Міністерство освіти і науки України

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Лабораторна робота №2.4

з дисципліни

«Алгоритми і структури даних»

Виконав:	Перевірила:
студент групи ІМ-13	Молчанова А. А.
Аненко Іван Ігорович	
номер у списку групи: 2	

Київ 2022

https://github.com/LikerFeed/ASD_Lab_2.4

Варіант 2

Загальна постановка задачі та завдання для конкретного варіанту

1. Представити напрямлений граф з заданими параметрами так само, як у лабораторній роботі №3.

Відміна: матриця A напрямленого графа за варіантом формується зафункціями:

srand(п1 п2 п3 п4);

T = randm(n,n);

A = mulmr((1.0 – n3*0.01 – n4*0.01 − 0.3)*T);

Перетворити граф у ненапрямлений.

2. Визначити степені вершин напрямленого і ненапрямленого графів.

Програма на екран виводить степені усіх вершин ненапрямленого графу і напівстепені виходу та заходу напрямленого графу. Визначити, чи граф є однорідним та якщо так, то вказати степінь однорідності графу.

3. Визначити всі висячі та ізольовані вершини. Програма на екран виводить перелік усіх висячих та ізольованих вершин графу.

4. Змінити матрицю графу за функцією

A = mulmr((1.0 – n3*0.005 – n4*0.005 − 0.27)*T);

Створити програму для обчислення наступних результатів:

1) матриця суміжності;

2) півстепені вузлів;

3) всі шляхи довжини 2 і 3;

4) матриця досяжності;

5) компоненти сильної зв’язності;

6) матриця зв’язності;

7) граф конденсації.

Шляхи довжиною 2 і 3 слід шукати за матрицями A2 і A3, відповідно. Матриця досяжності та компоненти сильної зв’язності слід шукати за допомогою операції транзитивного замикання.

