Министерство науки и высшего образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

**ОТЧЁТ**

По лабораторной работе №5

По курсу «Логика и основы алгоритмизации в инженерных задачах»

на тему «Определение характеристик графов»

**Выполнили**

**студенты группы 23ВВВ2:**

Пырков Д. А.

Родионов А. А.

**Приняли:**

Митрохин М. А.

Юрова О. В.

Пенза 2024

**Цель работы:** Изучить понятия графа, матрицы смежности и инцидентности, реализовать алгоритм определения размера графа, а также поиска изолированных, концевых и доминирующих в нём вершин.

**Задание 1:**

1. Сгенерируйте (используя генератор случайных чисел) матрицу смежности для неориентированного графа *G*. Выведите матрицу на экран.
2. Определите размер графа *G*, используя матрицу смежности графа.
3. Найдите изолированные, концевые и доминирующие вершины.

**\*Задание 2:**

1. Постройте для графа G матрицу инцидентности.
2. Определите размер графа *G*, используя матрицу инцидентности графа.
3. Найдите изолированные, концевые и доминирующие вершины.

**Ход работы**

**Задание 1**

**Описание программы. Общие сведения**

Программа создает и анализирует случайный неориентированный граф, а также выводит на экран его матрицу смежности. Она позволяет пользователю перед созданием графа выбрать количество его вершин и определить вероятность создания рёбер между ними. Программа работает следующим образом:

1. Функция main():
   * Запрашивает у пользователя количество вершин графа и вероятность создания ребра.
   * Создает граф, инициализирует его, выводит матрицу смежности и информацию о графе.
   * В конце освобождает выделенную память.
2. create\_graph(int size):
   * Создает двумерный массив для представления графа в виде матрицы смежности.
3. initialize\_graph(int size, int\_fast8\_t\*\* graph\_pointer, float edge\_probability):
   * Заполняет матрицу смежности случайным образом, основываясь на заданной вероятности.
   * Граф неориентированный, поэтому матрица симметрична относительно главной диагонали.
4. print\_graph(int size, int\_fast8\_t\*\* graph\_pointer):
   * Выводит матрицу смежности на экран.
5. graph\_info(int size, int\_fast8\_t\*\* graph\_pointer):
   * Анализирует граф и выводит информацию о нем:  
      Размер графа (количество ребер);  
      Изолированные вершины (вершины без ребер);  
      Концевые вершины (вершины с одним ребром);  
      Доминирующие вершины (вершины, соединенные со всеми остальными).
   * Для каждой вершины проверяется:
     1. Является ли она изолированной (нет ребер).
     2. Является ли она концевой (одно ребро).
     3. Является ли она доминирующей (соединена со всеми остальными).
   * Подсчитывается общее количество ребер.
   * Результаты выводятся на экран.

**Описание программы. Функция создания графа**

Функция, генерирующая матрицу смежности для неориентированного графа, имеет вид:

void initialize\_graph(int size, int\_fast8\_t\*\* graph\_pointer, float edge\_probability) {

srand((unsigned int)time(NULL));

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

for (int j = i + 1; j < size; j++) {

if ((float)rand() / (RAND\_MAX / 100) < edge\_probability) {

graph\_pointer[i][j] = 1;

graph\_pointer[j][i] = 1;

}

else {

graph\_pointer[i][j] = 0;

graph\_pointer[j][i] = 0;

}

}

graph\_pointer[i][i] = 0;

}

}

1. Инициализация генератора случайных чисел:

srand((unsigned int)time(NULL));

Эта строка инициализирует генератор случайных чисел текущим временем, чтобы каждый запуск программы давал разные результаты.

2. Внешний цикл:

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

Этот цикл перебирает все вершины графа.

3. Внутренний цикл:

for (int j = i + 1; j < size; j++)

Этот цикл начинается с j = i + 1, чтобы заполнить только верхнюю треугольную часть матрицы смежности. Это используется для оптимизации, так как граф неориентированный и матрица симметрична.

4. Генерация случайного числа и сравнение с вероятностью:

if ((float)rand() / (RAND\_MAX / 100) < edge\_probability)

Здесь генерируется случайное число от 0 до 100 и сравнивается с заданной вероятностью. Если случайное число меньше вероятности, создается ребро.

