Министерство образования Республики Беларусь

Учреждение образования

«Брестский государственный технический университет»

Кафедра ИИТ

Лабораторная работа №3

за 1 семестр

По дисциплине: «Дискретная математика»

Тема: «Графы. Остовные деревья. Кодирование деревьев. Алгоритм Дейкстра»

Выполнил:

Студент 2 курса

Группы ПО-4(2)

Кречко К. А.

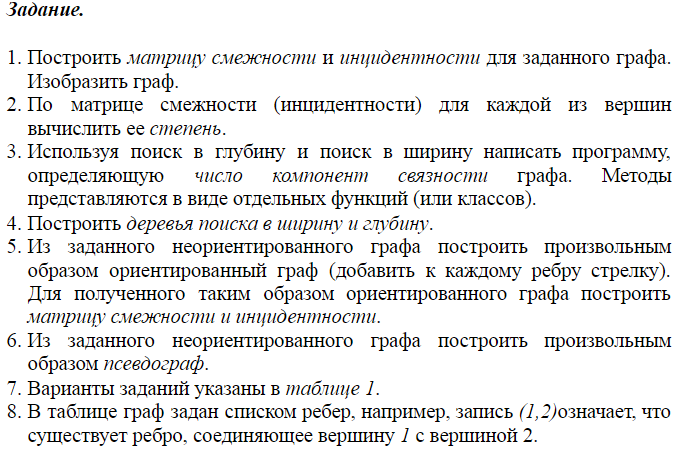
Проверила:

Глущенко Т.А.

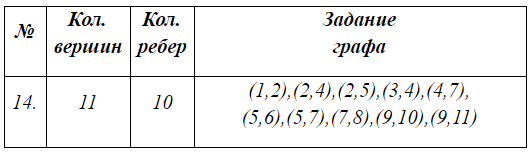
2020

**ЛАБОРАТОРНАЯ РАБОТА No 3.**

**Графы.**

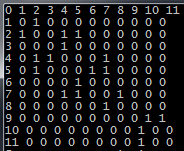


**Вариант: 14**

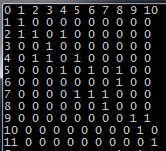
****

**1.**Построить *матрицу смежности* и *инцидентности* для заданного графа. Изобразить граф.

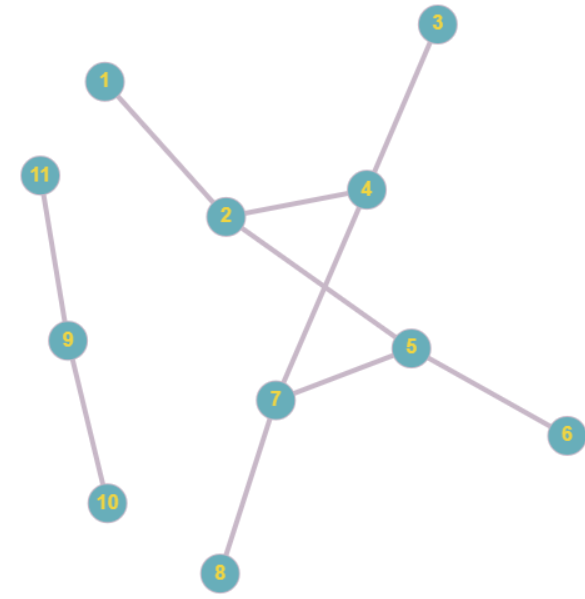
Матрица смежности:



Матрица инцидентности:



Граф:



**2.** По матрице смежности (инцидентности) для каждой из вершин вычислить ее *степень*.

Фрагмент кода:

for (int i = 1, n = 0; i < ROWS; i++)

{

for (int j = 1; j < COLS; j++)

{

n += arr[i][j];

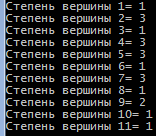
}

cout << "Степень вершины " << i << "= " << n << endl;

n = 0;

}

Результат:



**3.** Используя поиск в глубину и поиск в ширину написать программу, определяющую *число компонент связности* графа. Методы представляются в виде отдельных функций (или классов).

