МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Кафедра «Вычислительная техника»

# ОТЧЁТ

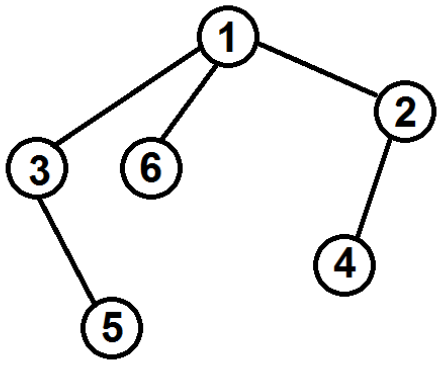
По лабораторной работе №9
По курсу «Логика и основы алгоритмизации в инженерных задачах»
На тему «Поиск расстояний в графе»

**Выполнили**
**студенты**
**группы**
**23ВВВ4:**
Святов И.Ю.
Епинин Д.В.

**Приняли:**
Юрова О.В.
Деев М.В.

Пенза 2024

**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.




$$G = \begin{Bmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{Bmatrix}$$

Рисунок 1 – Граф

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

## Задание:

**1.** Сгенерируйте (используя генератор случайных чисел) матрицу смежности для неориентированного графа *G*. Выведите матрицу на экран.

Ответ:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>

int main() {
    setlocale(LC_ALL, "Russian");
    srand(time(NULL));
    const int G = 5;  // Размер графа
    int M[G][G];      // Матрица смежности

    // Инициализация матрицы смежности
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            if (i == j) {
                M[i][j] = 0;  // Нет петли
            }
```

```
            else {
                M[i][j] = 0;  // Изначально нет ребер
            }
        }
    }
    // Генерация случайных рёбер для неориентированного графа
    for (int i = 0; i < G; i++) {
        for (int j = i + 1; j < G; j++) {
            M[i][j] = M[j][i] = rand() % 2;  // Случайное ребро между i и j
        }
    }
    // Вывод матрицы смежности на экран
    printf("Матрица смежности:\n");
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            printf("%d ", M[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

**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 C++.

Ответ:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>
#include <queue>

//const int G = 5;  // Размер графа
//int M[G][G];      // Матрица смежности

void bfs(int startVertex, int G, int** M) {
    std::queue<int> q;  // Очередь для BFS
    bool* visited = (bool*)malloc(G * sizeof(bool));
    int* distance = (int*)malloc(G * sizeof(int));
    for (int i = 0; i < G; i++) {
        visited[i] = false;
    }
   // bool visited[G] = { false };  // Массив для отслеживания посещенных вершин
    //int distance[G];  // Массив для хранения расстояний
    for (int i = 0; i < G; ++i) {
        distance[i] = -1;  // Изначально расстояние до всех вершин неопределено
    }

    visited[startVertex] = true;
    distance[startVertex] = 0;
    q.push(startVertex);

    while (!q.empty()) {
        int currentVertex = q.front();
        q.pop();
```

```
        // Смотрим соседей текущей вершины
        for (int i = 0; i < G; i++) {
            if (M[currentVertex][i] == 1 && !visited[i]) {   // Если есть ребро и не
посещена
                visited[i] = true;
                distance[i] = distance[currentVertex] + 1;  // Увеличиваем расстояние
                q.push(i);  // Добавляем в очередь
            }
        }
    }

    // Вывод расстояний от стартовой вершины
    printf("Расстояния от вершины %d:\n", startVertex);
    for (int i = 0; i < G; i++) {
        if (distance[i] != -1) {
            printf("До вершины %d: %d\n", i, distance[i]);
        }
        else {
            printf("До вершины %d: недоступно\n", i);
        }
    }
}

int main() {
    setlocale(LC_ALL, "Russian");
    srand(time(NULL));

    int G;
    int start;
    printf("Введите размер: ");
    scanf("%d", &G);
    int** M = (int**)malloc(G * sizeof(int*));
    for (int i = 0; i < G; i++) {
        M[i] = (int*)malloc(G * sizeof(int));
    }

    // Инициализация матрицы смежности
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            if (i == j) {
                M[i][j] = 0;  // Нет петли
            }
            else {
                M[i][j] = 0;  // Изначально нет ребер
            }
        }
    }
    // Генерация случайных рёбер для неориентированного графа
    for (int i = 0; i < G; i++) {
        for (int j = i + 1; j < G; j++) {
            M[i][j] = M[j][i] = rand() % 2;  // Случайное ребро между i и j
        }
    }

