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 №2.3

з дисципліни

«Алгоритми і структури даних»

Виконав:

студент групи ІМ-13

Аненко Іван Ігорович

номер у списку групи: 2

Перевірила:

Молчанова А. А.

Київ 2022

https://github.com/LikerFeed/ASD_Lab_2.3

Загальна постановка задачі та завдання для конкретного варіанту

1. Представити у програмі напрямлений і ненапрямлений графи з заданими параметрами:

— число вершин n;

— розміщення вершин;

— матриця суміжності A.

Параметри задаються на основі номера групи, представленого десятковими цифрами п1 = 1, п2 = 3 та номера студента у списку групи — десяткового числа п3 = 0, п4 = 2.

Число вершин п дорівнює 10 + п3.

Розміщення вершин:

— прямокутником (квадратом) при п4 = 2,3;

Текст програми мовою C

main.c

```
#include <stdlib.h>

#include <stdio.h>

#include <gtk/gtk.h>

#include <cairo.h>

#include <math.h>

#include "library.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;


//vertex coords
typedef struct node_pos
{
   double x;
   double y;
}
      node_pos_t;
```

```
//the number of vertices of the graph on each side of the field
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); //draw the connection


double dx = start_x - end_x;

double dy = start_y - end_y;

double length = sqrt(dx * dx + dy * dy); //line length

double ratio = ARROW_LENGTH / length; //the ratio of the length of the arrow to the length of the line


cairo_new_path(cr); //we create a new way of drawing in order not to paint other elements of the graph

double x_first = end_x + ratio * (dx * cos(ARROW_ANGLE) + dy * sin(ARROW_ANGLE)); //calculate by the formula of the coordinates of the ends of the arrow

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_node_positions()
    {
        node_pos_t *positions = 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 - 1, &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); //if we connect the vertex with the middle, we draw a curved line
                }
            }
        }
    }

    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); //draw an ellipse
        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)
{
    cairo_set_source_rgb(cr, 0, 0, 0);
    node_pos_t *positions = get_node_positions();

    draw_nodes(cr, positions);
    draw_connections(cr, positions, matrix);

    free(positions);
```

```
    }

    void set_window_size()
    {
        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);
    }

    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)
{
    draw_graph(cr, matrix);
    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("Print directed graph or not? (0 - no, any other - yes)\n");
    scanf("%d", &directed);
    directed = !directed ? 0 : 1;
}

void node_read()
{
    node_shown = -2;
    while (node_shown < -1 || node_shown >= NODE_COUNT)
    {
        printf("Index of node connections to show? (input index of node, '-1' - to show all)\n");
        scanf("%d", &node_shown);
    }
}

int main(int argc, char *argv[])
{
```

```
    directed_read();
    node_read();
    calculate_size();
    matrix = get_matrix();
    if (!directed)
    {
        to_undirected(matrix);
    }
    output_matrix(NODE_COUNT, NODE_COUNT, matrix);
    create_application(argc, argv);
    free_matrix(NODE_COUNT, matrix);
    return 1;
}
```

library.h

```
#ifndef LIBRARY_H_
# define LIBRARY_H_

extern const int NODE_COUNT;

double **get_matrix();
void output_matrix(int n, int m, double **matrix);
void to_undirected(double **matrix);
void free_matrix(int n, double **matrix);
```

```
#endif
```

library.c

```
#include <stdlib.h>

#include <stdio.h>


const int RAND_LIMIT = 2;

/*

parameters for randomization

*/

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()
```

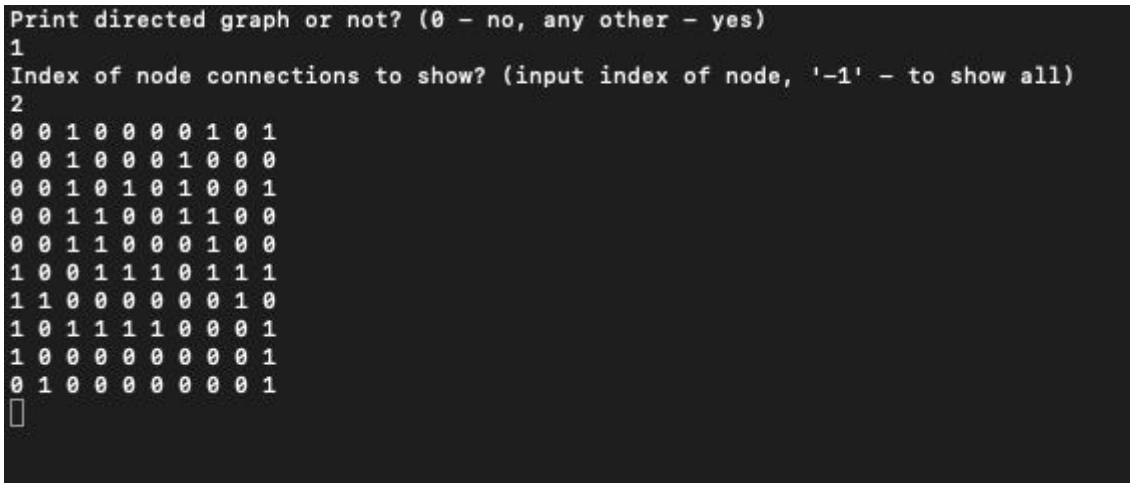
```
{
    return 1 - N3 * 0.02 - N4 * 0.005 - 0.25;
}

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;
}
```

