Министерство науки и высшего образования Российской Федерации

Пензенский государственный университет

Кафедра “Вычислительная техника”

**Отчет**

по лабораторной работе № 8

по курсу “Логика и основы алгоритмизации в инженерных задачах”

на тему “Обход графа в ширину”

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 обхода в ширину,

реализованную в соответствии с приведенным выше описанием. При

реализации алгоритма в качестве очереди используйте класс queue из

стандартной библиотеки С++.

3.\* Реализуйте процедуру обхода в ширину для графа, представленного

списками смежности.

Задание 2\*

1. Для матричной формы представления графов реализуйте алгоритм

обхода в ширину с использованием очереди, построенной на основе

структуры данных «список», самостоятельно созданной в лабораторной

работе № 3.

2. Оцените время работы двух реализаций алгоритмов обхода в ширину

(использующего стандартный класс queue и использующего очередь,

реализованную самостоятельно) для графов разных порядков.

**Листинг**

#include <iostream>

#include <ctime>

#include <cstdlib>

#include <queue>

#include <fstream>

#define X 1

#define Y 1

#define Z 1

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

struct Node {

int data;

Node\* next;

};

// Структура для представления элемента очереди

struct QueueNode {

int data;

QueueNode\* next;

};

// Структура для представления очереди

struct Queue {

QueueNode\* front;

QueueNode\* rear;

};

// Инициализация пустой очереди

Queue\* createQueue() {

Queue\* queue = new Queue;

queue->front = queue->rear = nullptr;

return queue;

}

// Проверка, пуста ли очередь

bool isEmpty(Queue\* queue) {

return queue->front == nullptr;

}

// 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 преобразования 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 вершины

for (int j = size - 1; j >= 0; j--) { // Обратный цикл для добавления вершин в обратном порядке

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

Node\* newNode = new Node;

newNode->data = j;

newNode->next = adjacencyList[i];

adjacencyList[i] = newNode;

}

}

}

return adjacencyList;

}

// Функция особождения памяти 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 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 освобождения памяти, выделенной под матрицу

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

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

delete[] matrix[i];

}

delete[] 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, разделенными пробелом

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

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

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

cout << matrix[i][j];

cout << " ";

}

cout << endl;

}

}

int\* breadthFirstSearch(int\*\* graph, int size, int startVertex) {

bool\* visited = new bool[size];

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

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

}

int\* traversalOrder = new int[size];

int currentIndex = 0;

queue<int> q;

q.push(startVertex);

visited[startVertex] = true;

while (!q.empty()) {

int currentVertex = q.front();

q.pop();

traversalOrder[currentIndex++] = currentVertex;

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

if (graph[currentVertex][i] == 1 && !visited[i]) {

q.push(i);

visited[i] = true;

}

}

}

delete[] visited;

return traversalOrder;

}

int\* bfsTraversal(Node\*\* adjacencyList, int size, int startVertex) {

bool\* visited = new bool[size];

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

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

}

int\* traversalOrder = new int[size];

int currentIndex = 0;

queue<int> q;

q.push(startVertex);

visited[startVertex] = true;

while (!q.empty()) {

int currentVertex = q.front();

q.pop();

traversalOrder[currentIndex++] = currentVertex;

// Проход по смежным вершинам списка смежности

Node\* current = adjacencyList[currentVertex];

while (current != nullptr) {

int neighbor = current->data;

if (!visited[neighbor]) {

q.push(neighbor);

visited[neighbor] = true;

}

current = current->next;

}

}

delete[] visited;

return traversalOrder;

}

// Функция для добавления вершины в список смежности

void addEdge(Node\*\* adjacencyList, int vertex, int data) {

Node\* newNode = new Node;

newNode->data = data;

newNode->next = adjacencyList[vertex];

adjacencyList[vertex] = newNode;

}

// Добавление элемента в очередь

void enqueue(Queue\* queue, int data) {

QueueNode\* newNode = new QueueNode;

newNode->data = data;

newNode->next = nullptr;

if (isEmpty(queue)) {

queue->front = queue->rear = newNode;

return;

}

queue->rear->next = newNode;

queue->rear = newNode;

}

// Извлечение элемента из очереди

int dequeue(Queue\* queue) {

if (isEmpty(queue)) {

cerr << "Очередь пуста!" << endl;

return -1; // Возвращаем -1, чтобы обозначить ошибку

}

int data = queue->front->data;

QueueNode\* temp = queue->front;

queue->front = queue->front->next;

delete temp;

if (queue->front == nullptr) {

queue->rear = nullptr; // Если очередь стала пустой

}

return data;

}

int\* BFScustomList(int\*\* matrix, int size, int startVertex) {

bool\* visited = new bool[size];

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

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

}

int\* traversalOrder = new int[size];

int currentIndex = 0;

Queue\* queue = createQueue();

enqueue(queue, startVertex);

visited[startVertex] = true;

while (!isEmpty(queue)) {

int currentVertex = dequeue(queue);

traversalOrder[currentIndex++] = currentVertex;

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

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

enqueue(queue, i);

visited[i] = true;

}

}

}

delete[] visited;

delete(queue);

return traversalOrder;

}

int main()