    // Вывод матрицы смежности на экран
    printf("Матрица смежности:\n");
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            printf("%d ", M[i][j]);
        }
        printf("\n");
    }

    // Запуск BFS от вершины 0 (можно выбрать любую)
```

```
        bfs(0, G, M);
        for (int i = 0; i < G; i++) {
            free(M[i]);
        }
        free(M);
        return 0;
    }
```

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

Ответ:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>
#include <queue>

#define G 5  // Размер графа
#define MAX_QUEUE_SIZE 100

typedef struct Node {
    int vertex;
    struct Node* next;
} Node;

typedef struct Graph {
    Node* adjLists[G];
} Graph;

// Создание графа
Graph* createGraph() {
    Graph* graph = (Graph*)malloc(sizeof(Graph));  // Явное приведение типа
    for (int i = 0; i < G; i++) {
        graph->adjLists[i] = NULL;
    }
    return graph;
}

// Добавление рёбер в граф
void addEdge(Graph* graph, int src, int dest) {
    Node* newNode = (Node*)malloc(sizeof(Node));  // Явное приведение типа
    newNode->vertex = dest;
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Для неориентированного графа добавляем обратное направление
    newNode = (Node*)malloc(sizeof(Node));  // Явное приведение типа
    newNode->vertex = src;
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Освобождение памяти
void freeGraph(Graph* graph) {
    for (int i = 0; i < G; i++) {
        Node* temp = graph->adjLists[i];
        while (temp) {
            Node* next = temp->next;
```

```
            free(temp);
            temp = next;
        }
    }
    free(graph);
}

// Реализация BFS
void bfs(Graph* graph, int startVertex) {
    int queue[MAX_QUEUE_SIZE];    // Очередь
    int front = 0, rear = 0;      // Указатели на начало и конец очереди
    int visited[G] = { 0 };          // Массив для отслеживания посещенных вершин
    int distance[G];               // Массив для хранения расстояний

    // Инициализация расстояний
    for (int i = 0; i < G; ++i) {
        distance[i] = -1;  // Изначально расстояние до всех вершин неопределено
    }

    // Начнем с начальной вершины
    visited[startVertex] = 1;
    distance[startVertex] = 0;
    queue[rear++] = startVertex;  // Добавляем начальную вершину в очередь

    while (front < rear) {
        int currentVertex = queue[front++];  // Получаем текущую вершину из очереди

        // Смотрим соседей текущей вершины
        Node* temp = graph->adjLists[currentVertex];
        while (temp) {
            int adjVertex = temp->vertex;
            if (!visited[adjVertex]) {  // Если сосед не посещен
                visited[adjVertex] = 1;
                distance[adjVertex]  =  distance[currentVertex]  +  1;    //  Увеличиваем
расстояние
                queue[rear++] = adjVertex;  // Добавляем в очередь
            }
            temp = temp->next;
        }
    }

    // Вывод расстояний от стартовой вершины
    printf("Расстояния от вершины %d:\n", startVertex);
    for (int i = 0; i < G; i++) {
        if (distance[i] != -1) {
            printf("До вершины %d: %d\n", i, distance[i]);
        }
        else {
            printf("До вершины %d: недоступно\n", i);
        }
    }
}

int main() {
    setlocale(LC_ALL, "Russian");
    srand(time(NULL));

    // Создание графа
    Graph* graph = createGraph();

    // Генерация случайных рёбер для неориентированного графа
    for (int i = 0; i < G; i++) {
        for (int j = i + 1; j < G; j++) {
            if (rand() % 2) {  // Случайное ребро между i и j
                addEdge(graph, i, j);
```

```
            }
        }
    }

    // Вывод списков смежности на экран
    printf("Списки смежности:\n");
    for (int i = 0; i < G; i++) {
        printf("%d: ", i);
        Node* temp = graph->adjLists[i];
        while (temp) {
            printf("%d ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }

    // Запуск BFS от вершины 0 (можно выбрать любую)
    bfs(graph, 0);

    // Освобождение памяти
    freeGraph(graph);

    return 0;
}
```

## Задание 2*

1. Реализуйте процедуру поиска расстояний на основе обхода в глубину.