Текст програми мовою C

main.c

```
#include <stdlib.h>
#include <stdio.h>
#include <gtk/gtk.h>
#include <cairo.h>
#include <math.h>
#include "../headers/matrix_tools.h"
#include "../headers/graph_operations.h"


int directed;
int node_shown;
const char APP_NAME_DIRECTED[] = "Directed Graph";
const char APP_NAME_UNDIRECTED[] = "Undirected Graph";
const int MARGIN = 50;
const int NODE_RADIUS = 35;
const int NODE_SPACING = NODE_RADIUS * 3;
const int LINE_WIDTH = 2;

const int CURVE_HOISTING = 20;

const int SELF_CONNECT_HOISTING = 20;

const int DOUBLE_OFFSET = 7;

const int OFFSET_MULTIPLIER_TILT = 8;

const int OFFSET_MULTIPLIER_CURVE = 2;

const int ARROW_LENGTH = 18;

const double ARROW_ANGLE = M_PI / 6;

const double WINDOW_HEIGHT_OFFSET_SIZE = 0.6;

int node_count;


typedef struct node_pos

{

    double x;

    double y;

}

node_pos_t;


typedef struct field

{

    int left;

    int right;

    int top;
    int bottom;
}
field_t;

int window_width;
int window_height;

struct field window_field;

double **matrix;

void draw_arrow(cairo_t *cr, double start_x, double start_y, double end_x,
double end_y)
{
    if (!directed)
    {
        return;
    }
    cairo_stroke(cr);
    double dx = start_x - end_x;
    double dy = start_y - end_y;
    double length = sqrt(dx * dx + dy * dy);
    double ratio = ARROW_LENGTH / length;
```

```
        cairo_new_path(cr);
        double x_first = end_x + ratio * (dx * cos(ARROW_ANGLE) + dy * sin(ARROW_ANGLE));
        double y_first = end_y + ratio * (dy * cos(ARROW_ANGLE) - dx * sin(ARROW_ANGLE));
        cairo_move_to(cr, x_first, y_first);
        cairo_line_to(cr, x_first, y_first);
        cairo_line_to(cr, end_x, end_y);
        double x_second = end_x + ratio * (dx * cos(ARROW_ANGLE) - dy * sin(ARROW_ANGLE));
        double y_second = end_y + ratio * (dy * cos(ARROW_ANGLE) + dx * sin(ARROW_ANGLE));
        cairo_line_to(cr, x_second, y_second);
        cairo_line_to(cr, x_first, y_first);
        cairo_close_path(cr);
        cairo_stroke_preserve(cr);
        cairo_fill(cr);
    }

    void connect_with_self(cairo_t *cr, node_pos_t node_n)
    {
        int y_offset_sign = node_n.y < window_height * WINDOW_HEIGHT_OFFSET_SIZE ? -1 : 1;
        double start_x = node_n.x;
        double start_y = node_n.y + y_offset_sign * NODE_RADIUS;
        double end_x = node_n.x + NODE_RADIUS;
        double end_y = node_n.y;
        double middle_x = end_x;
        double middle_y = end_y + y_offset_sign * SELF_CONNECT_HOISTING;
        cairo_move_to(cr, start_x, start_y);
        cairo_line_to(cr, start_x, start_y + y_offset_sign * SELF_CONNECT_HOISTING);
        cairo_line_to(cr, middle_x, middle_y);
        cairo_line_to(cr, end_x, end_y);
        draw_arrow(cr, middle_x, middle_y, end_x, end_y);
    }

    void connect_horizontal(cairo_t *cr, node_pos_t node_n, node_pos_t node_m, double offset)
    {
        double dx = node_m.x - node_n.x;
        if (fabs(dx) > NODE_SPACING * 2)
        {
            int y_offset_sign = node_n.y < window_height * WINDOW_HEIGHT_OFFSET_SIZE ? -1 : 1;
            double y_margin = y_offset_sign * CURVE_HOISTING;
            double x_margin = sqrt(NODE_RADIUS * NODE_RADIUS - y_margin * y_margin);
            x_margin = dx >= 0 ? x_margin : -x_margin;
            double start_x = node_n.x + x_margin;
            double start_y = node_n.y + y_margin;
            double middle_x = node_n.x + dx / 2;
            double middle_y = node_n.y + y_offset_sign * NODE_SPACING +
    y_margin + OFFSET_MULTIPLIER_CURVE * offset * DOUBLE_OFFSET;
            double end_x = node_m.x - x_margin;
            double end_y = node_m.y + y_margin;
            cairo_move_to(cr, start_x, start_y);
            cairo_curve_to(cr, start_x, start_y, middle_x, middle_y, end_x, end_y);
            draw_arrow(cr, middle_x, middle_y, end_x, end_y);
        }
        else
        {
            double y_margin = offset * DOUBLE_OFFSET;
            double x_margin = sqrt(NODE_RADIUS * NODE_RADIUS - y_margin
    * y_margin);
            x_margin = dx >= 0 ? x_margin : -x_margin;
            double start_x = node_n.x + x_margin;
            double start_y = node_n.y + y_margin;
            double end_x = node_m.x - x_margin;
            double end_y = node_m.y + y_margin;
            cairo_move_to(cr, start_x, start_y);
```

```
            cairo_line_to(cr, end_x, end_y);
            draw_arrow(cr, start_x, start_y, end_x, end_y);
        }
    }

    void connect_vertical(cairo_t *cr, node_pos_t node_n, node_pos_t node_m, double offset)
    {
        double dy = node_m.y - node_n.y;
        if (fabs(dy) > NODE_SPACING * 2)
        {
            double x_margin = -CURVE_HOISTING;
            double y_margin = sqrt(NODE_RADIUS * NODE_RADIUS - x_margin * x_margin);
            y_margin = dy >= 0 ? y_margin : -y_margin;
            double start_x = node_n.x + x_margin;
            double start_y = node_n.y + y_margin;
            double middle_x = node_n.x - NODE_SPACING * 2 - x_margin + OFFSET_MULTIPLIER_CURVE * offset * DOUBLE_OFFSET;
            double middle_y = node_n.y + dy / 2;
            double end_x = node_m.x + x_margin;
            double end_y = node_m.y - y_margin;
            cairo_move_to(cr, start_x, start_y);
            cairo_curve_to(cr, start_x, start_y, middle_x, middle_y, end_x, end_y);
            draw_arrow(cr, middle_x, middle_y, end_x, end_y);
        }
        else
        {
            double x_margin = offset * DOUBLE_OFFSET;
            double y_margin = sqrt(NODE_RADIUS * NODE_RADIUS - x_margin * x_margin);
            y_margin = dy >= 0 ? y_margin : -y_margin;
            double start_x = node_n.x + x_margin;
            double start_y = node_n.y + y_margin;
            double end_x = node_m.x + x_margin;
            double end_y = node_m.y - y_margin;
            cairo_move_to(cr, start_x, start_y);
            cairo_line_to(cr, end_x, end_y);
            draw_arrow(cr, start_x, start_y, end_x, end_y);
        }
    }

    void connect_tilted(cairo_t *cr, node_pos_t node_n, node_pos_t node_m, double offset)
    {
        double dx = node_m.x - node_n.x;
        double dy = node_m.y - node_n.y;
        double tangent = (double) fabs(dx) / fabs(dy);
        double y_margin = sqrt((NODE_RADIUS * NODE_RADIUS) / (1 + tangent * tangent));
        double x_margin = y_margin * tangent;
        y_margin = dy >= 0 ? y_margin : -y_margin;
        x_margin = dx >= 0 ? x_margin : -x_margin;
        double start_x = node_n.x + x_margin;
        double start_y = node_n.y + y_margin;
        double middle_x = node_n.x + dx / 2 + OFFSET_MULTIPLIER_TILT * offset * DOUBLE_OFFSET;
        double middle_y = node_n.y + dy / 2;
        double end_x = node_m.x - x_margin;
        double end_y = node_m.y - y_margin;
        cairo_move_to(cr, start_x, start_y);
        cairo_line_to(cr, middle_x, middle_y);
        cairo_line_to(cr, end_x, end_y);
        draw_arrow(cr, middle_x, middle_y, end_x, end_y);
    }

    void connect_nodes(cairo_t *cr, node_pos_t node_n, node_pos_t node_m, double offset)
    {
```

```
    if (node_n.x == node_m.x)
    {
        if (node_n.y == node_m.y)
        {
            connect_with_self(cr, node_n);
        }
        else
        {
            connect_vertical(cr, node_n, node_m, offset);
        }
    }
    else if (node_n.y == node_m.y)
    {
        connect_horizontal(cr, node_n, node_m, offset);
    }
    else
    {
        connect_tilted(cr, node_n, node_m, offset);
    }
    cairo_stroke(cr);
}
```

```
