Министерство образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

Пенза 2020

**ОТЧЕТ**

по лабораторной работе №6

по курсу «Логика и основы алгоритмизации в инженерных задачах»

на тему «Поиск расстояний в графе»

**Выполнил:**

Студенты группы 19ВВ2:

Юдин Д.

Анохин А.

Вожжов И.

**Приняли:**

Митрохин М.А.

Юрова О.В.

### **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:**

Написать функции и разработать программу по заданию.

**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.

### **Листинг:**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <conio.h>

#include <ctime>

#include <locale.h>

#include <queue>

#include <chrono>

using namespace std;

class Timer

{

private:

using clock\_t = std::chrono::high\_resolution\_clock;

using second\_t = std::chrono::duration<double, std::ratio<1> >;

std::chrono::time\_point<clock\_t> m\_beg;

public:

Timer() : m\_beg(clock\_t::now())

{

}

void reset()

{

m\_beg = clock\_t::now();

}

double elapsed() const

{

return std::chrono::duration\_cast<second\_t>(clock\_t::now() - m\_beg).count();

}

};

typedef struct Node {

int vertex;

struct Node\* next;

} Node;

void add\_Node\_to\_beginning(Node\*\* head, int data) //функция добавления узла в начало

{

Node\* tmp\_node = (Node\*)malloc(sizeof(Node)); // создаем новый узел

tmp\_node->vertex = data;

tmp\_node->next = (\*head); //присваиваем указателю tmp адрес след. узла

(\*head) = tmp\_node; //Присваиваем указателю head адрес tmp

} //после выхода из функции tmp уничтожается

int delete\_node(Node\*\* head) { //удаляем узел на который указывает адрес

Node\* prev = NULL;

int val;

prev = (\*head); //передаем во временную переменную prev адрес

val = prev->vertex;

(\*head) = (\*head)->next; //в head кладем адрес следующего узла

free(prev); //очищаем временную переменную prev

return val;

}

Node\* findLastHead(Node\* head) { //поиск адреса последнего элемента

if (head == NULL) {

return NULL;

}

while (head->next) {

head = head->next;

}

return head;

}

void add\_Node\_to\_end(Node\* head, int data) { //функция добавления нового узла в конец

Node\* last = findLastHead(head); //получаем указатель на последний элемент списка

Node\* tmp\_node = (Node\*)malloc(sizeof(Node)); // создаем новый узел

tmp\_node->vertex = data;

tmp\_node->next = NULL;

last->next = tmp\_node; //записываем в последний элемент списка указатель на новый узел

}

void createLinkedList(int\*\* mas, Node\*\* head, int n) //создаем связный список

{

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

{

add\_Node\_to\_beginning(&head[i], i); //передаем адрес вершины и ее номер

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

{

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

{

add\_Node\_to\_end(head[i], j); //добавляем к вершине связные с ней вершины

}

}

}

}

void printLinkedList(const Node\* head) {

printf("\n");

while (head) {

printf("%d -->", head->vertex + 1);

head = head->next;

}

printf("\n");

}

void BFS(int\*\* mas, int\* visited, int N, int v) //n-кол-во вершин, v- текущая вершина

{

queue <int> queue;

queue.push(v);

visited[v] = 0;

while (!queue.empty())

{

v = queue.front();

queue.pop();

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

{

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

{

queue.push(i);

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

}

}

}

}

void BFS\_ForLinkedList(Node\*\* head, int\* visited, int v)

{

Node\*\* tmp\_node = head;

queue <int> queue;

queue.push(tmp\_node[v]->vertex);

visited[tmp\_node[v]->vertex] = 0;

while (!queue.empty())

{

v = queue.front();

queue.pop();

while (tmp\_node[v])

{

if (visited[tmp\_node[v]->vertex] == -1)

{

queue.push(tmp\_node[v]->vertex);

visited[tmp\_node[v]->vertex] = visited[v] + 1;

}

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

}

}

}

void DFS(int\*\* mas, int\* visited, int N, int v, int distance)

{

visited[v] = distance;

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

{

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

{

DFS(mas, visited, N, i, distance + 1); //если для вершины есть путь вызываем функцию с count+1

}

if (mas[v][i] == 1 && (visited[i] > distance + 1))//если для вершины есть более ближний путь, чем тот, который уже найдет, вызывается это условие

{

DFS(mas, visited, N, i, distance + 1);

}

}

}

void DFS\_ForLinkedList(Node\*\* head, int\* visited, int v, int distance)

{

visited[v] = distance;

Node\* tmp = head[v];

while (tmp)

{

if (visited[tmp->vertex] == -1) {

DFS\_ForLinkedList(head, visited, tmp->vertex, distance + 1);

}

if (visited[tmp->vertex] > distance + 1){

DFS\_ForLinkedList(head, visited, tmp->vertex, distance + 1);

}

tmp = tmp->next;

}

}

void main()