5. Создание ребра:

graph\_pointer[i][j] = 1;

graph\_pointer[j][i] = 1;

Если условие выполняется, устанавливается 1 в соответствующих ячейках матрицы смежности. Нужно установить значение как для [i][j], так и для [j][i], чтобы сохранить симметрию матрицы.

6. Отсутствие ребра:

else {

graph\_pointer[i][j] = 0;

graph\_pointer[j][i] = 0;

}

Если условие не выполняется, устанавается 0, что означает отсутствие ребра.

7. Установка диагональных элементов:

graph\_pointer[i][i] = 0;

Эта строка устанавливает все диагональные элементы в 0, что означает отсутствие петель в графе.

Таким образом, функция initialize\_graph() создает случайный неориентированный граф без петель, где вероятность существования каждого ребра определяется параметром edge\_probability.

Листинг программы смотрите в Приложении А.

**Результат работы программы**

На рисунке приведён результат работы программы №1.

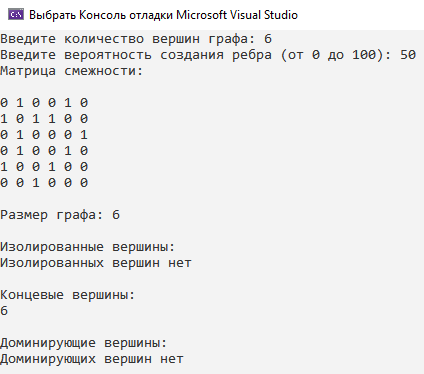


Рисунок 1 – результат работы программы №1

**\*Задание 2**

**Описание программы**

Программа создает и анализирует случайный неориентированный граф, а также выводит на экран его матрицу инцидентности. Она позволяет пользователю перед созданием графа выбрать количество его вершин и определить вероятность создания рёбер между ними. Программа работает следующим образом:

1. Функция main():
   * Запрашивает у пользователя количество вершин и вероятность создания ребра.
   * Вычисляет максимальное количество ребер для полного графа.
   * Создает и инициализирует граф.
   * Выводит матрицу инцидентности и информацию о графе.
   * Освобождает выделенную память.
2. create\_graph():  
   Создает двумерный массив для хранения матрицы инцидентности графа.
3. initialize\_graph():  
   Заполняет матрицу инцидентности случайным образом, учитывая заданную вероятность создания ребра.
4. print\_graph():  
   Выводит матрицу инцидентности на экран.
5. graph\_info():  
   Анализирует граф и выводит информацию о нем:
   * Размер графа (количество ребер).
   * Изолированные вершины (степень 0).
   * Концевые вершины (степень 1).
   * Доминирующие вершины (степень равна количеству вершин минус 1).

Особенности реализации:

* Граф представлен матрицей инцидентности, где строки соответствуют вершинам, а столбцы - ребрам.
* Используется тип int\_fast8\_t для экономии памяти.
* Программа использует динамическое выделение памяти и корректно ее освобождает.
* Реализована обработка ошибок ввода.
* Используется генератор случайных чисел для создания ребер с заданной вероятностью.

Алгоритм работы:

1. Пользователь вводит количество вершин и вероятность создания ребра.
2. Создается пустая матрица инцидентности.
3. Матрица заполняется случайным образом с учетом заданной вероятности.
4. Выводится матрица инцидентности.
5. Анализируется граф и выводится информация о нем.
6. Освобождается выделенная память.

Листинг программы смотрите в Приложении В.

**Результат работы программы**

На рисунке 2 приведён результат работы программы №2.

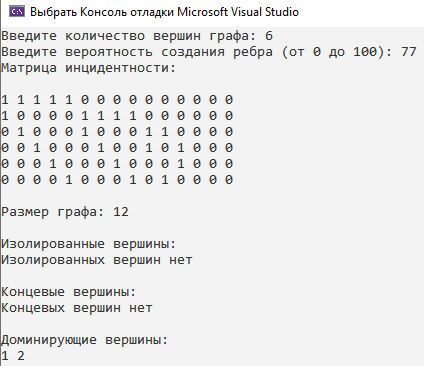


Рисунок 2 – результат работы программы №2

**Выводы**

Были изучены основные характеристики графов, реализованы алгоритмы определения размера графа, а также поиска изолированных, концевых и доминирующих в нём вершин для матриц смежности и инцидентности соответственно.

