Оглавление

[ВВЕДЕНИЕ 3](#_Toc59575144)

[1. Анализ проблемной области. 4](#_Toc59575145)

[2. Теоретическая часть. 6](#_Toc59575146)

[2.1. Теория по поиску центров на вершинах по матрице расстояний. 6](#_Toc59575147)

[2.2. Теория по алгоритму Хакими поиска абсолютного центра для неориентированного графа. 6](#_Toc59575148)

[2.3. Теория по алгоритму Хакими поиска абсолютного центра для ориентированного и неориентированного графов. 8](#_Toc59575149)

[3. Реализация алгоритмов для различных типов задач на языке программирования. 9](#_Toc59575150)

[3.1. Поиск центров на вершинах по матрице расстояний на языке программирования C++. 9](#_Toc59575151)

[3.2. Алгоритм Хакими для неориентированного графа на языке программирования C++. 10](#_Toc59575152)

[3.3. Алгоритм Хакими для ориентированного и неориентированного графов на языке программирования C++. 15](#_Toc59575153)

[ЗАКЛЮЧЕНИЕ 26](#_Toc59575154)

[СПИСОК ЛИТЕРАТУРЫ 27](#_Toc59575155)

[ПРИЛОЖЕНИЕ 28](#_Toc59575156)

# ВВЕДЕНИЕ

Во время практики по теории конечных графов ставилась цель изучить выбранную тему и выполнить проект, написав программу на языке программирования. В качестве темы проекта была выбрана «Размещение центров в графах» с написанием программы на языке программирования. В практической деятельности постоянно возникают задачи «наилучшего» размещения оборудования или средств обслуживания в сетях или графах. В более общей задаче требуется разместить несколько таких пунктов обслуживания. Поэтому задача размещения центров в графах актуальна и в настоящее время.

Цель проекта - написать программу для определения центров в графах.

Были поставлены задачи проекта:

- рассмотреть минимаксную задачу размещения центров в графах,

- провести анализ алгоритмов для различных типов задач,

- реализовать алгоритмы определения оптимального размещения центров в графах на языке программирования C++

# Анализ проблемной области.

В современном мире всё чаще приходится проектировать, строить, обслуживать системы, состоящие из большого количества элементов, соединённых многими связями друг с другом, так называемые «сложные системы». При этом связи между элементами с навешиваемыми на них скалярными характеристиками могут изменяться во времени. К таким практическим системам в первую очередь следует отнести современные коммуникационные и информационные сети, объём и масштаб которых в процессе роста, развития и эксплуатации может претерпевать значительные изменения.

В таких сетях хорошо известная в дискретной математике задача о назначениях, в частности, о размещении центров, приобретает новый смысл. Для решения современных задач структурной динамики становится где-то сложно, а где-то невозможно использовать классическую теорию графов, поскольку необходимо приходится моделировать динамические структуры с большим количеством элементов и ещё большим количеством отношений между ними.

В практической деятельности постоянно возникают задачи «наилучшего» размещения оборудования или средств обслуживания в сетях или на графах. В частности, если граф представляет сеть дорог, а вершины графа соответствуют отдельным районам, то можно поставить задачу оптимального размещения больниц, полицейских участков, пожарных частей и многих других крайне необходимых предприятий и служб. В таких случаях критерий оптимальности может состоять в минимизации суммарного расстояния, максимального расстояния, стоимости или времени проезда и т.д. от пункта обслуживания до самой удалённой вершины графа, т.е. в оптимизации «наихудшего варианта». В более общей задаче требуется разместить не один, а несколько таких пунктов обслуживания. При этом самая отдалённая вершина графа должна находиться на минимально возможном расстоянии по крайней мере от одного пункта обслуживания. К таким задачам прежде всего относят задачи размещения аварийных служб, поэтому объективным требованием в них является минимизация наибольшего расстояния от произвольной вершины графа до ближайшего к ней пункта обслуживания. По очевидным причинам задачи такого типа называются «минимаксными задачами размещения». Полученные при решении этих задач места размещения пунктов обслуживания называются «центрами» графа.

В некоторых задачах размещения лучше всего было бы минимизировать сумму всех расстояний от вершин графа до центра графа, если предполагать, что ищется место для размещения только одного такого пункта обслуживания. Такой критерий является наиболее подходящим, например, в задаче о размещении склада в сети дорог, где вершины сети представляют потребителей, обслуживаемых этим складом, или в задаче размещения телефонных станций в телефонной сети, где вершины сети представляют собой абонентов. Задачи такого типа вообще относятся к «минисуммным задачам размещения», хотя целевая функция является часто не просто суммой расстояний, а суммой различных функций от расстояний. Места размещения пунктов обслуживания, полученные в результате решения минисуммной задачи, называются «медианами» графа.

# Теоретическая часть.

## Теория по поиску центров на вершинах по матрице расстояний.

Матрица расстояний — это квадратная матрица типа «объект-объект» содержащая в качестве элементов расстояния между объектами в метрическом пространстве. Для каждой вершины находим число разделения. Среди всех чисел разделений находим наименьшее. Та вершина (или вершины), число разделений которой равно наименьшему, будет являться центром графа.

## Теория по алгоритму Хакими поиска абсолютного центра для неориентированного графа.

Этот метод очень прост и для неориентированного графа состоит в следующем:

1. Для каждого ребра  графа найти точки (или точку) на  которые имеют наименьшее число разделения.
2. Из всех точек  (k = 1, 2, …, m) в качестве абсолютного центра графа G выбрать точку с наименьшим числом разделения.

Первый шаг метода осуществляется следующим образом:

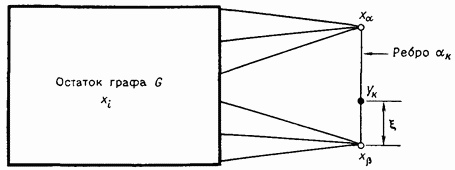


Рисунок 1 – Ребро  графа G

Возьмем ребро  графа G (рис. 1). Имеем:

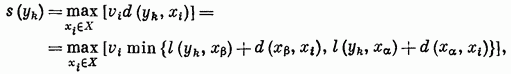


Рисунок 2 – Формула

поскольку расстояние d (, ) равно либо длине маршрута, проходящего через вершину , либо длине маршрута, проходящего через , ℓ (, ) и ℓ (, ) являются длинами соответствующих частей ребра .

Пусть ℓ (, ) = ξ. Поскольку ℓ (, ) = – ℓ (, ) = – ξ, то соотношение примет вид



Рисунок 3 – Формула

Для фиксированной вершины и при каждом значении  можно найти наименьшие значения выражений, заключенных в соотношении в квадратные скобки. Для этого выпишем отдельно два указанных выражения:



Рисунок 4 – Формула

и, рассматривая их относительно ξ  строим нижнюю «огибающую» для соответствующих им [прямых линий](http://scask.ru/a_lect_math1.php?id=25).

Повторяя эту процедуру для всех вершин ∈ X мы построим на одном и том же чертеже все остальные нижние «огибающие». Далее вычертим верхнюю «огибающую» для всех ранее полученных нижних «огибающих», которая дает числа разделения s () для всех значений параметра ξ, т. е. для всех точек  ребра  Построенная огибающая (она составлена из линейных кусков) может иметь несколько минимумов. Точка  соответствующая наименьшему из этих минимумов, является абсолютным центром , отвечающим дополнительному ограничению: он должен лежать на ребре  Абсолютным центром графа будет такая точка , которой соответствует наименьший из минимумов, определяющих указанные выше абсолютные центры на ребрах  (k = 1, 2, …, m).

## Теория по алгоритму Хакими поиска абсолютного центра для ориентированного и неориентированного графов.

Для ориентированного графа метод остается таким же, надо только каждое «неориентированное» понятие заменить его «ориентированным двойником».

# Реализация алгоритмов для различных типов задач на языке программирования.

Для реализации алгоритмов различных типов задач нахождения центров в графах мы выбрали язык программирования C++.

