Министерство науки и высшего образования Российской Федерации

Пензенский государственный университет

Кафедра “Вычислительная техника”

**Отчет**

по лабораторной работе № 9

по курсу “Логика и основы алгоритмизации в инженерных задачах”

на тему “Поиск расстояний в графе”

Выполнили студенты группы 22ВВВ3:

Шнайдер К. С.

Тельнов И. В.

Городничев М. И.

Приняли:

Юрова О. В.

Акифьев И. В.

Пенза 2023

**Лабораторное 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.

**Листинг**

#include <iostream>

#include <ctime>

#include <cstdlib>

#include <queue>

#include <stack>

#include <fstream>

#define X 1

using namespace std; // Директива для использования пространства имен std

struct Node

{

int data;

Node\* next;

};

// Функция для выделения памяти под двумерный массив

int\*\* createMatrix(int size)

{

int\*\* matrix = new int\* [size];

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

{

matrix[i] = new int[size];

}

return matrix;

}

// Функция для заполнения матрицы случайными значениями

void fillMatrixRandom(int\*\* matrix, int size)

{

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

{

for (int j = i; j < size; j++)

{

if (i == j) {

matrix[i][j] = 0;

}

else {

int randomValue = rand() % 2;

matrix[i][j] = randomValue;

matrix[j][i] = randomValue; // Матрица симметрична

}

}

}

}

// Функция для вывода матрицы с элементами

void printMatrix(int\*\* matrix, int size)

{

cout << "\nМатрица смежности:" << endl;

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

{

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

{

cout << matrix[i][j];

cout << " ";

}

cout << endl;

}

}

// Функция для освобождения памяти, выделенной под матрицу

void deleteMatrix(int\*\* matrix, int size)

{

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

{

delete[] matrix[i];

}

delete[] matrix;

}

// Функция вывода списка смежности

void printAdjacencyList(Node\*\* adjacencyList, int size)

{

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

{

cout << "Вершина " << i << " смежна с: ";

Node\* current = adjacencyList[i];

while (current != nullptr)

{

cout << current->data << " ";

current = current->next;

}

cout << endl;

}

}

// Функция освобождения памяти, выделенной для списка смежности

void freeAdjacencyList(Node\*\* adjacencyList, int size)

{

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

{

Node\* current = adjacencyList[i];

while (current != nullptr)

{

Node\* next = current->next;

delete current;

current = next;

}

}

delete[] adjacencyList;

}

// Функция перевода матрицы смежности в список смежности с обратным порядком соседних вершин

Node\*\* adjacencyMatrixToAdjacencyList(int\*\* matrix, int size)

{

Node\*\* adjacencyList = new Node \* [size];

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

{

adjacencyList[i] = nullptr; // Инициализация списка для текущей вершины

// Цикл в обратном порядке, чтобы добавить соседние вершины в обратном порядке

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

{

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

{

Node\* newNode = new Node;

newNode->data = j;

newNode->next = adjacencyList[i];

adjacencyList[i] = newNode;

}

}

}

return adjacencyList;

}

void bfsDistanceList(Node\*\* adjacencyList, int size, int startVertex, int\* distances)

{

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

{

distances[i] = -1; // Инициализируем расстояния как "не посещенные"

}

queue<int> q; // Создаем очередь для BFS

distances[startVertex] = 0; // Начальная вершина имеет расстояние 0

q.push(startVertex); // Помещаем начальную вершину в очередь

while (!q.empty())

{

int currentVertex = q.front();

q.pop();

Node\* adjacentNode = adjacencyList[currentVertex];

while (adjacentNode != nullptr)

{

int neighbor = adjacentNode->data;

if (distances[neighbor] == -1)

{

distances[neighbor] = distances[currentVertex] + 1;

q.push(neighbor);

}

adjacentNode = adjacentNode->next;

}

}

}

// Функция для поиска расстояний в графе с использованием матрицы смежности

void bfsDistanceMatrix(int\*\* matrix, int size, int startVertex, int\* distances)

{

bool\* visited = new bool[size];

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

{

distances[i] = -1; // Инициализируем расстояния как "не посещенные"

visited[i] = false; // Инициализируем все вершины как не посещенные

}

queue<int> q; // Создаем очередь для BFS

distances[startVertex] = 0; // Начальная вершина имеет расстояние 0

visited[startVertex] = true; // Помечаем начальную вершину как посещенную

q.push(startVertex); // Помещаем начальную вершину в очередь

while (!q.empty())

