# 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.

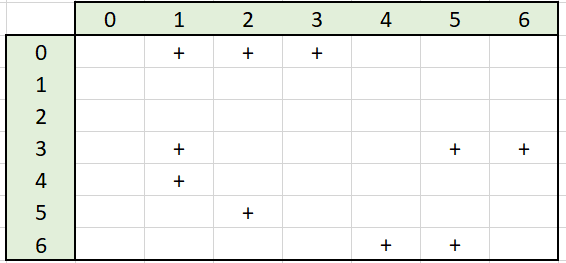
***Задание 1.*** Ориентированный граф **G** взять в соответствии с вариантом. Представить его в отчете в виде матрицы смежности, матрицы инцидентности, списка смежных вершин.

|  |  |
| --- | --- |
| Вариант | Граф G |
| 14 |  |

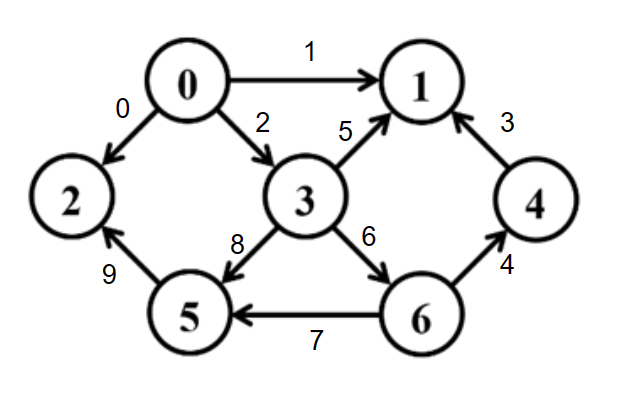
Матрица смежности - это квадратная матрица размера n x n, где n - число вершин в графе. Эта матрица используется для представления графа в виде таблицы, где каждый элемент матрицы a\_ij показывает, существует ли ребро между вершинами i и j.

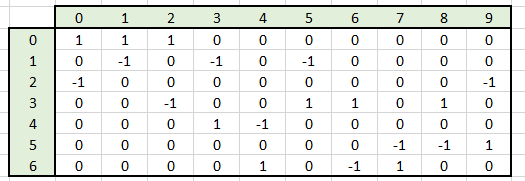
Если в графе есть ребро между вершинами i и j, то a\_ij равно 1. Если ребра между этими вершинами нет, то a\_ij равно 0. Так как граф ориентированный, матрица смежности не будет симметричной относительно главной диагонали.

**Матрица смежности:**









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.

Например, если в графе есть вершина A, которая имеет ребра, соединяющие ее с вершинами B, C и D, то список смежных вершин для вершины A будет содержать вершины B, C и D.

Список смежных вершин может быть представлен в виде массива, списка или любой другой структуры данных, в которой хранятся вершины, с которыми данная вершина имеет ребра.

**Список смежных вершин:**

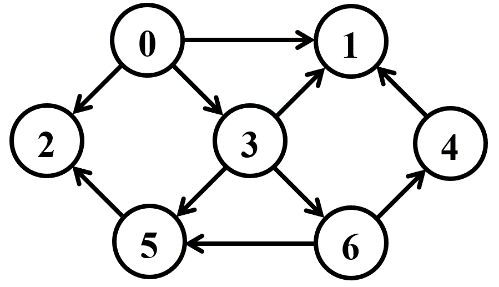
0 {1, 2, 3} 1 {-} 2 {-} 3 {1, 5, 6} 4 {1}