Ответ:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>

const int G = 5; // Размер графа
int M[G][G];     // Матрица смежности
int visited[G];  // Массив для отслеживания посещенных вершин
int distance[G]; // Массив для хранения расстояний

// Функция для выполнения обхода в глубину
void dfs(int vertex, int dist) {
    visited[vertex] = 1;              // Отмечаем вершину как посещенную
    distance[vertex] = dist;          // Устанавливаем расстояние до вершины

    for (int i = 0; i < G; i++) {
        if (M[vertex][i] == 1 && !visited[i]) { // Проверяем наличие ребра и
посещенность
            dfs(i, dist + 1);         // Рекурсивно вызываем dfs для соседней вершины
        }
    }
}

int main() {
    setlocale(LC_ALL, "Russian");
```

```
    srand(time(NULL));

    // Инициализация матрицы смежности
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            if (i == j) {
                M[i][j] = 0;  // Нет петли
            }
            else {
                M[i][j] = 0;  // Изначально нет ребер
            }
        }
    }

    // Генерация случайных рёбер для неориентированного графа
    for (int i = 0; i < G; i++) {
        for (int j = i + 1; j < G; j++) {
            M[i][j] = M[j][i] = rand() % 2; // Случайное ребро между i и j
        }
    }

    // Вывод матрицы смежности на экран
    printf("Матрица смежности:\n");
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            printf("%d ", M[i][j]);
        }
        printf("\n");
    }

    // Инициализация массивов
    memset(visited, 0, sizeof(visited));
    memset(distance, -1, sizeof(distance)); // Установка -1 для обозначения недоступных
вершин

    // Запуск DFS от стартовой вершины (например, 0)
    dfs(0, 0);

    // Вывод расстояний от стартовой вершины
    printf("Расстояния от вершины 0:\n");
    for (int i = 0; i < G; i++) {
        if (distance[i] != -1) {
            printf("До вершины %d: %d\n", i, distance[i]);
        }
        else {
            printf("До вершины %d: недоступно\n", i);
        }
    }

    return 0;
}
```

**2.** Реализуйте процедуру поиска расстояний на основе обхода в глубину для графа, представленного списками смежности.

Ответ:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>
#include <vector>

const int G = 5; // Размер графа
std::vector<int> adj[G]; // Списки смежности
int visited[G];   // Массив для отслеживания посещенных вершин
int distance[G];  // Массив для хранения расстояний

// Функция для выполнения обхода в глубину
void dfs(int vertex, int dist) {
    visited[vertex] = 1;               // Отмечаем вершину как посещенную
    distance[vertex] = dist;           // Устанавливаем расстояние до вершины

    for (int i = 0; i < adj[vertex].size(); i++) {
        int neighbor = adj[vertex][i]; // Получаем соседа
        if (!visited[neighbor]) {      // Проверяем посещенность
            dfs(neighbor, dist + 1);   // Рекурсивно вызываем dfs для соседней вершины
        }
    }
}

int main() {
    setlocale(LC_ALL, "Russian");
    srand(time(NULL));

    // Генерация случайных рёбер для неориентированного графа
    for (int i = 0; i < G; i++) {
        for (int j = i + 1; j < G; j++) {
            if (rand() % 2) { // Случайное ребро между i и j
                adj[i].push_back(j);
                adj[j].push_back(i);
            }
        }
    }

    // Вывод матрицы смежности на экран
    printf("Списки смежности:\n");
    for (int i = 0; i < G; i++) {
        printf("%d: ", i);
        for (size_t j = 0; j < adj[i].size(); j++) {
            printf("%d ", adj[i][j]);
        }
        printf("\n");
    }

    // Инициализация массивов
    memset(visited, 0, sizeof(visited));
    memset(distance,  -1,  sizeof(distance));  //  Установка  -1  для  обозначения
недоступных вершин

    // Запуск DFS от стартовой вершины (например, 0)
    dfs(0, 0);
```