Обход в ширину:

Код:

#include <iostream>

#include <queue> //библиотека для очереди

using namespace std;

int main() {

queue<int> q;

int mas[11][11] = { { 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, }, //строим матрицу смежности

{ 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0 },

{ 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0 } };

int nodes[11]; // вершины графа

for (int i = 0; i < 11; i++)

nodes[i] = 0; // присваиваем всем вершинам значение 0 (чтобы проверять задействовали ли мы её или нет)

for (int f = 0; f < 11; f++) {

q.push(f); // помещаем в очередь первую вершину

while (!q.empty()) { // проверяем пустая ли очередь

int node = q.front(); // извлекаем вершину

q.pop();

if (nodes[node] == 2) continue;

nodes[node] = 2; // отмечаем ее как посещенную

for (int j = 0; j < 11; j++)

{ // проверяем для нее все смежные вершины

if (mas[node][j] == 1 && nodes[j] == 0) // если вершина смежная и не обнаружена, то:

{

q.push(j); // добавляем ее в очередь

nodes[j] = 1; // отмечаем вершину как обнаруженную

}

}

cout << node + 1 << " - "; // выводим номер вершины

}

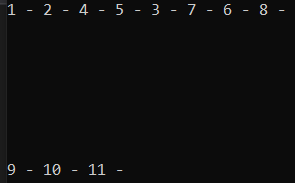
cout <<endl;

}

return 0;

}

Результат:



Обход в глубину:

Код:

#include <iostream>

#include <stack> // стек

using namespace std;

int main()

{

stack<int> Stack;

int mas[11][11] = { { 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, }, //строим матрицу смежности

{ 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0 },

{ 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0 } };

int nodes[11]; // вершины графа

for (int i = 0; i < 11; i++) // исходно все вершины равны 0

nodes[i] = 0;

Stack.push(0); // помещаем в очередь первую вершину

while (!Stack.empty())

{ // пока стек не пуст

int node = Stack.top(); // извлекаем вершину

Stack.pop();

if (nodes[node] == 2) continue;

nodes[node] = 2; // отмечаем ее как посещенную

for (int j = 10; j >= 0; j--)

{ // проверяем для нее все смежные вершины

if (mas[node][j] == 1 && nodes[j] != 2)

{ // если вершина смежная и не обнаружена

Stack.push(j); // добавляем ее в cтек

nodes[j] = 1; // отмечаем вершину как обнаруженную

}

}

cout << node + 1 << endl; // выводим номер вершины

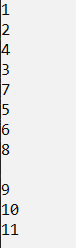
}

cin.get();

return 0;

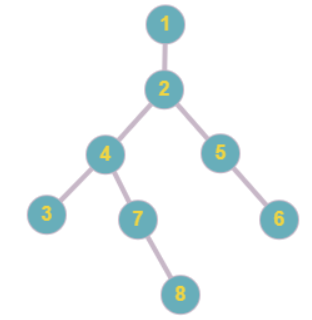
}

Резульатат:

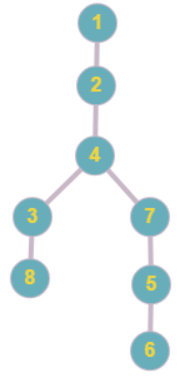
****

**4.** Построить *деревья поиска в ширину и глубину*.

Дерево поиска в ширину:

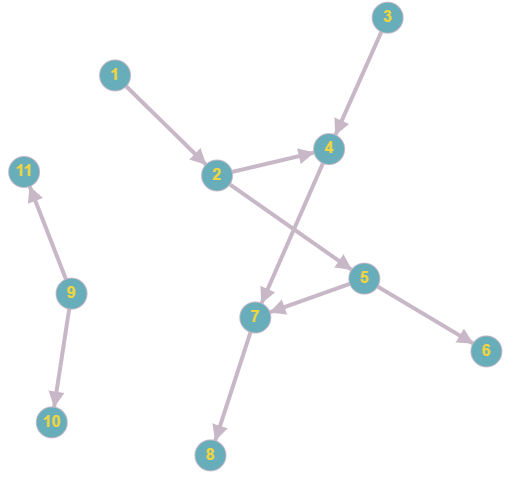


Дерево поиска в глубину:

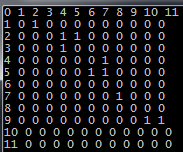


**5.** Из заданного неориентированного графа построить произвольным образом ориентированный граф (добавить к каждому ребру стрелку). Для полученного таким образом ориентированного графа построить *матрицу смежности и инцидентности*.

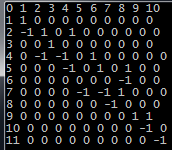
Граф:



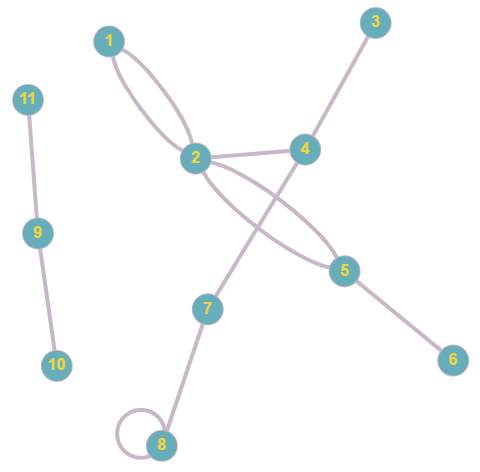
Матрица смежности:



Матрица инцидентности:



**6.** Из заданного неориентированного графа построить произвольным образом *псевдограф*.



**Вопросы:**

1. В каком из методов обхода графа путь в дереве поиска соответствует кратчайшему (т.е. содержащему наименьшее количество ребер) пути от вершины *s* до вершины *v*.

