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

ОТЧЁТ

По лабораторной работе №10

По курсу «Логика и основы алгоритмизации в инженерных задачах»

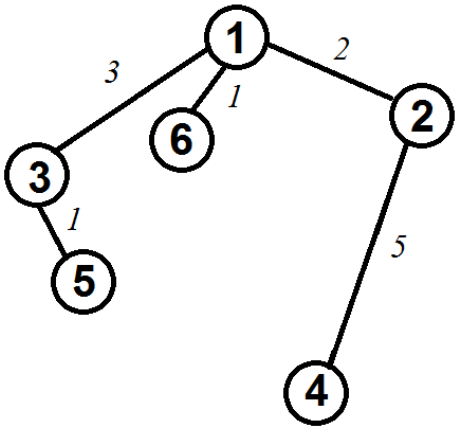
На тему «Поиск расстояний во взвешенном графе»

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

Приняли:
Юрова О.В.
Деев М.В.

Пенза 2024

Общие сведения:
Во взвешенном графе в отличие от не взвешенного каждое ребро имеет вес, отличный от нуля. Поэтому в матрице смежности взвешенного графа содержится информация не только о наличии ребра, но и о его весе.



$$G = \begin{bmatrix} 0 & 2 & 3 & 0 & 0 & 1 \\ 2 & 0 & 0 & 5 & 0 & 0 \\ 3 & 0 & 0 & 0 & 1 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Рисунок 1 – Граф

Поиск расстояний между вершинами в таком графе также возможно построить используя процедуры обхода графа. Отличие от поиска расстояний в не взвешенном графе будет состоять в том, что при обновлении расстояния до вершины при ее посещении оно будет увеличиваться не на 1, а на величину веса ребра.
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. *Сгенерируйте (используя генератор случайных чисел) матрицу смежности для ориентированного взвешенного графа *G*. Выведите матрицу на экран и осуществите процедуру поиска расстояний, реализованную в соответствии с приведенным выше описанием.

Ответ: во втором задание под цифрой 1.

Задание 2*

1. Для каждого из вариантов сгенерированных графов (ориентированного и не ориентированного) определите радиус и диаметр.

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>
#include <vector>
#include <limits.h>

void bfs(int startVertex, int G, int** M, std::vector<int>& distances) {
    std::queue<int> q;
    bool* visited = (bool*)malloc(G * sizeof(bool));

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

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

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

        for (int i = 0; i < G; i++) {
            if (M[currentVertex][i] != 0 && !visited[i]) {
                visited[i] = true;
                distances[i] = distances[currentVertex] + 1;
                q.push(i);
            }
        }
```

```
    }

    free(visited);
}

void findRadiusAndDiameter(int G, int** M) {
    int global_diameter = 0;
    int global_radius = INT_MAX;

    for (int i = 0; i < G; i++) {
        std::vector<int> distances(G, -1);
        bfs(i, G, M, distances);

        int max_dist = 0;
        for (int j = 0; j < G; j++) {
            if (distances[j] > max_dist) {
                max_dist = distances[j];
            }
        }

        if (max_dist > global_diameter) {
            global_diameter = max_dist;
        }

        if (max_dist < global_radius) {
            global_radius = max_dist;
        }
    }

    printf("Диаметр графа: %d\n", global_diameter);
    printf("Радиус графа: %d\n", (global_radius != INT_MAX) ? global_radius : -1); //
Если радиус остается MAX, значит, граф полностью недоступен
}

int main() {
    setlocale(LC_ALL, "Russian");
    srand(time(NULL));
    const int G = 5;  // Размер графа
    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");
    }

    // Выполнение BFS от вершины 0 и вывод расстояний
    std::vector<int> distances(G, -1);
    bfs(0, G, M, distances);
    printf("Расстояния от вершины 0:\n");
    for (int i = 0; i < G; i++) {
        if (distances[i] != -1) {
            printf("До вершины %d: %d\n", i, distances[i]);
        }
        else {
            printf("До вершины %d: недоступно\n", i);
        }
    }

    findRadiusAndDiameter(G, M);

