Министерство образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

**Выполнил:**

Студенты группы 19ВВ2:

Ильин С.

Пронин В.

Дерябина А.

**Приняли:**

Митрохин М.А.

Юрова О.В.

**ОТЧЕТ**

по лабораторной работе №6

по курсу «Логика и основы алгоритмизации в инженерных задачах»

на тему «Поиск расстояний в графе»

Пенза 2020

### **Название:**

Поиск расстояний в графе.

### **Задание:**

### **Задание 1**

1. Сгенерируйте (используя генератор случайных чисел) матрицу смежности для неориентированного графа *G*. Выведите матрицу на экран.
2. Для сгенерированного графа осуществите процедуру поиска расстояний, реализованную в соответствии с приведенным выше описанием. При реализации алгоритма в качестве очереди используйте класс **queue** из стандартной библиотеки С++.

**3.**\* Реализуйте процедуру поиска расстояний для графа, представленного списками смежности.

### **Задание 2\***

1. Реализуйте процедуру поиска расстояний на основе обхода в глубину.
2. Реализуйте процедуру поиска расстояний на основе обхода в глубину для графа, представленного списками смежности.
3. Оцените время работы реализаций алгоритмов поиска расстояний на основе обхода в глубину и обхода в ширину для графов разных порядков.

### **Цель работы:**

Разработать программы по данным заданиям.

**Общие сведения.**

Поиск расстояний – довольно распространенная задача анализа графов.

Для поиска расстояний можно использовать процедуры обхода графа. Для этого при каждом переходе в новую вершину необходимо запоминать, сколько шагов до нее мы сделали. При этом вектор, который хранил информацию о посещении вершин становится вектором расстояний. Довольно просто модернизировать для поиска расстояний в графе алгоритм обхода в ширину, т.к. этот алгоритм проходит вершины по уровням удаленности, то для не ориентированного графа для вершин каждого следующего уровня глубины расстояние от исходной вершины увеличивается на 1. Удалённость в данном случае понимается как количество ребер, по которым необходимо прейти до достижения вершины.



Рисунок 1 – Граф

Таким образом, можно предложить следующую реализацию алгоритма обхода в ширину.

**Вход**: G – матрица смежности графа, v – исходная вершина.

**Выход**: DIST – вектор расстояний до всех вершин от исходной.

**Алгоритм ПОШ**

1.1. для всех i положим DIST [i] = -1 пометим как "не посещенную";

1.2. **ВЫПОЛНЯТЬ** BFSD (v).

1.3 для всех i вывести DIST [i] на экран;

**Алгоритм** BFSD(v):

2.1. Создать пустую очередь Q = {};

2.2. Поместить v в очередь Q.push(v);

2.3. Обновить вектор расстояний DIST [ x ] = 0;

2.4. **ПОКА**  Q != ∅ очередь не пуста **ВЫПОЛНЯТЬ**

2.5. v = Q.front() установить текущую вершину;

2.6. Удалить первый элемент из очереди Q.pop();

2.7. вывести на экран v;

2.8. **ДЛЯ** i = 1 **ДО** size\_G **ВЫПОЛНЯТЬ**

2.9. **ЕСЛИ** G(v,i) = = 1**И** DIST = = -1

2.10. **ТО**

2.11. Поместить i в очередь Q.push(i);

2.12. Обновить вектор расстояний DIST [ i ] = DIST [ v ] + 1;

Реализация состоит из подготовительной части, в которой все вершины помечаются как не поcещенные (п.1.1). В отличие от алгоритма BFS не посещенные вершины помечаем -1, т.к. значение 0 и 1 могут быть расстояниями. Расстояние 0 – от исходной вершины до самой себя.

В самой процедуре как и в алгоритме BFS сначала создается пустая очередь (п. 2.1), в которую помещается исходная вершина, из которой начат обход (п.2.2). Расстояние до этой вершины (п.2.3) устанавливается равным 0 (расстояние до самой себя).

Далее итерационно, пока очередь не опустеет, из нее извлекается первый элемент, который становится текущей вершиной (п. 2.5, 2.6). Затем в цикле просматривается **v**-я строка матрицы смежности графа G(v,i). Как только алгоритм встречает смежную с **v** не посещенную вершину (п.2.9), эта вершина помещается в очередь (п.2.11) и для нее обновляется вектор расстояния (п.2.12). Расстояние до новой **i**-й вершины вычисляется как расстояние до текущей **v**-й вершины плюс 1 (так как ребра нашего графа не взвешенные).

После просмотра строки матрицы смежности алгоритм делает следующую итерацию цикла 2.4 или заканчивает работу, если очередь пуста.

Таким образом, если вершина помещается в очередь при просмотре сроки матрицы смежности на 1-й итерации, то они находятся на 1 уровне удаленности и расстояние до этих вершин будет равным 1.

DIST [ i ] = DIST [ v ] + 1, где DIST [ v ] = 0 – расстояние от исходной вершины до самой себя.