5 {2} 6 {4, 5}

***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

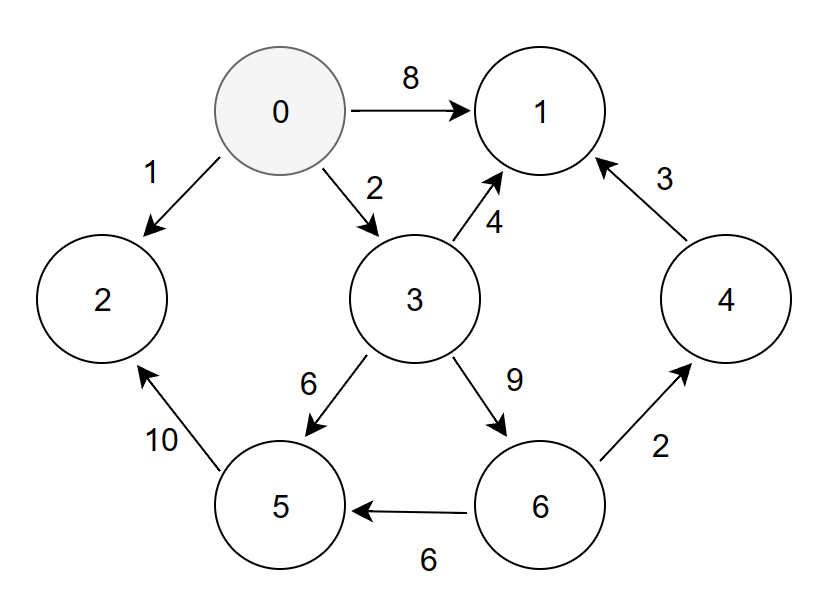
**Поиск в ширину:**



1.Посещённые вершины: {}

Очередь: {0}

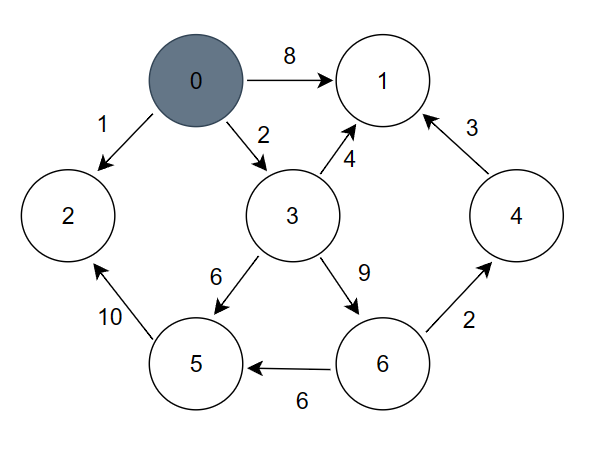
Текущая вершина: {}



2.Посещённые вершины: {0}

Очередь: {}

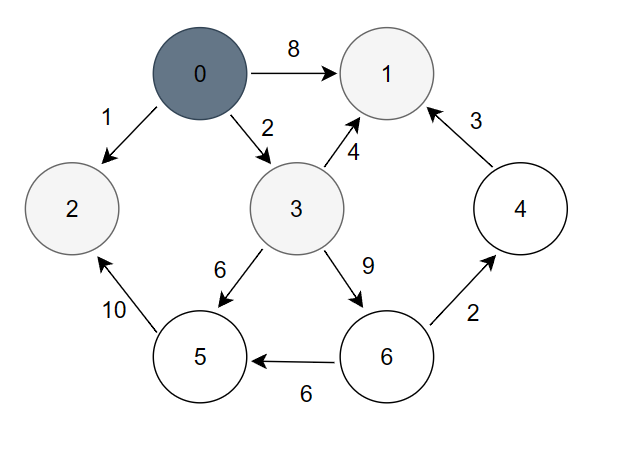
Текущая вершина: {0}



3.Посещённые вершины: {0}

Очередь: {1, 2, 3}

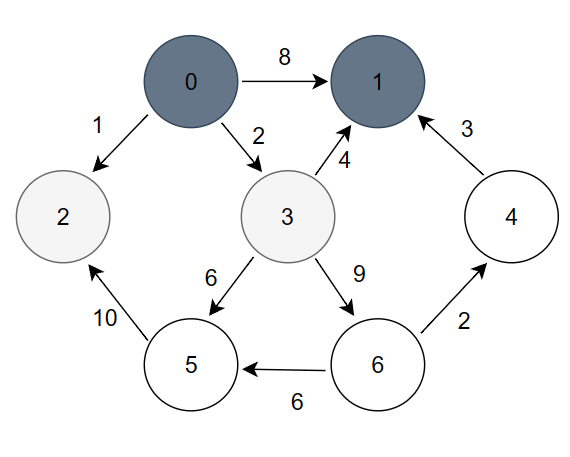
Текущая вершина: {0}



4.Посещённые вершины: {0, 1}

Очередь: {2, 3}

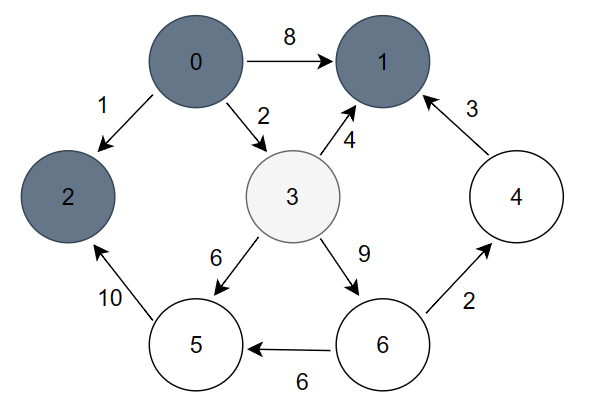
Текущая вершина: {1}



5.Посещённые вершины: {0, 1, 2}

Очередь: {3}

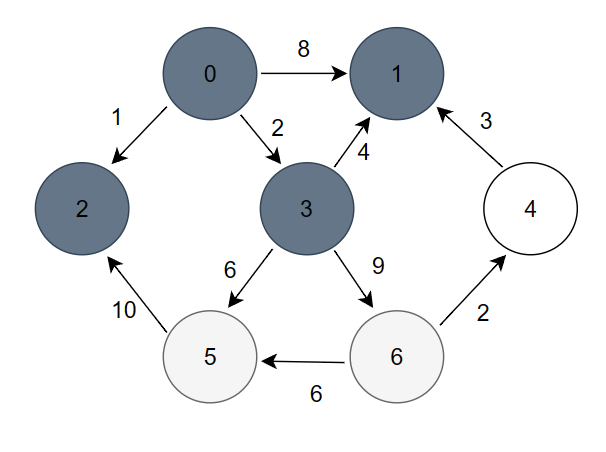
Текущая вершина: {2}



6.Посещённые вершины: {0, 1, 2, 3}

Очередь: {5, 6}

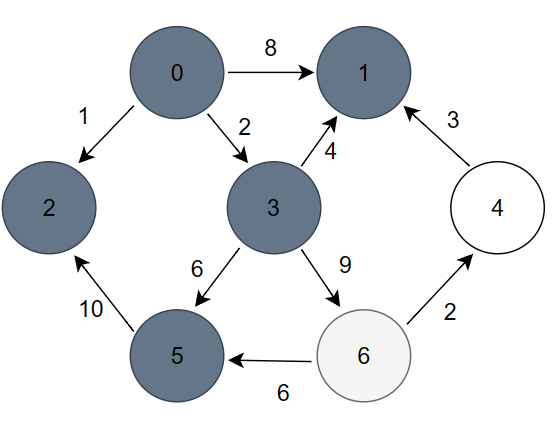
Текущая вершина: {3}



7.Посещённые вершины: {0, 1, 2, 3, 5}

Очередь: {6}

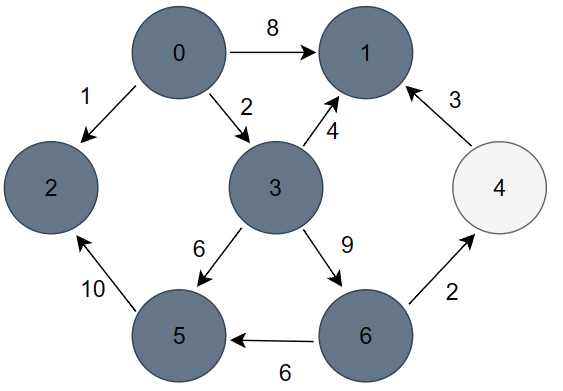
Текущая вершина: {5}



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

Очередь: {4}

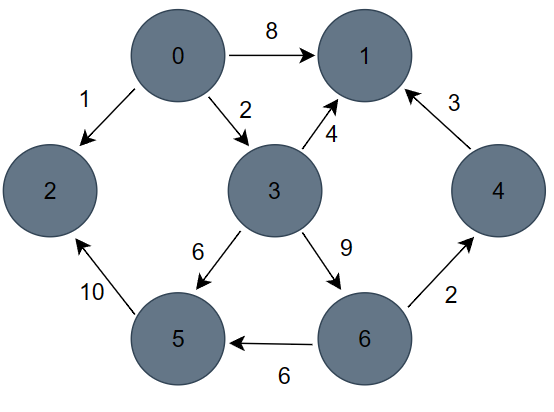
Текущая вершина: {6}



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

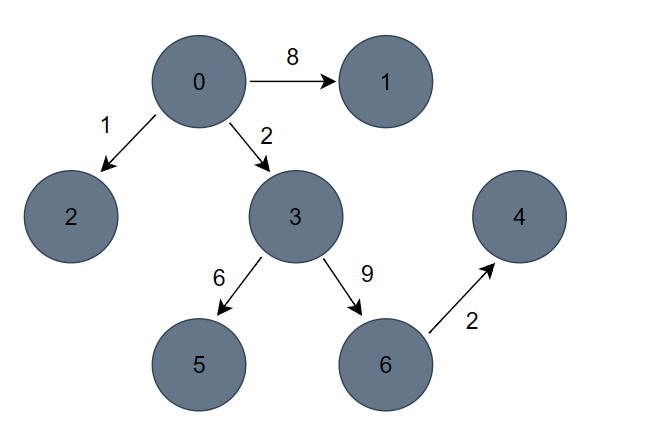
Очередь: {}

Текущая вершина: {4}



Очередь пуста – следует конец обхода.

***BFS-дерево*** – это дерево, множество вершин которого является подмножеством вершин исходного графа, связанных дугами в порядке их посещения (в соответствии с массивом **P**), а корнем – стартовая вершина.



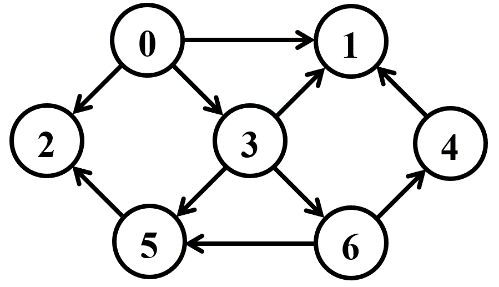
