Министерство образования Республики Беларусь

Учреждение образования

Брестский государственный технический университет

Кафедра ИИТ

Лабораторная работа №3

За 3 семестр

По дисциплине «Дискретная математика»

Тема: «Графы. Деревья.»

Выполнил: студент 2 курса

Группы ПО-4(2)

Коташевич С.Н.

Проверил: Глущенко Т.А.

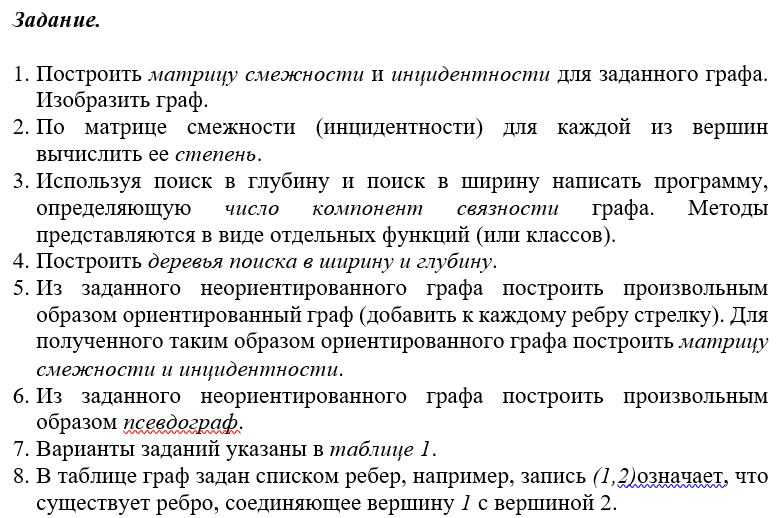
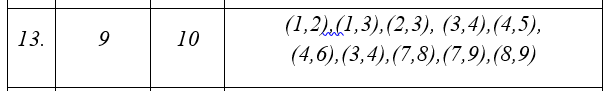
Брест 2020

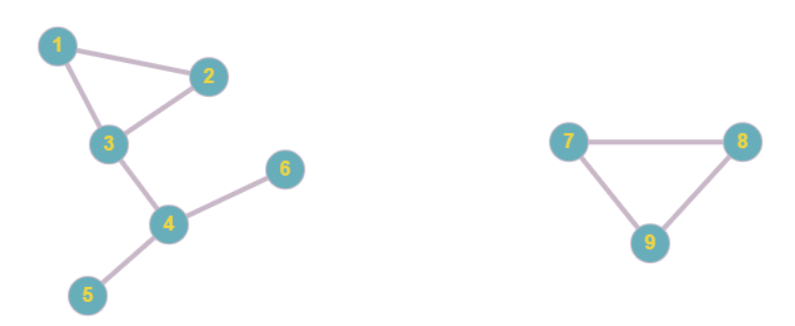
**Лабораторная работа №3**

**Графы. Деревья.**

**Графы**

**Вариант 2**



Матрица смежности:

Степени вершин: 1 – 4, 2 – 2, 3 – 2, 4 – 3, 5 – 3, 6 – 2.

**1 2 3 4 5 6 7 8 9**

**1** 0 1 1 0 0 0 0 0 0

**2** 1 0 1 0 0 0 0 0 0

**3** 1 1 0 1 0 0 0 0 0

**4** 0 0 1 0 1 1 0 0 0

**5** 0 0 0 1 0 0 0 0 0

**6** 0 0 0 1 0 0 0 0 0

**7** 0 0 0 0 0 0 0 1 1

**8** 0 0 0 0 0 0 1 0 1

**9** 0 0 0 0 0 0 1 1 0

Матрица инцидентности:

**1-2 1-3 2-3 3-4 4-5 4-6 7-8 7-9 8-9**

**1** 1 1 0 0 0 0 0 0 0

**2** 1 0 1 0 0 0 0 0 0

**3** 0 1 1 1 0 0 0 0 0

**4** 0 0 0 1 1 1 0 0 0

**5** 0 0 0 0 1 0 0 0 0

**6** 0 0 0 0 0 1 0 0 0

**7** 0 0 0 0 0 0 1 1 0

**8** 0 0 0 0 0 0 1 0 1

**9** 0 0 0 0 0 0 0 1 1

Поиск в глубину:

#include <iostream>

#include <list>

using namespace std;

class Graph

{

// количество вершин

int V;

list<int>\* adjList;

void util(int v, bool visited[]);

public:

Graph(int v);

void add\_edge(int v, int w);

void DFS();

};

Graph::Graph(int v)

{

V = v;

adjList = new list<int>[V];

}

void Graph::add\_edge(int v, int w)

{

v--; w--;

adjList[v].push\_back(w);

}

void Graph::util(int v, bool visited[])

{

list<int>::iterator i;

visited[v] = true; // текущий узел помечаем как посещенный

cout << v + 1 << " ";

// рекурсивно обрабатываем все соседние вершины узла

for (i = adjList[v].begin(); i != adjList[v].end(); ++i) {

if (!visited[\*i]) {

util(\*i, visited);

}

}

}

void Graph::DFS()

{

bool\* visited = new bool[V];

for (int i = 0; i < V; i++) {

visited[i] = false; // все вершины не посещенные

}

int count = 0;

cout << "Обход в глубину графа:" << "\n";

// исследуем вершины одну за другой

for (int i = 0; i < V; i++) {

if (!visited[i]) {

util(i, visited);

count++;

}

}

cout << "\nЧисло компонент связности: " << count;

cout << "\n";

}

int main()

{

setlocale(LC\_ALL, "rus");

