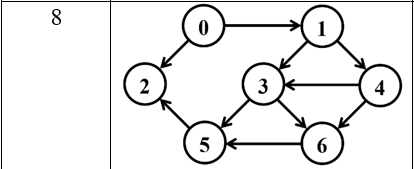
**Лабораторная работа 6. АЛГОРИТМЫ НА ГРАФАХ**

**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***

В соответствии с вариантом выбирает граф:



Составляем матрицу смежности графа:

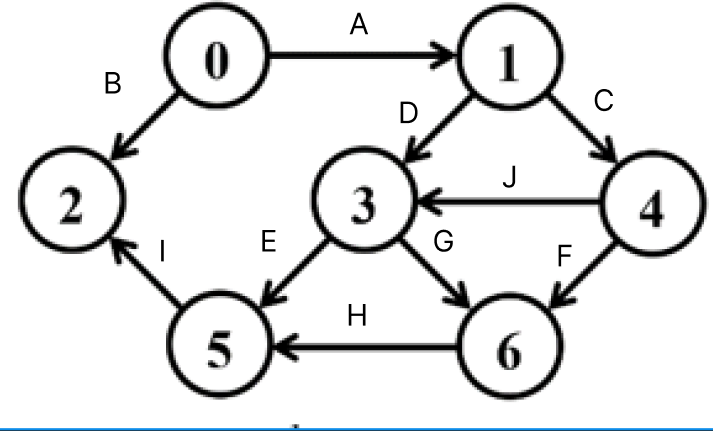
Чтобы составить матрицу смежности в ориентированном графе, мы смотрим на то, стрелка направленна на точку или от точки. Если стрелки направленны от точки, то ставим «1» на соответствующие значения если же нет или стрелки отсутствуют, то ставим «0».

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

Составляем матрицу инцидентности:

Чтобы составить матрицу инцидентности нам нужно пронумеровать все рёбра нашего графа и смотреть на направления стрелки. Если стрелка начинает своё движение, то ставим «1», а если заканчивает, то ставим «-1». Если же стрелки отсутствуют, то ставим «0».

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Матрица инцидентности | | | | | | | | | | |
|  | A | B | C | D | E | F | G | H | I | J |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | -1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | -1 | 0 |
| 3 | 0 | 0 | 0 | -1 | 1 | 0 | 1 | 0 | 0 | -1 |
| 4 | 0 | 0 | -1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 5 | 0 | 0 | 0 | 0 | -1 | 0 | 0 | -1 | 1 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | -1 | -1 | 1 | 0 | 0 |



Опишем список смежный вершин:

Для того, что бы описать список смежный вершин мы смотрим на направление стрелок. Нам нужно находить тех соседей нашей точки, к которым ведёт стрелка т.е

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

***Задание 2.***

**Алгоритм поиска в ширину (BFS).**

Алгоритм графа в ширину (BFS) используется для обхода графа и нахождения кратчайшего пути от начальной вершины до всех остальных вершин графа. Он работает следующим образом:

Создать очередь и добавить в нее начальную вершину.

Создать массив посещенных вершин и пометить начальную вершину как посещенную.

Пока очередь не пуста, извлечь из очереди первую вершину и для каждой смежной с ней вершины, которая еще не была посещена, пометить ее как посещенную и добавить в очередь.

Повторять шаг 3 до тех пор, пока очередь не станет пустой.

В результате работы алгоритма, для каждой вершины графа будет найден кратчайший путь от начальной вершины до нее.

Поиск в ширину:

0: q = (0),

1: 0, q = (1,2),

2: 1, q = (2,3,4),

3: 2, q = (3, 4),

4: 3, q = (4, 5, 6),

5: 4, q = (5,6)

6: 5, q = (6)

7: 6, q = (empty)

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

**Алгоритм обхода в глубину (DFS)**

Алгоритм обхода графа в глубину (Depth-First Search, DFS) — это алгоритм, который используется для обхода всех вершин графа. Он начинает с одной вершины и идет вглубь графа, пока не достигнет конца пути. Затем он возвращается на предыдущий уровень и продолжает обход до тех пор, пока не пройдет все вершины.

Поиск в глубину:

0: s = (0), d = (-),

1: 0, s = (1,2,0), d = (-),

2: 1, s = (3,4,1,2,0), d = (-),

3: 3, s = (5,6,3,4,1,2,0), d = (-),

4: 5 s = (6,3,4,1,2,0), d = (5),

5: 6, s = (3,4,1,2,0) d = (5,6),

6: 3, s = (4,1,2,0) d = (5,6,3),

7: 4, s = (1,2,0) d = (5,6,3,4),

8: 1, s = (2,0) d = (5,6,3,4,1),

9: 2, s = (0) d = (5,6,3,4,1,2),

10: 0, s = (0) d = (5,6,3,4,1,2,0),

11: 0, s = (empty) d = (5,6,3,4,1,2,0),

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

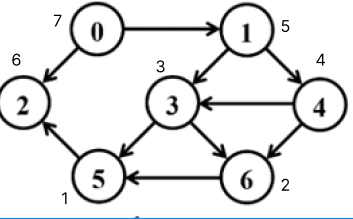
d – массив с топологически отсортированными вершинами (справа налево).

**Алгоритм топологической сортировки.**

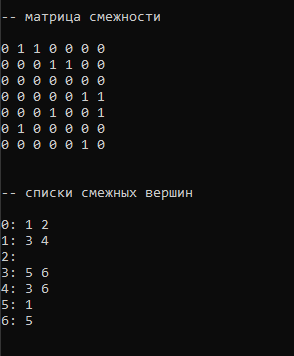
Топологическая сортировка (Topological sort) — один из основных алгоритмов на графах, который применяется для решения множества более сложных задач.