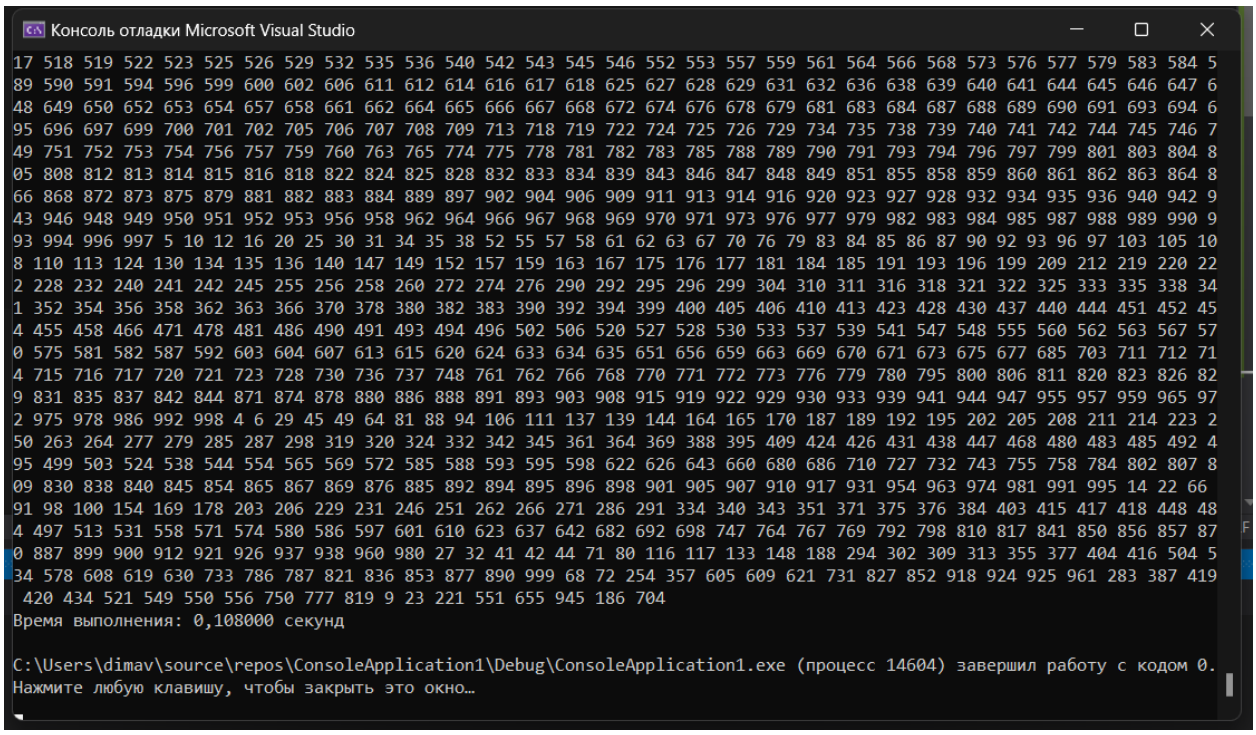
```
    // Вывод расстояний от стартовой вершины
    printf("Расстояния от вершины 0:\n");
    for (int i = 0; i < G; i++) {
        if (distance[i] != -1) {
            printf("До вершины %d: %d\n", i, distance[i]);
        }
        else {
            printf("До вершины %d: недоступно\n", i);
        }
    }

    return 0;
}
```

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

Использует обход в ширину:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>
#include <vector>
#include <ctime> // Для использования clock()
#include <time.h> // Для использования clock()

#define G 1000  // Размер графа
int M[G][G];      // Матрица смежности
bool visited[G];  // Массив для отслеживания посещенных вершин

void bfs(int start) {
    int q[G];
    int front = 0, rear = 0;
    visited[start] = true;
    q[rear++] = start;

    while (front != rear) {
        int current = q[front++];
        printf("%d ", current);

        for (int i = 0; i < G; i++) {
            if (M[current][i] == 1 && !visited[i]) {
                visited[i] = true;
                q[rear++] = i;
            }
        }
    }
}

