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 23ВВВ2:**

Пырков Д. А.

Родионов А. А.

**Приняли:**

Митрохин М. А.

Юрова О. В.

Пенза 2024

**Цель работы:** осуществить процедуру поиска расстояний для взвешенных неориентированных и ориентированных графов, определить радиус, диаметр, подмножества периферийных и центральных вершин, а также модернизировать программу так, чтобы получить возможность запуска программы с параметрами командной строки.

**Задание 1:**

1. Сгенерируйте (используя генератор случайных чисел) матрицу смежности для неориентированного взвешенного графа *G*. Выведите матрицу на экран. Для сгенерированного графа осуществите процедуру поиска расстояний, реализованную в соответствии с приведенным описанием. При реализации алгоритма в качестве очереди используйте класс **queue** из стандартной библиотеки С++.
2. (\*) 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 описанием.

**Задание 2:**

1. Для каждого из вариантов сгенерированных графов (ориентированного и не ориентированного) определите радиус и диаметр.
2. Определите подмножества периферийных и центральных вершин.

**\* Задание 3:**

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

**Ход работы**

**Задание 1**

**1.** **Поиск расстояний во взвешенном неориентированном графе**

1. Функция find\_distances:

void find\_distances(int size, int\_fast8\_t\*\* graph\_pointer) {

// Запрашивает начальную вершину у пользователя

int start\_vertex;

// Создает массив расстояний DIST, инициализируя все значения как -1

int\* DIST = new int[size];

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

DIST[i] = -1;

}

// Вызывает BFSD для поиска расстояний

BFSD(start\_vertex, size, graph\_pointer, DIST);

// Выводит найденные расстояния

}

1. Функция BFSD (BFS with Distance):

void BFSD(int v, int size, int\_fast8\_t\*\* graph\_pointer, int\* DIST) {

queue<int> Q; // Создает очередь для BFS

Q.push(v); // Добавляет начальную вершину в очередь

DIST[v] = 0; // Расстояние до начальной вершины = 0

while (!Q.empty()) {

int current = Q.front(); // Берет первую вершину из очереди

Q.pop();

// Проверяет все соседние вершины

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

// Если есть ребро и вершина еще не посещена

if (graph\_pointer[current][i] > 0 && DIST[i] == -1) {

Q.push(i); // Добавляет вершину в очередь

// Вычисляет расстояние как сумму расстояния до текущей вершины

// и веса ребра между текущей и новой вершиной

DIST[i] = DIST[current] + graph\_pointer[current][i];

}

}

}

}

Основные особенности работы алгоритма:

1. Использует массив DIST для хранения расстояний от начальной вершины до всех остальных
2. Начальные значения в DIST равны -1 (непосещенные вершины)
3. Для начальной вершины устанавливается расстояние 0
4. При обходе графа для каждой новой вершины расстояние вычисляется как:
   * Расстояние до текущей вершины (DIST[current])
   * Плюс вес ребра между текущей и новой вершиной (graph\_pointer[current][i])

Алгоритм гарантирует нахождение кратчайших путей в невзвешенном графе или в графе с положительными весами, так как рассматривает вершины в порядке возрастания расстояния от начальной вершины.

**Результат работы программы**

На рисунке 1 приведён результат работы программы №1.1.

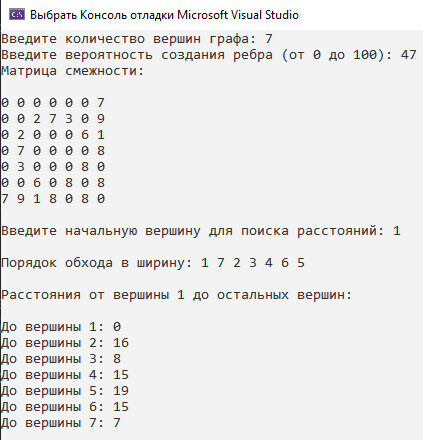


Рисунок 1 – результат работы программы №1.1

**2.** \* **Поиск расстояний во взвешенном ориентированном графе**

1. Функция find\_distances:

* Принимает размер графа и указатель на матрицу смежности
* Запрашивает начальную вершину у пользователя
* Создает массив DIST для хранения расстояний
* Вызывает BFSD для поиска расстояний
* Выводит найденные расстояния

1. Функция BFSD (Breadth-First Search with Distances):

void BFSD(int v, int size, int\_fast8\_t\*\* graph\_pointer, int\* DIST) {

queue<int> Q; // Создается очередь для BFS

// Инициализация массива расстояний бесконечностями

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

DIST[i] = INT\_MAX;

}

// Добавление начальной вершины

Q.push(v);

DIST[v] = 0; // Расстояние до начальной вершины = 0

while (!Q.empty()) { // Пока очередь не пуста

int current = Q.front(); // Берем первую вершину из очереди

Q.pop();

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

// Проверяем все возможные связи текущей вершины

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

if (graph\_pointer[i][current] > 0) { // Если есть ребро

// Вычисляем новое расстояние

int new\_dist = DIST[current] + graph\_pointer[i][current];

if (new\_dist < DIST[i]) { // Если новое расстояние меньше

DIST[i] = new\_dist; // Обновляем расстояние

// Если вершина еще не в очереди, добавляем её

if (find(Q.\_Get\_container().begin(), Q.\_Get\_container().end(), i) == Q.\_Get\_container().end()) {

Q.push(i);

}

}

}

}

}

}

Основные этапы работы алгоритма:

1. Инициализация:

* Создается очередь для BFS
* Все расстояния устанавливаются в бесконечность (INT\_MAX)
* Расстояние до начальной вершины устанавливается в 0

1. Основной цикл:

* Берется вершина из очереди
* Для каждого соседа этой вершины:
  + Вычисляется новое расстояние (текущее + вес ребра)
  + Если новое расстояние меньше существующего:
    - Обновляется расстояние
    - Вершина добавляется в очередь (если её там еще нет)

1. Результат:

* Массив DIST содержит кратчайшие расстояния от начальной вершины до всех остальных
* Если DIST[i] = INT\_MAX, значит вершина i недостижима

Этот алгоритм является модификацией классического BFS для работы с взвешенным графом и поиска кратчайших путей.

**Результат работы программы**

На рисунке 2 приведён результат работы программы №1.2.

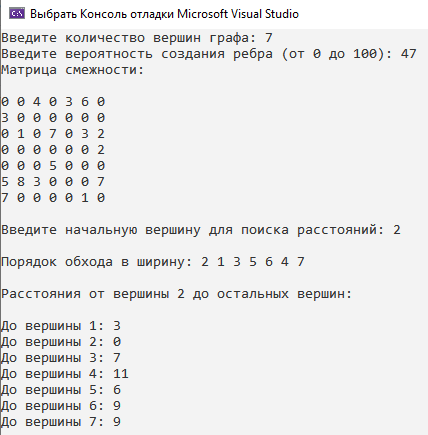


Рисунок 2 – результат работы программы №1.2

**Задание 2**

**1. Определение радиуса и диаметра для ориентированного и не ориентированного графов**

1. Функция find\_radius\_and\_diameter является основной для поиска радиуса и диаметра. Она использует несколько шагов:

void find\_radius\_and\_diameter(int size, int\_fast8\_t\*\* graph, int\* radius, int\* diameter) {

// Создание матрицы расстояний

int\*\* dist\_matrix = (int\*\*)malloc(sizeof(int\*) \* size);

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

dist\_matrix[i] = (int\*)malloc(sizeof(int) \* size);

}

// Применение алгоритма Флойда-Уоршелла

floyd\_warshall(size, graph, dist\_matrix);

\*radius = INT\_MAX;

\*diameter = 0;