{

setlocale(LC\_ALL, "Rus");

srand(time(0));

int choice;

do {

// Отображаем меню

cout << "Меню:" << endl;

cout << "1) Генерация матрицы и поиск в ширину" << endl;

cout << "2) Тест алгоритмов обхода в ширину" << endl;

cout << "3) Выход" << endl;

// Запрашиваем выбор пользователя

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

cin >> choice;

// Обработка выбора пользователя

switch (choice) {

case 1: {

cout << "Вы выбрали пункт №1" << endl;

// 1 задание

int size;

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

cin >> size;

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

fillMatrixRandom(adjacencyMatrix, size);

printMatrix(adjacencyMatrix, size);

// Перевод матрицы смежности в список смежности и его вывод

cout << "\nСписок смежности:\n";

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

printAdjacencyList(adjacencyList, size);

int startVertex;

cout << "\nВведите начальную вершину для обхода: ";

cin >> startVertex; cout << "\n";

int\* traversalOrder = breadthFirstSearch(adjacencyMatrix, size, startVertex);

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

cout << traversalOrder[i] << " ";

}

cout << " ВbreadthFirstSearch\n" << endl;

delete[] traversalOrder;

int\* traversalOrder1 = bfsTraversal(adjacencyList, size, startVertex);

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

cout << traversalOrder1[i] << " ";

}

cout << " bfsTraversal\n" << endl;

delete[] traversalOrder1;

int\* traversalOrder2 = BFScustomList(adjacencyMatrix, size, startVertex);

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

cout << traversalOrder2[i] << " ";

}

delete[] traversalOrder2;

cout << " BFScustomList\n" << endl;

deleteMatrix(adjacencyMatrix, size);

break;

}

// 2 задание

case 2:

{

int num\_vertex;

cout << "\nВы выбрали пункт №2" << endl;

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

cin >> num\_vertex;

cout << "\n";

// Размер графа для тестов

int\* sizes = new int[X \* Y \* Z];

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

{

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

{

for (int k = 0; k < Z; k++) {

\*(sizes + i \* Y \* Z + j \* Z + k) = num\_vertex;

}

}

}

cout << "Размер матрицы\t BFS матрица \t BFS список \t BFS свой\n";

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

int size = sizes[i];

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

fillMatrixRandom(adjacencyMatrix, size);

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

int startVertex = 0;

clock\_t startTime, endTime;

startTime = clock();

int\* traversalOrder = breadthFirstSearch(adjacencyMatrix, size, startVertex);

endTime = clock();

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

delete[] traversalOrder;

startTime = clock();

int\* traversalOrder1 = bfsTraversal(adjacencyList, size, startVertex);

endTime = clock();

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

delete[] traversalOrder1;

startTime = clock();

int\* traversalOrder2 = BFScustomList(adjacencyMatrix, size, startVertex);

endTime = clock();

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

delete[] traversalOrder2;

cout << " \t" << size << "\t\t" << bfsMatrixTime << "\t\t" << bfsListTime << "\t\t" << bfsCustomTime << "\n";

deleteMatrix(adjacencyMatrix, size);

freeAdjacencyList(adjacencyList, size);

}

delete[] sizes;

break; }

case 3:

cout << "Выход." << endl;

break;

default:

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

break;

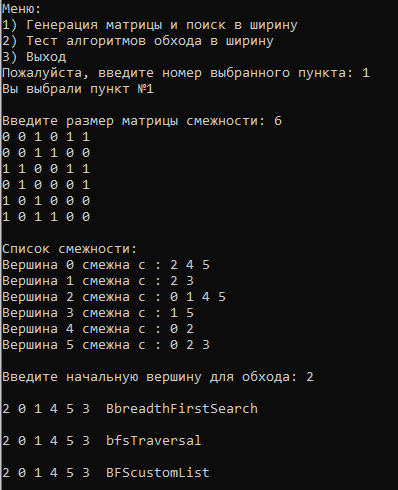
}

} while (choice != 3); // Выход из меню при выборе "Выход" (пункт 4)

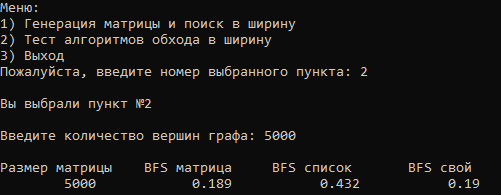
return 0;

}

**Результаты работы программы**

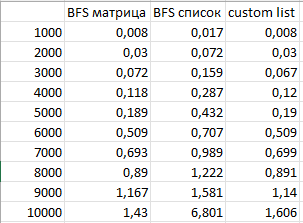


**Рисунок 1 - Результат работы программы**

****

**Рисунок 2 - Результат работы программы**

Пункт №2



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

смежности для неориентированного графа G, выведите матрицу на экран.

Для сгенерированного графа осуществили процедуру обхода в ширину, при

реализации алгоритма в качестве очереди использовали класс queue из

стандартной библиотеки С++. Реализовали процедуру обхода в ширину для графа, представленного списками смежности. Для матричной формы представления графов реализовали алгоритм обхода в ширину с использованием очереди, построенной на основе структуры данных «список», самостоятельно созданной в лабораторной работе № 3. Оценили время работы двух реализаций алгоритмов обхода в ширину

(использующего стандартный класс queue и использующего очередь,

реализованную самостоятельно) для графов разных порядков.