В методе обхода **в ширину** в дереве поиска показывается кратчайший путь от вершины s до вершины u, так как при методе поиска в ширину 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 (В графах из л.р. деревья совпали).

1. Для какого из обходов строится единственное (с точностью до изоморфизма) дерево поиска, а для какого можно построить их несколько.

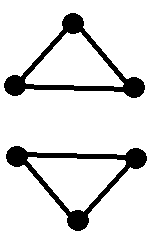
Единственное дерево существует только при обходе в ширину т.к. при построении дерева поиска в ширину от выбранной вершины все доступные вершины идут сразу. А при построении дерева поиска в глубину существует несколько вариаций, т.к. обход может выбираться по-разному.

1. Какое из утверждений является верным:  
   а) Полный граф всегда является   регулярным графом;  
   б) Регулярный граф всегда является   полным графом.

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

Регуля́рный граф — граф, степени всех вершин которого равны, то есть каждая вершина имеет одинаковое количество соседей.

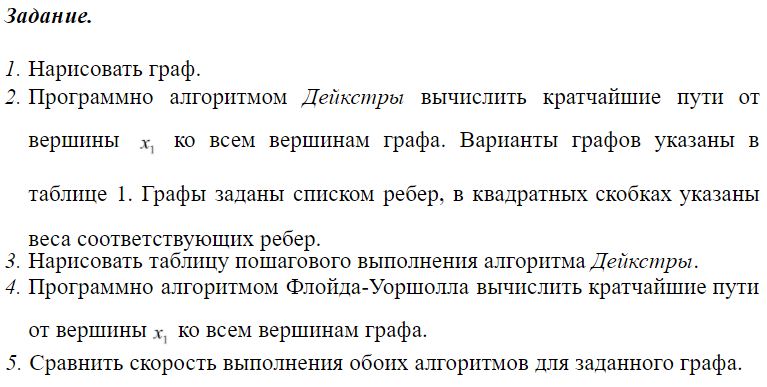
Соответсвенно каждый полный граф является регулярным (вершины имеют максимальную степень), но не каждый регулярный граф полный.

**** граф регулярный (степень 2) но не полный.

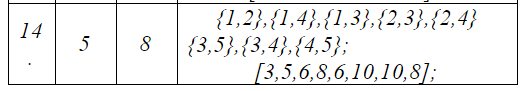
1. Что показывает сумма строк, столбцов матриц смежности и инцидентности для простого графа?

В матрице смежности для простого графа сумма элементов строки (или столбца) равна *степени* соответствующей вершины. В матрице инцидентности для *простого* графа сумма элементов строки также равна степени соответствующей вершины, а сумма элементов любого столбца равна *2.*

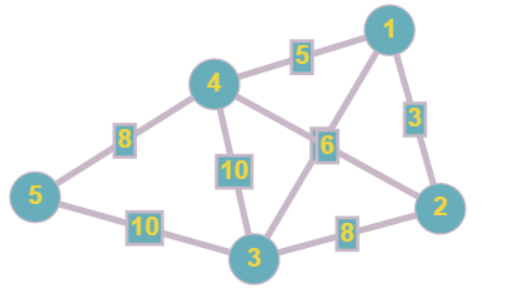
**Нахождение кратчайшего пути в графе (Дейкстра)**

****

**Вариант – 14**

****

1. Нарисовать граф.

****

1. Программно алгоритмом *Дейкстры* вычислить кратчайшие пути от вершины https://lh4.googleusercontent.com/NMWAPrEr096tAP6SxdMaETNRmRjU9FUObx_8JbvA43SeR_bwJSg84tLEKM5sz_U3dqbZCfD7en9GFvwGVkQohQg0DUx8WfbhjzjZWbmIIAjC5yWXwUt1blhDnFYcO4XSema1QTk ко всем вершинам графа. Варианты графов указаны в таблице 1. Графы заданы списком ребер, в квадратных скобках указаны веса соответствующих ребер.