## Поиск центров на вершинах по матрице расстояний на языке программирования C++.

В массиве s храним числа разделений, в min - минимальное число разделений:

int\* s = new int[m];

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

s[i] = 0;

int min = INT\_MAX;

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

{

Находим набольшее расстояние от вершины i до остальных вершин графа, то есть, находим число разделения:

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

{

if (d[i][j] > s[i])

{

s[i] = d[i][j];

}

}

Теперь находим наименьшее число разделений:

if (s[i] < min)

min = s[i];

}

После чего находим и выводим центры:

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

{

if (s[i] == min)

cout << "Center: " << i + 1 << endl << "Radius: " << s[i] << endl << endl;

}

}

Главная функция будет выглядеть таким образом:

int main()

{

Graph gr(5);

gr.InputTest1();

gr.PrintGraph();

gr.PrintDistances();

gr.PrintVertices();

gr.CVertexes();

return 0;

}

Результат работы программы:

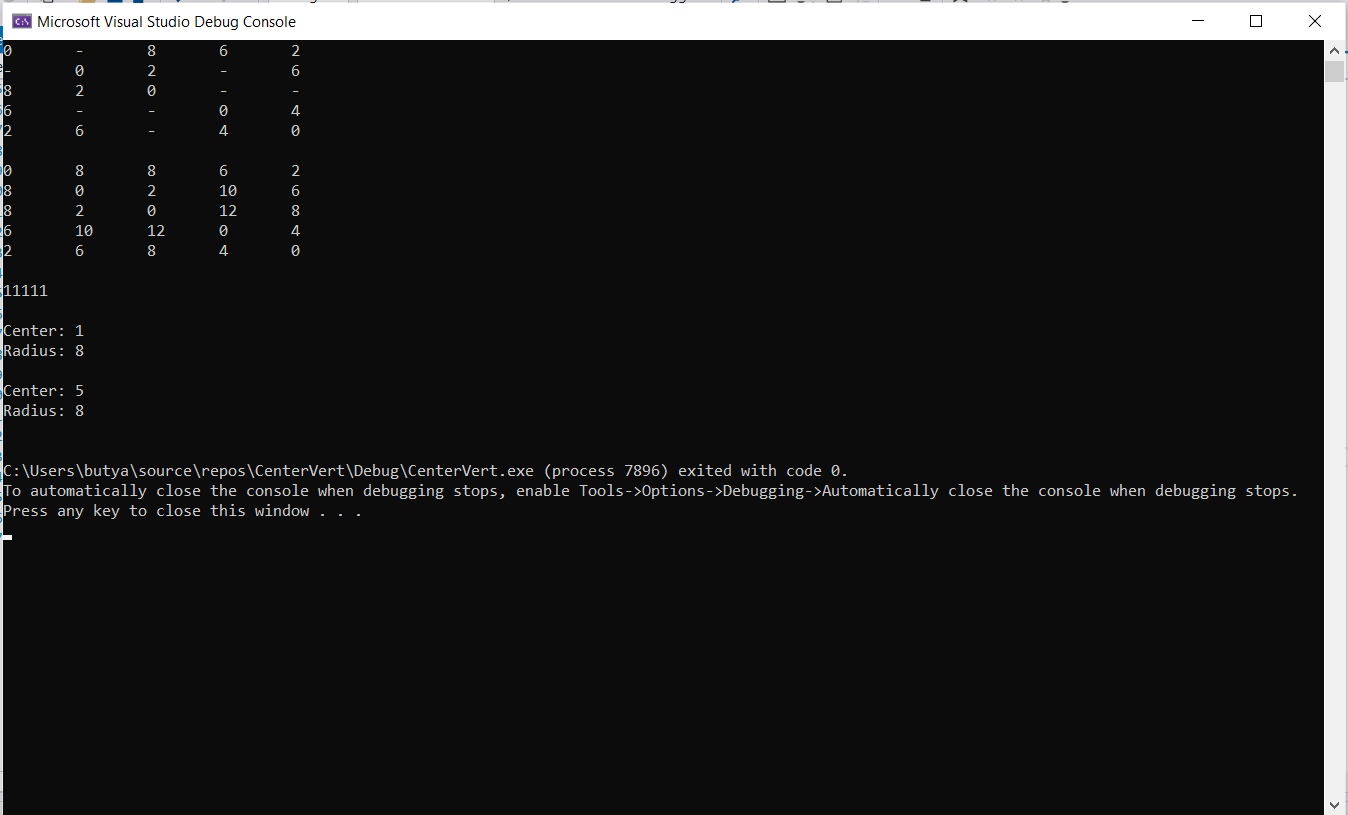


Рисунок 5 – Результат работы программы

## Алгоритм Хакими для неориентированного графа на языке программирования C++.

Поиск центров на вершинах:

vector<center\_on\_vertex> Centers\_on\_vertexes();

Поиск числе разделений для точек на ребре (a, b):

int\* Find\_S\_for\_all\_Y\_on\_AB(int a, int b);

Поиск локальных центров на ребер (a, b):

vector<center> LocalCenters(int a, int b);

Поиск абсолютных центров на гранях, не включая вершины:

vector<center> Centers\_on\_edges();

Поиск и вывод абсолютных центров:

void AbsCenters();

Следующие функции осуществляют алгоритм Хакими поиска абсолютных центров в неориентированном графе.

Поиск центров на вершинах:

vector<center\_on\_vertex> Graph::Centers\_on\_vertexes()

{

vector<center\_on\_vertex> Centers;

Ищем числа разделений для вершин и минимальное из них.

В массиве s будем хранить числа разделений. s[x] - число разделений вершины x. В min будем хранить минимальное число разделений.

int\* s = new int[m];

for (int x = 0; x < m; x++)

s[x] = 0;

int T;

int mins = INT\_MAX;

for (int x = 0; x < m; x++)

{

Находим максимальное расстояние от вершины x до остальных вершин графа, то есть, находим число разелений s[x]:

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

{

T = v[i] \* d[x][i];

if (T > s[x])

s[x] = T;

}

Ищем минимальное число разделений:

if (s[x] < mins)

mins = s[x];

}

Находим центры - вершины, с наименьшим числом разделений:

center\_on\_vertex c;

for (int x = 0; x < m; x++)

if (s[x] == mins)

{

c.a = x;

c.s = s[x];

Centers.push\_back(c);

}

return Centers;

}

Поиск числе разделений для точек на ребре (a, b), в массиве s хранятся числа разделений:

int\* Graph::Find\_S\_for\_all\_Y\_on\_AB(int a, int b)

{

int\* s = new int[g[a][b]];

for (int e = 0; e < g[a][b]; e++)

s[e] = 0;

Будем рассматривать точки, находящиеся на расстоянии 1, по всей длине ребра.

int T1; int T2;

for (int e = 1; e < g[a][b]; e++)

{

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

{

Теперь сравниваем пути в две стороны по ребру (a, b), выбирая наименьший:

T1 = v[i] \* (e + d[b][i]);

T2 = v[i] \* (g[a][b] - e + d[a][i]);

if (T1 < T2)

{

if (T1 > s[e])

s[e] = T1;

}

else

if (T2 > s[e])

s[e] = T2;

}

}

return s;

}

Поиск локальных центров на ребер (a, b), s - числа разделений точек на ребре (a, b):

vector<center> Graph::LocalCenters(int a, int b)

{

int\* s = Find\_S\_for\_all\_Y\_on\_AB(a, b);

int min = INT\_MAX;

center y;

y.a = a; y.b = b;

vector<center> ListOfY;

Находим наименьшее число разделений точек на ребре (a, b):

for (int e = 1; e < g[a][b]; e++)

if (s[e] < min)

min = s[e];

// Находим локальные центры на ребре (a, b)

for (int e = 1; e < g[a][b]; e++)

if (s[e] == min)

{

y.e = e;

y.s = s[e];

ListOfY.push\_back(y);

}

return ListOfY;

}

Поиск абсолютных центров на гранях, не включая вершины:

vector<center> Graph::Centers\_on\_edges()

{

center y;

vector<center> CentersOnAB;

vector<center> Centers;

int min = INT\_MAX;

Получаем список всех локальных центров Centers на ребрах графа:

for (int a = 0; a < m; a++)

for (int b = m - 1; b > a; b--)

if (g[a][b] < INT\_MAX)

{

CentersOnAB = LocalCenters(a, b);

if (CentersOnAB.front().s < min)

min = CentersOnAB.front().s;

while (!CentersOnAB.empty())

{

y = CentersOnAB.back();

Centers.push\_back(y);

CentersOnAB.pop\_back();

}

}

Ищем наименьшие из локальных центров:

vector<center> Abs;

int t = Centers.size();

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

if (Centers[i].s == min)

Abs.push\_back(Centers[i]);

return Abs;

}

Поиск и вывод абсолютных центров:

void Graph::AbsCenters()

{

vector<center\_on\_vertex> Centers = Centers\_on\_vertexes(); // центры на вершинах

vector<center> AbsCenters = Centers\_on\_edges(); // центры на гранях

Ищем абсолютные центры на вершинах и гранях:

center\_on\_vertex x;

center y;

x = Centers.front();

y = AbsCenters.front();

if (x.s > y.s)