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Массив предыдущих вершин | - | 0 | 0 | 0 | 6 | 3 | 3 |

**Алгоритм BFS** сводится к следующей последовательности шагов.

1. Инициализировать массивы **С**, **D**, **P**. Стартовую вершину **s** поместить в очередь **Q**. и окрасить в серый цвет: **C[s] = G**. Для стартовой вершины установить расстояние, равное нулю: **D[s] = 0**.
2. Если очередь **Q** пуста, то работа алгоритма завершена, в противном случае перейти к следующему шагу.
3. Выбрать из очереди **Q** вершину **k** и окрасить ее в черный цвет: **С[k] = B**.
4. Построить множества **J** вершин белого цвета смежных вершине **k**. Если таких вершин нет, то перейти к шагу 2, иначе – к следующему шагу.
5. Каждую вершину **j** из множества **J** поместить в очередь **Q**. Обычно (но не обязательно) в очередь вершины помещаются в порядке возрастания номеров.
6. Каждую вершину **j** из множества **J** окрасить в серый цвет: **С[j] = G**.
7. Для каждой вершины **j** из множества **J** вычислитьрасстояние: **D[j] = D[k] + 1**.
8. Для каждой вершины **j** из множества **J** указать предшествующую вершину: **P[j] = k**.
9. Перейти к шагу 3.

**Алгоритм поиска (или обхода) в глубину** (англ. depth-first search, DFS) позволяет построить обход графа, при котором посещаются все вершины, доступные из начальной вершины.