Код:

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include <stdio.h>

#include <stdlib.h>

#define SIZE 5

using namespace std;

int main()

{

int a[SIZE][SIZE]; // матрица весов

int d[SIZE]; // минимальное расстояние

int v[SIZE]; // посещенные вершины

int temp, minindex, min;

int begin\_index = 0;

system("chcp 1251");

system("cls");

system("color f0");

// Инициализация матрицы весов

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

{

a[i][i] = 0;

for (int j = i + 1; j < SIZE; j++) {

printf("Введите расстояние %d - %d: ", i + 1, j + 1);

scanf("%d", &temp);

a[i][j] = temp;

a[j][i] = temp;

}

}

// Вывод матрицы весов

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

{

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

printf("%5d ", a[i][j]);

printf("\n");

}

//Инициализация вершин и расстояний

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

{

d[i] = 10000;

v[i] = 1;

}

d[begin\_index] = 0;

// Шаг алгоритма

do {

minindex = 10000;

min = 10000;

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

{ // Если вершину ещё не обошли и вес меньше min

if ((v[i] == 1) && (d[i] < min))

{ // Переприсваиваем значения

min = d[i];

minindex = i;

}

}

// Добавляем найденный минимальный вес

// к текущему весу вершины

// и сравниваем с текущим минимальным весом вершины

if (minindex != 10000)

{

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

{

if (a[minindex][i] > 0)

{

temp = min + a[minindex][i];

if (temp < d[i])

{

d[i] = temp;

}

}

}

v[minindex] = 0;

}

} while (minindex < 10000);

// Вывод кратчайших расстояний до вершин

printf("\nКратчайшие расстояния до вершин: \n");

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

printf("%5d ", d[i]);

// Восстановление пути

int ver[SIZE]; // массив посещенных вершин

int end = 4; // индекс конечной вершины = 5 - 1

ver[0] = end + 1; // начальный элемент - конечная вершина

int k = 1; // индекс предыдущей вершины

int weight = d[end]; // вес конечной вершины

while (end != begin\_index) // пока не дошли до начальной вершины

{

for (int i = 0; i < SIZE; i++) // просматриваем все вершины

if (a[i][end] != 0) // если связь есть

{

int temp = weight - a[i][end]; // определяем вес пути из предыдущей вершины

if (temp == d[i]) // если вес совпал с рассчитанным

{ // значит из этой вершины и был переход

weight = temp; // сохраняем новый вес

end = i; // сохраняем предыдущую вершину

ver[k] = i + 1; // и записываем ее в массив

k++;

}

}

}

// Вывод пути (начальная вершина оказалась в конце массива из k элементов)

printf("\nВывод кратчайшего пути\n");

for (int i = k - 1; i >= 0; i--)

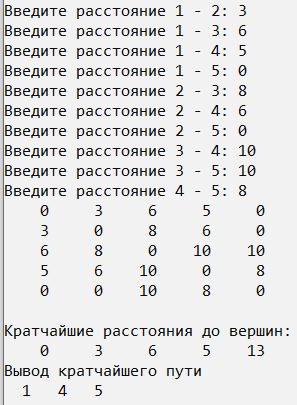
printf("%3d ", ver[i]);

getchar(); getchar();

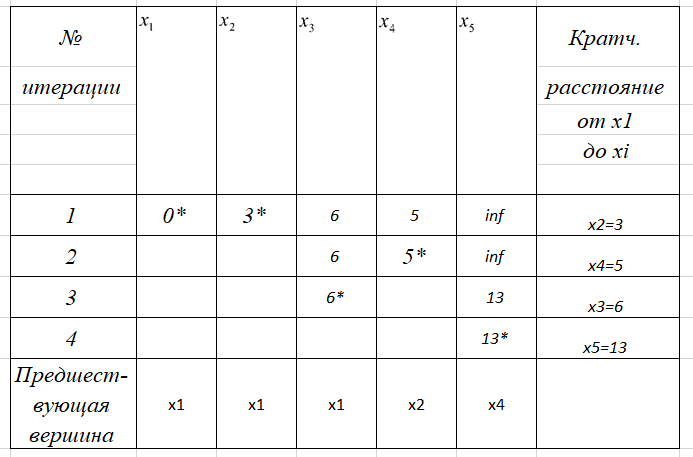
return 0;

}

**Результат:**

****

1. Нарисовать таблицу пошагового выполнения алгоритма *Дейкстры*.

****

1. Программно алгоритмом Флойда-Уоршолла вычислить кратчайшие пути от вершиныhttps://lh4.googleusercontent.com/NMWAPrEr096tAP6SxdMaETNRmRjU9FUObx_8JbvA43SeR_bwJSg84tLEKM5sz_U3dqbZCfD7en9GFvwGVkQohQg0DUx8WfbhjzjZWbmIIAjC5yWXwUt1blhDnFYcO4XSema1QTk ко всем вершинам графа.