Graph dg(9);

dg.add\_edge(1, 2);

dg.add\_edge(1, 3);

dg.add\_edge(2, 3);

dg.add\_edge(3, 4);

dg.add\_edge(4, 5);

dg.add\_edge(4, 6);

dg.add\_edge(3, 4);

dg.add\_edge(7, 8);

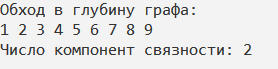
dg.add\_edge(7, 9);

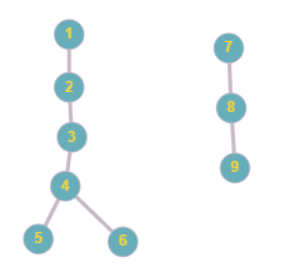
dg.add\_edge(8, 9);

dg.DFS();

return 0;

}





Поиск в ширину:

#include <iostream>

#include <list>

using namespace std;

class Graph

{

// количество вершин

int V;

list<int>\* adjList;

void util(int v, bool visited[]);

public:

Graph(int v);

void add\_edge(int v, int w);

void BFS();

};

Graph::Graph(int v)

{

V = v;

adjList = new list<int>[V];

}

void Graph::add\_edge(int v, int w)

{

v--; w--;

adjList[v].push\_back(w);

}

void Graph::util(int v, bool visited[])

{

list<int> queue;

list<int>::iterator i;

visited[v] = true; // текущий узел помечаем как посещенный

queue.push\_back(v);

while (!queue.empty()) {

v = queue.front();

cout << v + 1 << " ";

queue.pop\_front();

// исследуем вершины одну за другой

for (i = adjList[v].begin(); i != adjList[v].end(); ++i) {

if (!visited[\*i]) {

visited[\*i] = true;

queue.push\_back(\*i);

}

}

}

cout << "\n";

}

void Graph::BFS()

{

cout << "Обход графа в ширину: " << "\n";

int count = 0;

bool\* visited = new bool[V];

for (int i = 0; i < V; i++) {

visited[i] = false; // все вершины не посещенные

}

for (int i = 0; i < V; i++) {

if (!visited[i]) {

util(i, visited);

count++;

}

}

cout << "\nЧисло компонент связности: " << count;

}

int main()

{

setlocale(LC\_ALL, "rus");

Graph dg(9);

dg.add\_edge(1, 2);

dg.add\_edge(1, 3);

dg.add\_edge(2, 3);

dg.add\_edge(3, 4);

dg.add\_edge(4, 5);

dg.add\_edge(4, 6);

dg.add\_edge(3, 4);

dg.add\_edge(7, 8);

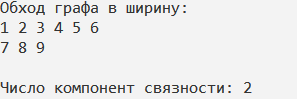
dg.add\_edge(7, 9);

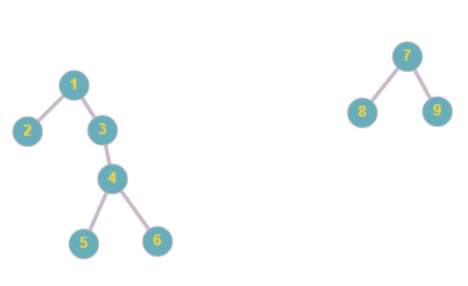
dg.add\_edge(8, 9);

dg.BFS();

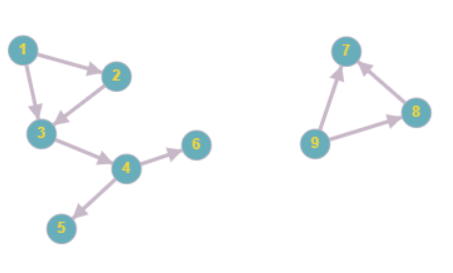
return 0;

}





Ориентированный граф:



Матрица смежности:

**1 2 3 4 5 6 7 8 9**

**1** 0 1 1 0 0 0 0 0 0

**2** 0 0 1 0 0 0 0 0 0

**3** 0 0 0 1 0 0 0 0 0

**4** 0 0 0 0 1 1 0 0 0

**5** 0 0 0 0 0 0 0 0 0

**6** 0 0 0 0 0 0 0 0 0

**7** 0 0 0 0 0 0 0 0 0

**8** 0 0 0 0 0 0 1 0 0

**9** 0 0 0 0 0 0 1 1 0

Матрица инцидентности:

**1-2 1-3 2-3 3-4 4-5 4-6 7-8 7-9 8-9**

**1** 1 1 0 0 0 0 0 0 0

**2 -**1 0 1 0 0 0 0 0 0

**3** 0 -1 -1 1 0 0 0 0 0

**4** 0 0 0 -1 1 1 0 0 0

**5** 0 0 0 0 -1 0 0 0 0

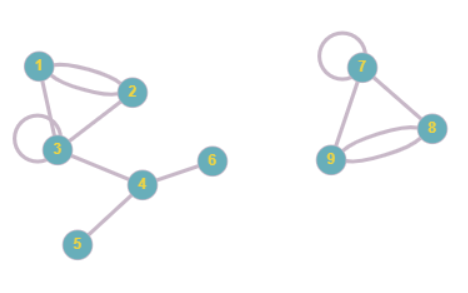
**6** 0 0 0 0 0 -1 0 0 0

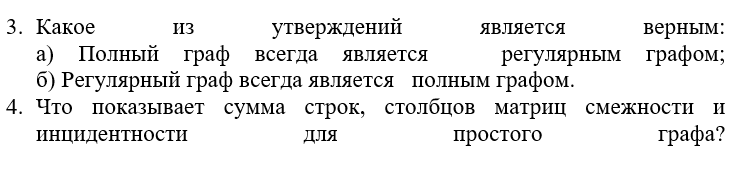
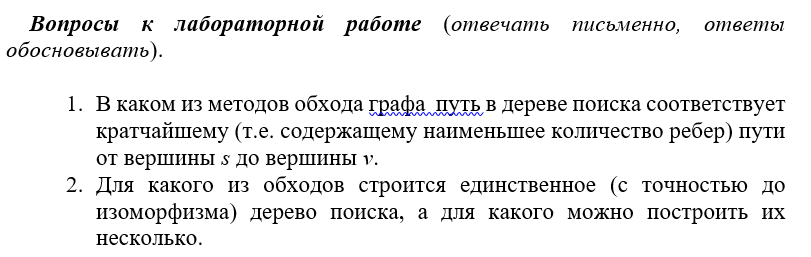
**7** 0 0 0 0 0 0 -1 -1 0

