Министерство науки и высшего образования РФ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Курский государственный университет»

Кафедра программного

обеспечения и администрирования

информационных систем

Направление подготовки

математическое обеспечение и

администрирование

информационных систем

Форма обучения очная

**Отчет**

**по лабораторной работе №9**

«Реализация базовых алгоритмов на графах»

Выполнил:

студент группы 213 Тихонов Е.Е.

Проверил:

профессор кафедры ПОиАИС Кудинов В.А.

Курск, 2022

**Цель работы**: Изучить основные алгоритмы теории графов.

**Задание**

*Задача 1*. В вершинах неориентированного графа "хранятся" положительные целые числа. Подсчитайте, сколько из них являются простыми (совершенными, дружественными).

*Задача 2*. Определите, существуют ли в заданном неориентированном графе циклы.

*Задача 3*. Реализуйте алгоритм проверки связности графа, основанный на следующем результате, а именно: "граф G связен в том и только в том случае, если после применения к нему алгоритмов обхода графа в глубину или в ширину все вершины будут помечены".

**Алгоритм решения задач**

Для решения поставленных задач в программе создадим структуру для представления графа, называемая структурой Вирта.

Для решения первой задачи будем пользоваться методом обхода графа в глубину, при обходе очередной вершины будем получать данные содержащиеся в вершине и сохранять в массив, далее отмечаем вершину как пройденную и переходим к следующей вершине. После того как все вершины будут пройдены мы будем иметь массив с числами, хранящимися в вершинах графа. Далее проходимся по массиву и вычисляем количество простых, совершенных, дружественных чисел.

Для решения второй задачи введем понятие раскраски вершины в графе. Изначально все вершины графа будут раскрашены в белый цвет. Затем из каждой белой вершины, запустим поиск в глубину, который при входе в вершину будет красить её в серый цвет, а при выходе из нее — в чёрный. И, если алгоритм пытается пойти в серую вершину, то это означает, что цикл найден. Информацию о цветах вершин будем хранить в массиве, где каждый индекс соответствует вершине, а значение по индексу соответствует цвету. Цвета представим с помощью перечисления enum, где 0 - белый цвет, 1 - серый, 2 – черный.

Для решения третьей задачи выполним обход графа в глубину, где будем отмечать каждую пройденную вершину как посещенную, для этого будем использовать массив, где каждый индекс соответствует вершине, а значение по индексу является флагом, отвечающим посетили мы вершину или нет. Значение 0 по индексу вершины говорит, что вершина не была посещена, значение 1 –вершина была посещена. Затем после прекращения обхода графа в глубину пройдемся по массиву и если встретим значение 0, значит, соответствующая вершина не была посещена, следовательно, граф не связный. Если все значения в массиве равны 1, то граф связный.

**Код программы**

#include <iostream>

#include <string>

using namespace std;

class Node

{

public:

string Name; //имя вершины

int Id; //номер вершины

int Data; //данные

Node\*\* Neighbours; //соседние(смежные) вершины

int NeighboursCount;

int MaxNeighboursCount;

Node(string Name, int Id, int Data = 0, int MaxNeighboursCount = 10)

{

this->Name = Name;

this->Id = Id;

this->Data = Data;

this->Neighbours = new Node\*[MaxNeighboursCount];

this->NeighboursCount = 0;

this->MaxNeighboursCount = MaxNeighboursCount;

}

void AddNeighbor(Node\* Neighbor, bool oriented = false) // добавить смежную вершину

{

if (NeighboursCount == MaxNeighboursCount)

{

cout << "Для вершины " << this->Name << " достигнут лимит соседей" << endl;

}

else

{

Neighbours[NeighboursCount] = Neighbor;

NeighboursCount++;

if (oriented == false)

{

Neighbor->AddNeighbor(this, true);

}

}

}

};

bool IsSimple(int n) //проверка на простоту

{

if (n < 2)

return false;

for (int i = 2; i <= n/2; i++)

if (n % i == 0)

return false;

return true;