{

setlocale(LC\_ALL, "RUS");

srand(time(NULL));

queue <int> queue;

int\*\* mas, N, num; //N-кол-во вершин, num-вершина для ввода

time\_t start, end;

printf("Укажите размер матрицы N\*N: ");

scanf("%d", &N);

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;

}

mas = (int\*\*)malloc(N \* sizeof(int\*));

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

{

mas[i] = (int\*)malloc(N \* sizeof(int));

}

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

{

mas[i][i] = 0;

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

{

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

mas[j][i] = mas[i][j];

}

}

printf("\n Вывод матрицы смежности\n ");

printf("\n ");

printf("\t");

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

printf("%d", i + 1);

}

for (int i = 0; i < N; i++) { // вывод матриц смежности

printf("\n");

printf("%d\t", i + 1);

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

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

}

}

printf("\n\nВведите вершину для начала обхода графа в глубину : ");

scanf("%d", &num);

num--;

time(&start);

Timer t;

BFS(mas, visited, N, num);

time(&end);

printf("\nРасстояние от вершины %d до других вершин, найденное с помощью обхода в ширину на матрице смежности:\n", num + 1);

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

{

printf("%d - %d\n", i+1, visited[i]);

}

printf("\nВремя на проход в ширину на матрице смежности %f\n", t.elapsed());

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

visited[i] = -1;

createLinkedList(mas, head, N);

Timer t1;

BFS\_ForLinkedList(head, visited, num);

printf("\nРасстояние от вершины %d до других вершин, найденное с помощью обхода в ширину на списке смежности:\n", num + 1);

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

{

printf("%d - %d\n", i + 1, visited[i]);

}

printf("\nВремя на проход в ширину на списке смежности %f\n", t1.elapsed());

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

visited[i] = -1;

int distance = 0;

Timer t2;

DFS(mas, visited, N, num, distance);

printf("\nРасстояние от вершины %d до других вершин, найденное с помощью обхода в глубину на матрице смежности:\n", num + 1);

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

{

printf("%d - %d\n", i + 1, visited[i]);

}

printf("\nВремя на проход в глубину на матрице смежности %f\n", t2.elapsed());

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

head[i] = NULL;

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

visited[i] = -1;

createLinkedList(mas, head, N);

Timer t3;

DFS\_ForLinkedList(head, visited, num, distance);

printf("\nРасстояние от вершины %d до других вершин, найденное с помощью обхода в глубину на списке смежности:\n", num + 1);

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

{

printf("%d - %d\n", i + 1, visited[i]);

}

printf("\nВремя на проход в глубину на списке смежности %f\n", t3.elapsed());

\_getch();