int main() {
    setlocale(LC_ALL, "Russian");
```

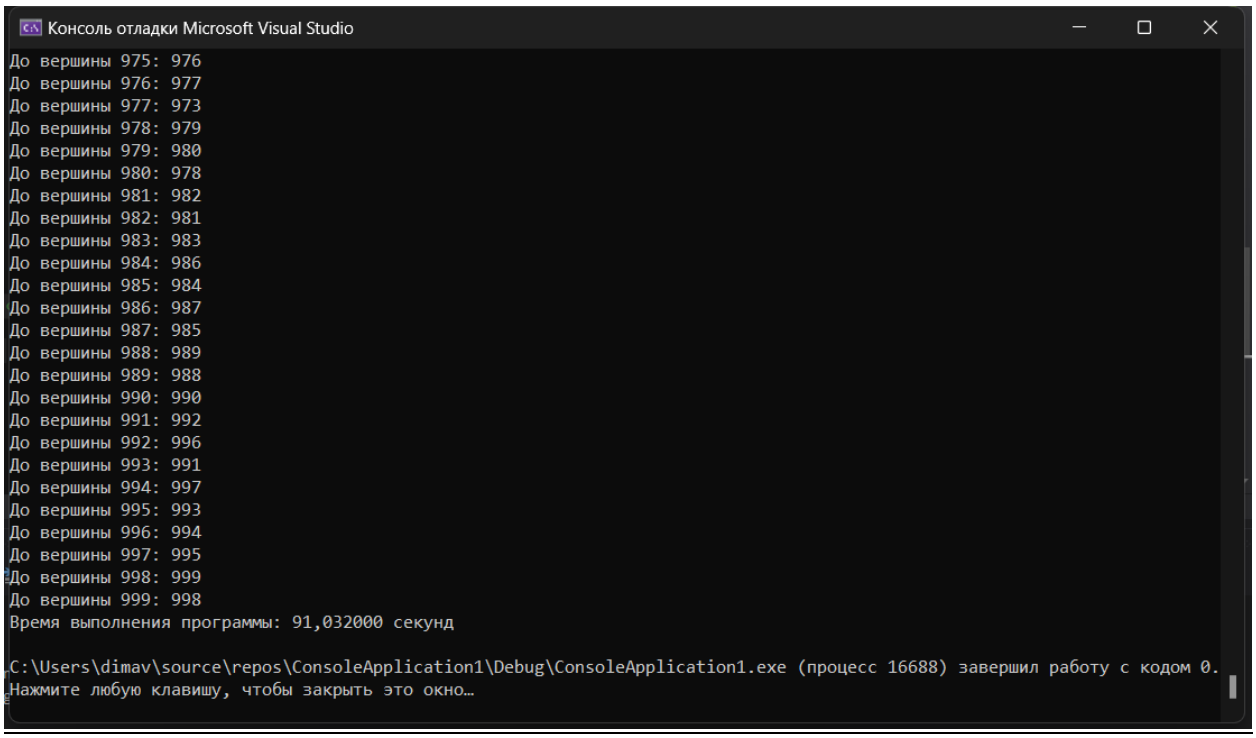
```
    srand(static_cast<unsigned int>(time(NULL)));

    // Инициализация матрицы смежности
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            M[i][j] = 0;  // Изначально нет рёбер
        }
    }

    // Генерация случайных рёбер для неориентированного графа
    for (int i = 0; i < G; i++) {
        for (int j = i + 1; j < G; j++) {
            M[i][j] = M[j][i] = rand() % 2;  // Случайное ребро между i и j
        }
    }

    // Вывод матрицы смежности на экран
    printf("Матрица смежности:\n");
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            printf("%d ", M[i][j]);
        }
        printf("\n");
    }

    // Измерение времени выполнения обхода в ширину
    clock_t start_time = clock();  // Запоминаем время начала
    // Обход в ширину, начиная с вершины 0
    printf("Обход в ширину: ");
    bfs(0);
    printf("\n");
    clock_t end_time = clock();  // Запоминаем время окончания

    // Вычисляем и выводим время выполнения
    double time_taken = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
    printf("Время выполнения: %f секунд\n", time_taken);