**Приложение А**

**Листинг программы для задания 1**

#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

#include <time.h>

#include <malloc.h>

#include <random>

#include <locale>

void initialize\_graph(int size, int\_fast8\_t\*\* graph\_pointer, float edge\_probability);

int\_fast8\_t\*\* create\_graph(int size);

void print\_graph(int size, int\_fast8\_t\*\* graph\_pointer);

void graph\_info(int size, int\_fast8\_t\*\* graph\_pointer);

int main() {

setlocale(LC\_ALL, "Russian");

int\_fast8\_t\*\* graph;

int size;

float edge\_probability;

fprintf(stdout, "Введите количество вершин графа: ");

if (!scanf("%d", &size) || size <= 1) {

fprintf(stderr, "Ошибка ввода размера массива");

exit(1);

}

fprintf(stdout, "Введите вероятность создания ребра (от 0 до 100): ");

if (!scanf("%f", &edge\_probability) || edge\_probability < 0 || edge\_probability > 100) {

fprintf(stderr, "Ошибка ввода вероятности");

exit(1);

}

graph = create\_graph(size);

initialize\_graph(size, graph, edge\_probability);

printf("Матрица смежности: \n\n");

print\_graph(size, graph);

graph\_info(size, graph);

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

free(graph[i]);

}

free(graph);

return 0;

}

int\_fast8\_t\*\* create\_graph(int size) {

int\_fast8\_t\*\* array = (int\_fast8\_t\*\*)malloc(sizeof(int\_fast8\_t\*) \* size);

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

array[i] = (int\_fast8\_t\*)malloc(sizeof(int\_fast8\_t) \* size);

}

if (array == NULL) {

fprintf(stderr, "Ошибка создания массива");

exit(1);

}

return array;

}

void initialize\_graph(int size, int\_fast8\_t\*\* graph\_pointer, float edge\_probability) {

srand((unsigned int)time(NULL));

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

for (int j = i + 1; j < size; j++) {

if ((float)rand() / (RAND\_MAX / 100) < edge\_probability) {

graph\_pointer[i][j] = 1;

graph\_pointer[j][i] = 1;

}

else {

graph\_pointer[i][j] = 0;

graph\_pointer[j][i] = 0;

}

}

graph\_pointer[i][i] = 0;

}

}

void print\_graph(int size, int\_fast8\_t\*\* graph\_pointer) {

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

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

fprintf(stdout, "%d ", graph\_pointer[j][i]);

}

fprintf(stdout, "\n");

}

}

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

int edge\_count = 0;

int\* isolated\_vertices = (int\*)calloc(size, sizeof(int));

int\* leaf\_vertices = (int\*)calloc(size, sizeof(int));

int\* dominating\_vertices = (int\*)calloc(size, sizeof(int));

int isolated\_count = 0, leaf\_count = 0, dominating\_count = 0;

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

int degree = 0;

int is\_isolated = 1;

int is\_dominating = 1;

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

if (graph\_pointer[i][j] == 1) {

is\_isolated = 0;

degree++;

if (i < j) {

edge\_count++;

}

}

else if (i != j) {

is\_dominating = 0;

}

}

if (is\_isolated) {

isolated\_vertices[isolated\_count++] = i + 1;

}

else if (degree == 1) {

leaf\_vertices[leaf\_count++] = i + 1;

}

if (is\_dominating) {

dominating\_vertices[dominating\_count++] = i + 1;

}

}

printf("\nРазмер графа: %d\n", edge\_count);

printf("\nИзолированные вершины:\n");

if (isolated\_count == 0) {

printf("Изолированных вершин нет\n");

}

else {

for (int i = 0; i < isolated\_count; i++) {

printf("%d ", isolated\_vertices[i]);

}

printf("\n");

}

printf("\nКонцевые вершины:\n");

if (leaf\_count == 0) {

printf("Концевых вершин нет\n");

}

else {

for (int i = 0; i < leaf\_count; i++) {

printf("%d ", leaf\_vertices[i]);

}

printf("\n");

}

printf("\nДоминирующие вершины:\n");

if (dominating\_count == 0) {

printf("Доминирующих вершин нет\n");

}

else {

for (int i = 0; i < dominating\_count; i++) {

printf("%d ", dominating\_vertices[i]);

}

printf("\n");

}

free(isolated\_vertices);

free(leaf\_vertices);

free(dominating\_vertices);

