**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра МО ЭВМ**

**ОТЧЕТ**

**по лабораторной работе № 1**

**по дисциплине «Программирование»**

**Тема:**

**Рекурсия**

|  |  |  |
| --- | --- | --- |
| Студент гр. 6383 |  | Быков И. В. |
| Преподаватель |  | Шолохова О.М. |

Санкт-Петербург

2017

1. **Цель работы.**

Ознакомление с основными понятиями и приёмами рекурсивного программирования, получение навыков программирования рекурсивных функций на языке программирования С++.

1. **Постановка задачи.**

Задание:

1. выделение рекурсивно определяемых информационных объектов и (или) действий;
2. разработка программы, использующую рекурсию;

Вариант 3.

Имеется n городов, пронумерованных от 1 до n. Некоторые пары городов соединены дорогами. Определить, можно ли попасть по этим дорогам из одного заданного города в другой заданный город. Входная информация о дорогах задаётся в виде последовательности пар чисел i и j (i < j и i, j=1..n), указывающих, что i-й и j-й города соединены дорогами.

1. **Основные теоретические положения.**

Полученное задание можно перефразировать так:

Задан неориентированный граф (задан ребрами). Требуется узнать, находятся ли 2 введенные вершины в одной компоненте связности.

*Граф — абстрактный математический объект, представляющий собой множество вершин графа и набор рёбер, то есть соединений между парами вершин.*

*Матрица смежности графа G с конечным числом вершин n (пронумерованных числами от 1 до n) — это квадратная матрица A размера n, в которой значение элемента aij равно числу рёбер из i-й вершины графа в j-ю вершину.*

*Компонента связности графа G — максимальный (по включению) связный подграф графа G.*

Таким образом, задачу можно разбить на 2 части:

1. Создание матрицы смежности.

Матрица смежности есть двумерный булевый массив. Заполнение: попарно вводим номера вершин графа u и w и записываем ребро, соединяющее их в матрицу (элементы a[u][w] = true и a[w][u] = true)

1. Нахождение компонент связности и принадлежности вершин к компоненте.

Для решения потребуется завести булевый массив visited, в котором будет храниться информация о посещении рекурсивным алгоритмом вершины; целочисленный массив components, в котором будет указан номер компоненты для каждой вершины. Алгоритм: проходим по вершинам в цикле, если вершина не посещена (!visited[i]), то

2.1) Запускаем функцию DFS

2.2) Увеличиваем число компонент связности (num\_components++).

Функция DFS – рекурсивный поиск в глубину. Идея алгоритма: перебираем все исходящие из рассматриваемой вершины рёбра. Если ребро ведёт в вершину, которая не была рассмотрена ранее, то запускаем алгоритм от этой нерассмотренной вершины, а после возвращаемся и продолжаем перебирать рёбра. В моем случае, алгоритм (на вход подается вершина v):

2.1.1) components[v] = num\_components;

2.1.2) В visited отмечаем, что вершина пройдена (visited[v]=true)

2.1.3) Запускаем цикл по строкам матрицы инцидентности

2.1.3.1) Если у вершины v есть путь в j и при этом вершина j не пройдена, то запускаем dfs по вершине j.

В результате мы получим массив components, заполненный номерами компонент связности для каждой вершины (вершины в роли индексов массива). Для решения задачи осталось узнать, совпадают ли значения элементов массива components с заданными индексами (components[v] =? components[w])

1. **Спецификация программы.**

Программа предназначена для проверки наличия пути между двумя заданными вершинами в неориентированном графе.

Программа написана на языке C++ с использованием компилятора mingw32- g++. Входными данными для программы являются:

n – количество вершин (unsigned long long [0, 18 446 744 073 709 551 615]) Проверка проводится на неравенство нулю. При вводе отрицательного числа возникает необработанная ошибка.

p – количество путей (unsigned long long [0, 18 446 744 073 709 551 615]) Проверка проводится на неравенство нулю. При вводе отрицательного числа возникает необработанная ошибка.

пары вершин i и j (unsigned long long [0, 18 446 744 073 709 551 615])

Всего вершин вводится по условию n, проверка проводится на соответствие условию 1<=i < j <=n. При вводе отрицательного числа возникает необработанная ошибка.