    return 0;
}
```




## Использует обход в глубину:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>
#include <vector>
#include <ctime> // Для использования clock()
const int G = 1000; // Размер графа
std::vector<int> adj[G]; // Списки смежности
int visited[G];   // Массив для отслеживания посещенных вершин
int distance[G];  // Массив для хранения расстояний

// Функция для выполнения обхода в глубину
void dfs(int vertex, int dist) {
    visited[vertex] = 1;               // Отмечаем вершину как посещенную
    distance[vertex] = dist;           // Устанавливаем расстояние до вершины

    for (int i = 0; i < adj[vertex].size(); i++) {
        int neighbor = adj[vertex][i]; // Получаем соседа
        if (!visited[neighbor]) {      // Проверяем посещенность
            dfs(neighbor, dist + 1);   // Рекурсивно вызываем dfs для соседней
вершины
        }
    }
}

int main() {
    setlocale(LC_ALL, "Russian");
    srand(static_cast<unsigned int>(time(NULL))); // Инициализация генератора
случайных чисел
```

```
    clock_t start = clock(); // Начало измерения времени

    // Генерация случайных рёбер для неориентированного графа
    for (int i = 0; i < G; i++) {
        for (int j = i + 1; j < G; j++) {
            if (rand() % 2) { // Случайное ребро между i и j
                adj[i].push_back(j);
                adj[j].push_back(i);
            }
        }
    }

    // Вывод списков смежности на экран
    printf("Списки смежности:\n");
    for (int i = 0; i < G; i++) {
        printf("%d: ", i);
        for (size_t j = 0; j < adj[i].size(); j++) {
            printf("%d ", adj[i][j]);
        }
        printf("\n");
    }

    // Инициализация массивов
    memset(visited, 0, sizeof(visited));
    memset(distance, -1, sizeof(distance)); // Установка -1 для обозначения
недоступных вершин

    // Запуск DFS от стартовой вершины (например, 0)
    dfs(0, 0);

    // Вывод расстояний от стартовой вершины
    printf("Расстояния от вершины 0:\n");
    for (int i = 0; i < G; i++) {
        if (distance[i] != -1) {
            printf("До вершины %d: %d\n", i, distance[i]);
        }
        else {
            printf("До вершины %d: недоступно\n", i);
        }
    }

    clock_t end = clock(); // Конец измерения времени
    double elapsed_time = double(end - start) / CLOCKS_PER_SEC; // Вычисление
прошедшего времени

    printf("Время выполнения программы: %.6f секунд\n", elapsed_time);

    return 0;
}
```




**Задание:** убрать лишние вектор visited. И второе, в списках смежности добавить динамические выделение памяти.

Ответ 1:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>
#include <queue>

//const int G = 5;  // Размер графа
//int M[G][G];      // Матрица смежности

void bfs(int startVertex, int G, int** M) {
    std::queue<int> q;  // Очередь для BFS
    int* distance = (int*)malloc(G * sizeof(int));

    for (int i = 0; i < G; ++i) {
        distance[i] = -1;  // Изначально расстояние до всех вершин неопределено
    }

    distance[startVertex] = 0;
    q.push(startVertex);

    while (!q.empty()) {
        int currentVertex = q.front();
        q.pop();
```

```
        // Смотрим соседей текущей вершины
        for (int i = 0; i < G; i++) {
            if (M[currentVertex][i] == 1 && distance[i] == -1) {  // Если есть
ребро и не посещена
                distance[i] = distance[currentVertex] + 1;  // Увеличиваем
расстояние
                q.push(i);  // Добавляем в очередь
            }
        }
    }

    // Вывод расстояний от стартовой вершины
    printf("Расстояния от вершины %d:\n", startVertex);
    for (int i = 0; i < G; i++) {
        if (distance[i] != -1) {
            printf("До вершины %d: %d\n", i, distance[i]);
        }
        else {
            printf("До вершины %d: недоступно\n", i);
        }
    }
    free(distance);
}

int main() {
    setlocale(LC_ALL, "Russian");
    srand(time(NULL));

    int G;
    int start;
    printf("Введите размер: ");
    scanf("%d", &G);
    int** M = (int**)malloc(G * sizeof(int*));
    for (int i = 0; i < G; i++) {
        M[i] = (int*)malloc(G * sizeof(int));
    }

    // Инициализация матрицы смежности
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            if (i == j) {
                M[i][j] = 0;  // Нет петли
            }
            else {
                M[i][j] = 0;  // Изначально нет ребер
            }
        }
    }
    // Генерация случайных рёбер для неориентированного графа
    for (int i = 0; i < G; i++) {
        for (int j = i + 1; j < G; j++) {
            M[i][j] = M[j][i] = rand() % 2;  // Случайное ребро между i и j
        }
    }

    // Вывод матрицы смежности на экран
    printf("Матрица смежности:\n");
    for (int i = 0; i < G; i++) {
```