}

**Приложение B**

**Листинг программы для задания 2**

#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

#include <time.h>

#include <malloc.h>

#include <random>

#include <locale>

void initialize\_graph(int vertices, int edges, int\_fast8\_t\*\* graph\_pointer, float edge\_probability);

int\_fast8\_t\*\* create\_graph(int vertices, int edges);

void print\_graph(int vertices, int edges, int\_fast8\_t\*\* graph\_pointer);

void graph\_info(int vertices, int edges, int\_fast8\_t\*\* graph\_pointer);

int main() {

setlocale(LC\_ALL, "Russian");

int\_fast8\_t\*\* graph;

int vertices, edges;

float edge\_probability;

fprintf(stdout, "Введите количество вершин графа: ");

if (!scanf("%d", &vertices) || vertices <= 1) {

fprintf(stderr, "Ошибка ввода количества вершин");

exit(1);

}

fprintf(stdout, "Введите вероятность создания ребра (от 0 до 100): ");

if (!scanf("%f", &edge\_probability) || edge\_probability < 0 || edge\_probability > 100) {

fprintf(stderr, "Ошибка ввода вероятности");

exit(1);

}

edges = vertices \* (vertices - 1) / 2;

graph = create\_graph(vertices, edges);

initialize\_graph(vertices, edges, graph, edge\_probability);

printf("Матрица инцидентности: \n\n");

print\_graph(vertices, edges, graph);

graph\_info(vertices, edges, graph);

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

free(graph[i]);

}

free(graph);

return 0;

}

int\_fast8\_t\*\* create\_graph(int vertices, int edges) {

int\_fast8\_t\*\* array = (int\_fast8\_t\*\*)malloc(sizeof(int\_fast8\_t\*) \* vertices);

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

array[i] = (int\_fast8\_t\*)calloc(edges, sizeof(int\_fast8\_t));

}

if (array == NULL) {

fprintf(stderr, "Ошибка создания массива");

exit(1);

}

return array;

}

void initialize\_graph(int vertices, int edges, int\_fast8\_t\*\* graph\_pointer, float edge\_probability) {

srand((unsigned int)time(NULL));

int edge\_count = 0;

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

for (int j = i + 1; j < vertices; j++) {

if ((float)rand() / (RAND\_MAX / 100) < edge\_probability && edge\_count < edges) {

graph\_pointer[i][edge\_count] = 1;

graph\_pointer[j][edge\_count] = 1;

edge\_count++;

}

}

}

}

void print\_graph(int vertices, int edges, int\_fast8\_t\*\* graph\_pointer) {

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

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

fprintf(stdout, "%d ", graph\_pointer[i][j]);

}

fprintf(stdout, "\n");

}

}

void graph\_info(int vertices, int edges, int\_fast8\_t\*\* graph\_pointer) {

int edge\_count = 0;

int\* isolated\_vertices = (int\*)calloc(vertices, sizeof(int));

int\* leaf\_vertices = (int\*)calloc(vertices, sizeof(int));

int\* dominating\_vertices = (int\*)calloc(vertices, sizeof(int));

int isolated\_count = 0, leaf\_count = 0, dominating\_count = 0;

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

int degree = 0;

int is\_isolated = 1;

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

if (graph\_pointer[i][j] == 1) {

is\_isolated = 0;

degree++;

edge\_count++;

}

}

if (is\_isolated) {

isolated\_vertices[isolated\_count++] = i + 1;

}

else if (degree == 1) {

leaf\_vertices[leaf\_count++] = i + 1;

}

if (degree == vertices - 1) {

dominating\_vertices[dominating\_count++] = i + 1;

}

}

edge\_count /= 2;

printf("\nРазмер графа: %d\n", edge\_count);

printf("\nИзолированные вершины:\n");

if (isolated\_count == 0) {

printf("Изолированных вершин нет\n");

}

else {

for (int i = 0; i < isolated\_count; i++) {

printf("%d ", isolated\_vertices[i]);

}

printf("\n");

}

printf("\nКонцевые вершины:\n");

if (leaf\_count == 0) {

printf("Концевых вершин нет\n");

}

else {

for (int i = 0; i < leaf\_count; i++) {

printf("%d ", leaf\_vertices[i]);

}

printf("\n");

}

printf("\nДоминирующие вершины:\n");

if (dominating\_count == 0) {

printf("Доминирующих вершин нет\n");

}

else {

for (int i = 0; i < dominating\_count; i++) {

printf("%d ", dominating\_vertices[i]);

}

printf("\n");

}

free(isolated\_vertices);

free(leaf\_vertices);

free(dominating\_vertices);

}