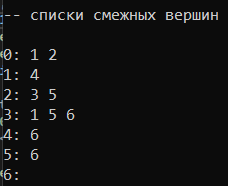


Результат сортировки

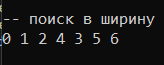


0 2 3 5 1 4 6

**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.



Списки смежных вершин



Поиск в ширину

|  |
| --- |
| #include "stdafx.h"  #include "BFS.h"  void BFS::init(const graph::AList& al, int s)  {  this->al = &al;  this->c = new Color[this->al->nV];  this->d = new int[this->al->nV];  this->p = new int[this->al->nV];  for (int i = 0; i < this->al->nV; 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;  } |

BFS.cpp

|  |
| --- |
| #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.h

|  |
| --- |
| #pragma once  #include <list>  #include "Graph.h"  namespace graph  {  struct AList;  struct AMatrix // матрица смежности  {  int nV; // количество вершин  int \*mr; // матрица  AMatrix(int n, int mr[]); // создать матрицу n\*n и  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 nV; // количество вершин  std::list<int> \*mr; // массив списков  void create(int n); // создать массив пустых списков  AList(int n, int mr[]); // создать списковое представление  AList(const AMatrix& am); // создать списковое представление  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.h

|  |
| --- |
| #include "stdafx.h"  #include <iostream>  #include "Graph.h"  #include "BFS.h"  #include "DFS.h"  int \_tmain(int argc, \_TCHAR\* argv[])  {  int m[7][7] = {  {0, 1, 1, 0, 0, 0, 0},  {0, 0, 0, 0, 1, 0, 0},  {0, 0, 0, 1, 0, 1, 0},  {0, 1, 0, 0, 0, 1, 1},  {0, 0, 0, 0, 0, 0, 1},  {0, 0, 0, 0, 0, 0, 1},  {0, 0, 0, 0, 0, 0, 0},  };  setlocale(LC\_ALL, "rus");  graph::AMatrix g1(7,(int\*) m);  std::cout<<std::endl;  std::cout<<std::endl<< "-- матрица смежности "<< std::endl;  for (int i = 0; i < g1.nV; i++)  {  std::cout<<std::endl;  for (int j = 0; j < g1.nV; 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.nV; 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;  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.nV; i++)  std::cout<< b2.get(i) << " ";  std::cout<<std::endl;  std::cout << std::endl << "Топологическая сортировка" << std::endl;  for (std::vector <int>::iterator i (b2.topological\_sort.begin ()); i != b2.topological\_sort.end (); ++i)  std::cout << \*i << ' ';  std::cout << std::endl;    system("pause");  return 0;  } |
|  |

Main.cpp

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



Результат выполнения

|  |
| --- |
| #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->nV];  this->d = new int[this->al->nV];  this->f = new int[this->al->nV];  this->p = new int[this->al->nV];  this->t = 0;  for (int i = 0; i < this->al->nV; i++)  {  this->c[i] = WHITE;  this->d[i] = this->f[i] = 0;  this->p[i] = NIL;  };  for (int i = 0; i < this->al->nV; 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->nV; k++)  if (this->f[k] < min1 && this->f[k] > min2)  {  min1 = this->f[k];  ntx = k;  };  min2 = min1; min1 = INF;  };  return ntx;}; |

DFS.cpp

|  |
| --- |
| #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.h

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



Результат выполнения

|  |
| --- |
| #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->nV];  this->d = new int[this->al->nV];  this->f = new int[this->al->nV];  this->p = new int[this->al->nV];  this->t = 0;  for (int i = 0; i < this->al->nV; i++)  {  this->c[i] = WHITE;  this->d[i] = this->f[i] = 0;  this->p[i] = NIL;  };  for (int i = 0; i < this->al->nV; 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->nV; k++)  if (this->f[k] < min1 && this->f[k] > min2)  {  min1 = this->f[k];  ntx = k;  };  min2 = min1; min1 = INF;  };  return ntx;  }; |