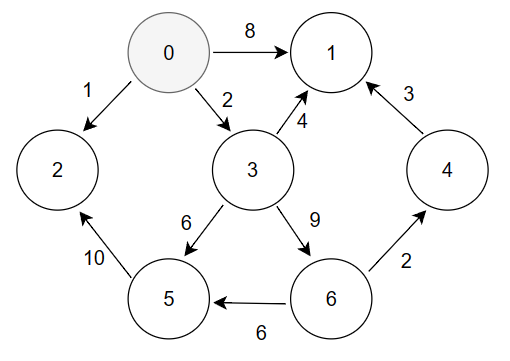
**Алгоритм поиска в глубину:**



1.Посещённые вершины: {}

Cтек: {0}

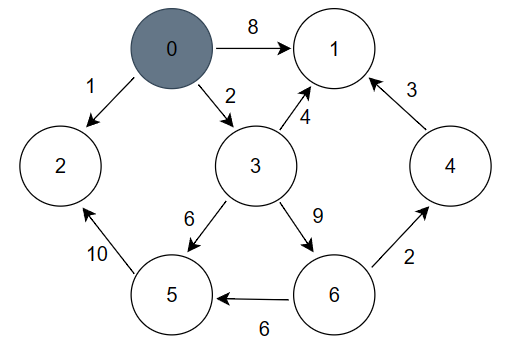
Текущая вершина: {}



2.Посещённые вершины: {0}

Cтек: {}

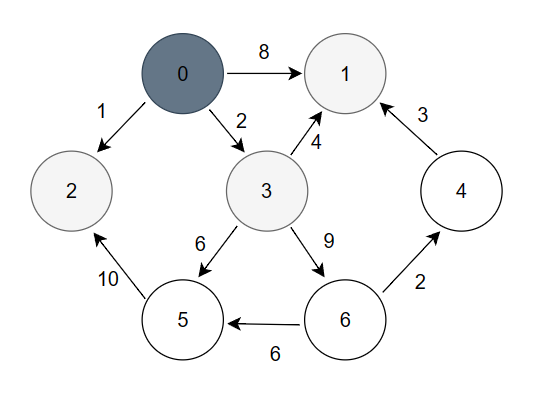
Текущая вершина: {0}



3.Посещённые вершины: {0}

Cтек: {1, 2, 3}

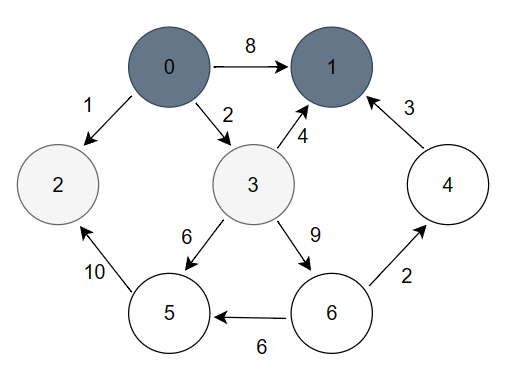
Текущая вершина: {0}



4.Посещённые вершины: {0, 1}

Cтек: { 2, 3}

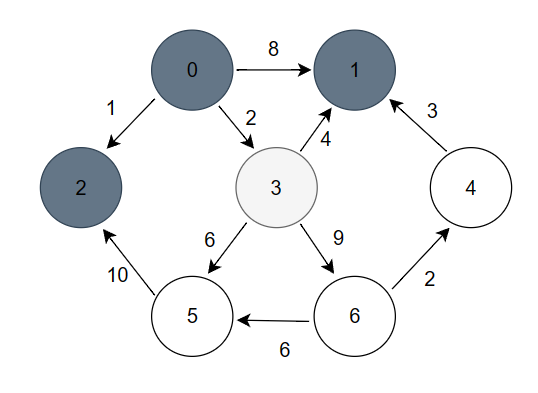
Текущая вершина: {1}



5.Посещённые вершины: {0, 1, 2}

Cтек: {3}

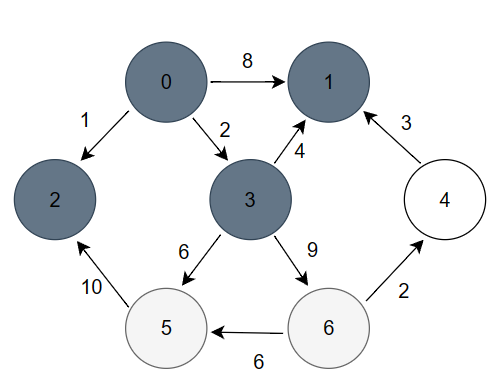
Текущая вершина: {2}



6.Посещённые вершины: {0, 1, 2, 3}

Cтек: {5, 6}

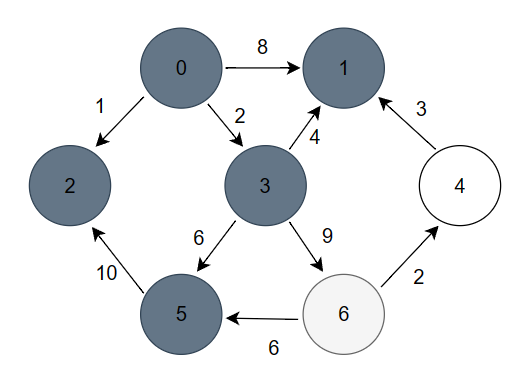
Текущая вершина: {3}



7.Посещённые вершины: {0, 1, 2, 3, 5}

Cтек: {6}

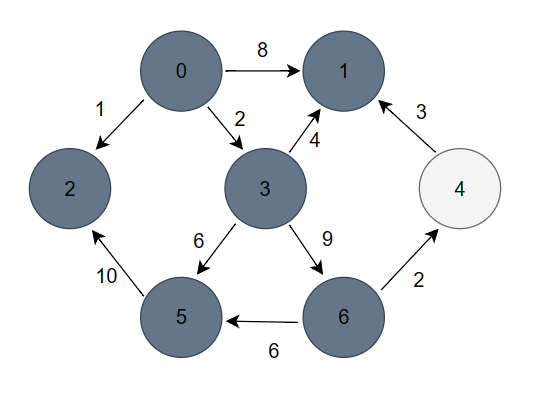
Текущая вершина: {5}



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

Cтек: {4}

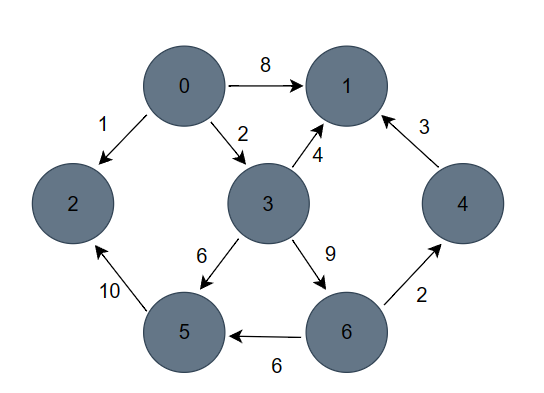
Текущая вершина: {6}



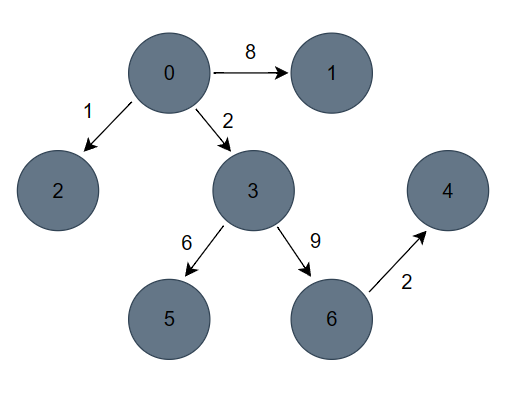
9.Посещённые вершины: {0, 1, 2, 3, 5, 6, 4}

Cтек: {}

Текущая вершина: {4}



Результирующий граф



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Номера предыдущих вершин | - | 0 | 0 | 0 | 6 | 3 | 3 |

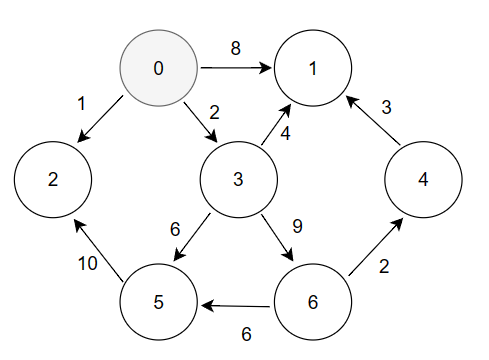
В основе алгоритма DFS лежит рекурсивная процедура **Visit**, имеющая один входной параметр **k** – вершину графа.