```
        for (int j = 0; j < G; j++) {
            printf("%d ", M[i][j]);
        }
        printf("\n");
    }

    // Запуск BFS от вершины 0 (можно выбрать любую)
    bfs(0, G, M);
    for (int i = 0; i < G; i++) {
        free(M[i]);
    }
    free(M);
    return 0;
}
```

Ответ 2:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>
#include <queue>
#include <time.h>

typedef struct Node {
    int vertex;
    struct Node* next;
} Node;

typedef struct Graph {
    int numVertices;  // Количество вершин
    Node** adjLists;  // Списки смежности
} Graph;

// Создание графа
Graph* createGraph(int vertices) {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->numVertices = vertices;
    graph->adjLists = (Node**)malloc(vertices * sizeof(Node*));

    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
    }
    return graph;
}

// Добавление рёбер в граф
void addEdge(Graph* graph, int src, int dest) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->vertex = dest;
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Для неориентированного графа добавим обратное направление
    newNode = (Node*)malloc(sizeof(Node));
    newNode->vertex = src;
    newNode->next = graph->adjLists[dest];
```

```
    graph->adjLists[dest] = newNode;
}

// Освобождение памяти
void freeGraph(Graph* graph) {
    for (int i = 0; i < graph->numVertices; i++) {
        Node* temp = graph->adjLists[i];
        while (temp) {
            Node* next = temp->next;
            free(temp);
            temp = next;
        }
    }
    free(graph->adjLists);
    free(graph);
}

// Реализация BFS
void bfs(Graph* graph, int startVertex) {
    int* queue = (int*)malloc(graph->numVertices * sizeof(int));
    int front = 0, rear = 0;
    int* distance = (int*)malloc(graph->numVertices * sizeof(int));

    for (int i = 0; i < graph->numVertices; ++i) {
        distance[i] = -1;  // Изначально расстояние до всех вершин неопределено
    }


    distance[startVertex] = 0;
    queue[rear++] = startVertex;  // Добавляем начальную вершину в очередь

    while (front < rear) {
        int currentVertex = queue[front++];  // Получаем текущую вершину из очереди

        Node* temp = graph->adjLists[currentVertex];
        while (temp) {
            int adjVertex = temp->vertex;
            if (distance[adjVertex] == -1) {  // Если сосед не посещен

                distance[adjVertex] = distance[currentVertex] + 1;   // Увеличиваем
расстояние
                queue[rear++] = adjVertex;  // Добавляем в очередь
            }
            temp = temp->next;
        }
    }

    // Вывод расстояний от стартовой вершины
    printf("Расстояния от вершины %d:\n", startVertex);
    for (int i = 0; i < graph->numVertices; i++) {
        if (distance[i] != -1) {
            printf("До вершины %d: %d\n", i, distance[i]);
        }
        else {
            printf("До вершины %d: недоступно\n", i);
        }
    }
```

```
    free(queue);
    free(distance);
}

int main() {
    setlocale(LC_ALL, "Russian");

    int numVertices;
    printf("Введите количество вершин графа: ");
    scanf("%d", &numVertices);

    // Создание графа с динамически выделенной памятью
    Graph* graph = createGraph(numVertices);
    srand(time(NULL));  // Инициализация генератора случайных чисел

    // Генерация случайных рёбер для неориентированного графа
    for (int i = 0; i < numVertices; i++) {
        for (int j = i + 1; j < numVertices; j++) {
            if (rand() % 2) {  // Случайное решение о добавлении рёбер
                addEdge(graph, i, j);
            }
        }
    }

    // Вывод списков смежности на экран
    printf("Списки смежности:\n");
    for (int i = 0; i < numVertices; i++) {
        printf("%d: ", i);
        Node* temp = graph->adjLists[i];
        while (temp) {
            printf("%d ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }

    // Запуск BFS от вершины 0 (можно выбрать любую)
    bfs(graph, 0);

    // Освобождение памяти
    freeGraph(graph);

    return 0;
}
```

**Вывод:** в ходе выполнения лабораторной работы были выполнены работы с графами, научились генерировать матрицы смежности и осуществлять обход в ширину различными методами. Оценили время работы этих методов и сделали вывод о том, какой из них более эффективен для решения конкретных задач.