### }**Результаты работы программы:**

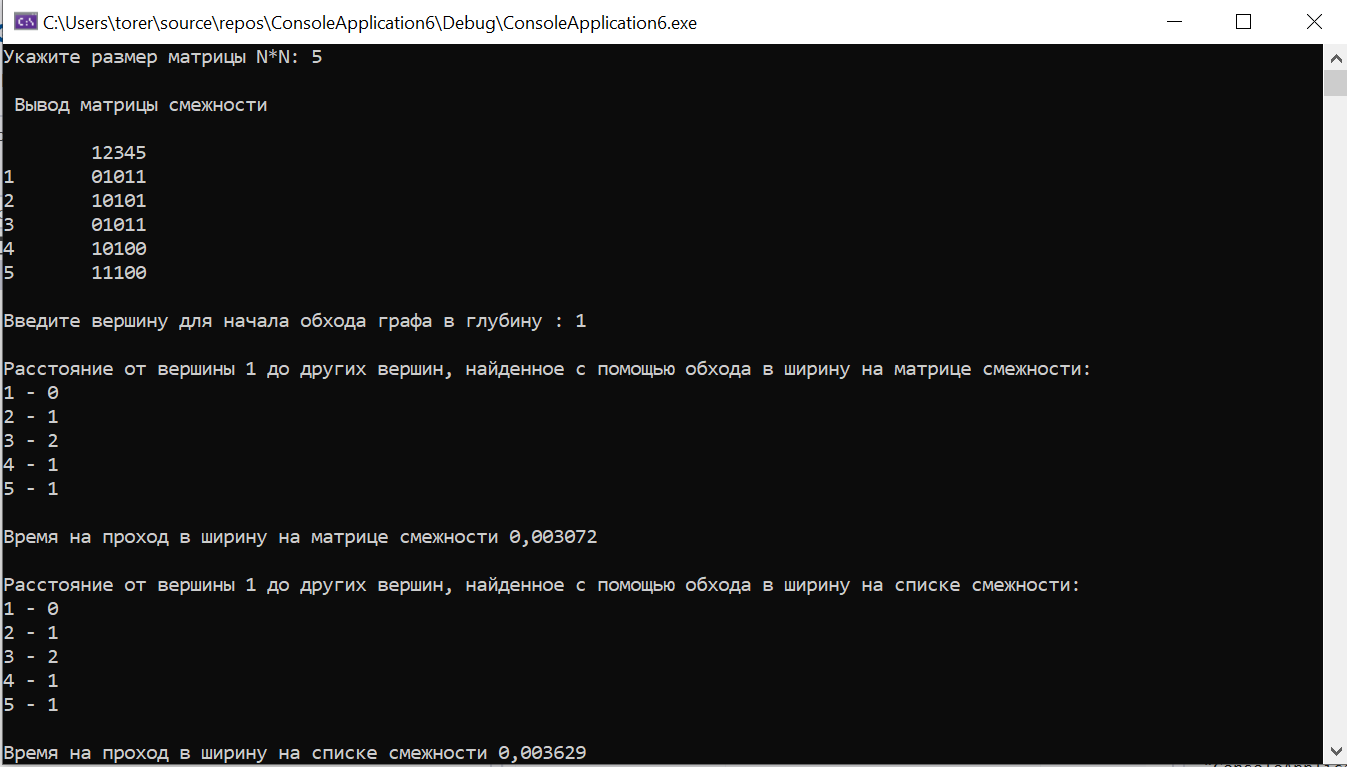


Рисунок 1 Результат работы программы

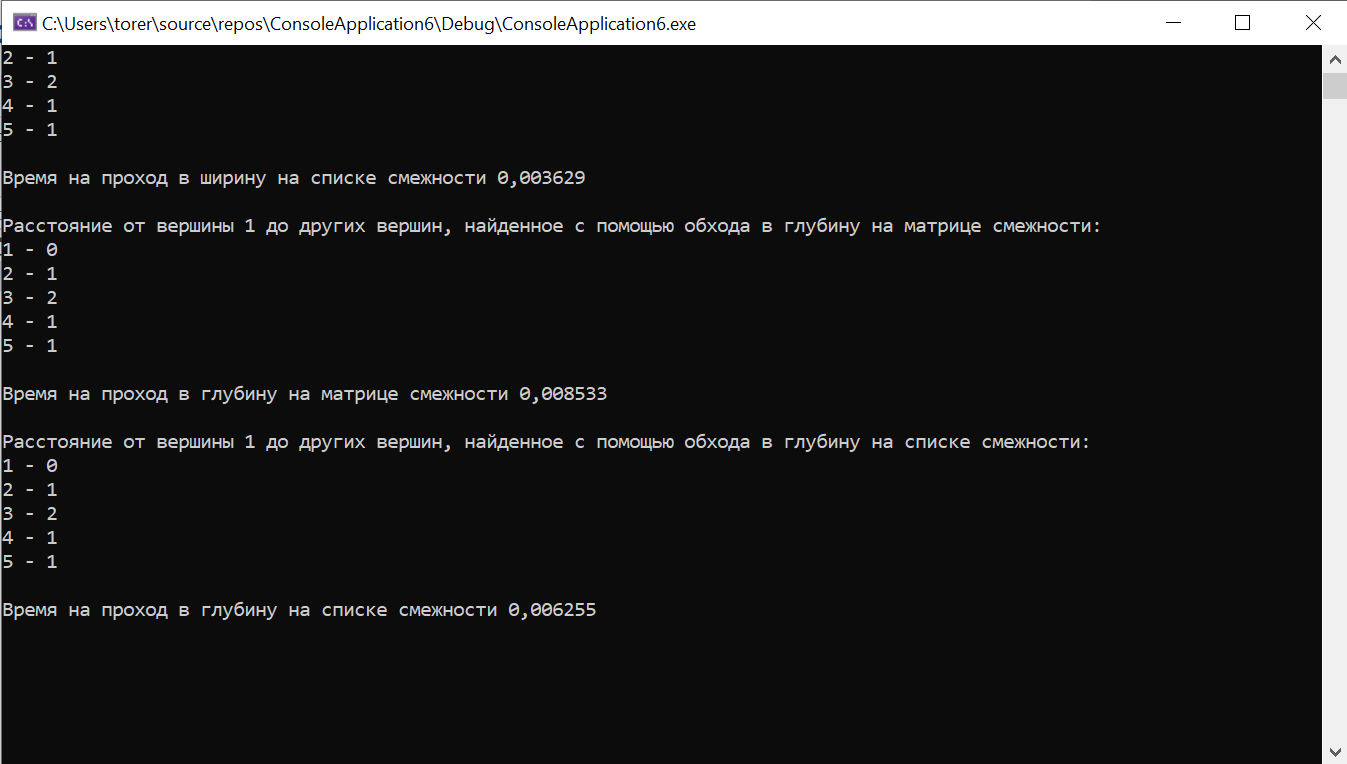


Рисунок 2 Результат работы программы

### **Вывод:**

В ходе выполнения лабораторной работы был реализован алгоритм поиска расстояний с помощью обхода графа в ширину и в глубину на основе матрицы смежности и списка смежности. Было сделано сравнение по времени работы данных алгоритмов.