Centers.clear();

else

if (y.s > x.s)

AbsCenters.clear();

Вывод:

while (!Centers.empty())

{

x = Centers.back();

cout << "Center:";

cout << x.a + 1 << endl;

cout << "Radius:" << x.s << endl;

Centers.pop\_back();

}

while (!AbsCenters.empty())

{

y = AbsCenters.back();

cout << "Absolut Center:";

cout << "(" << y.a + 1 << ", " << y.b + 1 << ") " << y.e << endl;

cout << "Radius:" << y.s << endl;

AbsCenters.pop\_back();

}

}

Главная функция будет выглядеть таким образом:

int main()

{

int m = 5;

Graph gr(m);

gr.InputTest1();

gr.PrintGraph();

gr.PrintDistances();

gr.AbsCenters();

return 0;

}

Результат работы программы:

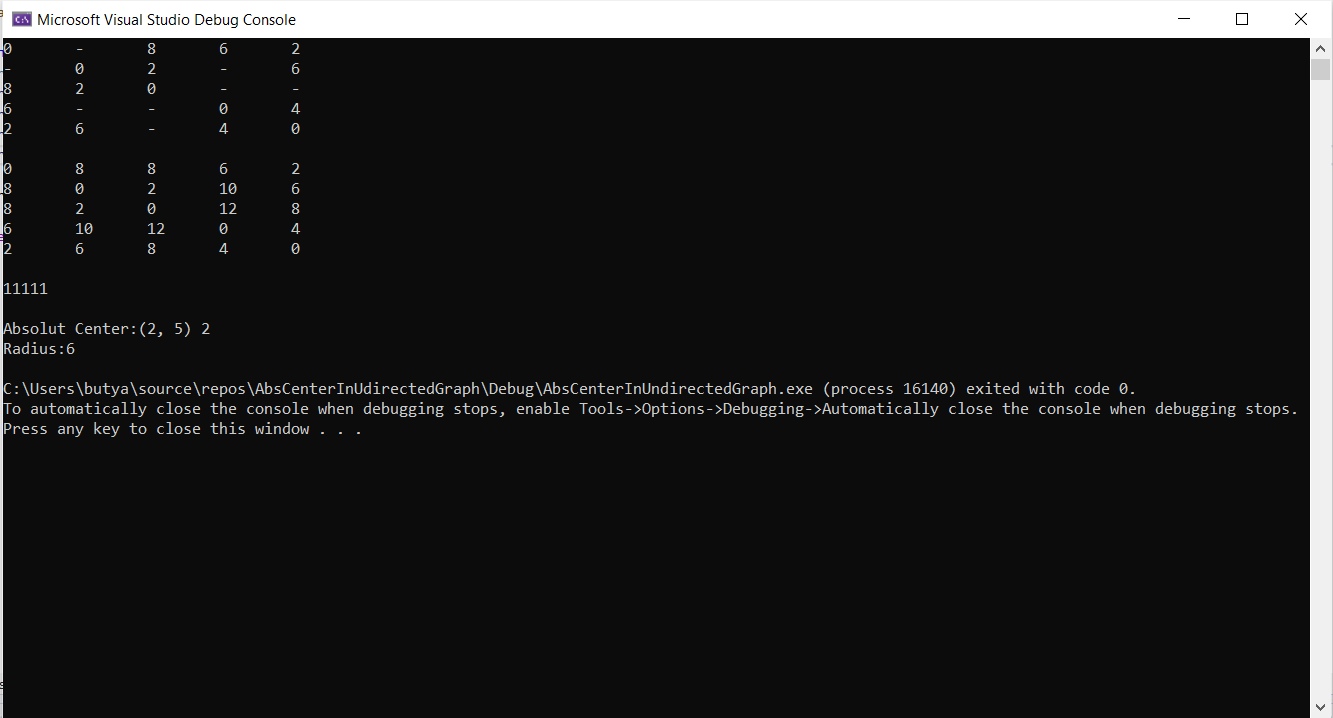


Рисунок 6 – Результат работы программы

## Алгоритм Хакими для ориентированного и неориентированного графов на языке программирования C++.

Находим числа центры среди вершин:

void Centers\_on\_vertexes(vector<center\_on\_vertex>& InCenters, vector<center\_on\_vertex>& ExCenters, vector<center\_on\_vertex>& Centers);

Теперь ищем внешние и внутренние числа разделений для всех точек Y на грани (a, b):

int\* Find\_S0\_for\_all\_Y\_on\_AB(int a, int b, bool undirected);

int\* Find\_St\_for\_all\_Y\_on\_AB(int a, int b, bool undirected);

//Находим локальные центры на грани (a, b)

void Local\_centers(int a, int b, vector<center>& mins0, vector<center>& minst, vector<center>& min);

Находим центры среди граней графа:

void Centers\_on\_edges(vector<center>& InAbsCenters, vector<center>& ExAbsCenters, vector<center>& AbsCenters);

//Находим абсолютные центры

void Abs\_centers();

};

Ищем числа разделений для вершин и минимальное из них. В массиве s0 будем хранить внешние числа разделений, в st – внутренние:

int\* s0 = new int[m];

int\* st = new int[m];

for (int x = 0; x < m; x++)

{

s0[x] = 0;

st[x] = 0;

}

int T;

В mins0 будем хранить минимальное число внешних разделений, в minst - внутренних, в mins - минимальную сумму внешнего и внутреннего числа разделений для одной точки, то есть, минимальное расстояние в одну сторону и обратно:

int mins0 = INT\_MAX;

int minst = INT\_MAX;

int mins = INT\_MAX;

for (int x = 0; x < m; x++)

{

Находим максимальное расстояние от вершины x до остальных вершин графа, то есть, находим числа разделений:

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

{

T = v[i] \* d[x][i];

if (T > s0[x])

s0[x] = T;

T = v[i] \* d[i][x];

if (T > st[x])

st[x] = T;

}

Ищем минимальные числа разделений:

if (s0[x] < mins0)

mins0 = s0[x];

if (st[x] < minst)

minst = st[x];

if (s0[x]+st[x] < mins)

mins = s0[x]+st[x];

}

Находим центры:

center\_on\_vertex c;

for (int x = 0; x < m; x++)

{

//внешние центры

if (s0[x] == mins0)

{

c.a = x;

c.s0 = s0[x];

c.st = st[x];

InCenters.push\_back(c);

}

//внутренние центры

if (st[x] == minst)

{

c.a = x;

c.s0 = s0[x];

c.st = st[x];

ExCenters.push\_back(c);

}

//центры в две стороны

if (s0[x]+st[x] == mins)

{

c.a = x;

c.s0 = s0[x];

c.st = st[x];

Centers.push\_back(c);

}

}

}

Находим внешние числа разделений для всех точек Y на грани (a, b):

int\* Graph::Find\_S0\_for\_all\_Y\_on\_AB(int a, int b, bool undirected)

{

В массив s0 будем хранить внешние числа разделений точек на ребре (a, b):

int\* s0 = new int[g[a][b]];

for (int e = 0; e < g[a][b]; e++)

s0[e] = 0;

//поиск на неориентированном ребре:

if (undirected)

{

int T;

for (int e = 1; e < g[a][b]; e++)

{

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

{

T = v[i] \* (e + d[b][i]);

if (T > s0[e])

s0[e] = T;

}

}

}

//поиск на ориентированном ребре

else

{

int T1; int T2;

for (int e = 1; e < g[a][b]; e++)

{

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

{

T1 = v[i] \* (e + d[b][i]);

T2 = v[i] \* (g[a][b] - e + d[a][i]);

if (T1 < T2)

{

if (T1 > s0[e])

s0[e] = T1;

}

else

if (T2 > s0[e])

s0[e] = T2;

}

}

}

return s0;

}

Находим внутренние числа разделений для всех точек Y на грани (a, b):

int\* Graph::Find\_St\_for\_all\_Y\_on\_AB(int a, int b, bool undirected)

{

В массив st будем хранить внутренние числа разделений точек на ребре (a, b):

int\* st = new int[g[a][b]];

for (int e = 0; e < g[a][b]; e++)

st[e] = 0;

//поиск на неориентированном ребре

if (undirected)

{

int T;

for (int e = 1; e < g[a][b]; e++)

{

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

{

T = v[i] \* (g[a][b] - e + d[i][a]);

if (T > st[e])

st[e] = T;

}

}

}

//поиск на ориентированном ребре

else

{

int T1; int T2;

for (int e = 1; e < g[a][b]; e++)

{

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

{

T1 = v[i] \* (e + d[b][i]);