1. **Пример диалога с пользователем.**

Введите количество городов:

16

Введите количество путей:

8

Введите номера городов:

1 2

2 3

3 6

4 5

6 7

8 9

9 11

11 15

Введите города, между которыми нужно узнать существование пути:

1 7

Матрица смежности:

0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0

0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0

0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0

0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0

0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0

0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0

0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Путь между городами 1 и 7 существует.

1. **Реализация.**

Функции:

1. bool find\_way(unsigned long long n, unsigned long long town\_i, unsigned long long town\_j, bool\*\* road);

*Функция, которая определяет, есть ли путь между двумя вершинами.*

*входные данные:*

*предусловие: n - число городов,*

*town\_i, town\_j - интересующие города*

*постусловие: road - матрица смежности*

*возвращает 1 или 0*

1. void dfs(unsigned long long n, unsigned long long top, unsigned long long\* components, unsigned long long& num\_components, bool\* visited, bool\*\* road);

*Рекурсивная функция "Поиск в глубину"*

*входные данные:*

*предусловие: n - число городов*

*top - текущая вершина*

*постусловие: components - указатель на массив номеров компонент связности для каждой вершины*

*num\_components - количество компонент связности вершины*

*visited - указатель на массив, в котром хранится информация о посещении вершин*

*road - двойной указатель на матрицу смежности*

1. Errors input\_and\_test\_town(unsigned long long n, bool\*\* road, std::istream& in);

*Функция ввода массива городов и интеграция их в матрицу смежности*

*входные данные:*

*предусловие: n - число городов*

*постусловие: road - указатель на булевый двумерный массив - матрица смежности*

*возвращает возможную ошибку при вводе городов ( 0 < town < n )*

1. Errors input\_and\_test\_n(unsigned long long& n, std::istream& in);

*Функция ввода количества городов*

*входные данные:*

*постусловие: n - число городов*

*возвращает возможную ошибку при вводе n (n > 0)*

1. Errors input\_and\_test\_my\_towns(unsigned long long n, unsigned long long& town, std::istream& in);

*Функция ввода городов, между котрыми требуется узнать путь*

*входные данные:*

*предусловие: n - число городов*

*постусловие: town - адрес на заданные города*

*возвращает возможную ошибку при вводе town (0 < town < n)*

1. Список возможных ошибок:

enum Errors

{

I\_IS\_NOT\_LESS\_THAN\_J, *// I >= J*

N\_IS\_NULL, *// N = 0*

INVALID\_VALUES\_IN\_ARRAY, *//неправильное значение в введенных городах*

INVALID\_VALUES\_IN\_MY\_TOWNS, *// неправильное значение в городах, между которыми ищем путь*

GOOD *// все ОК*

};

**Тестирование.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № |  | | Входные данные | | | Выходные данные (да – путь есть, нет – пути нет или ошибка) |
| n – число вершин (ull) | p – число путей(ull) | | i,j – пары вершин (ull) | w,v – вершины, между которыми определяем наличие пути |
| 1 | 0 | - | | - | - | Ошибка! n > 0 |
| 2 | 3 | 1 | | 2 1 | - | Ошибка! i < j |
| 3 | 6 | 3 | | 1 2  4 3  ---- | - | Ошибка! i < j |
| 4 | 6 | 3 | | 1 3  3 7  ---- | - | Ошибка! i,j <=n |
| 5 | 2 | 1 | | 1 2 | 3 1 | Ошибка! w <= n |
| 6 | 4 | 2 | | 1 3  2 4 | 1 2 | нет |
| 7 | 2 | 1 | | 1 2 | 1 2 | да |
| 8 | 8 | 4 | | 1 2  2 3  3 4  4 5 | 1 5 | да |
| 9 | 8 | 4 | | 1 2  2 3  3 4  4 5 | 5 1 | да |
| 10 | 10 | 5 | | 1 2  4 5  3 4  7 8  5 10 | 7 3 | нет |
| 11 | 50 | 25 | | 2 4  1 3  5 6  7 8  1 6  12 14  15 20  12 16  2 6  8 9  49 50  1 8  8 10  10 13  14 18  13 25  25 50  12 43  10 13  14 43  5 25  40 41  11 21  14 17  18 21 |  |  |

**Выводы.**

В ходе выполнения лабораторной работы были получены навыки реализации рекурсивных функций на языке программирования C++.