**Код:**

#include <iostream>

#include <algorithm>

#include <fstream>

#define INF 228

using namespace std;

void printMatrix(int\*\* matrix, int numberOfVert) {

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

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

if (matrix[i][j] == INF) {

cout << "-" << " ";

}

else {

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

}

}

cout << endl;

}

}

//matrix - матрица смежности

void originalFloydWarshall(int\*\* matrix, int numberOfVert) {

//Пробегаемся по всем вершинам и ищем более короткий путь

//через вершину k

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

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

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

//Новое значение ребра равно минимальному между старым

//и суммой ребер i <-> k + k <-> j (если через k пройти быстрее)

//выбирается минимальное

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

}

}

}

}

int main() {

ifstream file("hello.txt");

int numberOfVert = 5;

cout << numberOfVert <<endl;

//Матрица смежности с весами ребер графа(101 - ребра нет, 0 ребро в себя)

int\*\* matrix = new int\* [numberOfVert];

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

matrix[i] = new int[numberOfVert];

}

//Считываем матрицу весов ребер

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

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

file >> matrix[i][j];

}

}

file.close();

cout << "Old matrix" << endl;

printMatrix(matrix, numberOfVert);

originalFloydWarshall(matrix, numberOfVert);

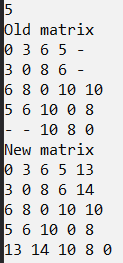
cout << "New matrix" << endl;

printMatrix(matrix, numberOfVert);

return 0;

}

**Результат:**

******

1. Сравнить скорость выполнения обоих алгоритмов для заданного графа.

Алгоритм Дейкстра работает быстрее (О(n2)), чем алгоритм Флойда-Уорршела (О(n3)). Последний 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.

**Вопросы:**

1. Что такое *«жадный»* алгоритм и какой из указанных алгоритмов является *«жадным»*? Указать *«О большое»* для обоих алгоритмов.

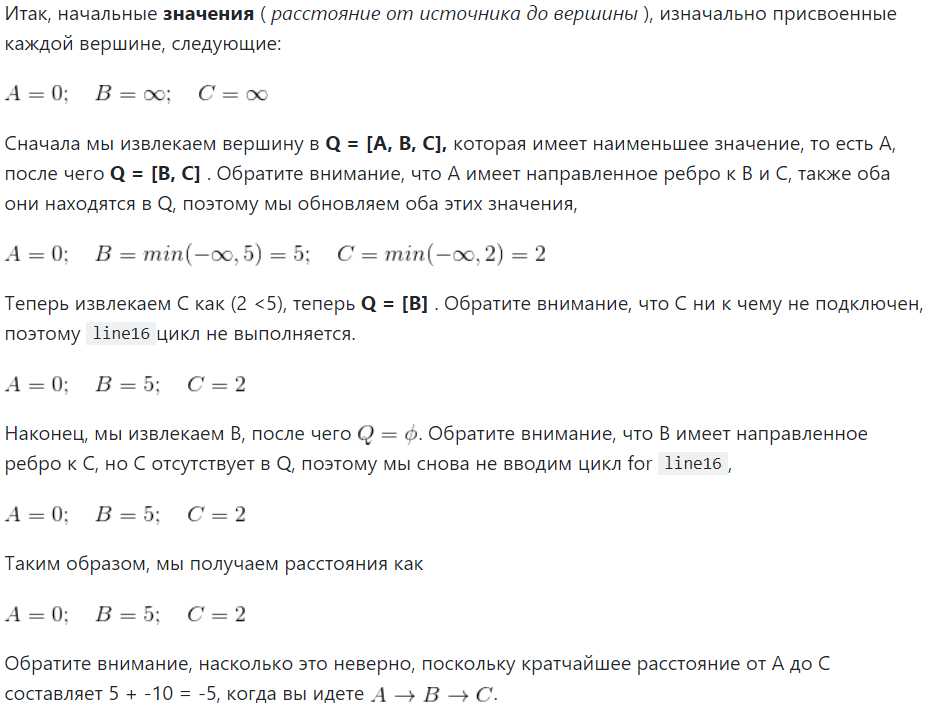
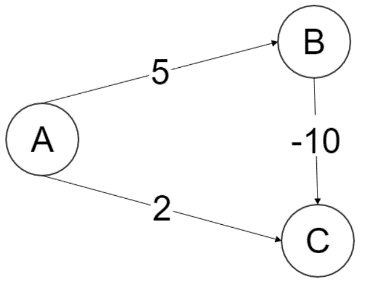
Жадный алгоритм — алгоритм, заключающийся в принятии локально [оптимальных решений](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D1%82%D0%B8%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D1%80%D0%B5%D1%88%D0%B5%D0%BD%D0%B8%D0%B5) на каждом этапе, допуская, что конечное решение также окажется оптимальным. “Жадным” алгоритмом здесь является алгоритм Флойда.

Для Дейкстры: О(n2), для Флойда: О(n3).