T2 = v[i] \* (g[a][b] - e + d[a][i]);

if (T1 < T2)

{

if (T1 > st[e])

st[e] = T1;

}

else

if (T2 > st[e])

st[e] = T2;

}

}

}

return st;

}

Поиск локальных центров на ребер (a, b):

void Graph::Local\_centers(int a, int b, vector<center>& mins0, vector<center>& minst, vector<center>& mins)

{

int\* s0 = Find\_S0\_for\_all\_Y\_on\_AB(a, b, g[a][b] != g[b][a]); //внешние числа разделений ребра (a, b)

int\* st = Find\_St\_for\_all\_Y\_on\_AB(a, b, g[a][b] != g[b][a]); //внутренние числа разделений ребра (a, b)

center y;

y.a = a; y.b = b;

int min0 = INT\_MAX;

int mint = INT\_MAX;

int min = INT\_MAX;

Находим наименьшие числа разделений точек на ребре (a, b):

for (int e = 1; e < g[a][b]; e++)

{

if (s0[e] < min0)

min0 = s0[e];

if (st[e] < mint)

mint = st[e];

if ((s0[e] + st[e]) < min)

min = (s0[e] + st[e]);

}

Находим локальные центры на ребре (a, b):

for (int e = 1; e < g[a][b]; e++)

{

//Внешние центры

if (s0[e] == min0)

{

y.e = e;

y.s0 = s0[e];

y.st = st[e];

mins0.push\_back(y);

}

//Внутренние центры

if (st[e] == mint)

{

y.e = e;

y.s0 = s0[e];

y.st = st[e];

minst.push\_back(y);

}

//Двусторонние центры

if (s0[e] + st[e] == min)

{

y.e = e;

y.s0 = s0[e];

y.st = st[e];

mins.push\_back(y);

}

}

}

Поиск абсолютных центров на гранях, не включая вершины:

void Graph::Centers\_on\_edges(vector<center>& InAbsCenters, vector<center>& ExAbsCenters, vector<center>& AbsCenters)

{

vector<center> LocalInAbsCenters; //локальные внешние центры на ребрах

vector<center> LocalExAbsCenters; //локальные внутренние центры на ребрах

vector<center> LocalAbsCenters; //локальные дусторонние центры на ребрях

vector<center> Localmins0;

vector<center> Localminst;

vector<center> Localmin;

int mins0 = INT\_MAX;

int minst = INT\_MAX;

int min = INT\_MAX;

center y;

Получаем список всех локальных центров на ребрах графа:

for (int a = 0; a < m; a++)

for (int b = 0; b < m; b++)

if (g[a][b] < INT\_MAX)

if (a < b || g[a][b] != g[b][a])

{

Local\_centers(a, b, Localmins0, Localminst, Localmin);

if (Localmins0.front().s0 <= mins0)

mins0 = Localmins0.front().s0;

if (Localminst.front().st < minst)

minst = Localminst.front().st;

if ((Localmin.front().s0 + Localmin.front().st) < min)

min = Localmin.front().s0 + Localmin.front().st;

while (!Localmins0.empty())

{

y = Localmins0.back();

LocalInAbsCenters.push\_back(y);

Localmins0.pop\_back();

}

while (!Localminst.empty())

{

y = Localminst.back();

LocalExAbsCenters.push\_back(y);

Localminst.pop\_back();

}

while (!Localmin.empty())

{

y = Localmin.back();

LocalAbsCenters.push\_back(y);

Localmin.pop\_back();

}

}

Ищем наименьшие из локальных центров:

//внутренние

while (!LocalInAbsCenters.empty())

{

if (LocalInAbsCenters.back().s0 == mins0)

{

InAbsCenters.push\_back(LocalInAbsCenters.back());

}

LocalInAbsCenters.pop\_back();

}

//внешние

while (!LocalExAbsCenters.empty())

{

if (LocalExAbsCenters.back().st == minst)

{

ExAbsCenters.push\_back(LocalExAbsCenters.back());

}

LocalExAbsCenters.pop\_back();

}

//двусторонние

while (!LocalAbsCenters.empty())

{

if ((LocalAbsCenters.back().s0 + LocalAbsCenters.back().st) == min)

{

AbsCenters.push\_back(LocalAbsCenters.back());

}

LocalAbsCenters.pop\_back();

}

}

Поиск и вывод абсолютных центров:

void Graph::Abs\_centers()

{

vector<center> InAbsCenters;

vector<center> ExAbsCenters;

vector<center> AbsCenters;

vector<center\_on\_vertex> InCenters;

vector<center\_on\_vertex> ExCenters;

vector<center\_on\_vertex> Centers;

Centers\_on\_vertexes(InCenters, ExCenters, Centers); //центры среди вершин

Centers\_on\_edges(InAbsCenters, ExAbsCenters, AbsCenters); // центры среди граней

center\_on\_vertex x;

center y;

Ищем абсолютные центры: сравниваем центры на вершинах и гранях.

x = InCenters.front();

y = InAbsCenters.front();

if (x.s0 > y.s0)

InCenters.clear();

else

if (y.s0 > x.s0)

InAbsCenters.clear();

x = ExCenters.front();

y = ExAbsCenters.front();

if (x.s0 > y.s0)

ExCenters.clear();

else

if (y.s0 > x.s0)

ExAbsCenters.clear();

x = Centers.front();

y = AbsCenters.front();

if (x.s0 > y.s0)

Centers.clear();

else

if (y.s0 > x.s0)

AbsCenters.clear();

Вывод:

while (!InCenters.empty())

{

x = InCenters.back();

cout << "Internal Center:";

cout << x.a+1 << endl;

cout << "Internal radius:" << x.s0 << endl;

cout << "External radius:" << x.st << endl << endl;

InCenters.pop\_back();

}

while (!ExCenters.empty())

{

x = ExCenters.back();

cout << "External Center:";

cout << x.a+1 << endl;

cout << "Internal radius:" << x.s0 << endl;

cout << "External radius:" << x.st << endl << endl;

ExCenters.pop\_back();

}

while (!Centers.empty())

{

x = Centers.back();

cout << "Center:";

cout << x.a+1 << endl;

cout << "Internal radius:" << x.s0 << endl;

cout << "External radius:" << x.st << endl;

cout << "Diameter:" << x.s0 + x.st << endl << endl;

Centers.pop\_back();

}

while (!InAbsCenters.empty())

{

y = InAbsCenters.back();

cout << "Absolut Internal Center:";

cout << "(" << y.a + 1 << ", " << y.b + 1 << ") " << y.e << endl;

cout << "Internal radius:" << y.s0 << endl;

cout << "External radius:" << y.st << endl << endl;

InAbsCenters.pop\_back();

}

while (!ExAbsCenters.empty())

{

y = ExAbsCenters.back();

cout << "Absolut External Center:";

cout << "(" << y.a + 1 << ", " << y.b + 1 << ") " << y.e << endl;

cout << "Internal radius:" << y.s0 << endl;

cout << "External radius:" << y.st << endl << endl;

ExAbsCenters.pop\_back();

}

while (!AbsCenters.empty())

{

y = AbsCenters.back();

cout << "Absolut Center:";

cout << "(" << y.a + 1 << ", " << y.b + 1 << ") " << y.e << endl;

cout << "Internal radius:" << y.s0 << endl;

cout << "External radius:" << y.st << endl;

cout << "Diameter:" << y.s0 + y.st << endl << endl;

AbsCenters.pop\_back();

}

}

Главная функция будет выглядеть таким образом:

int main()

{

int m = 6;

Graph gr(m);

gr.InputTest1();

gr.PrintGraph();

gr.PrintDistances();

gr.PrintVertices();

gr.Abs\_centers();

return 0;

}

Результат работы программы для ориентированного графа:

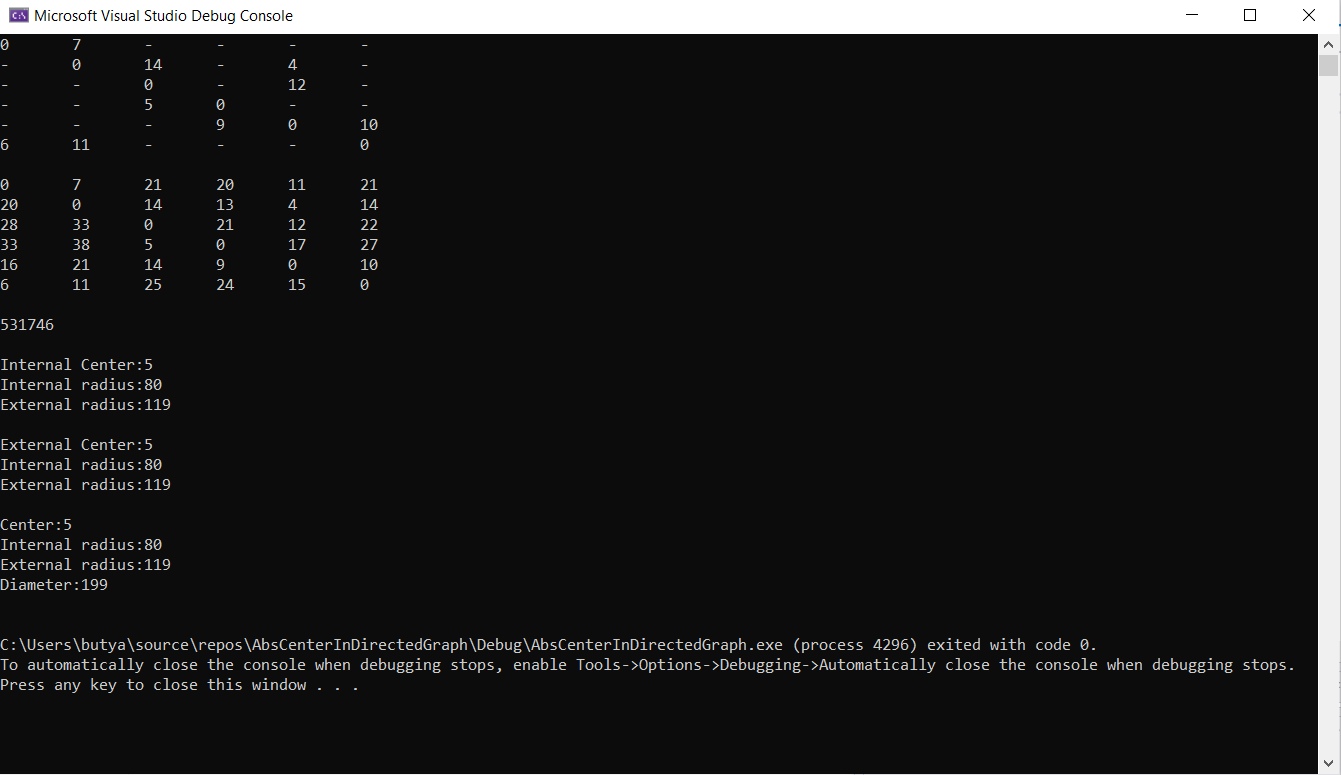


Рисунок 7 – Результат работы программы для ориентированного графа

Результат работы программы для неориентированного графа:

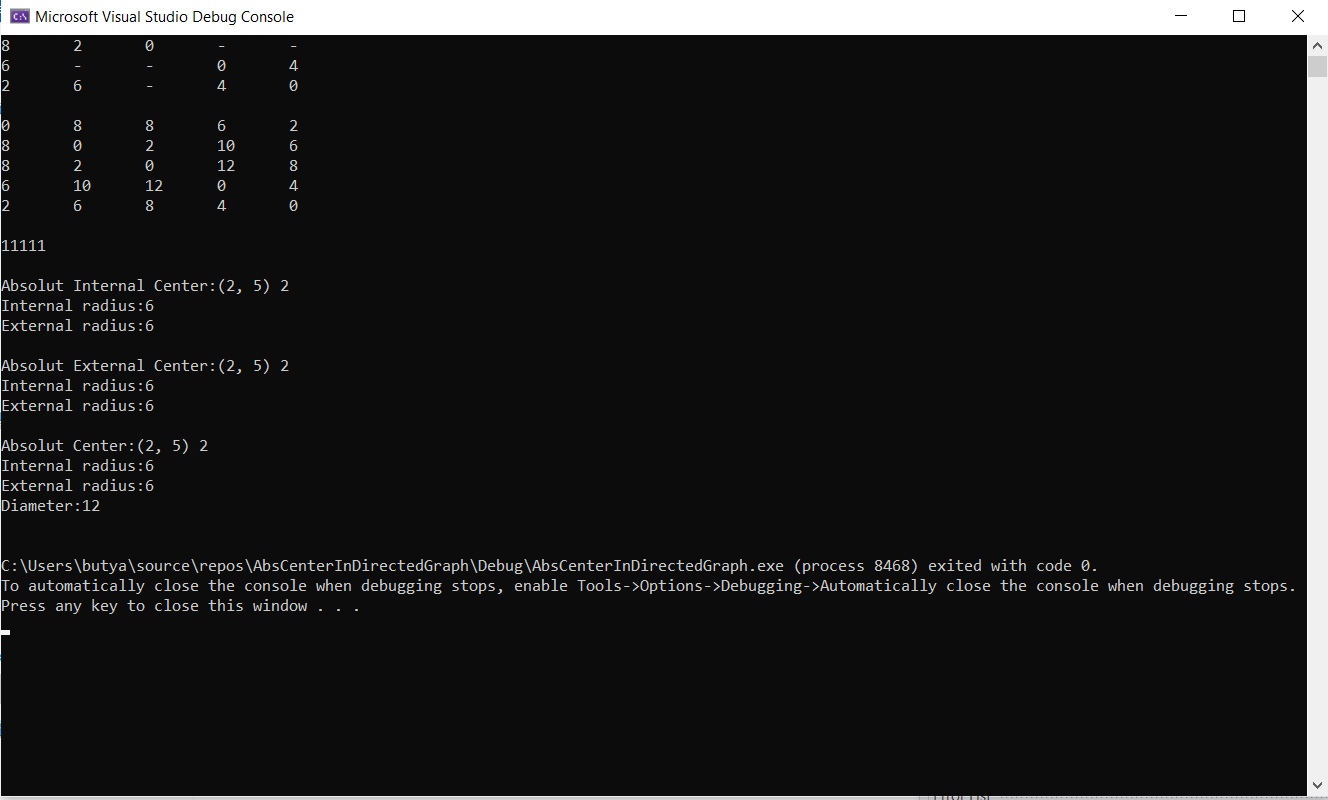


Рисунок 8 – Результат работы программы для неориентированного графа

# ЗАКЛЮЧЕНИЕ

Был выполнен проект по теме «Размещение центров в графах» с написанием программы на языке программирования C++.

С помощью справочников и учебников были изучены минимаксные задачи размещения и алгоритмы определения оптимального размещения центров в графах. Был проведён анализ алгоритмов для разных типов задач.

В результате проделанной работы мы можем искать центры среди вершин графа с помощью матрицы расстояний, а абсолютные центры методом Хакими. Эти методы используются в наших программах, которые решают практические задачи размещения специальных служб.

В перспективе возможно нахождение более эффективного решения поставленной задачи.

# СПИСОК ЛИТЕРАТУРЫ

1. Кристофидес Н. Теория графов. Алгоритмический подход / Кристофидес Н. – М.: Мир 1978. – 432 с.
2. Алексеев В. Е. Теория графов. Электронное учебно-методическое пособие / Алексеев В. Е., Захарова Д. В. – Нижний Новгород: Нижегородский госуниверситет, 2012. – 57 с.
3. Центр графа и его нахождение — CodeForces [Электронный ресурс]. —2020. — URL: [https://codeforces.com/blog/entry/17974?locale=ru](https://vk.com/away.php?to=https%3A%2F%2Fcodeforces.com%2Fblog%2Fentry%2F17974%3Flocale%3Dru&cc_key=) (даты обращения 16.12.2020).

# ПРИЛОЖЕНИЕ

Первая программа: поиск центров на вершинах по матрице расстояний.