void set_side_positions(node_pos_t *positions, int node_count, int *index,
node_pos_t(*get_pos)(int))
{
    int spaced = 0;
    for (int i = 0; i < node_count; i++)
    {
        node_pos_t pos = get_pos(spaced);
        positions[*index] = pos;
        *index += 1;
        spaced++;
    }
}

node_pos_t get_top_position(int spaced)
{
    node_pos_t pos;
    pos.x = MARGIN + (NODE_RADIUS * 2 + NODE_SPACING) * spaced +
NODE_RADIUS;
    pos.y = MARGIN + NODE_RADIUS;
    return pos;
}

node_pos_t get_right_position(int spaced)
```

```
    {
        node_pos_t pos;
        pos.x = window_width - MARGIN - NODE_RADIUS;
        pos.y = MARGIN + (NODE_RADIUS * 2 + NODE_SPACING) * (spaced + 1) + NODE_RADIUS;
        return pos;
    }

    node_pos_t get_bottom_position(int spaced)
    {
        node_pos_t pos;
        pos.x = window_width - (MARGIN + (NODE_RADIUS * 2 + NODE_SPACING) * (spaced + 1)) - NODE_RADIUS;
        pos.y = window_height - MARGIN - NODE_RADIUS;
        return pos;
    }

    node_pos_t get_left_position(int spaced)
    {
        node_pos_t pos;
        pos.x = MARGIN + NODE_RADIUS;
        pos.y = window_height - (MARGIN + (NODE_RADIUS * 2 + NODE_SPACING) * (spaced + 1)) - NODE_RADIUS;
        return pos;
    }

    node_pos_t get_center_position(int spaced)
    {
        node_pos_t pos;
        pos.x = window_width / 2;
        pos.y = window_height / 2;
        return pos;
    }

    node_pos_t *get_node_positions()
    {
        node_pos_t *positions = (node_pos_t *)malloc(sizeof(node_pos_t) *
node_count);

        int index = 0;
        set_side_positions(positions, window_field.top + 1, &index,
get_top_position);

        set_side_positions(positions, window_field.right - 1, &index,
get_right_position);
```

```
    set_side_positions(positions, window_field.bottom - 1, &index, get_bottom_position);

    set_side_positions(positions, window_field.left - 2, &index, get_left_position);


    return positions;
}

void draw_connections(cairo_t *cr, node_pos_t *positions, double **matrix) {
    int start_index = node_shown == -1 ? 0 : node_shown;
    int end_index = node_shown == -1 ? node_count : node_shown + 1;
    for (int i = start_index; i < end_index; i++) {
        for (int j = 0; j < node_count; j++) {
            if (!matrix[i][j]) continue;
            if (directed && i != j && matrix[j][i] == 1) {
                if (i < j || j < start_index) {
                    connect_nodes(cr, positions[i], positions[j], 1);
                    if (j < end_index && j >= start_index) {
                        connect_nodes(cr, positions[j], positions[i], -1);
                    }
                }
            }
            else if (directed || i <= j)
            {
                connect_nodes(cr, positions[i], positions[j],
                              j == (node_count - 1) && !directed ? 1 : 0);
            }
        }
    }
}

void draw_node(cairo_t *cr, node_pos_t pos, char *text)
{
    cairo_move_to(cr, pos.x + NODE_RADIUS, pos.y);
    cairo_arc(cr, pos.x, pos.y, NODE_RADIUS, 0, 2 * M_PI); //малюємо еліпс
    cairo_stroke_preserve(cr);
    cairo_set_font_size(cr, NODE_RADIUS);
    if (strlen(text) > 1) {
        cairo_move_to(cr, pos.x - NODE_RADIUS / 2, pos.y + NODE_RADIUS
/ 3);
    } else {
        cairo_move_to(cr, pos.x - NODE_RADIUS / 3.5, pos.y +
NODE_RADIUS / 3);
    }
    cairo_show_text(cr, text);
}

void draw_nodes(cairo_t *cr, node_pos_t *positions)
{
    for (int i = 1; i <= node_count; i++)
    {
        char text[3];
        sprintf(text, "%d", i);
        draw_node(cr, positions[i - 1], text);
    }
}

void draw_graph(cairo_t *cr, double **matrix, node_pos_t *positions)
{
    cairo_set_source_rgb(cr, 0, 1, 1);
    draw_nodes(cr, positions);
    draw_connections(cr, positions, matrix);
    free(positions);
}

void set_window_size()
{
    if (node_count > 5) {
        window_height = 2 * MARGIN + window_field.right * NODE_RADIUS * 2 + NODE_SPACING * (window_field.right - 1);
        window_width = 2 * MARGIN + window_field.bottom * NODE_RADIUS * 2 + NODE_SPACING * (window_field.bottom - 1);
    }
    else
    {
        window_width = 2 * MARGIN + node_count * NODE_RADIUS * 2 + NODE_SPACING * (node_count - 1);
        window_height = 2 * MARGIN + NODE_RADIUS * 2;
    }
}

void calculate_size()
{
    int free_count = node_count - 4 - 1;
    int vertical = 2 + free_count / 4;
    window_field.left = vertical;
    window_field.right = vertical;
    window_field.top = vertical;
    window_field.bottom = vertical;
    int lefover = free_count % 4;
    window_field.top += lefover / 2;
    window_field.bottom += lefover - lefover / 2;
    set_window_size();
}

static gboolean on_draw_event(GtkWidget *widget, cairo_t *cr, gpointer user_data)
{
    node_pos_t *positions = get_node_positions();
    draw_graph(cr, matrix, positions);
    return FALSE;
}

GtkWidget *create_window(GtkApplication *app)
{
    GtkWidget *window = gtk_application_window_new(app);
    gtk_window_set_title(GTK_WINDOW(window), directed ? APP_NAME_DIRECTED : APP_NAME_UNDIRECTED);
    gtk_window_set_default_size(GTK_WINDOW(window), window_width, window_height);
    return window;
}
```

```
GtkWidget *create_darea(GtkWidget *window)
{
   GtkWidget *darea = gtk_drawing_area_new();
   gtk_container_add(GTK_CONTAINER(window), darea);
   g_signal_connect(G_OBJECT(darea),                         "draw",
G_CALLBACK(on_draw_event), NULL);
   g_signal_connect(window,   "destroy",   G_CALLBACK(gtk_main_quit),
NULL);
   return darea;
}

void on_app_activate(GtkApplication *app, gpointer data)
{
   GtkWidget *window = create_window(app);
   GtkWidget *darea = create_darea(window);
   gtk_widget_show_all(window);
   gtk_main();
}

void create_application(int argc, char *argv[])
{
   GtkApplication     *app     =     gtk_application_new("ivan.anenko",
G_APPLICATION_FLAGS_NONE);
```

```
    g_signal_connect(app, "activate", G_CALLBACK(on_app_activate), NULL);

    g_application_run(G_APPLICATION(app), argc, argv);

}


void directed_read()

{

    printf("Output directed graph? (0 for no, any other number for yes)\n");

    scanf("%d", &directed);

    directed = !directed ? 0 : 1;

}


void node_read()

{

    node_shown = -2;

    while (node_shown < -1 || node_shown >= node_count)

    {

        printf("What node connections to show? (input index of node or -1 to show all node connections)\n");

        scanf("%d", &node_shown);

    }

}
```

```
int type_read() {
    int type = 0;
    while (type != 1 && type != 2)
    {
        printf("Input type of matrix:\n");
        scanf("%d", &type);
    }
    return type;
}

int condensed_read() {
    int condensed;
    printf("Condense the graph?\n");
    scanf("%d", &condensed);
    return !condensed ? 0 : 1;
}

int main(int argc, char *argv[])
{
    directed_read();
    node_read();
    int type = type_read();
    int condensed = 0;
}

if (directed) {
    condensed = condensed_read();
}


matrix = get_matrix(type);
node_count = NODE_COUNT;
if (!directed)
{
    to_undirected(matrix);
}


printf("\nOriginal matrix:\n");
output_matrix(node_count, node_count, matrix);


if (condensed)
{
    double **temp = matrix;
    matrix = get_condensed_matrix(node_count, matrix);
    free_matrix(node_count, temp);
    node_count = get_condensed_matrix_size();
}


set_directed(directed);
```

```
    set_count(node_count);
    if (!condensed) additional_output(type, matrix);
    calculate_size();
    create_application(argc, argv);
    free_matrix(node_count, matrix);
    return 1;
}
```