{

int currentVertex = q.front();

q.pop();

for (int neighbor = 0; neighbor < size; neighbor++)

{

if (matrix[currentVertex][neighbor] == 1 && !visited[neighbor])

{

distances[neighbor] = distances[currentVertex] + 1;

visited[neighbor] = true;

q.push(neighbor);

}

}

}

delete[] visited;

}

void dfsDistanceMatrix(int\*\* matrix, int size, int startVertex, int\* distances)

{

bool\* visited = new bool[size];

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

{

distances[i] = -1;

visited[i] = false;

}

stack<int> s;

distances[startVertex] = 0;

visited[startVertex] = true;

s.push(startVertex);

while (!s.empty()) {

int currentVertex = s.top();

s.pop();

for (int neighbor = 0; neighbor < size; neighbor++)

{

if (matrix[currentVertex][neighbor] == 1 && !visited[neighbor])

{

distances[neighbor] = distances[currentVertex] + 1;

visited[neighbor] = true;

s.push(neighbor);

}

}

}

delete[] visited;

}

void dfsDistanceList(Node\*\* adjacencyList, int size, int startVertex, int\* distances)

{

bool\* visited = new bool[size];

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

{

distances[i] = -1;

visited[i] = false;

}

stack<int> s;

distances[startVertex] = 0;

visited[startVertex] = true;

s.push(startVertex);

while (!s.empty())

{

int currentVertex = s.top();

s.pop();

Node\* adjacentNode = adjacencyList[currentVertex];

while (adjacentNode != nullptr) {

int neighbor = adjacentNode->data;

if (!visited[neighbor]) {

distances[neighbor] = distances[currentVertex] + 1;

visited[neighbor] = true;

s.push(neighbor);

}

adjacentNode = adjacentNode->next;

}

}

delete[] visited;

}

int main()

{

setlocale(LC\_ALL, "Rus");

srand(time(0));

int choice;

do

{

// Отображаем меню

cout << "Меню:" << endl;

cout << "1) Генерация графа и поиск расстояний" << endl;

cout << "2) Тест алгоритмов поиска расстояний" << endl;

cout << "3) Выход" << endl;

// Ввод номера пункта

cout << "\n";

cout << "Пожалуйста, введите номер выбранного пункта: ";

cin >> choice;

cout << "\n";

// Обработка выбора пользователя

switch (choice)

{

case 1:

{

cout << "Вы выбрали пункт 1." << endl;

int size;

cout << "\nВведите размер матрицы смежности: ";

cin >> size;

int\*\* adjacencyMatrix = createMatrix(size);

fillMatrixRandom(adjacencyMatrix, size);

printMatrix(adjacencyMatrix, size);

cout << "\nСписок смежности:\n";

Node\*\* adjacencyList = adjacencyMatrixToAdjacencyList(adjacencyMatrix, size);

printAdjacencyList(adjacencyList, size);

// Вызов функции поиска расстояний с указанием начальной вершины

int startVertex;

cout << "\nВведите начальную вершину для поиска расстояний: ";

cin >> startVertex;

if (startVertex >= 0 && startVertex < size)

{

int\* distances = new int[size];

bfsDistanceList(adjacencyList, size, startVertex, distances);

cout << "\nBFSD list: Расстояния от вершины " << startVertex << " до других вершин:\n";

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

{

cout << "Вершина " << i << ": " << distances[i] << endl;

}

bfsDistanceMatrix(adjacencyMatrix, size, startVertex, distances);

cout << "\nBFSD matrix: Расстояния от вершины " << startVertex << " до других вершин:\n";

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

{

cout << "Вершина " << i << ": " << distances[i] << endl;

}

dfsDistanceMatrix(adjacencyMatrix, size, startVertex, distances);

cout << "\nDFSD matrix: Расстояния от вершины " << startVertex << " до других вершин:\n";

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

{

cout << "Вершина " << i << ": " << distances[i] << endl;

}

dfsDistanceList(adjacencyList, size, startVertex, distances);

cout << "\nDFSD list: Расстояния от вершины " << startVertex << " до других вершин:\n";

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

{

cout << "Вершина " << i << ": " << distances[i] << endl;

}

cout << "\n";

delete[] distances;