#include<iostream>

#include<climits>

using namespace std;

struct center

{

int a;

int b;

int e;

int s0;

};

class Graph

{

int m;  
 int n;

int\* v;

int\*\* g;

int\*\* d;   
public:

Graph(int m);

Graph(const Graph& ob);

~Graph();

void InputVertices();

void Input();

void InputTest1();

void PrintGraph();

void PrintVertices();

void PrintDistances();

int GetM() { return m; }

int GetN() { return n; }

void ShortestPaths();

void CVertexes();

};

Graph::Graph(int m)

{

this->m = m;

n = 0;

v = new int[m];

g = new int\* [m];

d = new int\* [m];

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

{

v[i] = 1;

g[i] = new int[m];

d[i] = new int[m];

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

{

g[i][j] = (i == j) ? 0 : INT\_MAX;

d[i][j] = (i == j) ? 0 : INT\_MAX;

}

}

}

Graph::Graph(const Graph& ob)

{

m = ob.m;

n = ob.n;

v = new int[m];

g = new int\* [m];

d = new int\* [m];

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

{

v[i] = ob.v[i];

d[i] = new int[m];

g[i] = new int[m];

}

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

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

{

g[i][j] = ob.g[i][j];

d[i][j] = ob.d[i][j];

}

}

Graph::~Graph()

{

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

{

delete[] g[i];

delete[] d[i];

}

delete[] v;

delete[] g;

delete[] d;

}

void Graph::PrintGraph()

{

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

{

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

if (g[i][j] != INT\_MAX)

cout << g[i][j] << "\t";

else

cout << "-\t";

cout << endl;

}

cout << endl;

}

void Graph::PrintVertices()

{

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

cout << v[i];

cout << endl << endl;

}

void Graph::PrintDistances()

{

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

{

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

if (d[i][j] != INT\_MAX)

cout << d[i][j] << "\t";

else

cout << "-\t";

cout << endl;

}

cout << endl;

}

void Graph::InputVertices()

{

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

cin >> v[i];

cout << endl;

}

void Graph::Input()

{

cout << "Если ребро отсутствует, ввeдите -1" << endl;

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

{

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

{

cin >> g[i][j];

if (g[i][j] == -1)

g[i][j] = INT\_MAX;

if (g[i][j] != g[j][i])

n++;

}

cout << endl;

}

ShortestPaths();

}

void Graph::ShortestPaths()

{

int\*\* paths = new int\* [m];

int\*\* vertexes = new int\* [m];

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

{

paths[i] = new int[m];

vertexes[i] = new int[m];

}

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

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

{

paths[i][j] = g[i][j];

if (g[i][j] != INT\_MAX)

vertexes[i][j] = j;

else

vertexes[i][j] = -1;

}

for (int k = 0; k < m; k++)

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

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

if (i != j && j != k && i != k)

if (paths[i][k] != INT\_MAX && paths[k][j] != INT\_MAX)

if (paths[i][k] + paths[k][j] < paths[i][j])

{

paths[i][j] = paths[i][k] + paths[k][j];

vertexes[i][j] = vertexes[i][k];

}

d = paths;

}

void Graph::CVertexes()

{

int\* s = new int[m];

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

s[i] = 0;

int min = INT\_MAX;

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

{

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

{

if (d[i][j] > s[i])

{

s[i] = d[i][j];

}

}

if (s[i] < min)

min = s[i];

}

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

{

if (s[i] == min)

cout << "Center: " << i + 1 << endl << "Radius: " << s[i] << endl << endl;

}

}

void Graph::InputTest1()

{

g[0][1] = INT\_MAX;

g[0][2] = 8;

g[0][3] = 6;

g[0][4] = 2;

g[1][0] = INT\_MAX;

g[1][2] = 2;

g[1][3] = INT\_MAX;

g[1][4] = 6;

g[2][0] = 8;

g[2][1] = 2;

g[2][3] = INT\_MAX;

g[2][4] = INT\_MAX;

g[3][0] = 6;

g[3][1] = INT\_MAX;

g[3][2] = INT\_MAX;

g[3][4] = 4;

g[4][0] = 2;

g[4][1] = 6;

g[4][2] = INT\_MAX;

g[4][3] = 4;

n = 10;

ShortestPaths();

}

int main()

{

Graph gr(5);

gr.InputTest1();

gr.PrintGraph();

gr.PrintDistances();

gr.PrintVertices();

gr.CVertexes();

return 0;

}

Вторая программа: алгоритм Хакими для неориентированного графа.

#include<iostream>

#include<climits>

#include<vector>

using namespace std;

struct center

{

int a;

int b;

int e;

int s;

};

struct center\_on\_vertex

{

int a;

int s;

};

class Graph

{

int m;  
 int n;  
 int\* v;

int\*\* g;

int\*\* d;

public:

Graph(int m);

Graph(const Graph& ob);

~Graph();

void InputVertices();

void Input();

void InputTest1();

void PrintGraph();

void PrintDistances();

int GetM() { return m; }

int GetN() { return n; }

void ShortestPaths();

vector<center\_on\_vertex> Centers\_on\_vertexes();

int\* Find\_S\_for\_all\_Y\_on\_AB(int a, int b); vector<center> LocalCenters(int a, int b); vector<center> Centers\_on\_edges();   
 void AbsCenters();

};

Graph::Graph(int m)

{

this->m = m;

n = 0;

v = new int[m];

g = new int\* [m];

d = new int\* [m];

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

{

v[i] = 1;

g[i] = new int[m];

d[i] = new int[m];

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

{

g[i][j] = (i == j) ? 0 : INT\_MAX;

d[i][j] = (i == j) ? 0 : INT\_MAX;

}

}

}

Graph::Graph(const Graph& ob)

{

m = ob.m;

n = ob.n;

v = new int[m];

g = new int\* [m];

d = new int\* [m];

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

{

v[i] = ob.v[i];

d[i] = new int[m];

g[i] = new int[m];

}

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

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

{

g[i][j] = ob.g[i][j];

d[i][j] = ob.d[i][j];

}

}

Graph::~Graph()

{

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

{

delete[] g[i];

delete[] d[i];

}

delete[] v;

delete[] g;

delete[] d;

}

void Graph::PrintGraph()

{

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

{

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

if (g[i][j] != INT\_MAX)

cout << g[i][j] << "\t";

else

cout << "-\t";

cout << endl;

}

cout << endl;

}

void Graph::PrintDistances()

{

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

{

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

if (d[i][j] != INT\_MAX)

cout << d[i][j] << "\t";

else

cout << "-\t";

cout << endl;

}

cout << endl;

}

void Graph::InputVertices()

{

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

cin >> v[i];

cout << endl;

}

void Graph::Input()

{

cout << "Если ребро отсутствует, вводите 0" << endl;

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

{

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

{

cin >> g[i][j];

if (g[i][j] == 0 && i != j)

g[i][j] = INT\_MAX;

if (g[i][j] != g[j][i])

n++;

}

cout << endl;

}

ShortestPaths();

}

void Graph::ShortestPaths()

{

int\*\* paths = new int\* [m];

int\*\* vertexes = new int\* [m];

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

{

paths[i] = new int[m];

vertexes[i] = new int[m];

}

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

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

{

paths[i][j] = g[i][j];

if (g[i][j] != INT\_MAX)

vertexes[i][j] = j;

else

vertexes[i][j] = -1;

}

for (int k = 0; k < m; k++)

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

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

if (i != j && j != k && i != k)

if (paths[i][k] != INT\_MAX && paths[k][j] != INT\_MAX)

if (paths[i][k] + paths[k][j] < paths[i][j])

{

paths[i][j] = paths[i][k] + paths[k][j];

vertexes[i][j] = vertexes[i][k];

}

d = paths;

}

void Graph::InputTest1()

{

g[0][1] = INT\_MAX;

g[0][2] = 8;

g[0][3] = 6;

g[0][4] = 2;

g[1][0] = INT\_MAX;

g[1][2] = 2;

g[1][3] = INT\_MAX;

g[1][4] = 6;

g[2][0] = 8;

g[2][1] = 2;

g[2][3] = INT\_MAX;

g[2][4] = INT\_MAX;

g[3][0] = 6;

g[3][1] = INT\_MAX;

g[3][2] = INT\_MAX;

g[3][4] = 4;

g[4][0] = 2;

g[4][1] = 6;

g[4][2] = INT\_MAX;

g[4][3] = 4;

n = 10;

ShortestPaths();

}

vector<center\_on\_vertex> Graph::Centers\_on\_vertexes()

{

vector<center\_on\_vertex> Centers;

int\* s = new int[m];

for (int x = 0; x < m; x++)

s[x] = 0;

int T;

int mins = INT\_MAX;

for (int x = 0; x < m; x++)

{

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

{

T = v[i] \* d[x][i];

if (T > s[x])

s[x] = T;

}

if (s[x] < mins)

mins = s[x];

}

center\_on\_vertex c;

for (int x = 0; x < m; x++)

if (s[x] == mins)

{

c.a = x;

c.s = s[x];

Centers.push\_back(c);

}

return Centers;

}

int\* Graph::Find\_S\_for\_all\_Y\_on\_AB(int a, int b)

{

int\* s = new int[g[a][b]];

for (int e = 0; e < g[a][b]; e++)

s[e] = 0;

int T1; int T2;

for (int e = 1; e < g[a][b]; e++)

{

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

{

T1 = v[i] \* (e + d[b][i]);

T2 = v[i] \* (g[a][b] - e + d[a][i]);

if (T1 < T2)

{

if (T1 > s[e])

s[e] = T1;