1. Почему классический алгоритм *Дейкстры* не работает для отрицательных весов?

Алгоритм Дейкстры предполагает, что пути могут становиться только «тяжелее», поэтому, если у вас есть путь от A до B с весом 3 и путь от A до C с весом 3, вы не можете добавить ребро и добраться из A в B через C с весом менее 3.

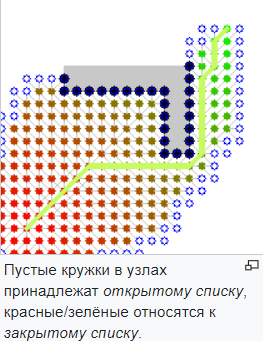
Это предположение делает алгоритм быстрее, чем алгоритмы, которые должны учитывать отрицательные веса.



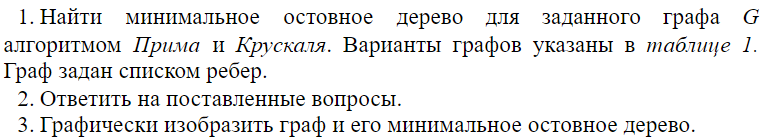
1. Описать алгоритм *А стар (А\*)* и область его применения

A\* пошагово просматривает все пути, ведущие от начальной вершины в конечную, пока не найдёт минимальный. Как и все [информированные алгоритмы поиска](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA), он просматривает сначала те маршруты, которые «кажутся» ведущими к цели. От [жадного алгоритма](https://ru.wikipedia.org/wiki/%D0%96%D0%B0%D0%B4%D0%BD%D1%8B%D0%B9_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC), 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.

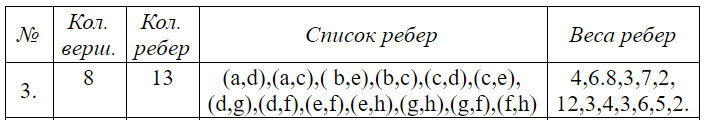
В начале работы просматриваются узлы, смежные с начальным; выбирается тот из них, который имеет минимальное значение *f(x)*, после чего этот узел раскрывается. На каждом этапе алгоритм оперирует с множеством путей из начальной точки до всех ещё не раскрытых (листовых) вершин графа — множеством частных решений, — которое размещается в [очереди с приоритетом](https://ru.wikipedia.org/wiki/%D0%9E%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C_%D1%81_%D0%BF%D1%80%D0%B8%D0%BE%D1%80%D0%B8%D1%82%D0%B5%D1%82%D0%BE%D0%BC_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). 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.



***Нахождение минимального остовного дерева связанного неориентированного графа.***

****

**Вариант – 3**

****

**1.** Найти минимальное остовное дерево для заданного графа *G* алгоритмом *Прима* и *Крускаля*. Варианты графов указаны в *таблице 1.* Граф задан списком ребер.

**Алгоритм Прима:**

На вход алгоритма подаётся связный неориентированный граф. Для каждого ребра задаётся его стоимость.

Сначала берётся произвольная вершина и находится ребро, инцидентное данной вершине и обладающее наименьшей стоимостью. Найденное ребро и соединяемые им две вершины образуют дерево. Затем, рассматриваются рёбра графа, один конец которых — уже принадлежащая дереву вершина, а другой — нет; из этих рёбер выбирается ребро наименьшей стоимости. Выбираемое на каждом шаге ребро присоединяется к дереву. Рост дерева происходит до тех пор, пока не будут исчерпаны все вершины исходного графа.

Результатом работы алгоритма является остовное дерево минимальной стоимости

Код:

#include<iostream>

#include<conio.h>

using namespace std;

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

int visited[10] = { 0 }, mini, mincost = 0, cost[10][10];

int main()

{

system("color F0");

setlocale(LC\_ALL, "rus");

int path[100] = { 0 }; //В этот массив будут записываться вершины, по которым составиться путь

int path\_index = 0;

cout << "Введи количество вершин "; cin >> n;

cout << "Введи матрицу смежности\n";

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

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

{

cin >> cost[i][j];

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

cost[i][j] = 999; //999 - это что-типа бесконечности. Должно быть больше чем значения веса каждого из ребер в графе

}

visited[1] = 1;

cout << "\n";

while (ne < n)

{

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

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

if (cost[i][j] < mini)

if (visited[i] != 0)

{

mini = cost[i][j];

a = u = i;

b = v = j;

}

if (visited[u] == 0 || visited[v] == 0)

{

path[path\_index] = b;

path\_index++;

//cout<<"\n "<<ne++<<" "<<a<<" "<<b<<min; //Можно вывести так

ne++; //если строчку выше раскомментировать - эту закомментировать

mincost += mini;

visited[b] = 1;

}

cost[a][b] = cost[b][a] = 999;