}

else

{

cout << "\n";

cout << "Ошибка: Введена некорректная начальная вершина." << endl;

}

// Освобождение памяти

deleteMatrix(adjacencyMatrix, size);

freeAdjacencyList(adjacencyList, size);

break;

}

case 2:

{

cout << "Вы выбрали пункт 2." << endl;

int num\_vertex;

cout << "\nВведите количество вершин графа: ";

cin >> num\_vertex;

cout << "\n";

// Размер графа для тестов

int\* sizes = new int[X];

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

{

\*(sizes + i) = num\_vertex;

}

for (int i = 0; i < 1; i++)

{

int size = sizes[i];

int\*\* adjacencyMatrix = createMatrix(size);

fillMatrixRandom(adjacencyMatrix, size);

Node\*\* adjacencyList = adjacencyMatrixToAdjacencyList(adjacencyMatrix, size);

int startVertex = 0;

int\* distances = new int[size];

clock\_t startTime, endTime;

startTime = clock();

bfsDistanceList(adjacencyList, size, startVertex, distances);

endTime = clock();

double bfsDistanceListTime = double(endTime - startTime) / CLOCKS\_PER\_SEC;

startTime = clock();

bfsDistanceMatrix(adjacencyMatrix, size, startVertex, distances);

endTime = clock();

double bfsDistanceMatrixTime = double(endTime - startTime) / CLOCKS\_PER\_SEC;

startTime = clock();

dfsDistanceMatrix(adjacencyMatrix, size, startVertex, distances);

endTime = clock();

double dfsDistanceMatrixTime = double(endTime - startTime) / CLOCKS\_PER\_SEC;

startTime = clock();

dfsDistanceList(adjacencyList, size, startVertex, distances);

endTime = clock();

double dfsDistanceListTime = double(endTime - startTime) / CLOCKS\_PER\_SEC;

cout << size << "\t\t " << bfsDistanceListTime << "\t\t" << bfsDistanceMatrixTime << "\t\t\t" << dfsDistanceMatrixTime << "\t\t\t" << dfsDistanceListTime << "\n";

deleteMatrix(adjacencyMatrix, size);

freeAdjacencyList(adjacencyList, size);

}

delete[] sizes;

break;

}

case 3:

cout << "Выход." << endl;

break;

default:

cout << "Неправильный выбор. Пожалуйста, введите верный номер пункта." << endl;

break;

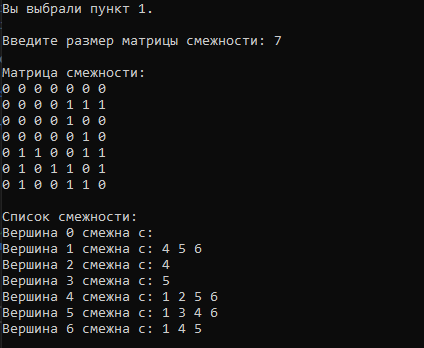
}

} while (choice != 3);

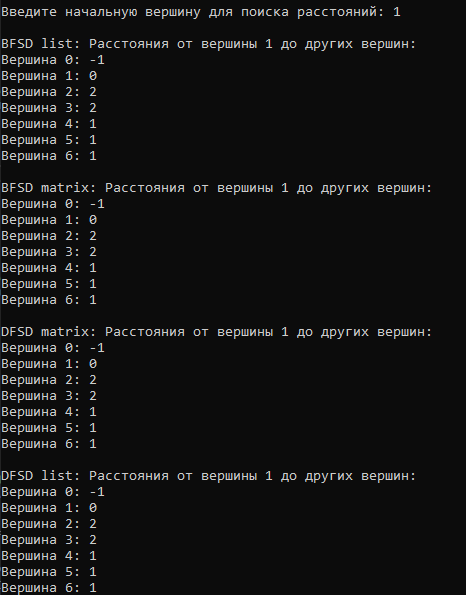
return 0;

}

**Результаты работы программы**

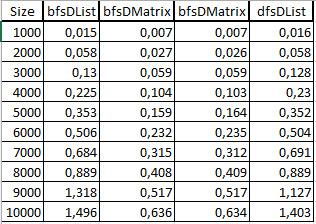


**Рисунок 1 - Результат работы программы**



**Рисунок 2 - Результат работы программы**

Пункт №2



**Вывод:** в ходе выполнения лабораторной работы сгенерировали 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. 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.