// Находим минимальный и максимальный эксцентриситет

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

int ecc = eccentricity(i, size, dist\_matrix);

if (ecc != 0) {

if (ecc < \*radius) \*radius = ecc;

if (ecc > \*diameter) \*diameter = ecc;

}

}

}

1. Алгоритм использует функцию floyd\_warshall для нахождения кратчайших путей между всеми парами вершин:

void floyd\_warshall(int size, int\_fast8\_t\*\* graph, int\*\* distances) {

// Инициализация матрицы расстояний

// Поиск кратчайших путей между всеми парами вершин

}

1. Функция eccentricity вычисляет эксцентриситет вершины:

int eccentricity(int vertex, int size, int\*\* distances) {

int max\_distance = 0;

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

if (distances[vertex][i] != INT\_MAX &&

distances[vertex][i] > max\_distance) {

max\_distance = distances[vertex][i];

}

}

return max\_distance;

}

Основные понятия:

* Эксцентриситет вершины - это максимальное расстояние от данной вершины до любой другой вершины графа
* Радиус графа - минимальный эксцентриситет среди всех вершин
* Диаметр графа - максимальный эксцентриситет среди всех вершин

Алгоритм работает следующим образом:

1. Создаёт матрицу расстояний
2. Находит все кратчайшие пути с помощью алгоритма Флойда-Уоршелла
3. Вычисляет эксцентриситет для каждой вершины
4. Находит минимальный (радиус) и максимальный (диаметр) эксцентриситет

**2. Определение подмножества периферийных и центральных вершин для ориентированного и не ориентированного графов**

В программе периферийные и центральные вершины определяются с помощью двух функций:

1. Функция поиска периферийных вершин:

void find\_peripheral\_vertices(int size, int\_fast8\_t\*\* graph, int diameter) {

// Создание матрицы расстояний

int\*\* dist\_matrix = (int\*\*)malloc(sizeof(int\*) \* size);

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

dist\_matrix[i] = (int\*)malloc(sizeof(int) \* size);

}

// Применение алгоритма Флойда-Уоршелла

floyd\_warshall(size, graph, dist\_matrix);

// Поиск вершин с эксцентриситетом равным диаметру

printf("Периферийные вершины: ");

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

int eccentr = eccentricity(i, size, dist\_matrix);

if (eccentr == diameter) {

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

}

}

printf("\n");

}

1. Функция поиска центральных вершин:

void find\_central\_vertices(int size, int\_fast8\_t\*\* graph, int radius) {

// Создание матрицы расстояний

int\*\* dist\_matrix = (int\*\*)malloc(sizeof(int\*) \* size);

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

dist\_matrix[i] = (int\*)malloc(sizeof(int) \* size);

}

// Применение алгоритма Флойда-Уоршелла

floyd\_warshall(size, graph, dist\_matrix);

// Поиск вершин с эксцентриситетом равным радиусу

printf("Центральные вершины: ");

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

int eccentr = eccentricity(i, size, dist\_matrix);

if (eccentr == radius) {

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

}

}

printf("\n");

}

Основные принципы определения:

1. Периферийные вершины:

* Это вершины, эксцентриситет которых равен диаметру графа
* Имеют максимальное расстояние до самой удалённой вершины
* Находятся путём сравнения эксцентриситета каждой вершины с диаметром

1. Центральные вершины:

* Это вершины, эксцентриситет которых равен радиусу графа
* Имеют минимальное расстояние до самой удалённой вершины
* Находятся путём сравнения эксцентриситета каждой вершины с радиусом

Алгоритм для обоих случаев:

1. Создаёт матрицу расстояний
2. Применяет алгоритм Флойда-Уоршелла для нахождения кратчайших путей
3. Для каждой вершины:
   * Вычисляет её эксцентриситет
   * Сравнивает с радиусом (для центральных) или диаметром (для периферийных)
   * Если совпадает - включает вершину в соответствующее подмножество

Результатом работы этих функций является вывод номеров вершин, принадлежащих к периферийным или центральным подмножествам.