matrix_tools.c

```
#include <stdlib.h>
#include <stdio.h>

const int RAND_LIMIT = 2;
const int N1 = 1;
const int N2 = 3;
const int N3 = 0;
const int N4 = 2;
const int NODE_COUNT = 10 + N3;

double get_seed() {
    return N1 * 1000 + N2 * 100 + N3 * 10 + N4;
}

double get_coef(int type) {
    if (type == 2) {
        return 1 - N3 * 0.005 - N4 * 0.005 - 0.27;
    } else {
        return 1 - N3 * 0.01 - N4 * 0.01 - 0.3;
    }
}

double ranged_rand() {
    return (double)rand() / ((double)RAND_MAX / RAND_LIMIT);
}

double **randm(int n, int m) {
    double **matrix = (double **)malloc(sizeof(double *) * n);
    for (int i = 0; i < n; i++) {
        double *row = (double *)malloc(sizeof(double) * m);
        matrix[i] = row;
        for (int j = 0; j < m; j++) {
            row[j] = ranged_rand();
        }
    }
    return matrix;
}
```

```
void mulmr(double coef, int n, int m, double **matrix) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            matrix[i][j] = matrix[i][j] * coef >= 1 ? 1 : 0;
        }
    }
}

void output_matrix(int n, int m, double **matrix) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            printf("%.0f ", matrix[i][j]);
        }
        printf("\n");
    }
}

void to_undirected(double **matrix) {
    for (int i = 0; i < NODE_COUNT; i++) {
        for (int j = 0; j < NODE_COUNT; j++) {
            if (matrix[i][j] == 1) {
                matrix[j][i] = 1;
            }
        }
    }
}

double **get_matrix(int type) {
    srand(get_seed());
    double **matrix = randm(NODE_COUNT, NODE_COUNT);
    mulmr(get_coef(type), NODE_COUNT, NODE_COUNT, matrix);
    return matrix;
}

void free_matrix(int n, double **matrix) {
    for (int i = 0; i < n; i++) {
        free(matrix[i]);
    }
    free(matrix);
}
```