Задача топологической сортировки графа состоит в следующем: указать такой линейный порядок на его вершинах, чтобы любое ребро вело от вершины с меньшим номером к вершине с большим номером. Очевидно, что если в графе есть циклы, то такого порядка не существует.

Топологическая сортировка сводится к обходу графа в глубину, так что   
Отсортированный граф будет выглядеть так:

****

***Задание 3***

******

#### Результат выполнения программы

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

#### Рисунок 3.1 – 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  }; |

#### Рисунок 3.2 – BFS.h

|  |
| --- |
| #pragma once  #include <list>  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 };}; |

#### Рисунок 3.3 – Graph.h

|  |
| --- |
| #include <iostream>  #include "Graph.h"  #include "BFS.h"  #include "DFS.h"  int main()  {  int m[7][7] = {  {0, 1, 1, 0, 0,0,0},  {0, 0, 0, 1, 1,0,0},  {0, 0, 0, 0, 0,0,0},  {0, 0, 0, 0, 0,1,1},  {0, 0, 0, 1, 0,0,1},  {0, 1, 0, 0, 0,0,0},  {0, 0, 0, 0, 0,1,0}  };  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.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(7, (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[7][7] = {  {0, 1, 1, 0, 0,0,0},  {0, 0, 0, 1, 1,0,0},  {0, 0, 0, 0, 0,0,0},  {0, 0, 0, 0, 0,1,1},  {0, 0, 0, 1, 0,0,1},  {0, 1, 0, 0, 0,0,0},  {0, 0, 0, 0, 0,1,0}  };  graph::AList g5(7, (int\*)m3);  DFS b3(g5);  std::cout << std::endl;  std::cout << std::endl << "-- поиск в глубину (ориентированный граф) "  << std::endl;  for (int i = 0; i < g5.n\_vertex; i++) std::cout << b3.get(i) << " ";  std::cout << std::endl;  std::cout << std::endl << "Топологическая сортировка" << std::endl;  for (std::vector <int>::iterator i(b3.topological\_sort.begin()); i != b3.topological\_sort.end(); ++i) std::cout << \*i << ' ';  std::cout << std::endl;  system("pause");  return 0;  } |

#### Рисунок 3.4 – main.cpp

***Задание 4-5.***

#### 

#### Рисунок 4 – Результат выполнения

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

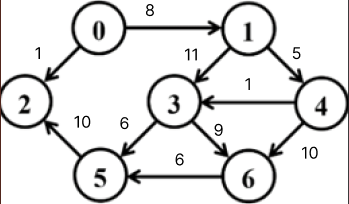
#### Рисунок 4.1 – DFS.cpp

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

#### Рисунок 4.2 – DFS.h

***Задание 6***

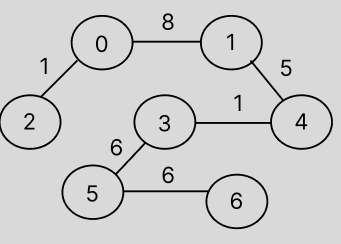
Составим минимальное дерево по алгоритму Прима. Для начала расставим вес ребра из данных по условию значения. После расставления веса рёбер мы начинаем из точки «0» и выбираем максимально оптимальные маршруты с минимальным весом. Суть алгоритма – пройти все узлы по одному разу по рёбрам с минимальным весом.



Начальная вершина 0:

1. Кратчайший путь от 0 к другой вершине - 1, это вершина 2
2. Кратчайший путь из вершин 0, 2: это ребро с весом 8 от 0 к другой вершине 1
3. Кратчайший путь из имеющихся в дереве вершин: от 1 к вершине 4
4. Кратчайший путь из имеющихся в дереве вершин: от 4 к вершине 3
5. Кратчайший путь из имеющихся в дереве вершин: от 3 к вершине 5
6. Кратчайший путь из имеющихся в дереве вершин: от 5 к вершине 6

Остовое дерево имеет вид:



Вес минимального остового дерева равен 27

***Задание 7.***

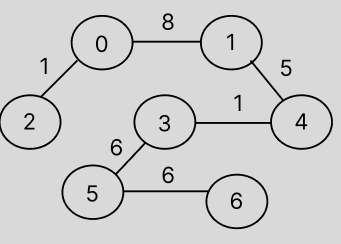
В начале текущее множество рёбер устанавливается пустым. Затем, пока это возможно, проводится следующая операция: из всех рёбер, добавление которых к уже имеющемуся множеству не вызовет появление в нём цикла, выбирается ребро минимального веса и добавляется к уже имеющемуся множеству. Когда таких рёбер больше нет, алгоритм завершён. Подграф данного графа, содержащий все его вершины и найденное множество рёбер, является его остовым деревом минимального веса.

Начальное ребро {0,2} = 1

1. {0,2} => ({3,4} = 1),
2. {3,4} => ({1,4} = 5),
3. {1,4} => ({3,5} = 6),
4. {3,5} => ({5,6} = 6),
5. {5,6} => ({0,1} = 8),

Вес минимального остового дерева равен 27

Остовое дерево имеет вид:



Вес минимального остового дерева получился равным и в алгоритме Прима, и в алгоритме Крускала

**Вывод:** в ходе лабораторной работы я освоил сущность и программную реализацию: а) способов представления графов; б) алгоритмов поиска в ширину и глубину; в) алгоритма топологической сортировки графов; разобрал алгоритм Прима и алгоритм Крускала для нахождения минимального остовного дерева.