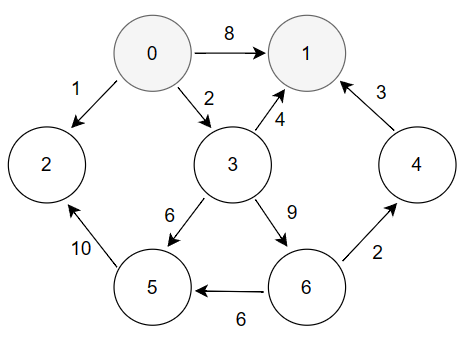
Опишем пошагово процедуру **Visit**.

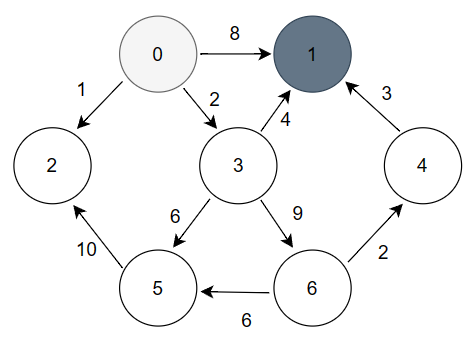
1. Принять параметр **k** – вершину графа.
2. Вершину **k**  окрасить в серый цвет: **C[k] = G**.
3. Увеличить номер шага: **t = t + 1**.
4. Подсчитать расстояние до вершины: **D[k] = t**. Расстояние до вершины в алгоритме DFS совпадает с номером шага, на котором эта вершина была обнаружена (окрашена в серый цвет).
5. Построить множества **J** вершин белого цвета, смежных вершине **k**. Если таких вершин нет, то перейти к шагу 8.
6. Для каждой вершины **j** из множества **J** указать предшествующую вершину: **P[j] = k**.
7. Для каждой вершины **j** из множества **J** выполнить процедуру **Visit**.
8. Вершину **k**  окрасить в черный цвет: **C[k] = B**.
9. Увеличить номер шага: **t = t + 1**.
10. Отметить время фиксации вершины: **F[k] = t**.

**Топологическая сортировка**

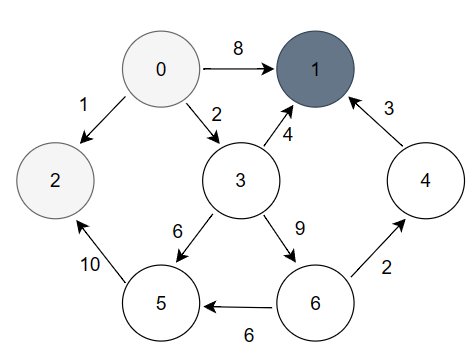
**Топологическая сортировка −** это процедура упорядочивания вершин бесконтурного ориентированного графа, не имеющего циклов (ациклического графа). В результате топологической сортировки для вершин графа определяется такой порядок, что если их расположить на рисунке в соответствии с этим порядком сверху вниз, то дуги будут направлены только от верхних вершин к нижним**.**

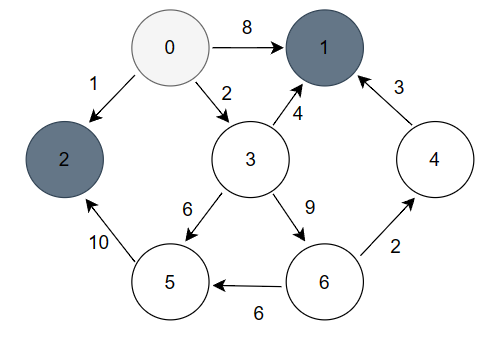




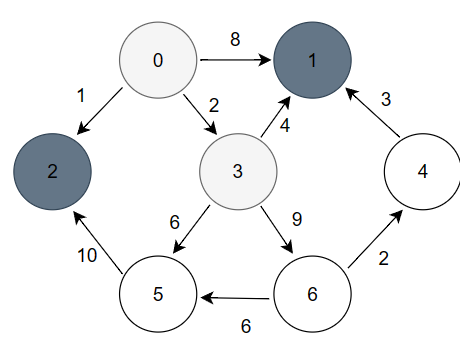


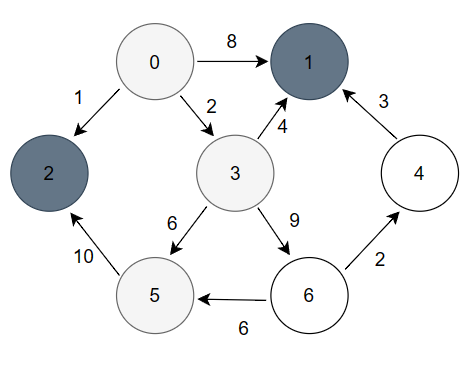
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  | **1** |

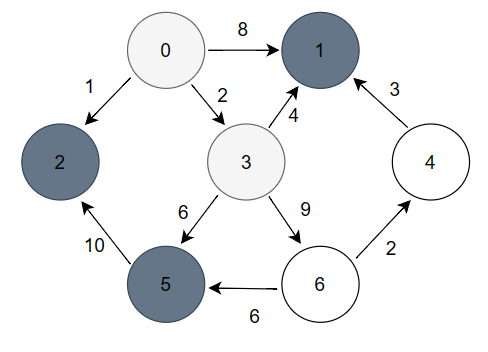




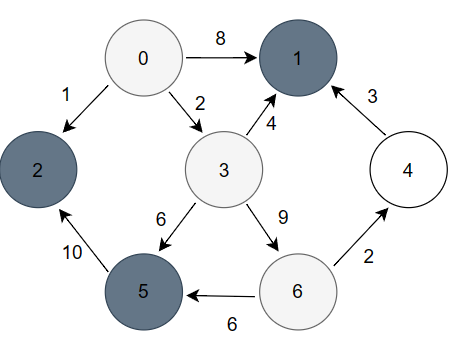
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | **2** | **1** |