**Результат работы программы**

На рисунке 3 приведён результат работы программы №2.

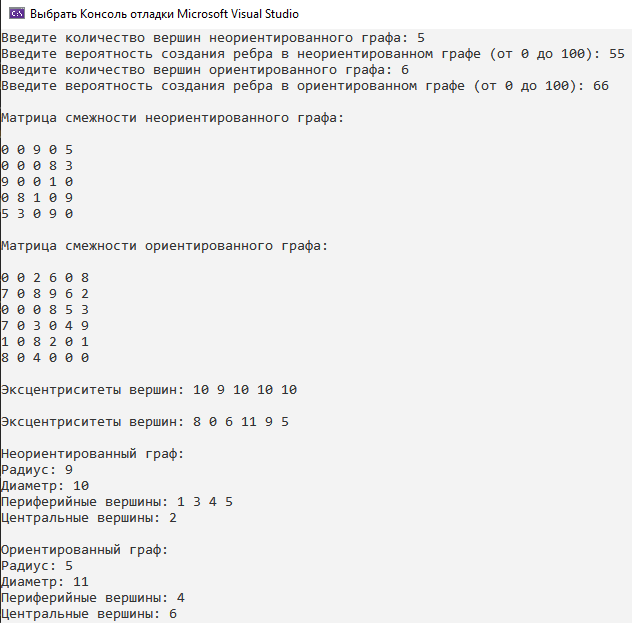


Рисунок 3 – результат работы программы №2

**Задание 3**

**Запуск программы с параметрами командной строки**

1. Формат запуска:

program.exe -w [yes|no] -o [yes|no]

1. Параметры командной строки:

* -w yes|no - определяет, будет ли граф взвешенным (yes) или невзвешенным (no)
* -o yes|no - определяет, будет ли граф ориентированным (yes) или неориентированным (no)

1. Примеры запуска:

program.exe -w yes -o no // создаст взвешенный неориентированный граф

program.exe -w no -o yes // создаст невзвешенный ориентированный граф

1. Проверка аргументов:

* Программа ожидает ровно 4 аргумента (не считая имени программы)
* Если количество аргументов неверное или они некорректны, выводится справка (функция print\_guide())

1. Результат работы:

* Создает матрицу смежности графа заданного типа
* Выводит её на экран
* Вычисляет и выводит радиус и диаметр графа
* Находит периферийные и центральные вершины

1. Константы программы:

* Size\_graph (6) - размер создаваемого графа
* Edge\_probability (50) - вероятность создания ребра между вершинами в процентах

Если программа запущена без аргументов или с неправильными аргументами, она выведет инструкцию по использованию.

**Результат работы программы**

На рисунке 4 приведён результат работы программы №3.

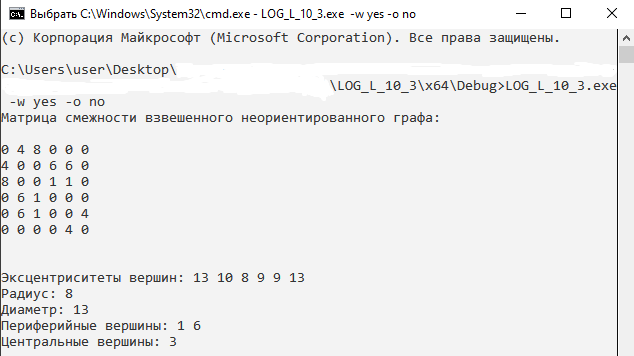


Рисунок 4 – результат работы программы №3

**Выводы**

Был реализован алгоритм поиска расстояний во взвешенных неориентированных и ориентированных графах, определения радиуса, диаметра, центральных и периферийных вершин, а также была осуществлена возможность запуска программы с параметрами командной строки.