graph_operations.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "../headers/matrix_tools.h"
```

```
}

int is_directed;

int node_count;

int condensed_size;


double **new_matrix(int n) {

    double **matrix = (double **)malloc(sizeof(double *) * n);

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

        double *row = (double *)malloc(sizeof(double) * n);

        matrix[i] = row;

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

            row[j] = 0;

        }

    }

    return matrix;

}


int *new_array(int n) {

    int *array = (int *)malloc(n * sizeof(int));

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

        array[i] = 0;

    }

    return array;

}
```

```
double **matrix_square_multiply(int n, double **a, double **b) {
    double **multiplied = new_matrix(n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                multiplied[i][j] += a[k][j] * b[i][k];
            }
        }
    }
    return multiplied;
}

double **matrix_transpose(int n, double **matrix) {
    double **transpose = new_matrix(n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            transpose[i][j] = matrix[j][i];
        }
    }

    return transpose;
}
```

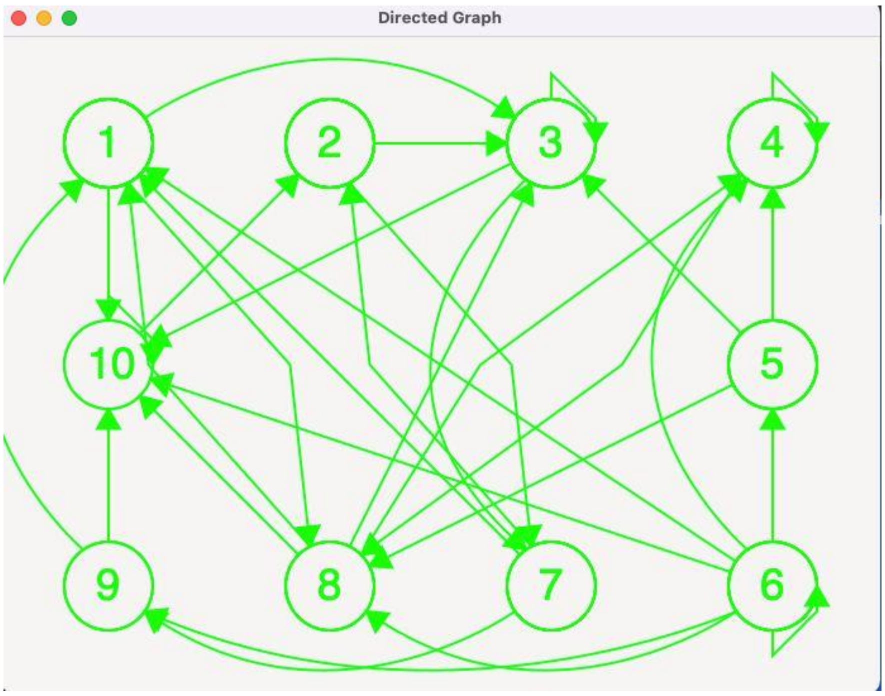
```
void set_directed(int directed) {
    is_directed = directed;
}

void set_count(int n) {
    node_count = n;
}

void output_array(int n, int array[n]) {
    if (n == 0) {
        printf("none\n");
        return;
    }
    printf("[");
    for (int i = 0; i < n; i++) {
        if (i < n - 1) {
            printf("%d, ", array[i]);
        }
        else {
            printf("%d]\n", array[i]);
        }
    }
```

```c
}


int get_component_num(int n, int *components) {
    int component_num = 0; //шукаємо максимум
    for (int i = 0; i < n; i++) {
        if (components[i] > component_num) {
            component_num = components[i];
        }
    }
    return component_num;
}


void output_components(int n, int *components) {
    int component_num = get_component_num(n, components);
    for (int i = 1; i <= component_num; i++) {
        printf("Component %d: [ ", i);
        for (int j = 0; j < n; j++) {
            if (components[j] == i) {
                printf("%d ", j + 1);
            }
        }
        printf("]\n");
    }
```