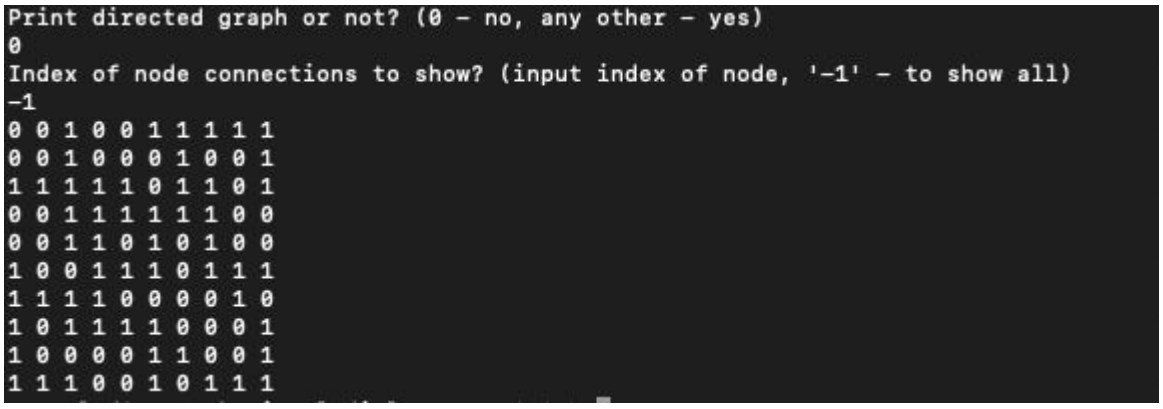
```
void mulmr(double coef, int n, int m, double **matrix)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            matrix[i][j] = matrix[i][j] * coef >= 1 ? 1 : 0;
        }
    }
}

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()
{
    srand(get_seed());
    double **matrix = randm(NODE_COUNT, NODE_COUNT);
    mulmr(get_coef(), 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);
}
```

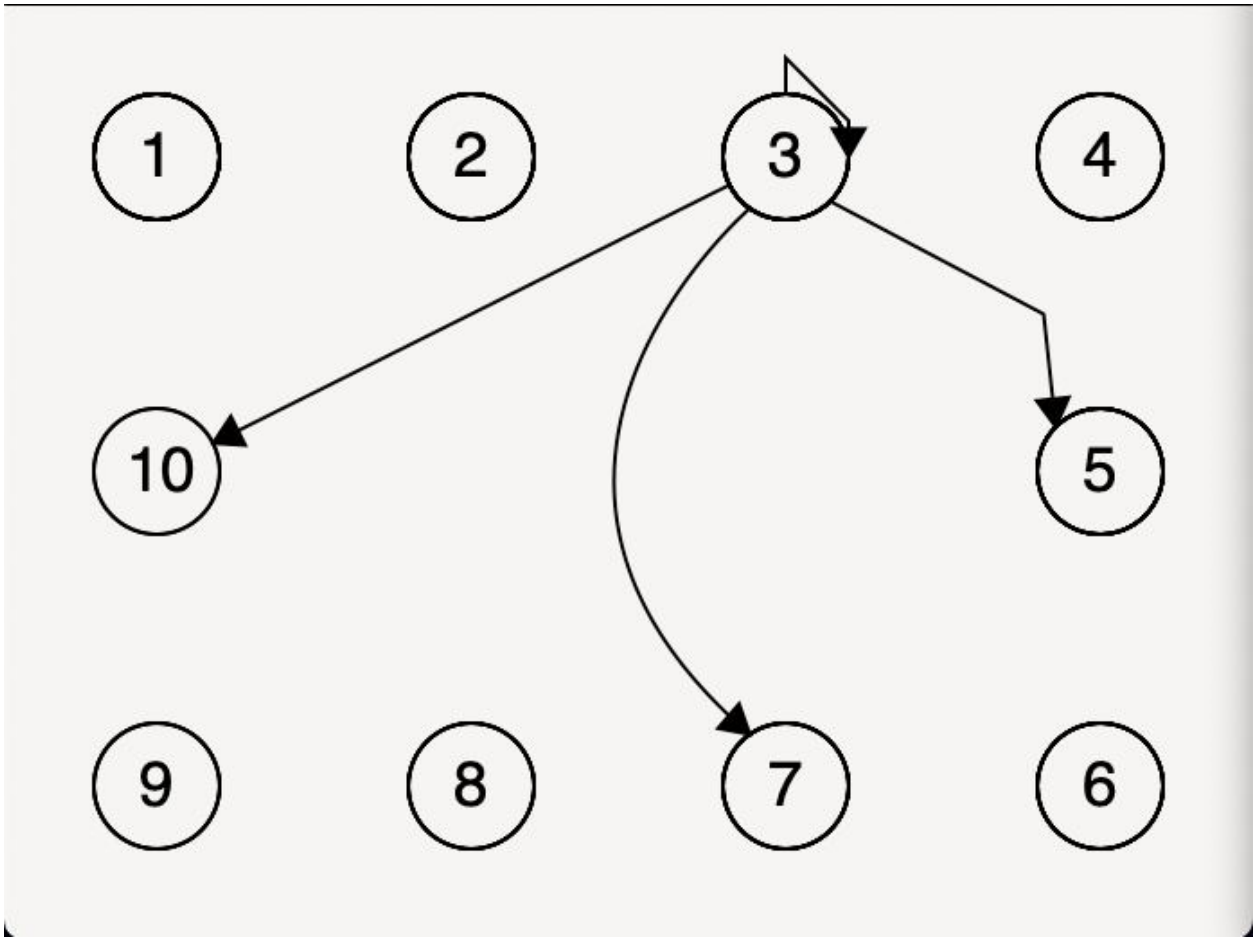
Згенеровані матриці суміжності напямленого і ненапрямленого графів