Так же я научился работать с неориентированными графами, а именно создавать матрицу инцидентности, зная информацию о ребрах графа; определять компоненты связности графа и находить путь между двумя вершинами с помощью рекурсивного алгоритма поиска в глубину. К недостаткам решения данной задачи требуется отнести, что при довольно большом количестве вершин требуется не рекурсивное решение задачи, что будет более оптимальным. Стоит заметить, что данный алгоритм так же может найти кратчайший путь, но только в невзвешенном графе. Для поиска кратчайшего пути в взвешенном графе рекомендуется использовать алгоритм Дейкстры.

**Приложение А. Исходный код.**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Быков И. В. 6383 гр, л/р 1, вариант 3.

Условие:

Имеется n городов, пронумерованных от 1 до n. Некоторые пары городов соединены дорогами.

Определить, можно ли попасть по этим дорогам из одного заданного города в другой заданный город.

Входная информация о дорогах задаётся в виде последовательности пар чисел i и j (i < j и i, j 1..n),

указывающих, что i-й и j-й города соединены дорогами.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include <fstream>

#include <cmath>

using namespace std;

enum Errors

{

N\_IS\_ODD\_NUMBER, // N - нечетное число

I\_IS\_NOT\_LESS\_THAN\_J, // I >= J

N\_IS\_NULL, // N = 0

TOWN\_IS\_GREATER\_THAN\_N, // введенный город > n

INVALID\_VALUES\_IN\_ARRAY, // неправильное значение в введенных городах

INVALID\_VALUES\_IN\_MY\_TOWNS, // неправильное значение в городах, между которыми ищем путь

GOOD // все ОК

};

/\* Функция, которая определяет, есть ли путь между двумя вершинами.

Для этого создаем массив в котором храним номер компоненты связности для каждой вершины. Затем достаточно узнать,

в одной ли компоненте связности находятся заданные вершины

входные данные:

предусловие: n - число городов

town\_i, town\_j - интересующие города

постусловие: road - матрица смежности

возвращает 1 или 0

\*/

bool find\_way(unsigned long long n, unsigned long long town\_i, unsigned long long town\_j, bool\*\* road);

/\* Рекурсивная функция "Поиск в глубину"

Идея: для каждой непройденной вершины находим все непройденные смежные вершины и повторяем писк для них

Сложность O(V+E)

входные данные:

предусловие: n - число городов

top - текущая вершина

постусловие: components - указатель на массив номеров компонент связности для каждой вершины

num\_components - количество компонент связности вершины

visited - указатель на массив, в котром хранится информация о посещении вершин

road - двойной указатель на матрицу смежности

ничего не возвращает

\*/

void dfs(unsigned long long n, unsigned long long top, unsigned long long\* components, unsigned long long& num\_components, bool\* visited, bool\*\* road);

/\* Функция ввода массива городов и интеграция их в матрицу смежности

входные данные:

предусловие: n - число городов

постусловие: road - указатель на булевый двумерный массив - матрица смежности

возвращает возможную ошибку при вводе городов ( 0 < town < n )

\*/

Errors input\_and\_test\_town(unsigned long long n, bool\*\* road, std::istream& in);

/\* Функция ввода количества городов

входные данные:

постусловие: n - число городов

возвращает возможную ошибку при вводе n (n > 0)

\*/

Errors input\_and\_test\_n(unsigned long long& n, std::istream& in);

/\* Функция ввода городов, между котрыми требуется узнать путь

входные данные:

предусловие: n - число городов

постусловие: town - адрес на заданные города

возвращает возможную ошибку при вводе town (0 < town < n)

\*/

Errors input\_and\_test\_my\_towns(unsigned long long n, unsigned long long& town, std::istream& in);

int main()

{

setlocale(LC\_ALL, "Russian");

unsigned long long n = 0; // количество городов

bool \*\*road; // матрица смежности

unsigned long long town\_j = 0; // город j

unsigned long long town\_i = 0; // город i

cout << "Введите количество городов: " << endl;

Errors res\_n = input\_and\_test\_n(n, cin);

if (res\_n == N\_IS\_ODD\_NUMBER)

{

cout << "Введенное количество городов НЕчетно. Нужно ввести четное количество городов." << endl;

return -1;