**8** 0 0 0 0 0 0 1 0 -1

**9** 0 0 0 0 0 0 0 1 1

Псевдограф:





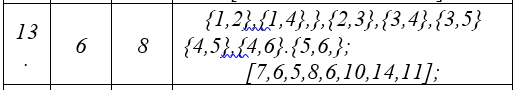
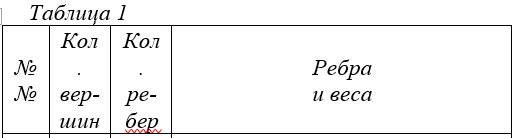
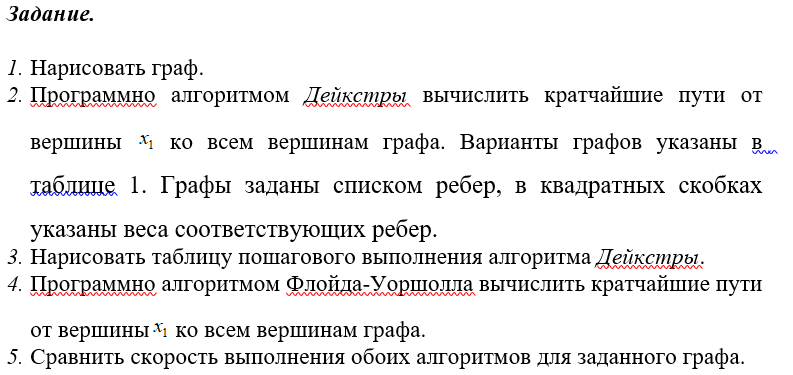
1. В методе обхода в ширину в дереве поиска показывается кратчайший путь от вершины s до вершины и все вершины, связанные с выбранной, помечаются как доступные, что отображается на дереве поиска. При методе обхода в глубину выбирается первая доступная вершина для выбранной и делается обход, поэтому количество ребер, необходимых для доступа в вершину, может увеличиться.
2. Для обхода в ширину можно построить единственное дерево поиска, а для поиска в глубину – несколько, т.к. обход может выбираться по-разному.
3. а)

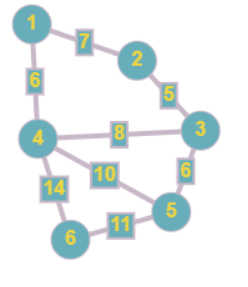
Полный граф — простой неориентированный граф, в котором каждая пара различных вершин смежна

1. Степень вершины графа

**Дейкстры**

**Вариант 13**





Алгоритм Дейкстры:

#include <iostream>

#include <fstream>

using namespace std;

void Read(int\*\* GR, int N)

{

ifstream Ft;

Ft.open("matrix.txt");

for (int i = 0; i < N; i++)

for (int j = 0; j < N; j++)

Ft >> GR[i][j];

Ft.close();

}

void Dijkstra(int\*\* GR, int st, int N)

{

int index, u, m = st + 1;

int\* distance = new int[N];

bool\* visited = new bool[N];

for (int i = 0; i < N; i++)

{

distance[i] = INT\_MAX;

visited[i] = false;

}

distance[st] = 0;

for (int count = 0; count < N - 1; count++)

{

int min = INT\_MAX;

for (int i = 0; i < N; i++)

if (!visited[i] && distance[i] <= min)

{

min = distance[i];

index = i;

}

u = index;

visited[u] = true;

for (int i = 0; i < N; i++)

if (!visited[i] && GR[u][i] && distance[u] != INT\_MAX &&

distance[u] + GR[u][i] < distance[i])

distance[i] = distance[u] + GR[u][i];

}

cout << "Стоимость пути из начальной вершины до остальных:\n";

for (int i = 0; i < N; i++)

if (distance[i] != INT\_MAX)

cout << m << " -> " << i + 1 << " = " << distance[i] << endl;

else

cout << m << " -> " << i + 1 << " - " << "маршрут недоступен" << endl;

delete[] distance;

delete[] visited;

}

int main()

{

setlocale(LC\_ALL, "Rus");

int N, start;

cout << "Введите размер матрицы: "; cin >> N;

int\*\* GR = new int\* [N];

for (int i = 0; i < N; i++)

GR[i] = new int[N];

Read(GR, N);

cout << "Матрица смежности:\n";

for (int i = 0; i < N; i++)

{

for (int j = 0; j < N; j++)

cout << GR[i][j] << " ";

cout << endl;

}

cout << "Начальная вершина >> "; cin >> start;

Dijkstra(GR, start - 1, N);

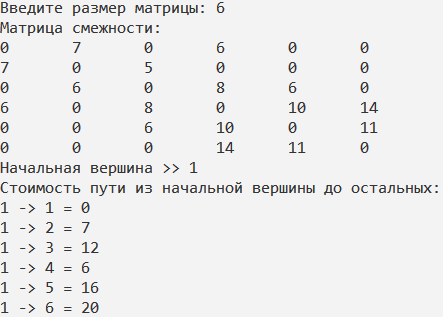
for (int i = 0; i < N; i++)