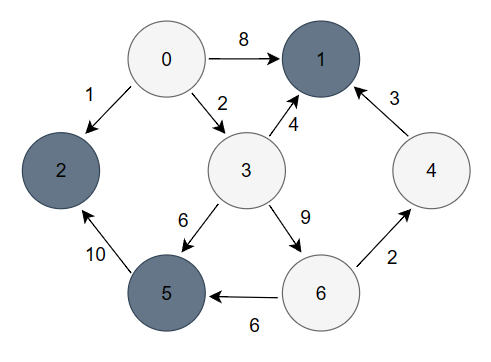


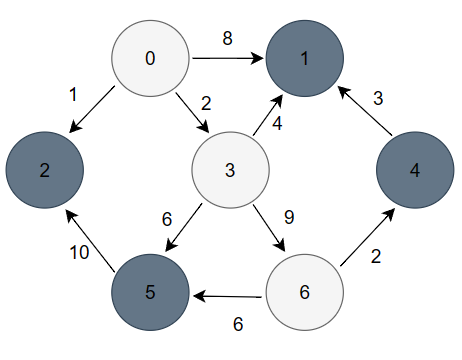




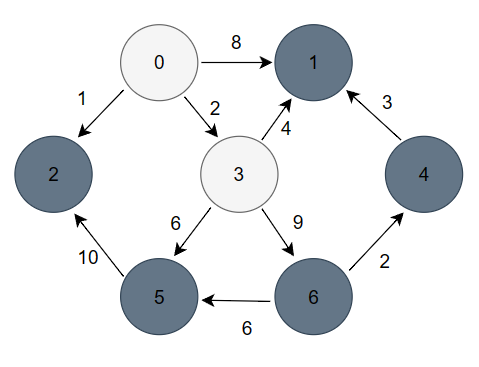
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | **5** | **2** | **1** |



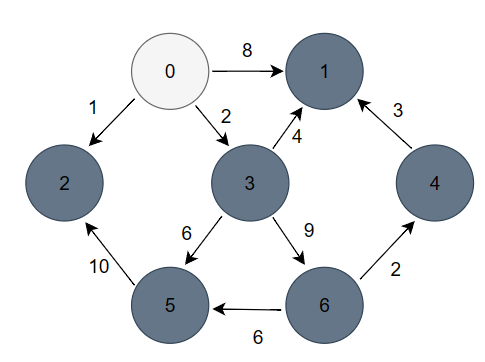




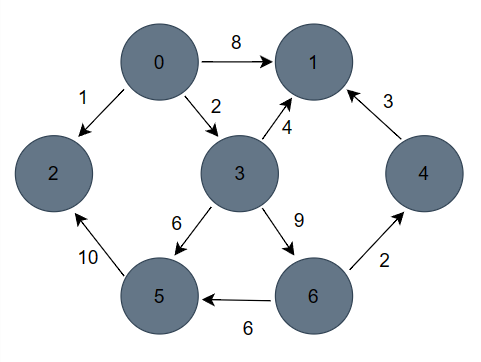
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | **4** | **5** | **2** | **1** |



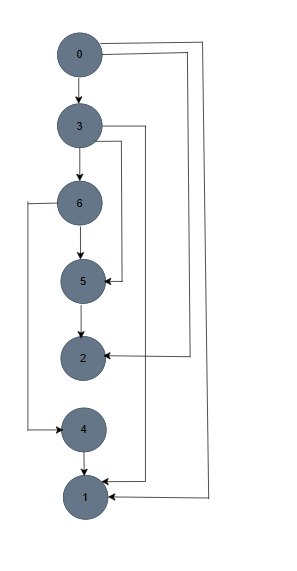
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | **6** | **4** | **5** | **2** | **1** |



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



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



**Результат: 0-3-6-4-5-2-1**

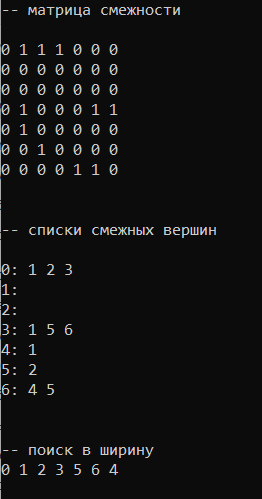
***Задание 3.*** Осуществить программную реализацию алгоритмов на C++. Разработать структуры **AMatrix** и **АList**  для представления ориентированного графа матричным и списковым способом. Разработать функции преобразования из одного способа представления в другой. Разработать функцию **BFS** обхода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

|  |
| --- |
| // ---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  }; |

Листинг 3.1 — файл BFS.h

|  |
| --- |
| // ---BFS.cpp  //  #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 3.2 — файл BFS.cpp

******

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

|  |
| --- |
| // ---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  }; |

Листинг 4.1 — файл DFS.h

|  |
| --- |
| // ---DFS.cpp  //  #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 4.2 — файл DFS.cpp