DFS.cpp

|  |
| --- |
| #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.h

**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;

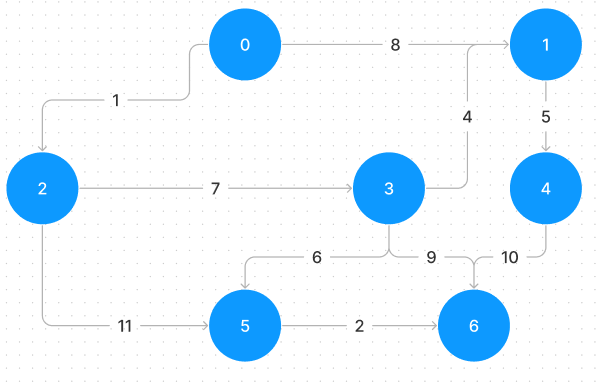
W(e4,3)=4; W(e3,4)=1;

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

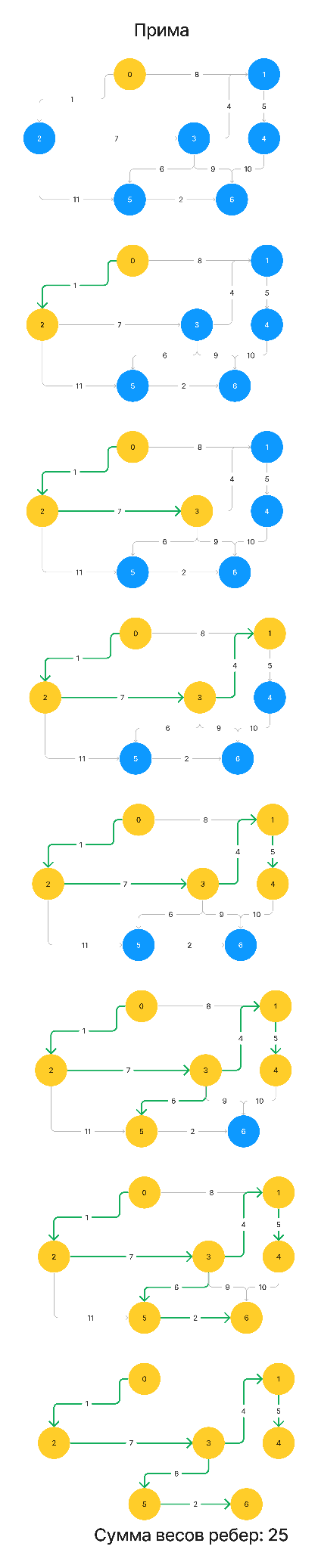
W(e5,6)=2; W(e6,5)=6;

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

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

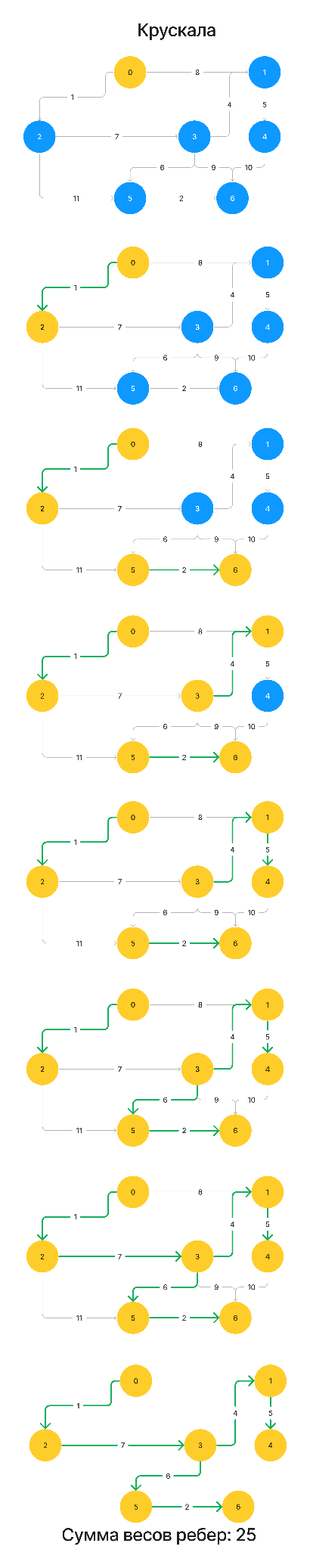


Решение



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

Минимальное дерево по алгоритму Крускала. Для построения дерева по алгоритму Крускала мы должны выделять самые минимальные пары узлов. Для начала мы выделили. В результате мы можем увидеть что алгоритм Прима и Крускала дали одинаковый результат.



**Вывод**: Мы освоили 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.