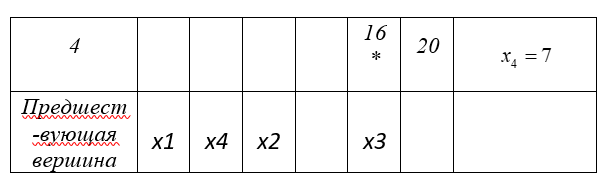
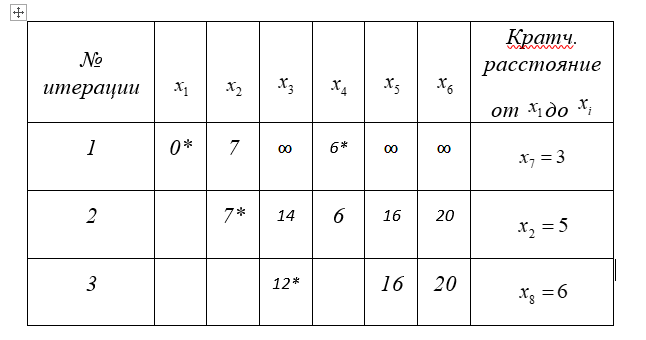
delete[] GR[i];

delete[] GR;

return 0;

}





Алгоритм Флойда-Уоршолла:

#include <iostream>

#include <iomanip>

#include <vector>

using namespace std;

class Graph

{

private:

// количество вершин

int V;

vector<vector<int>> G; // граф

public:

Graph(int v);

void add\_edge(int u, int v, int w);

void floyd();

void display();

};

Graph::Graph(int v)

{

V = v;

vector<vector<int>> g(V, vector<int>(V));

for (int i = 0; i < V; i++) {

for (int j = 0; j < V; j++) {

g[i][j] = (i == j) ? 0 : 1000;

}

}

G = g;

}

void Graph::add\_edge(int u, int v, int w)

{

u--, v--;

G[u][v] = G[v][u] = w;

}

void Graph::floyd()

{

for (int k = 0; k < V; k++) {

for (int i = 0; i < V; i++) {

for (int j = 0; j < V; j++) {

G[i][j] = min(G[i][j], G[i][k] + G[k][j]);

}

}

}

}

void Graph::display()

{

for (int i = 0; i < V; i++) {

for (int j = 0; j < V; j++) {

cout << setw(4) << G[i][j] << " ";

}

cout << "\n";

}

}

int main()

{

Graph gf(6);

gf.add\_edge(1, 2, 7);

gf.add\_edge(2, 3, 5);

gf.add\_edge(4, 3, 8);

gf.add\_edge(1, 4, 6);

gf.add\_edge(3, 5, 6);

gf.add\_edge(4, 5, 10);

gf.add\_edge(4, 6, 14);

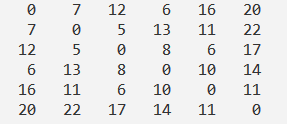
gf.add\_edge(5, 6, 11);

gf.floyd();

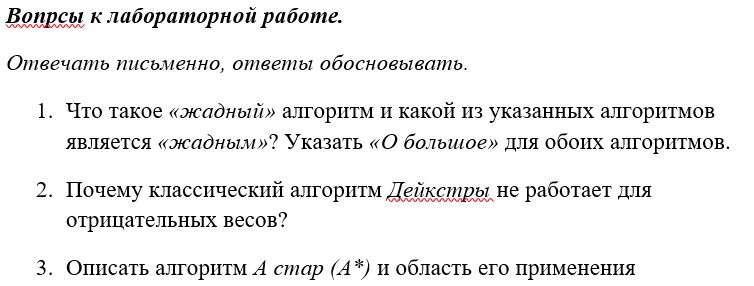
gf.display();

return 0;

}



Алгоритм Флойда — Уоршелла является эффективным для расчёта всех кратчайших путей в плотных графах, когда имеет место большое количество пар рёбер между парами вершин. В случае разреженных графов с рёбрами неотрицательного веса лучшим выбором считается использование алгоритма Дейкстры. При таком выборе сложность составляет O(|V|\*|E| log |V|) применении двоичной кучи, что лучше, чем O(|V|^{3}) алгоритма Флойда — Уоршелла тогда, когда |E| существенно меньше |V|^{2} (условие разреженности графа).



1. Алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным

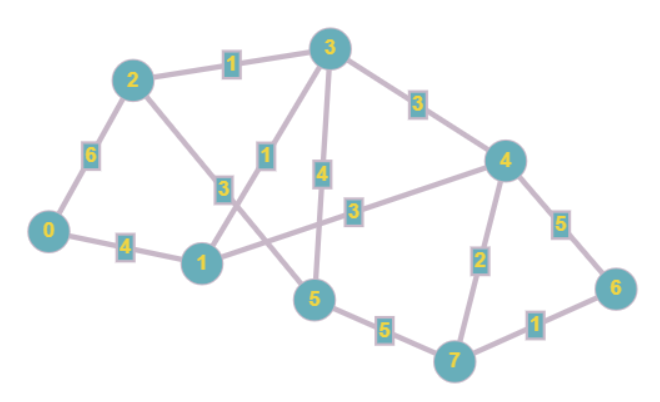
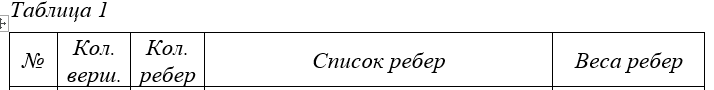
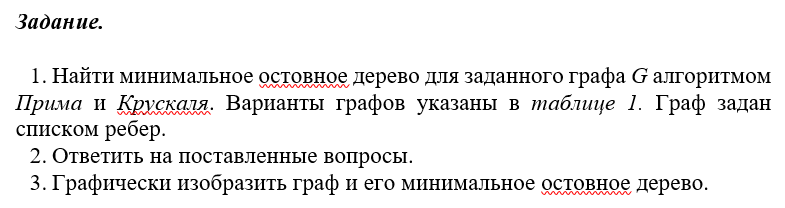
Дейкстра О(n^2)), Флойда-Уоршолла(O(n^3/k) k - длина битовой маски