}

else

if (T2 > s[e])

s[e] = T2;

}

}

return s;

}

vector<center> Graph::LocalCenters(int a, int b)

{

int\* s = Find\_S\_for\_all\_Y\_on\_AB(a, b);

int min = INT\_MAX;

center y;

y.a = a; y.b = b;

vector<center> ListOfY;

for (int e = 1; e < g[a][b]; e++)

if (s[e] < min)

min = s[e];

for (int e = 1; e < g[a][b]; e++)

if (s[e] == min)

{

y.e = e;

y.s = s[e];

ListOfY.push\_back(y);

}

return ListOfY;

}

vector<center> Graph::Centers\_on\_edges()

{

center y;

vector<center> CentersOnAB;

vector<center> Centers;

int min = INT\_MAX;

for (int a = 0; a < m; a++)

for (int b = m - 1; b > a; b--)

if (g[a][b] < INT\_MAX)

{

CentersOnAB = LocalCenters(a, b);

if (CentersOnAB.front().s < min)

min = CentersOnAB.front().s;

while (!CentersOnAB.empty())

{

y = CentersOnAB.back();

Centers.push\_back(y);

CentersOnAB.pop\_back();

}

}

vector<center> Abs;

int t = Centers.size();

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

if (Centers[i].s == min)

Abs.push\_back(Centers[i]);

return Abs;

}

void Graph::AbsCenters()

{

vector<center\_on\_vertex> Centers = Centers\_on\_vertexes(); // Центры на верщинах

vector<center> AbsCenters = Centers\_on\_edges();

center\_on\_vertex x;

center y;

x = Centers.front();

y = AbsCenters.front();

if (x.s > y.s)

Centers.clear();

else

if (y.s > x.s)

AbsCenters.clear();

while (!Centers.empty())

{

x = Centers.back();

cout << "Center:";

cout << x.a + 1 << endl;

cout << "Radius:" << x.s << endl;

Centers.pop\_back();

}

while (!AbsCenters.empty())

{

y = AbsCenters.back();

cout << "Absolut Center:";

cout << "(" << y.a + 1 << ", " << y.b + 1 << ") " << y.e << endl;

cout << "Radius:" << y.s << endl;

AbsCenters.pop\_back();

}

}

int main()

{

int m = 5;

Graph gr(m);

gr.InputTest1();

gr.PrintGraph();

gr.PrintDistances();

gr.AbsCenters();

return 0;

}

Третья программа: алгоритм Хакими для ориентированного и неориентированного графов.

#include<iostream>

#include<climits>

#include<vector>

using namespace std;

struct center

{

int a;

int b;

int e;

int s0;

int st;

};

struct center\_on\_vertex

{

int a;

int s0;

int st;

};

class Graph

{

int m;

int n;

int\* v;

int\*\* g;

int\*\* d;

public:

Graph(int m);

Graph(const Graph& ob);

~Graph();

void InputVertices();

void Input();

void InputTest0();  
 void InputTest1(); \

void PrintGraph();

void PrintDistances();

void PrintVertices();

int GetM() { return m; }

int GetN() { return n; }

void ShortestPaths();

void Centers\_on\_vertexes(vector<center\_on\_vertex>& InCenters, vector<center\_on\_vertex>& ExCenters, vector<center\_on\_vertex>& Centers);

int\* Find\_S0\_for\_all\_Y\_on\_AB(int a, int b, bool undirected);

int\* Find\_St\_for\_all\_Y\_on\_AB(int a, int b, bool undirected);

void Local\_centers(int a, int b, vector<center>& mins0, vector<center>& minst, vector<center>& min);

void Centers\_on\_edges(vector<center>& InAbsCenters, vector<center>& ExAbsCenters, vector<center>& AbsCenters);

void Abs\_centers();

};

Graph::Graph(int m)

{

this->m = m;

n = 0;

v = new int[m];

g = new int\* [m];

d = new int\* [m];

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

{

v[i] = 1;

g[i] = new int[m];

d[i] = new int[m];

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

{

g[i][j] = (i == j) ? 0 : INT\_MAX;

d[i][j] = (i == j) ? 0 : INT\_MAX;

}

}

}

Graph::Graph(const Graph& ob)

{

m = ob.m;

n = ob.n;

v = new int[m];

g = new int\* [m];

d = new int\* [m];

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

{

v[i] = ob.v[i];

d[i] = new int[m];

g[i] = new int[m];

}

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

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

{

g[i][j] = ob.g[i][j];

d[i][j] = ob.d[i][j];

}

}

Graph::~Graph()

{

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

{

delete[] g[i];

delete[] d[i];

}

delete[] v;

delete[] g;

delete[] d;

}

void Graph::PrintGraph()

{

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

{

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

if (g[i][j] != INT\_MAX)

cout << g[i][j] << "\t";

else

cout << "-\t";

cout << endl;

}

cout << endl;

}

void Graph::PrintDistances()

{

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

{

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

if (d[i][j] != INT\_MAX)

cout << d[i][j] << "\t";

else

cout << "-\t";

cout << endl;

}

cout << endl;

}

void Graph::PrintVertices()

{

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

cout << v[i];

cout << endl << endl;

}

void Graph::InputVertices()

{

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

cin >> v[i];

cout << endl;

}

void Graph::Input()

{

cout << "Если ребро отсутствует, вводите 0" << endl;

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

{

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

{

cin >> g[i][j];

if (g[i][j] == 0 && i != j)

g[i][j] = INT\_MAX;

if (g[i][j] != g[j][i])

n++;

}

cout << endl;

}

ShortestPaths();

}

void Graph::ShortestPaths()

{

int\*\* paths = new int\* [m];

int\*\* vertexes = new int\* [m];

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

{

paths[i] = new int[m];

vertexes[i] = new int[m];

}

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

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

{

paths[i][j] = g[i][j];

if (g[i][j] != INT\_MAX)

vertexes[i][j] = j;

else

vertexes[i][j] = -1;

}

for (int k = 0; k < m; k++)

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

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

if (i != j && j != k && i != k)

if (paths[i][k] != INT\_MAX && paths[k][j] != INT\_MAX)

if (paths[i][k] + paths[k][j] < paths[i][j])

{

paths[i][j] = paths[i][k] + paths[k][j];

vertexes[i][j] = vertexes[i][k];

}

d = paths;

}

void Graph::InputTest0()

{

g[0][1] = INT\_MAX;

g[0][2] = 8;

g[0][3] = 6;

g[0][4] = 2;

g[1][0] = INT\_MAX;

g[1][2] = 2;

g[1][3] = INT\_MAX;

g[1][4] = 6;

g[2][0] = 8;

g[2][1] = 2;

g[2][3] = INT\_MAX;

g[2][4] = INT\_MAX;

g[3][0] = 6;

g[3][1] = INT\_MAX;

g[3][2] = INT\_MAX;

g[3][4] = 4;

g[4][0] = 2;

g[4][1] = 6;

g[4][2] = INT\_MAX;

g[4][3] = 4;

n = 10;

ShortestPaths();

}

void Graph::InputTest1()

{

g[0][1] = 7;

g[1][2] = 14;

g[1][4] = 4;

g[2][4] = 12;

g[3][2] = 5;

g[4][3] = 9;

g[4][5] = 10;

g[5][0] = 6;

g[5][1] = 11;

v[0] = 5;

v[1] = 3;

v[2] = 1;

v[3] = 7;

v[4] = 4;

v[5] = 6;

n = 9;

ShortestPaths();

}

void Graph::Centers\_on\_vertexes(vector<center\_on\_vertex>& InCenters, vector<center\_on\_vertex>& ExCenters, vector<center\_on\_vertex>& Centers)

{

int\* s0 = new int[m];

int\* st = new int[m];

for (int x = 0; x < m; x++)

{

s0[x] = 0;

st[x] = 0;

}

int T;

int mins0 = INT\_MAX;

int minst = INT\_MAX;

int mins = INT\_MAX;

for (int x = 0; x < m; x++)

{

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

{

T = v[i] \* d[x][i];

if (T > s0[x])

s0[x] = T;

T = v[i] \* d[i][x];

if (T > st[x])

st[x] = T;

}

if (s0[x] < mins0)

mins0 = s0[x];

if (st[x] < minst)

minst = st[x];

if (s0[x]+st[x] < mins)

mins = s0[x]+st[x];

}

center\_on\_vertex c;

for (int x = 0; x < m; x++)

{

if (s0[x] == mins0)

{

c.a = x;

c.s0 = s0[x];

c.st = st[x];

InCenters.push\_back(c);

}

if (st[x] == minst)