}

else if (res\_n == N\_IS\_NULL)

{

cout << "Вы ввели 0. Число городов должно быть положительным." << endl;

return -1;

}

road = new bool\*[n];

for (unsigned long long i = 0; i < n; i++)

road[i] = new bool[n];

for (unsigned long long i = 0; i < n; i++)

for (unsigned long long j = 0; j < n; j++)

road[i][j] = 0;

cout << "Введите номера городов: " << endl;

Errors res\_towns = input\_and\_test\_town(n, road, cin);

if (res\_towns == INVALID\_VALUES\_IN\_ARRAY)

{

cout << "Неверно введены даннные о городах (выход за пределы границ 1<=town<=n)" << endl;

return -1;

}

if (res\_towns == I\_IS\_NOT\_LESS\_THAN\_J)

{

cout << "В соседней паре i < j по условию. Ошибка!" << endl;

return -1;

}

cout << "Введите города, между которыми нужно узнать существование пути: " << endl;

Errors res\_town\_i = input\_and\_test\_my\_towns(n, town\_i, cin);

cout << endl;

if (res\_town\_i == INVALID\_VALUES\_IN\_MY\_TOWNS)

{

cout << "Cтартовый город указан неверно." << endl;

return -1;

}

Errors res\_town\_j = input\_and\_test\_my\_towns(n, town\_j, cin);

if (res\_town\_j == INVALID\_VALUES\_IN\_MY\_TOWNS)

{

cout << "Конечный город указан неверно." << endl;

return -1;

}

cout << "Матрица смежности: \n";

for (unsigned long long i = 0; i < n; i++)

{

for (unsigned long long j = 0; j < n; j++)

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

cout << endl;

}

if(find\_way(n, town\_i, town\_j, road))

cout << "\nПуть между городами " << town\_i + 1 << " и " << town\_j + 1 << " существует." << endl;

else

cout << "\nПуть между городами " << town\_i + 1 << " и " << town\_j + 1 << " НЕ существует." << endl;

for (unsigned long long i = 0; i < n; i++) delete [] road[i];

delete [] road;

road = NULL;

return 0;

}

Errors input\_and\_test\_town(unsigned long long n, bool\*\* road, std::istream& in)

{

unsigned long long town\_i, town\_j, k = 2;

while (k <= n)

{

in >> town\_i;

in >> town\_j;

if (town\_i >= town\_j)return I\_IS\_NOT\_LESS\_THAN\_J;

if ((town\_i < 1) || (town\_i > n) || (town\_j < 1) || (town\_j > n))

return INVALID\_VALUES\_IN\_ARRAY;

road[town\_i - 1][town\_j - 1] = true;

road[town\_j - 1][town\_i - 1] = true;

k += 2;}

return GOOD;}

bool find\_way(unsigned long long n, unsigned long long town\_i, unsigned long long town\_j, bool\*\* road)

{

bool \*visited;

unsigned long long \*components;

unsigned long long num\_components = 0;

components = new unsigned long long[n];

visited = new bool[n];

for (unsigned long long i = 0; i < n; i++)

{

visited[i] = false;

components[i] = 0;}

for (unsigned long long i = 0; i < n; i++)

if (!visited[i])

{

dfs(n, i, components, num\_components, visited, road);

num\_components++;}

for (unsigned long long i = 0; i < n; i++)

if (components[town\_i] == components[town\_j]) return true;

else return false;}

void dfs(unsigned long long n,unsigned long long top, unsigned long long\* components, unsigned long long& num\_components, bool\* visited, bool\*\* road)

{

components[top] = num\_components;

visited[top] = true;

for (unsigned long long j = 0; j < n; j++)

if (road[top][j] != 0)

if (!visited[j])

dfs(n, j, components, num\_components, visited, road);}

Errors input\_and\_test\_n(unsigned long long& n, std::istream& in)

{

in >> n;

if (n <= 0) return N\_IS\_NULL;

if (n % 2 != 0) return N\_IS\_ODD\_NUMBER;

return GOOD;}

Errors input\_and\_test\_my\_towns(unsigned long long n, unsigned long long& town, std::istream& in)

{

in >> town;

if (town < 1 || town > n)

return INVALID\_VALUES\_IN\_MY\_TOWNS;

town--;

return GOOD;}