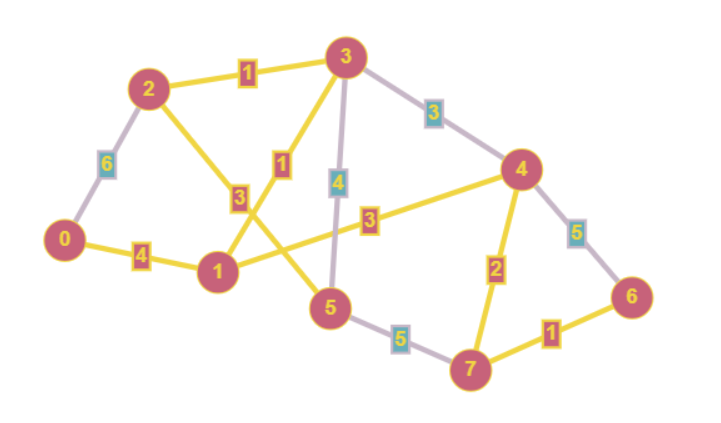
1. Алгоритм Дейкстры является в некотором роде "жадным" - найдя один раз минимальный путь до вершины, он фиксирует его как минимальный навсегда - поскольку путь через другие вершины не может быть короче найденного. Но в графе с отрицательными дугами это не так!
2. A\* пошагово просматривает все пути, ведущие от начальной вершины в конечную, пока не найдёт минимальный. Как и все информированные алгоритмы поиска, он просматривает сначала те маршруты, которые «кажутся» ведущими к цели. От жадного алгоритма, который тоже является алгоритмом поиска по первому лучшему совпадению, его отличает то, что при выборе вершины он учитывает, помимо прочего, весь пройденный до неё путь. Составляющая g(x) — это стоимость пути от начальной вершины, а не от предыдущей, как в жадном алгоритме.

В начале работы просматриваются узлы, смежные с начальным; выбирается тот из них, который имеет минимальное значение f(x), после чего этот узел раскрывается. На каждом этапе алгоритм оперирует с множеством путей из начальной точки до всех ещё не раскрытых (листовых) вершин графа — множеством частных решений, — которое размещается в очереди с приоритетом. Приоритет пути определяется по значению f(x) = g(x) + h(x). Алгоритм продолжает свою работу до тех пор, пока значение f(x) целевой вершины не окажется меньшим, чем любое значение в очереди, либо пока всё дерево не будет просмотрено. Из множества решений выбирается решение с наименьшей стоимостью.

**Остовные деревья**

**Вариант 2**





Алгоритм Прима:

#include <fstream>

#include<iostream>

#include <iomanip>

using namespace std;

const int MAX = 101;

int main()

{

setlocale(0, "");

system("color F0");

int n = 8; // размер матрицы смежностей

ifstream fin("matrix1.txt");

int\*\* graph = new int\* [n];

for (int i = 0; i < n; i++) { //матрица смежности

graph[i] = new int[n];

}

bool\* visited = new bool[n]; // массив, хранящий вершины уже добавленные в минимальное остовное дерево

for (int i = 0; i < n; i++) { //считываем матрицу с файла

for (int j = 0; j < n; j++) {

fin >> graph[i][j];

if (graph[i][j] == 0)

graph[i][j] = MAX; // нули меняем на 'бесконечность'

cout << setw(4) << graph[i][j];

}

cout << endl;

visited[i] = false; //инициализация - вершин добавленых в дерево нет

}

fin.close();

int v1, v2; // v1-вершина уже добавленая, v2 - будет добавлена

int min;

visited[0] = true; // добавляем в дерево первую вершину

for (int tek = 1; tek < n; tek++) { // повторяем н - 1 раз(н - 1 : оставшееся количество вершин)

min = MAX; // мин = \*бесконечность\*

for (int i = 0; i < n; i++) { // тут ищем потенциально возможные новые рёбра для дерева, для минимального ребра сохраняем вершины при нем

if (visited[i] == true) {

for (int d = 0; d < n; d++) {

if (min > graph[i][d] && !visited[d]) {

v1 = i; //уже добавили

min = graph[i][d];

v2 = d; //будет добавлена

}

}

}

}

cout << endl << v1 << ' ' << v2;

visited[v2] = true;

}

cout << endl;

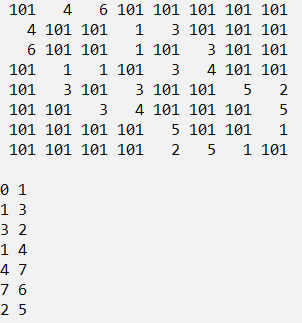
for (int i = 0; i < n; i++)

delete[] graph[i];

system("pause");

return 0;

}



Алгоритм Крускаля:

#define \_CRT\_SECURE\_NO\_WARNINGS

#include<stdio.h>

#include<stdlib.h>

#include <iostream>

using namespace std;

int i, j, k, a, b, u, v, n, ne = 1;

int min, mincost = 0, cost[9][9], parent[9];

int find(int);

int uni(int, int);

int main()