{

c.a = x;

c.s0 = s0[x];

c.st = st[x];

ExCenters.push\_back(c);

}

if (s0[x]+st[x] == mins)

{

c.a = x;

c.s0 = s0[x];

c.st = st[x];

Centers.push\_back(c);

}

}

}

int\* Graph::Find\_S0\_for\_all\_Y\_on\_AB(int a, int b, bool undirected)

{

int\* s0 = new int[g[a][b]];

for (int e = 0; e < g[a][b]; e++)

s0[e] = 0;

if (undirected)

{

int T;

for (int e = 1; e < g[a][b]; e++)

{

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

{

T = v[i] \* (e + d[b][i]);

if (T > s0[e])

s0[e] = T;

}

}

}

else

{

int T1; int T2;

for (int e = 1; e < g[a][b]; e++)

{

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

{

T1 = v[i] \* (e + d[b][i]);

T2 = v[i] \* (g[a][b] - e + d[a][i]);

if (T1 < T2)

{

if (T1 > s0[e])

s0[e] = T1;

}

else

if (T2 > s0[e])

s0[e] = T2;

}

}

}

return s0;

}

int\* Graph::Find\_St\_for\_all\_Y\_on\_AB(int a, int b, bool undirected)

{

int\* st = new int[g[a][b]];

for (int e = 0; e < g[a][b]; e++)

st[e] = 0;

if (undirected)

{

int T;

for (int e = 1; e < g[a][b]; e++)

{

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

{

T = v[i] \* (g[a][b] - e + d[i][a]);

if (T > st[e])

st[e] = T;

}

}

}

else

{

int T1; int T2;

for (int e = 1; e < g[a][b]; e++)

{

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

{

T1 = v[i] \* (e + d[b][i]);

T2 = v[i] \* (g[a][b] - e + d[a][i]);

if (T1 < T2)

{

if (T1 > st[e])

st[e] = T1;

}

else

if (T2 > st[e])

st[e] = T2;

}

}

}

return st;

}

void Graph::Local\_centers(int a, int b, vector<center>& mins0, vector<center>& minst, vector<center>& mins)

{

int\* s0 = Find\_S0\_for\_all\_Y\_on\_AB(a, b, g[a][b] != g[b][a]); int\* st = Find\_St\_for\_all\_Y\_on\_AB(a, b, g[a][b] != g[b][a]);

center y;

y.a = a; y.b = b;

int min0 = INT\_MAX;

int mint = INT\_MAX;

int min = INT\_MAX;

for (int e = 1; e < g[a][b]; e++)

{

if (s0[e] < min0)

min0 = s0[e];

if (st[e] < mint)

mint = st[e];

if ((s0[e] + st[e]) < min)

min = (s0[e] + st[e]);

}

for (int e = 1; e < g[a][b]; e++)

{

if (s0[e] == min0)

{

y.e = e;

y.s0 = s0[e];

y.st = st[e];

mins0.push\_back(y);

}

if (st[e] == mint)

{

y.e = e;

y.s0 = s0[e];

y.st = st[e];

minst.push\_back(y);

}

if (s0[e] + st[e] == min)

{

y.e = e;

y.s0 = s0[e];

y.st = st[e];

mins.push\_back(y);

}

}

}

void Graph::Centers\_on\_edges(vector<center>& InAbsCenters, vector<center>& ExAbsCenters, vector<center>& AbsCenters)

{

vector<center> LocalInAbsCenters;  
 vector<center> LocalExAbsCenters;  
 vector<center> LocalAbsCenters;

vector<center> Localmins0;

vector<center> Localminst;

vector<center> Localmin;

int mins0 = INT\_MAX;

int minst = INT\_MAX;

int min = INT\_MAX;

center y;

for (int a = 0; a < m; a++)

for (int b = 0; b < m; b++)

if (g[a][b] < INT\_MAX)

if (a < b || g[a][b] != g[b][a])

{

Local\_centers(a, b, Localmins0, Localminst, Localmin);

if (Localmins0.front().s0 <= mins0)

mins0 = Localmins0.front().s0;

if (Localminst.front().st < minst)

minst = Localminst.front().st;

if ((Localmin.front().s0 + Localmin.front().st) < min)

min = Localmin.front().s0 + Localmin.front().st;

while (!Localmins0.empty())

{

y = Localmins0.back();

LocalInAbsCenters.push\_back(y);

Localmins0.pop\_back();

}

while (!Localminst.empty())

{

y = Localminst.back();

LocalExAbsCenters.push\_back(y);

Localminst.pop\_back();

}

while (!Localmin.empty())

{

y = Localmin.back();

LocalAbsCenters.push\_back(y);

Localmin.pop\_back();

}

}

while (!LocalInAbsCenters.empty())

{

if (LocalInAbsCenters.back().s0 == mins0)

{

InAbsCenters.push\_back(LocalInAbsCenters.back());

}

LocalInAbsCenters.pop\_back();

}

while (!LocalExAbsCenters.empty())

{

if (LocalExAbsCenters.back().st == minst)

{

ExAbsCenters.push\_back(LocalExAbsCenters.back());

}

LocalExAbsCenters.pop\_back();

}

while (!LocalAbsCenters.empty())

{

if ((LocalAbsCenters.back().s0 + LocalAbsCenters.back().st) == min)

{

AbsCenters.push\_back(LocalAbsCenters.back());

}

LocalAbsCenters.pop\_back();

}

}

void Graph::Abs\_centers()

{

vector<center> InAbsCenters;

vector<center> ExAbsCenters;

vector<center> AbsCenters;

vector<center\_on\_vertex> InCenters;

vector<center\_on\_vertex> ExCenters;

vector<center\_on\_vertex> Centers;

Centers\_on\_vertexes(InCenters, ExCenters, Centers); Centers\_on\_edges(InAbsCenters, ExAbsCenters, AbsCenters);

center\_on\_vertex x;

center y;

x = InCenters.front();

y = InAbsCenters.front();

if (x.s0 > y.s0)

InCenters.clear();

else

if (y.s0 > x.s0)

InAbsCenters.clear();

x = ExCenters.front();

y = ExAbsCenters.front();

if (x.s0 > y.s0)

ExCenters.clear();

else

if (y.s0 > x.s0)

ExAbsCenters.clear();

x = Centers.front();

y = AbsCenters.front();

if (x.s0 > y.s0)

Centers.clear();

else

if (y.s0 > x.s0)

AbsCenters.clear();

//Вывод

while (!InCenters.empty())

{

x = InCenters.back();

cout << "Internal Center:";

cout << x.a+1 << endl;

cout << "Internal radius:" << x.s0 << endl;

cout << "External radius:" << x.st << endl << endl;

InCenters.pop\_back();

}

while (!ExCenters.empty())

{

x = ExCenters.back();

cout << "External Center:";

cout << x.a+1 << endl;

cout << "Internal radius:" << x.s0 << endl;

cout << "External radius:" << x.st << endl << endl;

ExCenters.pop\_back();

}

while (!Centers.empty())

{

x = Centers.back();

cout << "Center:";

cout << x.a+1 << endl;

cout << "Internal radius:" << x.s0 << endl;

cout << "External radius:" << x.st << endl;

cout << "Diameter:" << x.s0 + x.st << endl << endl;

Centers.pop\_back();

}

while (!InAbsCenters.empty())

{

y = InAbsCenters.back();

cout << "Absolut Internal Center:";

cout << "(" << y.a + 1 << ", " << y.b + 1 << ") " << y.e << endl;

cout << "Internal radius:" << y.s0 << endl;

cout << "External radius:" << y.st << endl << endl;

InAbsCenters.pop\_back();

}

while (!ExAbsCenters.empty())

{

y = ExAbsCenters.back();

cout << "Absolut External Center:";

cout << "(" << y.a + 1 << ", " << y.b + 1 << ") " << y.e << endl;

cout << "Internal radius:" << y.s0 << endl;

cout << "External radius:" << y.st << endl << endl;

ExAbsCenters.pop\_back();

}

while (!AbsCenters.empty())

{

y = AbsCenters.back();

cout << "Absolut Center:";

cout << "(" << y.a + 1 << ", " << y.b + 1 << ") " << y.e << endl;

cout << "Internal radius:" << y.s0 << endl;

cout << "External radius:" << y.st << endl;

cout << "Diameter:" << y.s0 + y.st << endl << endl;

AbsCenters.pop\_back();

}

}

int main()

{

int m = 6;

Graph gr(m);

gr.InputTest1();

gr.PrintGraph();

gr.PrintDistances();

gr.PrintVertices();

gr.Abs\_centers();

return 0;

}