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

|  |
| --- |
| // Функция для топологической сортировки  void topologicalSort(int v, bool visited[], std::stack<int>& Stack, graph::AList& g)  {  // Помечаем текущую вершину как посещенную  visited[v] = true;  // Рекурсивно вызываем функцию для всех смежных вершин  // Если смежная вершина не была посещена, то рекурсивно вызываем функцию  for (int i = 0; i < g.size(v); i++)  if (!visited[g.get(v, i)])  topologicalSort(g.get(v, i), visited, Stack, g);  // Помещаем текущую вершину в стек  Stack.push(v);  }  . . .  std::cout << std::endl << "-- топологическая сортировка" << std::endl;  // Создаем стек для хранения топологической сортировки  std::stack<int> Stack;  // Массив для хранения информации о посещении вершин  bool \*visited = new bool[g5.n\_vertex];  for (int i = 0; i < g5.n\_vertex; i++) visited[i] = false;  // Вызываем функцию для топологической сортировки  for (int i = 0; i < g5.n\_vertex; i++)  if (visited[i] == false)  topologicalSort(i, visited, Stack, g5);  // Выводим топологически отсортированный граф  while (Stack.empty() == false)  {  std::cout << Stack.top() << " ";  Stack.pop();  }  std::cout << std::endl; |

Листинг 5.1— функция топологической сортировки



***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;

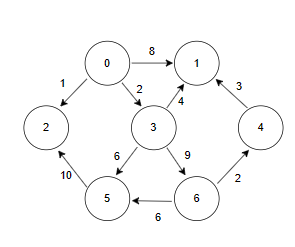
W(e4,3)=4; W(e3,4)=1;

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

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

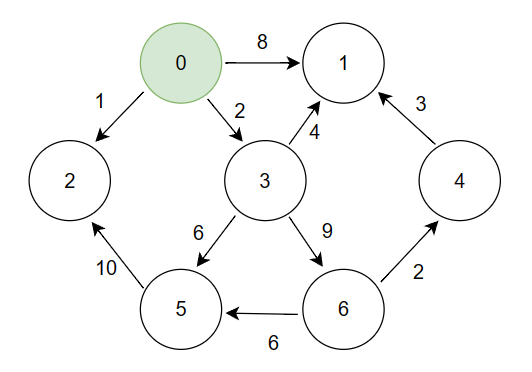
W(e5,3)=3; W(e3,5)=6;

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

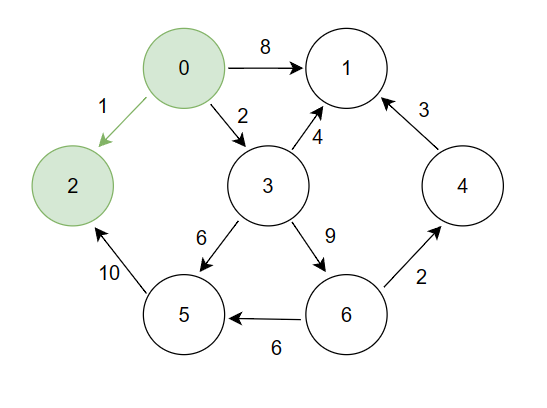


Шаг 1:

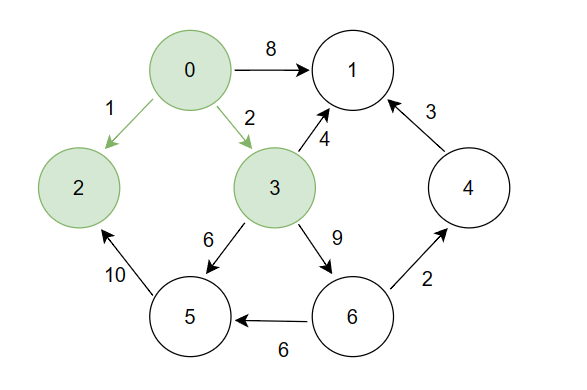
На первом шаге выбирается стартовая вершина (вершина 0) и окрашивается.



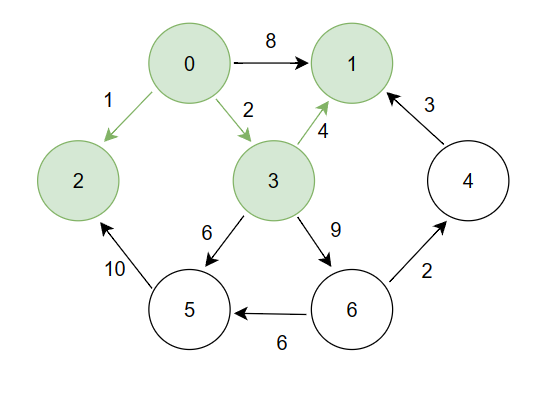
Шаг 2:

Среди всех ребер, инцидентных стартовой вершине, отыскивается ребро, имеющее наименьшую длину (на рисунке – ребро (0, 2)). Вторая (неокрашенная) вершина ребра окрашивается, а само ребро вместе с концевыми вершинами включается в будущее минимальное остовное дерево. 

Шаг 3:

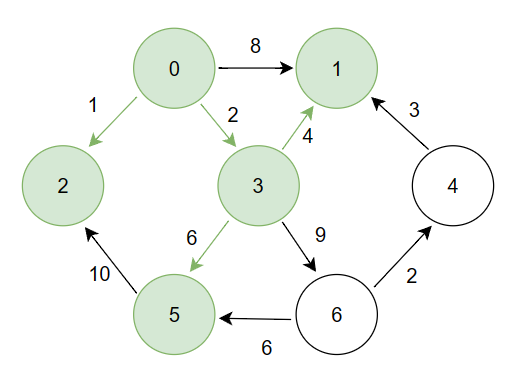
Выбирается ребро с минимальной длиной и одной неокрашенной концевой вершиной. Это ребро (0,3) Неокрашенная вершина окрашивается, выбранное ребра пополняет строящееся минимальное остовное дерево. 

Шаг 4:

Выбирается ребро с минимальной длиной и одной неокрашенной концевой вершиной. Это ребро (3, 1) Неокрашенная вершина окрашивается, выбранное ребра пополняет строящееся минимальное остовное дерево. 

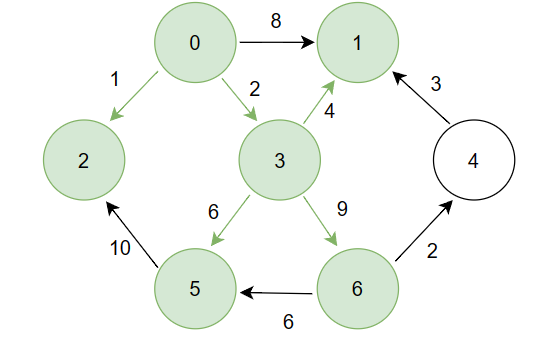
Шаг 5:

Выбирается ребро с минимальной длиной и одной неокрашенной концевой вершиной. Это ребро (3, 5) Неокрашенная вершина окрашивается, выбранное ребра пополняет строящееся минимальное остовное дерево.