{

setlocale(0, "");

system("color F0");

int min;

printf("Введите количество вершин: ");

scanf("%d", &n);

printf("\nВведите матрицу: \n");

for (i = 1; i <= n; i++) {

for (j = 1; j <= n; j++) {

scanf("%d", &cost[i][j]);

if (cost[i][j] == 0)

cost[i][j] = 999;

}

}

while (ne < n) { //просматривает матрицу и находит в ней наименьшее значение

for (i = 1, min = 999; i <= n; i++) {

for (j = 1; j <= n; j++) {

if (cost[i][j] < min) {

min = cost[i][j];

a = u = i;

b = v = j;

}

}

}

u = find(u);

v = find(v);

if (uni(u, v)) {

printf("%d edge (%d, %d) =%d\n", ne++, a, b, min);

mincost += min;

}

cost[a][b] = 999;

}

printf("\n\tМинимальная сумма = %d\n", mincost);

}

//находит корни деревьев, которым u и v принадлежат

int find(int i)

{

while (parent[i])

i = parent[i];

return i;

}

//Если два корня совпадают, код ничего не делает, ребро не используется

int uni(int i, int j)

{

if (i != j) {

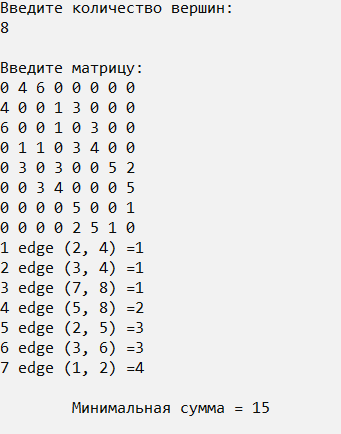
parent[j] = i;

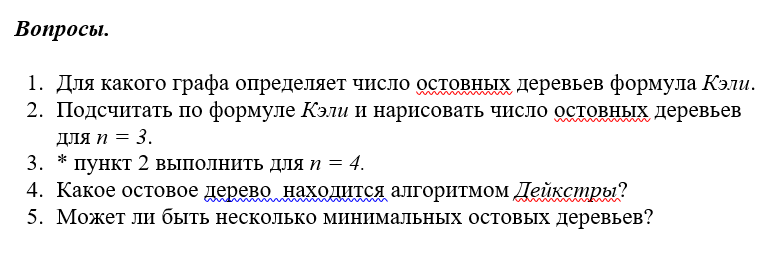
return 1;

}

return 0;

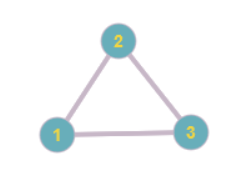
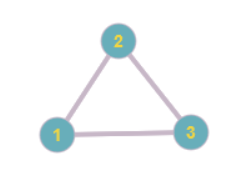
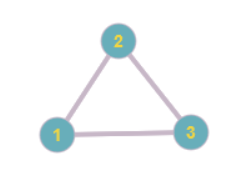
}



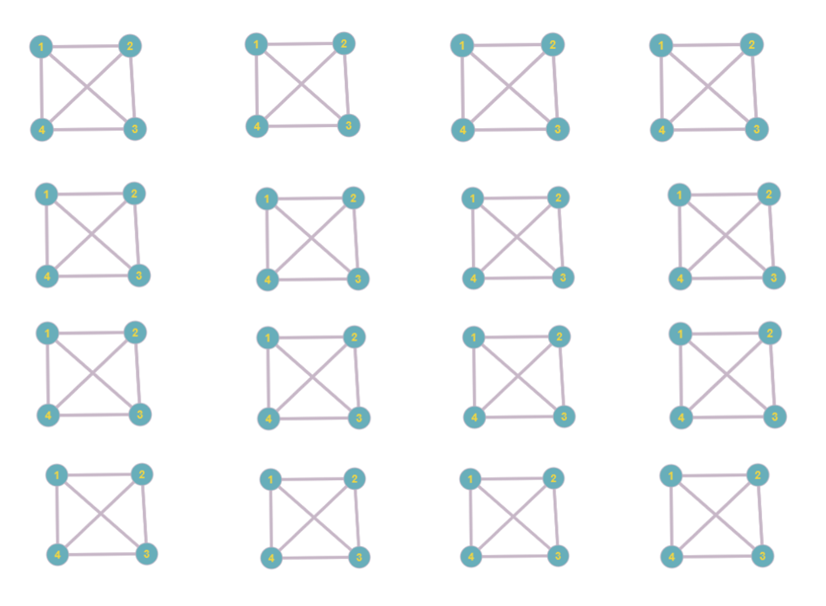


1. Для полных помеченных (взвешенных) графов

2. 3^(3-2) = 3

3. 4^(4-2) = 16



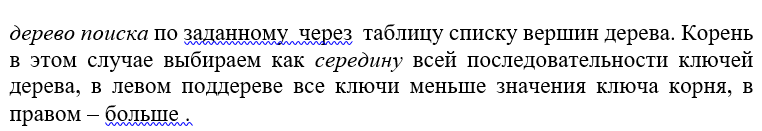
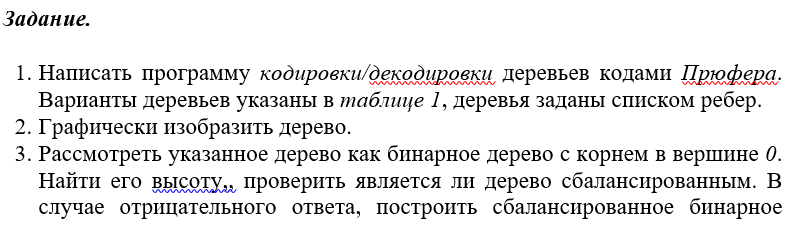
4. Обход вершин которого стоит меньше всего (минимальное остовное дерево).

