Белорусский государственный технологический университет

Факультет информационных технологий

Кафедра программной инженерии

 Лабораторная работа № 6

По дисциплине «Математическое программирование»

На тему «Алгоритмы на графах»

Выполнила:

Студентка 2 курса 10 группы

Рублевская Маргарита Владимировна

Преподаватель: асс. Ромыш А.С.

2025, Минск

**Вариант № 10**

**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.

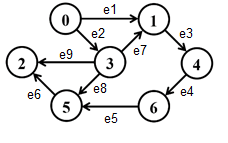
**Способы представления графов**

Исходный граф:



Матрица смежности:

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



Матрица инцидентности:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | e1 | e2 | e3 | e4 | e5 | e6 | e7 | e8 | e9 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | -1 | 0 | 1 | 0 | 0 | 0 | -1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | -1 | 0 | 0 | -1 |
| 3 | 0 | -1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 4 | 0 | 0 | -1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | -1 | 1 | 0 | -1 | 0 |
| 6 | 0 | 0 | 0 | -1 | 1 | 0 | 0 | 0 | 0 |

Список смежных вершин:

0: 1 3

1: 4

2: -

3: 1 2 5

4: 6

5: 2

6: 5

**Алгоритм поиска в ширину (BFS)**

Исходный граф:



Шаг 1

Текущая вершина: 0

Состояние очереди: q = {1, 3}.

Шаг 2

Текущая вершина: 1.

Состояние очереди: q = {3, 4}.

Шаг 3

Текущая вершина: 3.

Состояние очереди: q = {4, 2, 5}.

Шаг 4

Текущая вершина: 4.

Состояние очереди: q = {2, 5, 6}.

Шаг 5

Текущая вершина: 2.

Состояние очереди: q = {5, 6}.

Шаг 6

Текущая вершина: 5.

Состояние очереди: q = {6}.

Шаг 7

Текущая вершина: 6.

Состояние очереди: q = {}.

Итоговая последовательность вершин: 0-1-3-4-2-5-6

**Алгоритм поиска в глубину (DFS) и топологическая сортировка**

Исходный граф:



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.

Шаг 1

Текущая вершина: 0

Состояние стека: q = {1, 3}.

Шаг 2

Текущая вершина: 1

Состояние стека: q = {4, 3}.

Шаг 3

Текущая вершина: 4

Состояние стека: q = {6, 3}.

Шаг 4

Текущая вершина: 6

Состояние стека: q = {5, 3}.

Шаг 5

Текущая вершина: 5.

Состояние стека: q = {2, 3}.

Шаг 6

Текущая вершина: 2.

Состояние стека: q = {3}.

Шаг 7

Текущая вершина: 3.

Состояние стека: q = {}.

Итоговая последовательность вершин: 0-1-4-6-5-2-3.

Топологическая сортировка – это процедура упорядочивания вершин ориентированного графа, не имеющего циклов.

При реализации топологической сортировки с помощью алгоритма поиска в глубину используется массив меток вершин, с помощью которого моделируется удаление вершин из графа и сохраняются новые номера вершин.

Шаг 1

Текущая вершина: 0.

Состояние стека: s = {1, 3, 0}.

Массив топологически отсортированных вершин: d = {}.

Шаг 2

Текущая вершина: 1.

Состояние стека: s = {4, 1, 3, 0}.

Массив топологически отсортированных вершин: d = {}.

Шаг 3

Текущая вершина: 4.

Состояние стека: s = {6, 4, 1, 3, 0}.

Массив топологически отсортированных вершин: d = {}.

Шаг 4

Текущая вершина: 6.

Состояние стека: s = {5, 6, 4, 1, 3, 0}.

Массив топологически отсортированных вершин: d = {}.

Шаг 5

Текущая вершина: 5.

Состояние стека: s = {2, 5, 6, 4, 1, 3, 0}.

Массив топологически отсортированных вершин: d = {}.

Шаг 6

Текущая вершина: 2.

Состояние стека: s = {5, 6, 4, 1, 3, 0}.

Массив топологически отсортированных вершин: d = {2}.

Шаг 7

Текущая вершина: 5.

Состояние стека: s = {6, 4, 1, 3, 0}.

Массив топологически отсортированных вершин: d = {2, 5}.

Шаг 8

Текущая вершина: 6.

Состояние стека: s = {4, 1, 3, 0}.

Массив топологически отсортированных вершин: d = {2, 5, 6}.

Шаг 9

Текущая вершина: 4.

Состояние стека: s = {1, 3, 0}.

Массив топологически отсортированных вершин: d = {2, 5, 6, 4}.

Шаг 10

Текущая вершина: 1.

Состояние стека: s = {3, 0}.

Массив топологически отсортированных вершин: d = {2, 5, 6, 4, 1}.

Шаг 11

Текущая вершина: 3.

Состояние стека: s = {0}.

Массив топологически отсортированных вершин: d = {2, 5, 6, 4, 1, 3}.

Шаг 12

Текущая вершина: 0.

Состояние стека: s = {}.

Массив топологически отсортированных вершин: d = {2, 5, 6, 4, 1, 3, 0}.