```
}


int *get_undirected_degrees(int n, double **matrix) {
    int *degrees = new_array(n);
    for (int i = 0; i < n; i++) {
        degrees[i] = 0;
        for (int j = 0; j < n; j++) {
            if (matrix[i][j]) {
                degrees[i]++;
            }
        }
    }

    return degrees;
}


int *get_in_degrees(int n, double **matrix) {
    int *degrees = new_array(n);
    for (int j = 0; j < n; j++) {
        degrees[j] = 0;
        for (int i = 0; i < n; i++) {
            if (matrix[i][j]) {
                degrees[j]++;
            }
        }
    }
    return degrees;
}

void output_node_types(int n, int *degrees) {
    int leaf_count = 0;
    int isolated_count = 0;
    int leaves[n];
    int isolated[n];
    for (int i = 0; i < n; i++) {
        if (degrees[i] == 0) {
            isolated[isolated_count] = i + 1;
            isolated_count++;
        }
        else if (degrees[i] == 1) {
            leaves[leaf_count] = i + 1;
            leaf_count++;
        }
    }
    printf("Isolated: ");
    output_array(isolated_count, isolated);
    printf("Leaves: ");
    output_array(leaf_count, leaves);
}


void output_is_regular(int n, int *degrees) {
    int is_regular = 1;
    for (int i = 1; i < n; i++) {
        if (degrees[i] != degrees[i - 1]) {
            is_regular = 0;
            break;
        }
    }
    if (is_regular) {
        printf("\nThe graph is regular. It`s degree is %d\n\n", degrees[0]);
    }
    else {
        printf("\nThe graph is not regular\n\n");
    }
}


int *get_total(int n, int *out_degrees, int *in_degrees) {
    int *total = new_array(n * 2);
    memcpy(total, out_degrees, n * sizeof(int));
    memcpy(total + n, in_degrees, n * sizeof(int));
    return total;
}

int *get_summed(int n, int *out_degrees, int *in_degrees) {
    int *summed = new_array(n);
    for (int i = 0; i < n; i++) {
        summed[i] = out_degrees[i] + in_degrees[i];
    }
    return summed;
}

void output_directed_degree_info(int n, double **matrix) {
    printf("Out degrees: ");
    int *out_degrees = get_undirected_degrees(n, matrix);
    output_array(n, out_degrees);
    printf("In degrees: ");
    int *in_degrees = get_in_degrees(n, matrix);
    output_array(n, in_degrees);
    int* total = get_total(n, out_degrees, in_degrees);
    output_is_regular(n * 2, total);
    free(total);
    int* summed = get_summed(n, out_degrees, in_degrees);
    output_node_types(n, summed);
    free(summed);
    free(out_degrees);
    free(in_degrees);
}

void output_degree_operations(double **matrix) {
    printf("\nDegrees of every node:\n");
    if (!is_directed) {
        int *degrees = get_undirected_degrees(node_count, matrix);
        output_array(node_count, degrees);
        output_is_regular(node_count, degrees);
        output_node_types(node_count, degrees);
        free(degrees);
    } else {
        output_directed_degree_info(node_count, matrix);
    }
}

void get_paths(int length, int n, double **matrix) {
    if (length < 1 || length > 3) {
        printf("unsupported length!\n");
        return;
}
printf("{\n");
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (!matrix[i][j]) continue;
        if (length == 1) {
            printf("%d->%d\n", i + 1, j + 1);
            continue;
        }
        for (int k = 0; k < n; k++) {
            if (!matrix[j][k]) continue;
            if (length == 2) {
                printf("%d->%d->%d\n", i + 1, j + 1, k + 1);
                continue;
            }
            for (int l = 0; l < n; l++) {
                if (!matrix[k][l]) continue;
                printf("%d->%d->%d->%d\n", i + 1, j + 1, k + 1, l + 1);
            }
        }
    }
}
printf("}\n");
```

```
}

void output_path_operations(double **matrix) {
    printf("\nMatrix of power 2: \n");
    double **matrix_2 = matrix_square_multiply(node_count, matrix, matrix);
    output_matrix(node_count, node_count, matrix_2);
    printf("\nPaths of length 2: \n");
    get_paths(1, node_count, matrix_2);
    printf("\nMatrix of power 3: \n");
    double **matrix_3 = matrix_square_multiply(node_count, matrix, matrix_2);
    output_matrix(node_count, node_count, matrix_3);
    printf("\nPaths of length 3: \n");
    get_paths(1, node_count, matrix_3);
}

void search_components(double **matrix, int index, int component_num, int *used, int *components) {
    used[index] = 1;
    components[index] = component_num;
    for (int i = 0; i < node_count; i++) {
        if (!used[i] && matrix[index][i]) {
            search_components(matrix, i, component_num, used, components);
        }
    }
}

int *get_connection_components(int n, double **matrix) {
    int component_num = 1;
    int *used = new_array(n);
    for (int i = 0; i < n; i++) {
        used[i] = 0;
    }
    int *components = new_array(n);
    for (int i = 0; i < n; i++) {
        if (!used[i]) {
            search_components(matrix, i, component_num, used, components);
            component_num++;
        }
    }

    free(used);
    return components;
}

double **get_connection_matrix(int n, double **reachability) {
    double **transpose = matrix_transpose(n, reachability);
    double **connection_matrix = new_matrix(n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            connection_matrix[i][j] = reachability[i][j] && transpose[i][j];
        }
    }
    free(transpose);
    return connection_matrix;
}