5. Если каждое ребро имеет отдельный вес, тогда будет только одно уникальное минимальное остовное дерево. В противном случае, может существовать несколько минимальных остовов.

**Кодирование деревьев**

**Вариант 6**





#include<iostream>

#include <vector>

#include <set>

#include <climits>

using namespace std;

class Graph

{

private:

// количество вершин

int V;

int E;

vector<pair<int, int>> G; // граф исходный

vector<pair<int, int>> D; // граф декодированный

vector<int> Degree;

vector<int> Code;

public:

Graph(int v);

void add\_edge(int u, int v);

void prufer\_encode();

void prufer\_decode();

void display();

};

Graph::Graph(int v)

{

V = v;

E = V - 1;

vector<int> d(V, 0);

Degree = d;

Code.clear();

D.clear();

}

void Graph::add\_edge(int u, int v)

{

G.push\_back(make\_pair(u, v));

Degree[u]++;

Degree[v]++;

}

void Graph::display()

{

cout << "Исходное дерево:\n";

for (int i = 0; i < G.size(); i++) {

cout << G[i].first << " " << G[i].second << "\n";

}

cout << "Код Прюфера: ";

for (int i = 0; i < Code.size(); i++) {

cout << Code[i] << " ";

}

cout << "\n";

cout << "Декодированное дерево:\n";

for (int i = 0; i < D.size(); i++) {

cout << D[i].first << " " << D[i].second << "\n";

}

}

void Graph::prufer\_encode()

{

int x;

for (int i = 0; i < V - 2; i++) {

int min = INT\_MAX;

// выбрать вершину с наименьшим индексом и равную 1

for (int j = 0; j < E; j++) {

if (Degree[G[j].first] == 1 && Degree[G[j].second] > 0) {

if (min > G[j].first) {

min = G[j].first;

x = j;

}

}

if (Degree[G[j].second] == 1 && Degree[G[j].first] > 0) {

if (min > G[j].second) {

min = G[j].second;

x = j;

}

}

}

// уменьшить значение вершин на 1

if (Degree[G[x].first] > 0) {

Degree[G[x].first]--;

}

if (Degree[G[x].second] > 0) {

Degree[G[x].second]--;

}

// сохранить вершину, из которой удаляется лист

if (Degree[G[x].first] == 0) {

Code.push\_back(G[x].second);

}

else {

Code.push\_back(G[x].first);

}

}

}

void Graph::prufer\_decode()

{

int n = (int)Code.size() + 2;

vector<int> degree(n, 1);

for (int i = 0; i < n - 2; ++i) {

++degree[Code[i]];

}

set<int> leaves;

for (int i = 0; i < n; ++i) {

if (degree[i] == 1) {

leaves.insert(i);

}

}

for (int i = 0; i < n - 2; ++i) {

int leaf = \*leaves.begin();

leaves.erase(leaves.begin());

int v = Code[i];

D.push\_back(make\_pair(leaf, v));

if (--degree[v] == 1) {

leaves.insert(v);

}

}

D.push\_back(make\_pair(\*leaves.begin(), \*--leaves.end()));

}

int main()

{

setlocale(LC\_ALL, "rus");

Graph gp(13);

gp.add\_edge(0, 1);

gp.add\_edge(0, 2);

gp.add\_edge(1, 3);

gp.add\_edge(1, 4);

gp.add\_edge(2, 5);

gp.add\_edge(2, 6);

gp.add\_edge(4, 7);

gp.add\_edge(7, 8);

gp.add\_edge(7, 9);

gp.add\_edge(9, 10);

gp.add\_edge(9, 11);

gp.add\_edge(11, 12);

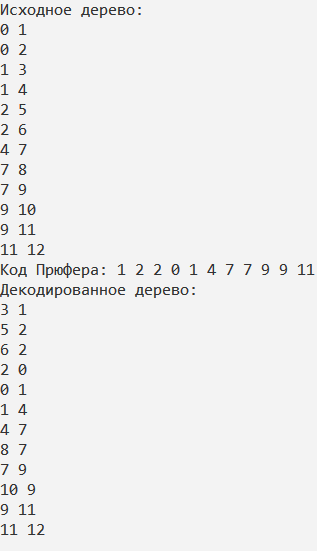
gp.prufer\_encode();

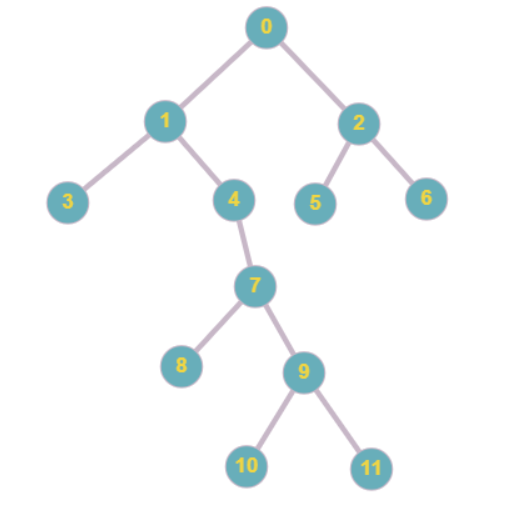
gp.prufer\_decode();

gp.display();

return 0;