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

    return 0;
}
```

2. Определите подмножества периферийных и центральных вершин.

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>
#include <vector>
#include <limits.h>

void bfs(int startVertex, int G, int** M, std::vector<int>& distances) {
    std::queue<int> q;
    bool* visited = (bool*)malloc(G * sizeof(bool));

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

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

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

        for (int i = 0; i < G; i++) {
            if (M[currentVertex][i] != 0 && !visited[i]) {
                visited[i] = true;
                distances[i] = distances[currentVertex] + 1;
                q.push(i);
            }
        }
    }

    free(visited);
}

void findRadiusAndDiameter(int G, int** M, std::vector<int>& peripheral,
std::vector<int>& central) {
    int global_diameter = 0;
    int global_radius = INT_MAX;

    for (int i = 0; i < G; i++) {
        std::vector<int> distances(G, -1);
        bfs(i, G, M, distances);

        int max_dist = 0;
        for (int j = 0; j < G; j++) {
            if (distances[j] > max_dist) {
                max_dist = distances[j];
            }
        }

        if (max_dist > global_diameter) {
            global_diameter = max_dist;
        }

        if (max_dist < global_radius) {
            global_radius = max_dist;
        }

        // Определение периферийных вершин
        if (max_dist == global_diameter) {
            peripheral.push_back(i);
        }

        // Определение центральных вершин
        if (max_dist == global_radius) {
            central.push_back(i);
        }
    }

    printf("Диаметр графа: %d\n", global_diameter);
    printf("Радиус графа: %d\n", (global_radius != INT_MAX) ? global_radius : -1);
// Если радиус остается MAX, значит, граф полностью недоступен
```

```
}

int main() {
    setlocale(LC_ALL, "Russian");
    srand(time(NULL));
    const int G = 5;  // Размер графа
    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");
    }

    // Выполнение BFS от вершины 0 и вывод расстояний
    std::vector<int> distances(G, -1);
    bfs(0, G, M, distances);
    printf("Расстояния от вершины 0:\n");
    for (int i = 0; i < G; i++) {
        if (distances[i] != -1) {
            printf("До вершины %d: %d\n", i, distances[i]);
        }
        else {
            printf("До вершины %d: недоступно\n", i);
        }
    }

    std::vector<int> peripheral;
    std::vector<int> central;
    findRadiusAndDiameter(G, M, peripheral, central);

    printf("Периферийные вершины: ");
    for (int v : peripheral) {
        printf("%d ", v);
    }
```

```
    printf("\n");

    printf("Центральные вершины: ");
    for (int v : central) {
        printf("%d ", v);
    }
    printf("\n");

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

    return 0;
}
```

Задание 3*

1. Модернизируйте программу так, чтобы получить возможность запуска программы с параметрами командной строки (см. описание ниже). В качестве параметра должны указываться тип графа (взвешенный или нет) и наличие ориентации его ребер (есть ориентация или нет).

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>
#include <vector>
#include <limits.h>
#include <ctime>
#include <limits.h>

void bfs(int startVertex, int G, int** M, std::vector<int>& distances) {
    std::queue<int> q;
    bool* visited = (bool*)malloc(G * sizeof(bool));