double **get_reachability_matrix(int n, double **matrix) {
    double **reachability = new_matrix(n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            reachability[i][j] = matrix[i][j];
        }
    }
    double **powered = matrix_square_multiply(node_count, matrix, matrix);
    for (int time = 0; time < n - 1; time++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                reachability[i][j] = reachability[i][j] || powered[i][j];
            }
        }
        double **temp = matrix_square_multiply(node_count, powered, matrix);
        free(powered);
        powered = temp;
    }

    return reachability;
}

int get_condensed_matrix_size() {
    return condensed_size;
}

double **get_condensed_matrix(int n, double **matrix) {
    double **reachability = get_reachability_matrix(n, matrix);
    double **connection_matrix = get_connection_matrix(n, reachability);
    int *components = get_connection_components(n, connection_matrix);
    output_components(n, components);
    free_matrix(n, connection_matrix);
    free_matrix(n, reachability);
    int component_num = get_component_num(n, components);
    condensed_size = component_num;
    double **condensed = new_matrix(component_num);
    for (int i = 0; i < component_num; i++) {
        for (int j = 0; j < n; j++) {
            if (components[j] == i + 1) {
                for (int k = 0; k < n; k++) {
                    if (components[k] != i + 1) {
                        if (matrix[j][k]) {
                            condensed[i][components[k] - 1] = 1;
                        } else if (matrix[k][j]) {
                            condensed[components[k] - 1][i] = 1;
                        }
                    }
                }
            }
        }
    }
    printf("\n");
    output_matrix(component_num, component_num, condensed);
    free(components);
    return condensed;
}

void output_reachability_operations(double **matrix) {
    double **reachability = get_reachability_matrix(node_count, matrix);
    printf("\nMatrix of accessiblenes:\n");
    output_matrix(node_count, node_count, reachability);
    double **connection_matrix = get_connection_matrix(node_count, reachability);
    printf("\nConnections (matrix):\n");
    output_matrix(node_count, node_count, connection_matrix);
    int *components = get_connection_components(node_count, connection_matrix);
    printf("\nComponents:\n");
    output_components(node_count, components);
    free_matrix(node_count,reachability);
    free_matrix(node_count, connection_matrix);
    free(components);
}


void additional_output(int type, double **matrix) {
    output_degree_operations(matrix);
    if (type == 2) {
        output_path_operations(matrix);
        output_reachability_operations(matrix);
    }
}
```

Результати тестування програми



Output directed graph? (0 for no, any other number for yes)

1

What node connections to show? (input index of node or -1 to show all node connections)

-1

Input type of matrix:

1

Condense the graph?

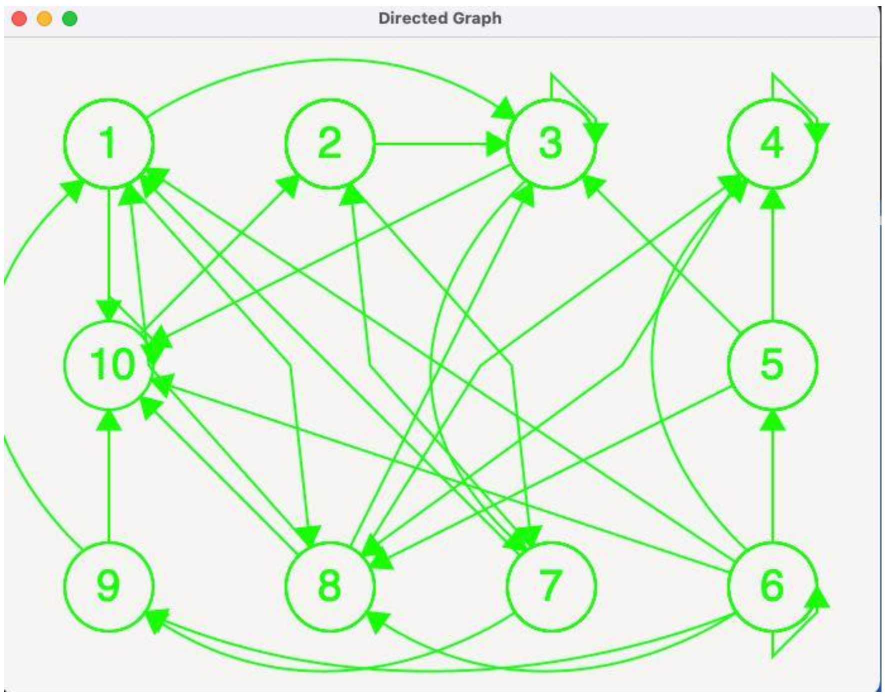
0

Original matrix:

0 0 1 0 0 0 0 1 0 1

0 0 1 0 0 0 1 0 0 0

```
0 0 1 0 0 0 1 0 0 1

0 0 0 1 0 0 0 1 0 0

0 0 1 1 0 0 0 1 0 0

1 0 0 1 1 1 0 1 1 1

1 1 0 0 0 0 0 0 1 0

1 0 1 1 0 0 0 0 0 1

1 0 0 0 0 0 0 0 0 1

0 1 0 0 0 0 0 0 0 1


Degrees of every node:

Out degrees: [3, 2, 3, 2, 3, 7, 3, 4, 2, 2]

In degrees: [4, 2, 5, 4, 1, 1, 2, 4, 2, 6]


The graph is not regular


Isolated: none

Leaves: none
```



Output directed graph? (0 for no, any other number for yes)

1

What node connections to show? (input index of node or -1 to show all node connections)

-1

Input type of matrix:

2

Condense the graph?

0

Original matrix:

0 0 1 0 0 0 0 1 0 1

0 0 1 0 0 0 1 0 0 0

0 0 1 0 1 0 1 0 0 1

```
0 0 1 1 0 0 1 1 0 0

0 0 1 1 0 0 0 1 0 0

1 0 0 1 1 1 0 1 1 1

1 1 0 0 0 0 0 0 1 0

1 0 1 1 1 1 0 0 0 1

1 0 0 0 0 0 0 0 0 1

0 1 0 0 0 0 0 0 0 1


Degrees of every node:

Out degrees: [3, 2, 4, 4, 3, 7, 3, 6, 2, 2]

In degrees: [4, 2, 6, 4, 3, 2, 3, 4, 2, 6]


The graph is not regular


Isolated: none

Leaves: none


Matrix of power 2:

1 1 2 1 2 1 1 0 0 3

1 1 1 0 1 0 1 0 1 1

1 2 2 1 1 0 1 1 1 2

2 1 3 2 2 1 2 1 1 2

1 0 3 2 2 1 2 1 0 2
```

```
3 1 4 4 2 2 1 4 1 5

1 0 2 0 0 0 1 1 0 2

1 1 4 3 2 1 2 4 1 4

0 1 1 0 0 0 0 1 0 2

0 1 1 0 0 0 1 0 0 1


Paths of length 2:

{

1->1

1->2

1->3

1->4

1->5

1->6

1->7

1->10

2->1

2->2

2->3

2->5

2->7

2->9

2->10
```

3->1

3->2

3->3

3->4

3->5

3->7

3->8

3->9

3->10

4->1

4->2

4->3

4->4

4->5

4->6

4->7

4->8

4->9

4->10

5->1

5->3

5->4

5->5

5->6

5->7

5->8

5->10

6->1

6->2

6->3

6->4

6->5

6->6

6->7

6->8

6->9

6->10

7->1

7->3

7->7

7->8

7->10

8->1

8->2

8->3

8->4

```
8->5

8->6

8->7

8->8

8->9

8->10

9->2

9->3

9->8

9->10

10->2

10->3

10->7

10->10

}


Matrix of power 3:

2 4 7 4 3 1 4 5 2 7

2 2 4 1 1 0 2 2 1 4

3 3 8 3 3 1 5 3 1 7

5 4 11 6 5 2 6 7 3 10

4 4 9 6 5 2 5 6 3 8

8 6 18 12 10 6 9 11 3 19
```

```
2 3 4 1 3 1 2 1 1 6

8 6 15 10 9 5 8 7 3 15

1 2 3 1 2 1 2 0 0 4

1 2 2 0 1 0 2 0 1 2


Paths of length 3:

{

1->1

1->2

1->3

1->4

1->5

1->6

1->7

1->8

1->9

1->10

2->1

2->2

2->3

2->4

2->5

2->7
```

2->8

2->9

2->10

3->1

3->2

3->3

3->4

3->5

3->6

3->7

3->8

3->9

3->10

4->1

4->2

4->3

4->4

4->5

4->6

4->7

4->8

4->9

4->10

5->1

5->2

5->3

5->4

5->5

5->6

5->7

5->8

5->9

5->10

6->1

6->2

6->3

6->4

6->5

6->6

6->7

6->8

6->9

6->10

7->1

7->2

7->3

7->4

7->5

7->6

7->7

7->8

7->9

7->10

8->1

8->2

8->3

8->4

8->5

8->6

8->7

8->8

8->9

8->10

9->1

9->2

9->3

9->4

9->5

9->6

```
9->7
9->10
10->1
10->2
10->3
10->5
10->7
10->9
10->10
}

Matrix of accessiblenes:
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
```

```
Connections (matrix):
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1

Components:
Component 1: [ 1 2 3 4 5 6 7 8 9 10 ]
```

```
Output directed graph? (0 for no, any other number for yes)
1
What node connections to show? (input index of node or -1 to show all node connections)
-1
Input type of matrix:
```

```
2

Condense the graph?

1


Original matrix:

0 0 1 0 0 0 0 1 0 1

0 0 1 0 0 0 1 0 0 0

0 0 1 0 1 0 1 0 0 1

0 0 1 1 0 0 1 1 0 0

0 0 1 1 0 0 0 1 0 0

1 0 0 1 1 1 0 1 1 1

1 1 0 0 0 0 0 0 1 0

1 0 1 1 1 1 0 0 0 1

1 0 0 0 0 0 0 0 0 1

0 1 0 0 0 0 0 0 0 1

Component 1: [ 1 2 3 4 5 6 7 8 9 10 ]


0
```