Итоговая последовательность вершин: 2-5-6-4-1-3-0

**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.

Код файла Graph.h представлен в листинге 1.1.

|  |
| --- |
| #pragma once  #include <list>  #include "Graph.h"  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  };  } |

Листинг 1.1 – Код файла Graph.h

Реализация методов структур Alist и AMatrix представлена в листинге 1.2.

|  |
| --- |
| #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 1.2 – Реализация методов структур AList и AMatrix

Код файла BFS.h представлен в листинге 1.3.

|  |
| --- |
| #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 1.3 – Код файла BFS.h

Реализация методов структуры BFS представлена в листинге 1.4.

|  |
| --- |
| #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 1.4 – Реализация методов структуры BFS

Результат работы функций представлен на рисунке 1.1.

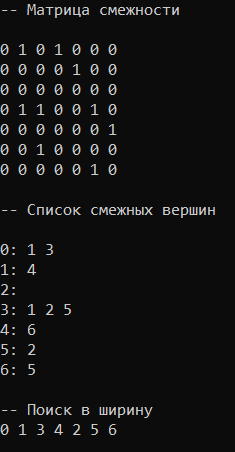


Рисунок 1.1 – Результат работы функций

**Задание №4-5**

Разработать функцию **DFS** обхода вершин графа, используя метод поиска глубину. Продемонстрировать работу функции.

Доработайте функцию **DFS**,для выполнения топологической сортировки графа.

Код файла DFS.h представлен в листинге 1.5.

|  |
| --- |
| #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 1.5 – Код файла DFS.h

Реализация методов структуры DFS представлена в листинге 1.6.

|  |
| --- |
| #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 1.6 – Реализация методов структуры DFS

Результат работы функций представлен на рисунке 1.2.

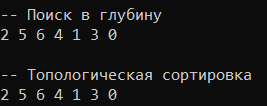
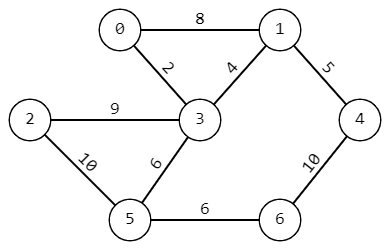


Рисунок 1.2 – Результат работы функций

**Задание №6**

По графу, соответствующему варианту составить минимальное остовное дерево по алгоритму Прима.

Исходный граф:



Шаг 1

Текущие вершины дерева: 0

Выбираем ребро 0-3

Шаг 2

Текущие вершины дерева: 0, 3

Выбираем ребро 3-1

Шаг 3

Текущие вершины дерева: 0, 3, 1

Выбираем ребро 1-4

Шаг 4

Текущие вершины дерева: 0, 3, 1, 4

Выбираем ребро 3-5

Шаг 5

Текущие вершины дерева: 0, 3, 1, 4, 5

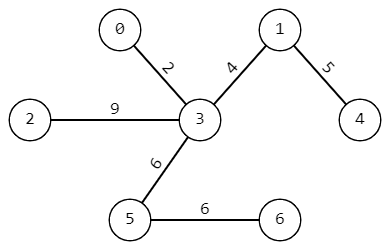
Выбираем ребро 3-2

Шаг 6

Текущие вершины дерева: 0, 3, 1, 4, 5, 2

Выбираем ребро 5-6

Полученное минимальное остовное дерево:



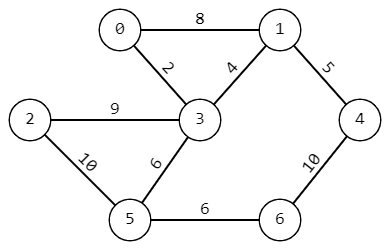
Вес дерева: 32.

**Задание №7**

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

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.

Исходный граф:



Отсортируем все ребра по возрастанию веса:

|  |  |
| --- | --- |
| Ребро | Вес |
| 0-3 | 2 |
| 1-3 | 4 |
| 1-4 | 5 |
| 3-5 | 6 |
| 5-6 | 6 |
| 0-1 | 8 |
| 2-3 | 9 |
| 2-5 | 10 |
| 4-6 | 10 |

Начальное ребро – это ребро с минимальным весом, а именно 0-3.

Шаг 1

Множество ребер остовного дерева: {(0, 3)}.

Шаг 2

Множество ребер остовного дерева: {(0, 3), (1, 3)}.

Шаг 3

Множество ребер остовного дерева: {(0, 3), (1, 3), (1, 4)}.

Шаг 4

Множество ребер остовного дерева: {(0, 3), (1, 3), (1, 4), (3, 5)}.

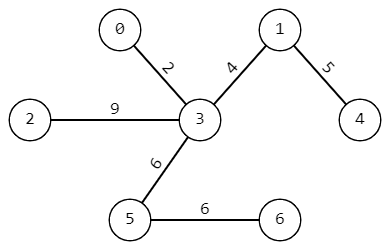
Шаг 5

Множество ребер остовного дерева: {(0, 3), (1, 3), (1, 4), (3, 5), (5, 6)}.

Шаг 6

Множество ребер остовного дерева: {(0, 3), (1, 3), (1, 4), (3, 5), (5, 6), (2, 3)}.

Полученное минимальное остовное дерево:



Вес дерева: 32.

**Вывод:** в ходе лабораторной работы были освоены сущность и программная реализация: а) способов представления графов; б) алгоритмов поиска в ширину и глубину; в) алгоритма топологической сортировки графов и разобрала алгоритм Прима и алгоритм Крускала.