Далее, начинают просматриваться вершины первого уровня и соответствующие им строки матрицы смежности. При добавлении смежных с вершинами первого уровня вершин, расстояния до них будут равны 2.

DIST [ i ] = DIST [ v ] + 1, где DIST [ v ] = 1 – расстояние от исходной вершины до вершин 1 уровня.

После того, как все вершины первого уровня будут просмотрены и извлечены из очереди, начнется просмотр вершин 2 уровня и соответствующих им строк матрицы смежности. При добавлении смежных с вершинами второго уровня вершин, расстояния до них будут равны 3.

DIST [ i ] = DIST [ v ] + 1, где DIST [ v ] = 2 – расстояние от исходной вершины до вершин 2 уровня.

И так далее, алгоритм проходит вершины по уровням, пока очередь не опустеет.

### **Описание метода решения задачи:**

### **Задание 1**

Сгенерировали (используя генератор случайных чисел) матрицу смежности для неориентированного графа *G*. Вывели матрицу на экран.

Для сгенерированного графа осуществили процедуру поиска расстояний, реализованную в соответствии с приведенным выше описанием. При реализации алгоритма в качестве очереди использовали класс **queue** из стандартной библиотеки С++.

Реализовали процедуру поиска расстояний для графа, представленного списками смежности.

### **Задание 2\***

Реализовали процедуру поиска расстояний на основе обхода в глубину.

Реализовали процедуру поиска расстояний на основе обхода в глубину для графа, представленного списками смежности.

Оценили время работы реализаций алгоритмов поиска расстояний на основе обхода в глубину и обхода в ширину для графов разных порядков.

### **Листинг:**

#include <stdlib.h>

#include <ctime>

#include <locale.h>

#include <queue>

#include <iostream>

#include <iomanip>

#include <windows.h>

using namespace std;

typedef struct Node {

int ver;

struct Node\* next;

} Node;

void push(Node\*\* head, int data)

{

Node\* tmp = (Node\*)malloc(sizeof(Node));

tmp->ver = data;

tmp->next = (\*head);

(\*head) = tmp;

}

int pop(Node\*\* head) {

Node\* prev = NULL;

int val;

if (head == NULL) {

exit(-1);

}

prev = (\*head);

val = prev->ver;

(\*head) = (\*head)->next;

free(prev);

return val;

}

Node\* getLast(Node\* head) {

if (head == NULL) {

return NULL;

}

while (head->next) {

head = head->next;

}

return head;

}

void pushBack(Node\* head, int value) {

Node\* last = getLast(head);

Node\* tmp = (Node\*)malloc(sizeof(Node));

tmp->ver = value;

tmp->next = NULL;

last->next = tmp;

}

void initializinglist(int\*\* a, Node\*\* head, int n)

{

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

{

push(&head[i], i);

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

{

if (a[i][j] == 1)

{

pushBack(head[i], j);

}

}

}

}

void printLinkedList(const Node \* head) {

if (head) {

cout << head->ver + 1;

head = head->next;

}

while (head) {

cout << "--->" << head->ver + 1;

head = head->next;

}

printf("\n");

}

void BFS(int\*\* a, int\* visited, int n, int v)

{

queue <int> q;

q.push(v);

visited[v] = 0;

while (!q.empty())

{

v = q.front();

q.pop();

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

{

if (a[v][i] == 1 && visited[i] == -1)

{

q.push(i);

visited[i] = visited[v] + 1;

}

}

}

}

void BFSlist(Node \* \*head, int\* visited, int v)

{

Node\*\* tmp = head;

queue <int> q;

q.push(tmp[v]->ver);

visited[tmp[v]->ver] = 0;

while (!q.empty())

{

v = q.front();

q.pop();

while (tmp[v])

{

if (visited[tmp[v]->ver] == -1)

{

q.push(tmp[v]->ver);

visited[tmp[v]->ver] = visited[v] + 1;

}

tmp[v] = tmp[v]->next;

}

}

}

void DFS(int\*\* a, int\* visited, int n, int v, const int count)

{

visited[v] = count;

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

{

if (a[v][i] == 1 && visited[i] == -1)

{

DFS(a, visited, n, i, count + 1);

}

if (a[v][i] == 1 && (visited[i] > count + 1))

{

DFS(a, visited, n, i, count + 1);

}

}

}

void DFSlist(Node\*\* head, int\* visited, int v, const int count)

{

visited[v] = count;

Node\* tmp = head[v];

while (tmp)

{

if (visited[tmp->ver] == -1)

DFSlist(head, visited, tmp->ver, count + 1);

if (visited[tmp->ver] > count + 1)

DFSlist(head, visited, tmp->ver, count + 1);

tmp = tmp->next;

}

}

void main()

{

setlocale(LC\_ALL, "RUS");

srand(time(NULL));

LARGE\_INTEGER start, end, f;

queue <int> q;

int\*\* a, n = 9, num, v;

Node\*\* head = (Node\*\*)malloc(n \* sizeof(Node\*));