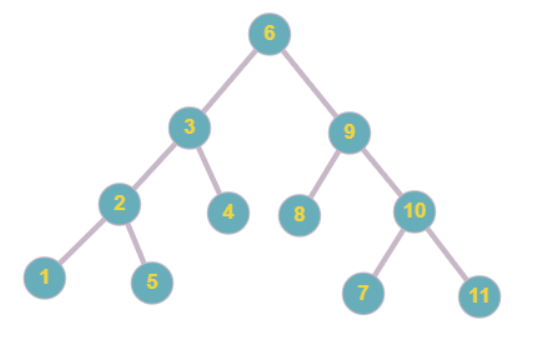
}

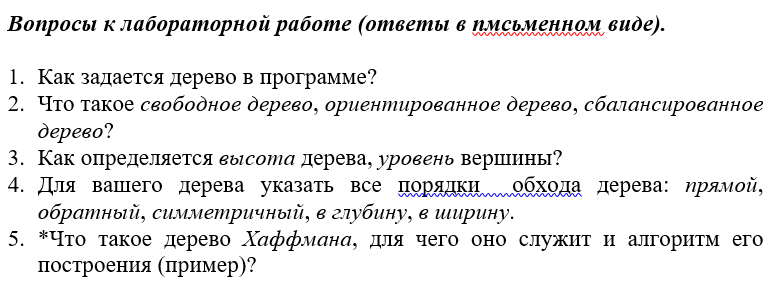




Высота дерева – 5.

Сбалансированное дерево:





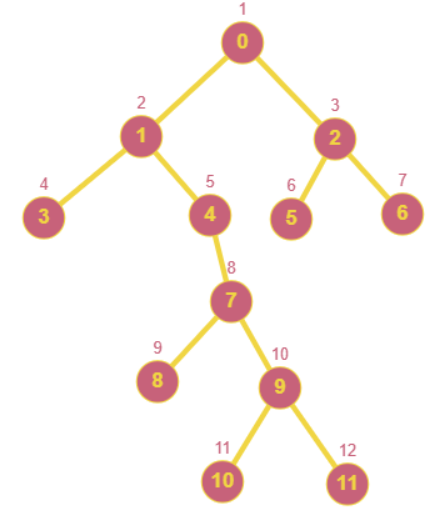
1. Граф задается в виде списка ребер, представленных векторами, где каждый элемент вектора – пара вершин дерева.
2. Свободное дерево – дерево, в котором не выделен корень.

Ориентированное дерево – дерево, ребрам которого присвоено направление.

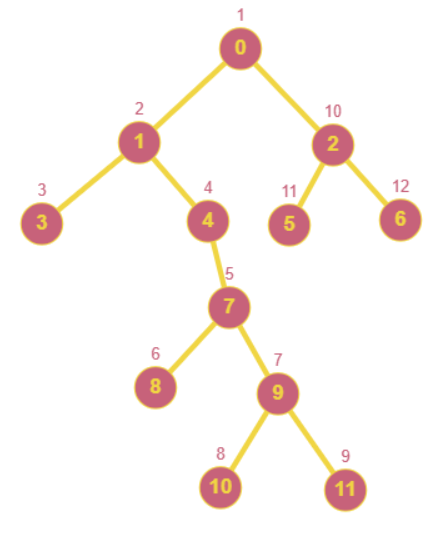
Сбалансированное дерево – дерево, у которого высота левого и правого поддерева каждого узла этого дерева отличается максимум на 1.

1. Высота дерева равна длине самого длинного пути от корня дерева до листа. Уровень вершины дерева - удаленность вершины от корня. Уровень корня дерева равен 0.

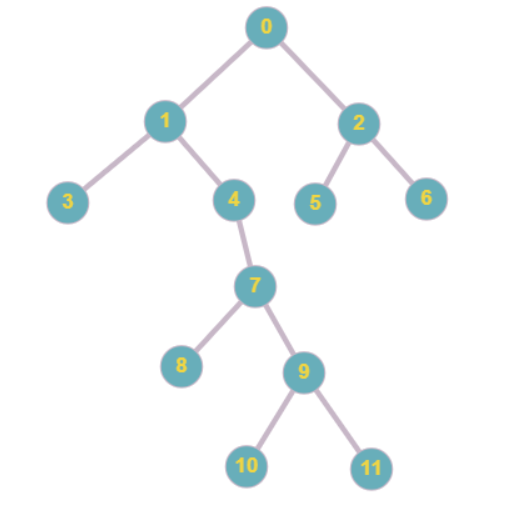
В ширину: 0 1 2 3 4 5 6 7 8 9 10 11



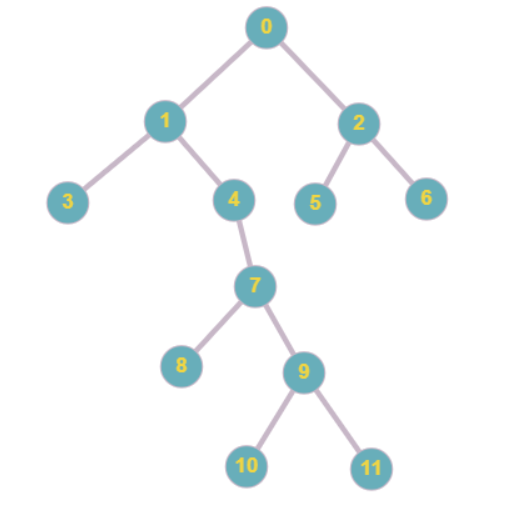
В глубину:



Прямой: 0 1 3 4 7 8 9 10 1 2 5 6



Обратный: 11 10 9 8 7 4 1 5 6 2 0



Симметричный: 3 1 8 7 10 9 11 4 0 5 2 6