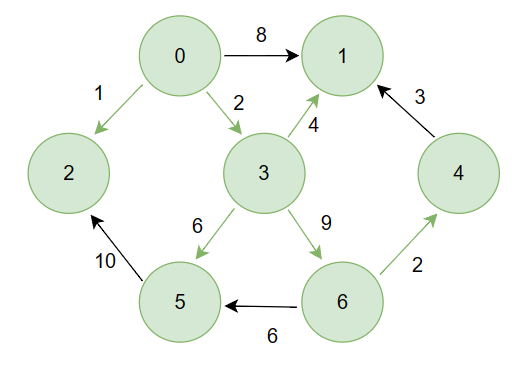
Шаг 6:

Выбирается ребро с минимальной длиной и одной неокрашенной концевой вершиной. Это ребро (3, 6) Неокрашенная вершина окрашивается, выбранное ребра пополняет строящееся минимальное остовное дерево.

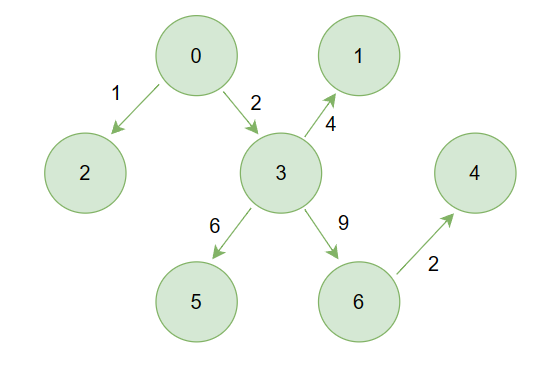


Шаг 7:

Выбирается ребро с минимальной длиной и одной неокрашенной концевой вершиной. Это ребро (5,6) Неокрашенная вершина окрашивается, выбранное ребра пополняет строящееся минимальное остовное дерево.

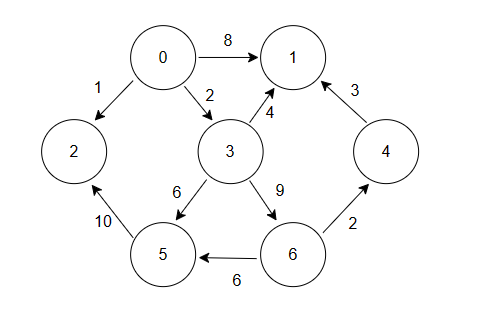


**Результат:**



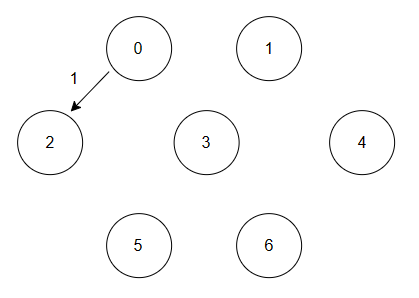
Вес минимального остовного дерева: 24

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

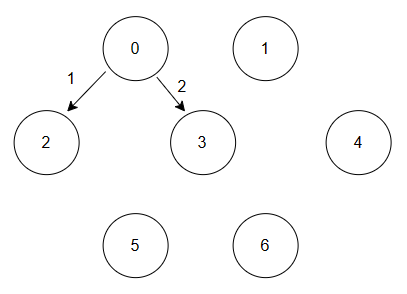


Шаг 1:

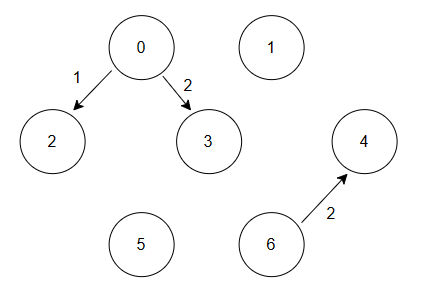
Из всех ребер выбираем ребро с минимальной стоимостью (весом) и включаем это ребро в остовное дерево. Повторяем так далее, однако смотрим на то, чтобы следующее ребро соединяло вершины из разных компонент (чтобы не было циклов)



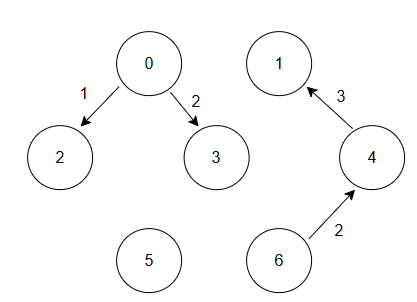
Шаг 2:



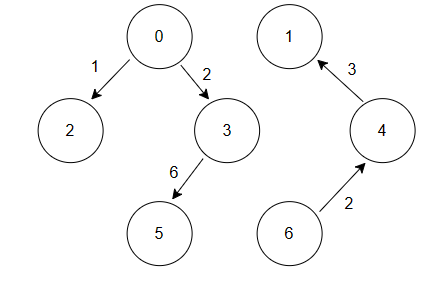
Шаг 3:



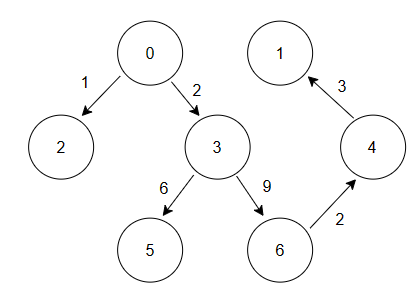
Шаг 4:



Шаг 5:



Шаг 6:



Вес минимального остовного дерева: 23

Вывод: Были освоены сущность и программная реализация способов представления графов; алгоритмов поиска в ширину и глубину; алгоритма топологической сортировки графов. Разобраны алгоритм Прима и алгоритм Крускала.