}

cout << "\n";

cout << 1 << " --> ";

for (int i = 0; i < n - 1; i++)

{

cout << path[i];

if (i < n - 2) cout << " --> ";

}

cout << "\n Минимальная стоимость " << mincost;

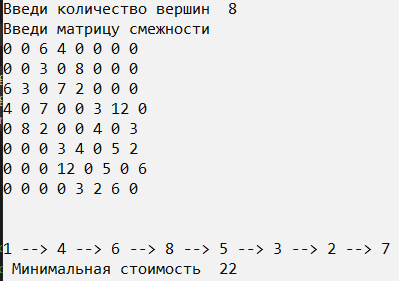
cin.get();

cin.get();

return 0;

}

Результат:



Алгоритм Крускаля:

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.

Код:

#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(LC\_ALL, "ru");

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("\nМинимальная 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;

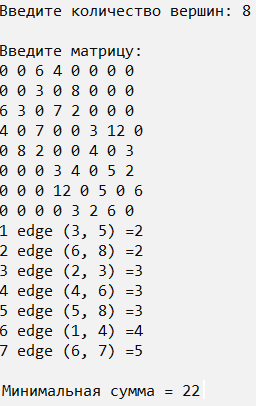
return 1;

}

return 0;

}

Результат:



Вопросы:

**1.** Для какого графа определяет число остовных деревьев формула *Кэли*?

Число остовых деревьев для   *n* размеченных вершин (множество из *n* вершин размечено, если каждой вершине приписано единственное натуральное число между *1* и *n*) определяется формулой *Кэли* для дерева.

**2.** Подсчитать по формуле *Кэли* и нарисовать число остовных деревьев для *n = 3*.

Формула: 

Соответсвенно:

33-2 = 3.

**3.** \* пункт 2 выполнить для *n = 4.*

44-2 = 16

**4.** Какое остовое дерево находится алгоритмом *Дейкстры*?

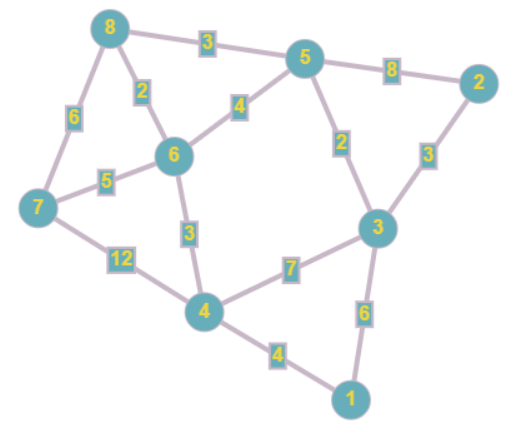
Обход вершин которого стоит меньше всего.

**5.** Может ли быть несколько минимальных остовых деревьев?

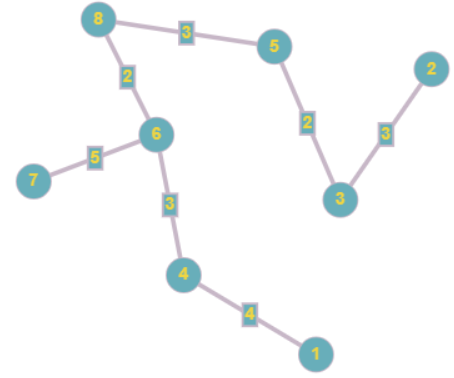
Если каждое ребро имеет отдельный вес, тогда будет только одно уникальное минимальное остовное дерево. В противном случае, может существовать несколько минимальных остовов.

**4.** Графически изобразить граф и его минимальное остовное дерево.

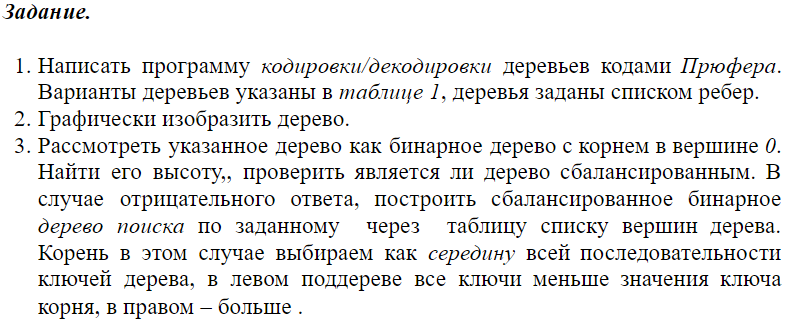
Граф:



Минимальное остовное дерево:



***Кодирование деревьев.***



**Вариант – 7**

****

**Код:**

#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> Mass;

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);

Mass = d;

Code.clear();

D.clear();