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

head[i] = NULL;

int\* visited = (int\*)malloc(n \* sizeof(int));

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

visited[i] = -1;

a = (int\*\*)malloc(n \* sizeof(int\*));

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

{

a[i] = (int\*)malloc(n \* sizeof(int));

}

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

{

for (int j = n - 1; j >= 0; j--)

{

a[j][i] = a[i][j] = rand() % 2;

}

}

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

{

cout << "\n\n";

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

{

cout << a[i][j] << setw(5);

}

}

cout << "\n\nВыберите вершину с которой начинать поиск растояний, используя обход в ширину: ";

cin >> v;

v--;

QueryPerformanceCounter(&start);

BFS(a, visited, n, v);

QueryPerformanceCounter(&end);

QueryPerformanceFrequency(&f);

double sec = double(end.QuadPart - start.QuadPart) / f.QuadPart;

cout << "\nРасстояния от вешины "<< v + 1 << " до всех остальныйх используя обход в ширину на матрице смежности:\n";

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

{

cout << i + 1 << " - " << visited[i] << endl;

}

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

visited[i] = -1;

initializinglist(a, head, n);

QueryPerformanceCounter(&start);

BFSlist(head, visited, v);

QueryPerformanceCounter(&end);

double sec1 = double(end.QuadPart - start.QuadPart) / f.QuadPart;

cout << "\nРасстояния от вешины " << v + 1 << " до всех остальныйх используя обход в ширину на списках смежности::\n";

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

{

cout << i + 1 << " - " << visited[i] << endl;

}

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

visited[i] = -1;

const int count = 0;

QueryPerformanceCounter(&start);

DFS(a, visited, n, v, count);

QueryPerformanceCounter(&end);

double sec2 = double(end.QuadPart - start.QuadPart) / f.QuadPart;

cout << "\nРасстояния от вешины " << v + 1 << " до всех остальныйх используя обход в глубину на матрице смежности:\n";

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

{

cout << i + 1 << " - " << visited[i] << endl;

}

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

head[i] = NULL;

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

visited[i] = -1;

initializinglist(a, head, n);

QueryPerformanceCounter(&start);

DFSlist(head, visited, v, count);

QueryPerformanceCounter(&end);

double sec3 = double(end.QuadPart - start.QuadPart) / f.QuadPart;

cout << "\nРасстояния от вешины " << v + 1 << " до всех остальныйх используя обход в глубину на списках смежности:\n";

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

{

cout << i + 1 << " - " << visited[i] << endl;

}

cout << "Время работы поиска расстояний в графк с помощью обхода в ширину на матрицах смежности: " << sec << endl;

cout << "Время работы поиска расстояний в графк с помощью обхода в ширину на списках смежности: " << sec1 << endl;

cout << "Время работы поиска расстояний в графк с помощью обхода в глубину на матрицах смежности: " << sec2 << endl;

cout << "Время работы поиска расстояний в графк с помощью обхода в глубину на списках смежности: " << sec3 << endl;

}

### **Результаты работы программы:**

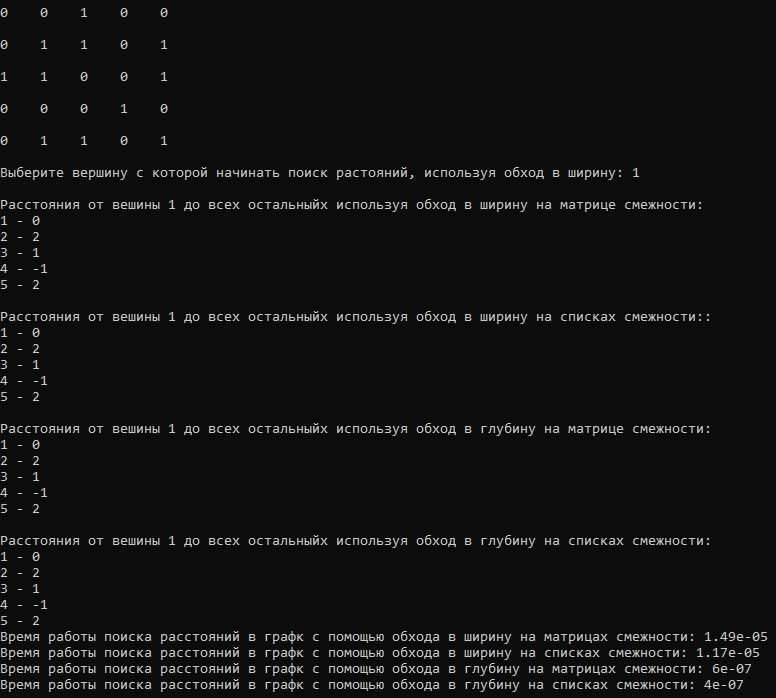


Рисунок 1 Результат работы программы

### **Вывод:**

В ходе выполнения лабораторной работы мы научились осуществлять поиск расстояний в графе. Были разработаны программы, соответствующие данным заданиям.