}

bool IsPerfect(int n) //проверка на совершенность (сумма делителей равна самому числу)

{

int sum = 0;

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

//Проверка на делимость без остатка

if (n % i == 0)

//Прибавляем делитель к суммае

sum += i;

//Проверка на равенство суммы делителей и первоначального числа

if (n == sum)

return true;

else

return false;

}

bool IsFriendly(int n1, int n2) //проверка на дружественность (сумма делителей первого числа равна второму числу и обратно)

{

int sum1 = 0;

for (int i = 1; i < n1; i++)

if (n1 % i == 0)

sum1 += i;

int sum2 = 0;

for (int i = 1; i < n2; i++)

if (n2 % i == 0)

sum2 += i;

if (sum1 == n2 && sum2 == n1)

return true;

else

return false;

}

void DataDepthSearch(Node\* FirstNode, int\* visited, int\* DataArray, int &DataCount)

{

DataArray[DataCount] = FirstNode->Data;

DataCount++;

visited[FirstNode->Id] = 1;

for (int i = 0; i < FirstNode->NeighboursCount; i++)

{

Node\* NextNode = FirstNode->Neighbours[i];

if (visited[NextNode->Id] == 0)

{

DataDepthSearch(NextNode, visited, DataArray, DataCount);

}

}

}

enum Color {white, grey, black};

void FindСycle(Node\* FirstNode, int\* colors, bool& HaveCycle, Node\* PrevNode = nullptr)

{

colors[FirstNode->Id] = grey;

for (int i = 0; i < FirstNode->NeighboursCount; i++)

{

Node\* NextNode = FirstNode->Neighbours[i];

if (colors[NextNode->Id] == white)

{

FindСycle(NextNode, colors, HaveCycle, FirstNode);

}

else if(colors[NextNode->Id] == grey && NextNode != PrevNode)

{

HaveCycle = true;

return;

}

colors[FirstNode->Id] = black;

}

}

void DepthSearch(Node\* FirstNode, bool\* visited)

{

visited[FirstNode->Id] = 1;

for (int i = 0; i < FirstNode->NeighboursCount; i++)

{

Node\* NextNode = FirstNode->Neighbours[i];

if (visited[NextNode->Id] == 0)

{

DepthSearch(NextNode, visited);

}

}

}

bool IsConnectivity(Node\* FirstNode, int Count)

{

bool\* visited = new bool[Count];

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

visited[i] = false;

DepthSearch(FirstNode, visited);

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

{

if (!visited[i])

{

return false;

}

}

return true;

}

int main()

{

setlocale(LC\_ALL, "Russian");

Node\* v1 = new Node("v1", 0, 13);

Node\* v2 = new Node("v2", 1, 7);

Node\* v3 = new Node("v3", 2, 6);

Node\* v4 = new Node("v4", 3, 220);

Node\* v5 = new Node("v5", 4, 284);

v1->AddNeighbor(v2);

v1->AddNeighbor(v4);

v2->AddNeighbor(v3);

v2->AddNeighbor(v4);

v3->AddNeighbor(v4);

v4->AddNeighbor(v5);

cout << "1. Обходы графов" << endl;

//4

int\* visited = new int[5]{0,0,0,0,0}; // посещенные вершины

int\* DataArray = new int[5]; // массив для хранения чисел из вершин

int DataCount = 0;

DataDepthSearch(v1, visited, DataArray, DataCount); //обход в глубину и сбор данных с графа в массив DataArray

//подсчет

int SIMPLE\_COUNT = 0;

int PERFECT\_COUNT = 0;

int FRIENDLY\_COUNT = 0;

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

{

if (IsSimple(DataArray[i]))

{

SIMPLE\_COUNT++;

}

if (IsPerfect(DataArray[i]))

{

PERFECT\_COUNT++;

}

for (int j = i+1; j < DataCount; j++)

{

if (IsFriendly(DataArray[i], DataArray[j]))