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

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

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

```
        q.pop();

        for (int i = 0; i < G; i++) {
            if (M[currentVertex][i] != 0 && !visited[i]) {
                visited[i] = true;
                distances[i] = distances[currentVertex] + 1;
                q.push(i);
            }
        }
    }

    free(visited);
}

void findRadiusAndDiameter(int G, int** M, std::vector<int>& peripheral,
std::vector<int>& central) {
    int global_diameter = 0;
    int global_radius = INT_MAX;

    for (int i = 0; i < G; i++) {
        std::vector<int> distances(G, -1);
        bfs(i, G, M, distances);

        int max_dist = 0;
        for (int j = 0; j < G; j++) {
            if (distances[j] > max_dist) {
                max_dist = distances[j];
            }
        }

        if (max_dist > global_diameter) {
            global_diameter = max_dist;
        }

        if (max_dist < global_radius) {
            global_radius = max_dist;
        }

        // Определение периферийных вершин
        if (max_dist == global_diameter) {
            peripheral.push_back(i);
        }

        // Определение центральных вершин
        if (max_dist == global_radius) {
            central.push_back(i);
        }
    }

    printf("Диаметр графа: %d\n", global_diameter);
    printf("Радиус графа: %d\n", (global_radius != INT_MAX) ? global_radius : -1);
}

int main(int argc, char* argv[]) {
    setlocale(LC_ALL, "Russian");
    srand(time(NULL));

    const int G = 5;  // Размер графа
```

```
    int** M = (int**)malloc(G * sizeof(int*));
    for (int i = 0; i < G; i++) {
        M[i] = (int*)malloc(G * sizeof(int));
    }

    bool isWeighted = false;
    bool isDirected = false;

    // Обработка аргументов командной строки
    if (argc > 1) {
        for (int i = 1; i < argc; i++) {
            if (strcmp(argv[i], "-v") == 0 || strcmp(argv[i], "-vzvesh") == 0) {
                if (i + 1 < argc && (strcmp(argv[i + 1], "da") == 0 || strcmp(argv[i
+ 1], "1") == 0)) {
                    isWeighted = true;
                    i++;  // Пропускаем следующий аргумент, так как он уже обработан
                }
            }
            if (strcmp(argv[i], "-o") == 0 || strcmp(argv[i], "-orient") == 0) {
                if (i + 1 < argc && (strcmp(argv[i + 1], "da") == 0 || strcmp(argv[i
+ 1], "1") == 0)) {
                    isDirected = true;
                    i++;  // Пропускаем следующий аргумент, так как он уже обработан
                }
            }
        }
    }
    else {
        // Ввод параметров графа с клавиатуры
        char qwer[100];
        printf("Взвешенный граф или нет (da/net): ");
        scanf("%s", qwer);

        char qwert[100];
        printf("Ориентированный граф или нет (da/net): ");
        scanf("%s", qwert);

        isWeighted = (strcmp(qwer, "da") == 0);  // Определяем, взвешенный граф или
нет
        isDirected = (strcmp(qwert, "da") == 0);  // Определяем, ориентированный граф
или нет
    }

    // Вывод параметров графа
    printf("Тип графа: %s\n", isWeighted ? "взвешенный" : "невзвешенный");
    printf("Наличие ориентации рёбер: %s\n", isDirected ? "да" : "нет");

    // Инициализация матрицы смежности
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            M[i][j] = (i == j) ? 0 : (rand() % 2);  // Нет петель, случайные ребра
        }
    }

    // Если граф ориентированный, можно оставить как есть. Если нет, симметризуем
матрицу
    if (!isDirected) {
        for (int i = 0; i < G; i++) {
```

```
            for (int j = i + 1; j < G; j++) {
                if (M[i][j] == 1) {
                    M[j][i] = 1; // Сдвигаем ребро обратно, если граф не
ориентированный
                }
            }
        }
    }

    // Если граф взвешенный, задаем случайные веса
    if (isWeighted) {
        for (int i = 0; i < G; i++) {
            for (int j = 0; j < G; j++) {
                if (M[i][j] == 1) {
                    M[i][j] = rand() % 10 + 1; // Устанавливаем случайный вес от 1 до
10
                }
            }
        }
    }

    // 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");
    }

    std::vector<int> distances(G, -1);
    bfs(0, G, M, distances);
    printf("Расстояния от вершины 0:\n");
    for (int i = 0; i < G; i++) {
        if (distances[i] != -1) {
            printf("До вершины %d: %d\n", i, distances[i]);
        }
        else {
            printf("До вершины %d: недоступно\n", i);
        }
    }

    std::vector<int> peripheral;
    std::vector<int> central;
    findRadiusAndDiameter(G, M, peripheral, central);

    printf("Периферийные вершины: ");
    for (int v : peripheral) {
        printf("%d ", v);
    }
    printf("\n");

    printf("Центральные вершины: ");
    for (int v : central) {
        printf("%d ", v);
    }
    printf("\n");
```

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

    return 0;
}
```

```
C:\Users\dimav>"C:\Users\dimav\source\repos\ConsoleApplication2\Debug\ConsoleApplication2" -v da -o net
Тип графа: взвешенный
Наличие ориентации рёбер: нет
Матрица смежности:
0 0 3 0 7
4 0 9 10 0
1 1 0 0 2
0 7 0 0 0
10 0 2 0 0
Расстояния от вершины 0:
До вершины 0: 0
До вершины 1: 2
До вершины 2: 1
До вершины 3: 3
До вершины 4: 1
Диаметр графа: 3
Радиус графа: 2
Периферийные вершины: 0 3 4
Центральные вершины: 0 1 2
```

Вывод: в ходе выполнения лабораторной работы был достигнут ряд значительных результатов: успешно сгенерированы и проанализированы графы, реализованы алгоритмы поиска кратчайших путей, проведены вычисления радиусов и диаметров, а также определены центральные и периферийные вершины. Модернизация программы добавила комфорта и гибкости в использовании. Лабораторная работа значительно углубила знания о математической теории графов и их практическом применении, а также повысила квалификацию в программировании на C++.