}

void Graph::add\_edge(int u, int v) { //добавление ребра

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

Mass[u]++;

Mass[v]++;

}

void Graph::display() { //просмотр

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

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

min = G[j].first; //находим наименьшее

x = j;

}

}

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

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

min = G[j].second; //находим наименьшее

x = j;

}

}

}

// уменьшить значение вершин на 1

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

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

}

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

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

}

// сохранить вершину, из которой удаляется лист

if (Mass[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> Mass(n, 1);

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

++Mass[Code[i]];

}

set<int> leaves;

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

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

leaves.insert(v);

}

}

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

}

int main() {

setlocale(0, "");

Graph rebro(14);

rebro.add\_edge(0, 1);

rebro.add\_edge(0, 2);

rebro.add\_edge(1, 3);

rebro.add\_edge(1, 4);

rebro.add\_edge(2, 5);

rebro.add\_edge(2, 6);

rebro.add\_edge(3, 7);

rebro.add\_edge(3, 8);

rebro.add\_edge(6, 9);

rebro.add\_edge(8, 10);

rebro.add\_edge(8, 11);

rebro.add\_edge(10, 12);

rebro.add\_edge(11, 13);

rebro.Prufer\_encode();

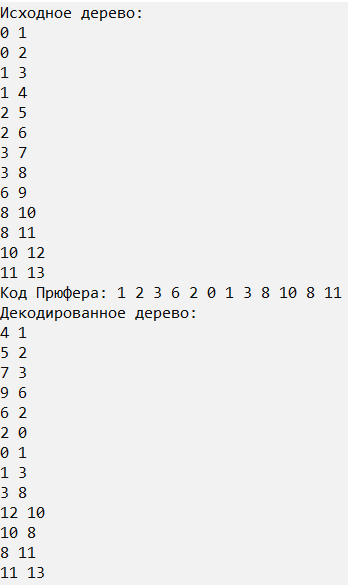
rebro.Prufer\_decode();

rebro.display();

return 0;

}

**Результат:**

****

2. Графически изобразить дерево.

3. Рассмотреть указанное дерево как бинарное дерево с корнем в вершине *0*. Найти его высоту,, проверить является ли дерево сбалансированным. В случае отрицательного ответа, построить сбалансированное бинарное *дерево поиска* по заданному  через  таблицу списку вершин дерева. Корень в этом случае выбираем как *середину* всей последовательности ключей дерева, в левом поддереве все ключи меньше значения ключа корня, в правом – больше .

Исходное дерево не является сбалансированным.

Сбалансированное дерево:

Вопросы:

1.Как задается дерево в программе?

Граф задается в виде списка ребер, представленных векторами, где каждый элемент вектора – пара вершин дерева.

2.Что такое свободное дерево, ориентированное дерево, сбалансированное дерево?

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.

3.Как определяется высота дерева, уровень вершины?

Высота дерева равна длине самого длинного пути от корня дерева до листа.

Уровень вершины дерева - удаленность вершины от корня. Уровень корня дерева равен 0.

4.Для вашего дерева указать все порядки обхода дерева: прямой, обратный, симметричный, в глубину, в ширину.

прямой: 0 1 3 4 2 5 6 7 8 10 9 11 12

обратный: 3 4 12 1 11 9 10 8 7 6 5 2 0

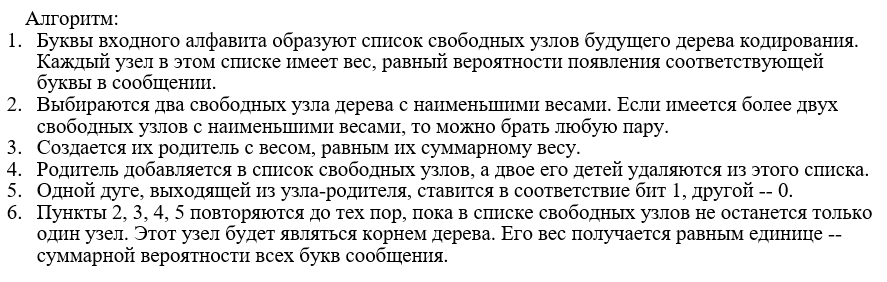
симметричный: 5 3 6 1 4 7 10 0 2 9 11 12

в глубину(префиксный): 5 3 6 1 4 8 0 2 7 10 9 11 12

в ширину: 5 3 1 0 2 4 6 8 7 10 9 11 12

5.\*Что такое дерево Хаффмана, для чего оно служит и алгоритм его построения (пример)?

Дерево Хаффмана - это, фактически, двоичное дерево, по которому можно узнать (при наличии статистических данных) какой последовательностьяЯю бит выгоднее всего кодировать данный символ.



**Вывод:** Яизучил принципы работы с деревьями.