{

FRIENDLY\_COUNT+=2;

}

}

}

cout << "простых чисел " << SIMPLE\_COUNT << "\nсовершенных чисел " << PERFECT\_COUNT << "\nдружественных чисел " << FRIENDLY\_COUNT << endl;

cout << "2. Цепи и циклы неориентированного графа." << endl;

//2

//граф имеющий цикл(квадрат)

bool HaveCycle = false;

visited = new int[5]{ 0,0,0,0,0 };

FindСycle(v1, visited, HaveCycle);

if (HaveCycle)

{

cout << "имеет цикл"<< endl;

}

else

{

cout << "не имеет цикл" << endl;

}

//граф не имеющий цикл(ломаная)

Node\* n1 = new Node("n1", 0, 0);

Node\* n2 = new Node("n2", 1, 0);

Node\* n3 = new Node("n3", 2, 0);

n1->AddNeighbor(n2);

n1->AddNeighbor(n3);

HaveCycle = false;

int\* colors = new int[3]{ white, white, white };

FindСycle(n1, colors, HaveCycle);

if (HaveCycle)

{

cout << "имеет цикл" << endl;

}

else

{

cout << "не имеет цикл" << endl;

}

cout << "3. Связность" << endl;

//6

//проверка на связность обходом в глубину

//проверим предыдущие 2 графа(не ориентированные)

if (IsConnectivity(v1, 5))

{

cout << "граф связный" << endl;

}

else

{

cout << "граф не связный" << endl;

}

if (IsConnectivity(n1, 3))

{

cout << "граф связный" << endl;

}

else

{

cout << "граф не связный" << endl;

}

//проверим третий ориентированный граф

Node\* m1 = new Node("m1", 0);

Node\* m2 = new Node("m2", 1);

Node\* m3 = new Node("m3", 2);

Node\* m4 = new Node("m4", 3);

m1->AddNeighbor(m2, true);

m3->AddNeighbor(m2, true);

m4->AddNeighbor(m1, true);

m4->AddNeighbor(m3, true);

if (IsConnectivity(m1, 4))

{

cout << "граф связный" << endl;

}

else

{

cout << "граф не связный" << endl;

}

}

**Тест программы**

Тест программы представлен на рисунке 1. Внешний вид графов для тестирования программы представлен на рисунках 2-4.

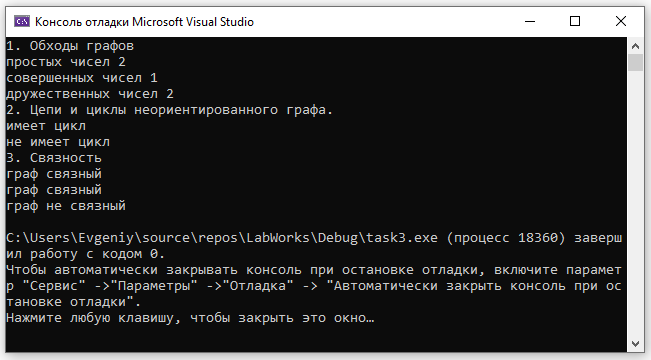


Рисунок 1 - Тест программы 1

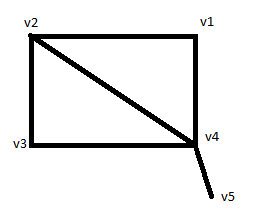


Рисунок 2 – Внешний вид первого неориентированного графа

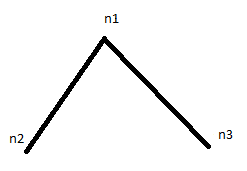


Рисунок 3 - Внешний вид второго неориентированного графа

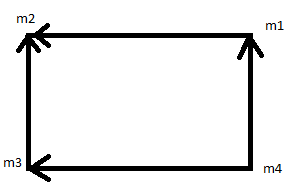


Рисунок 4 - Внешний вид третьего не связного графа