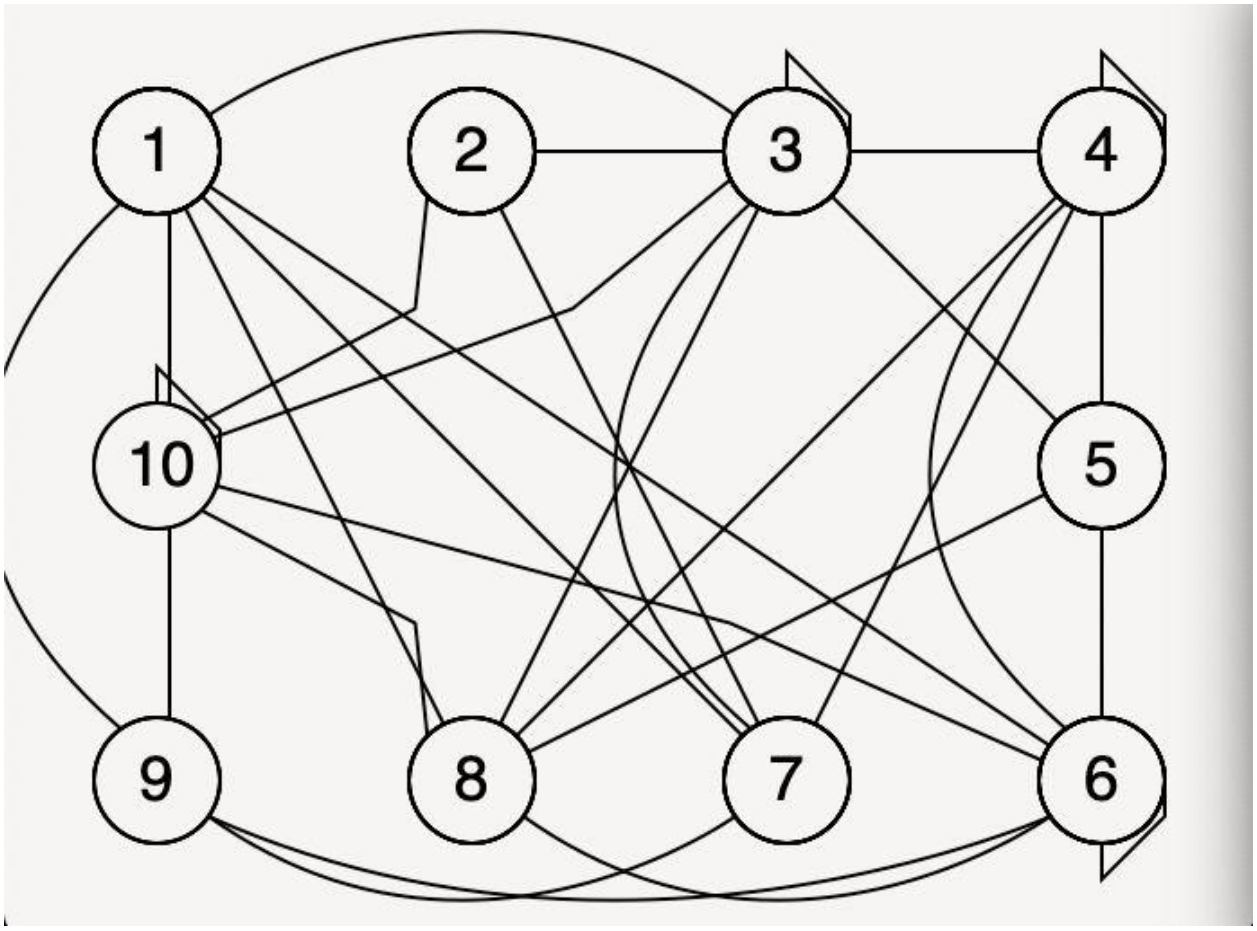


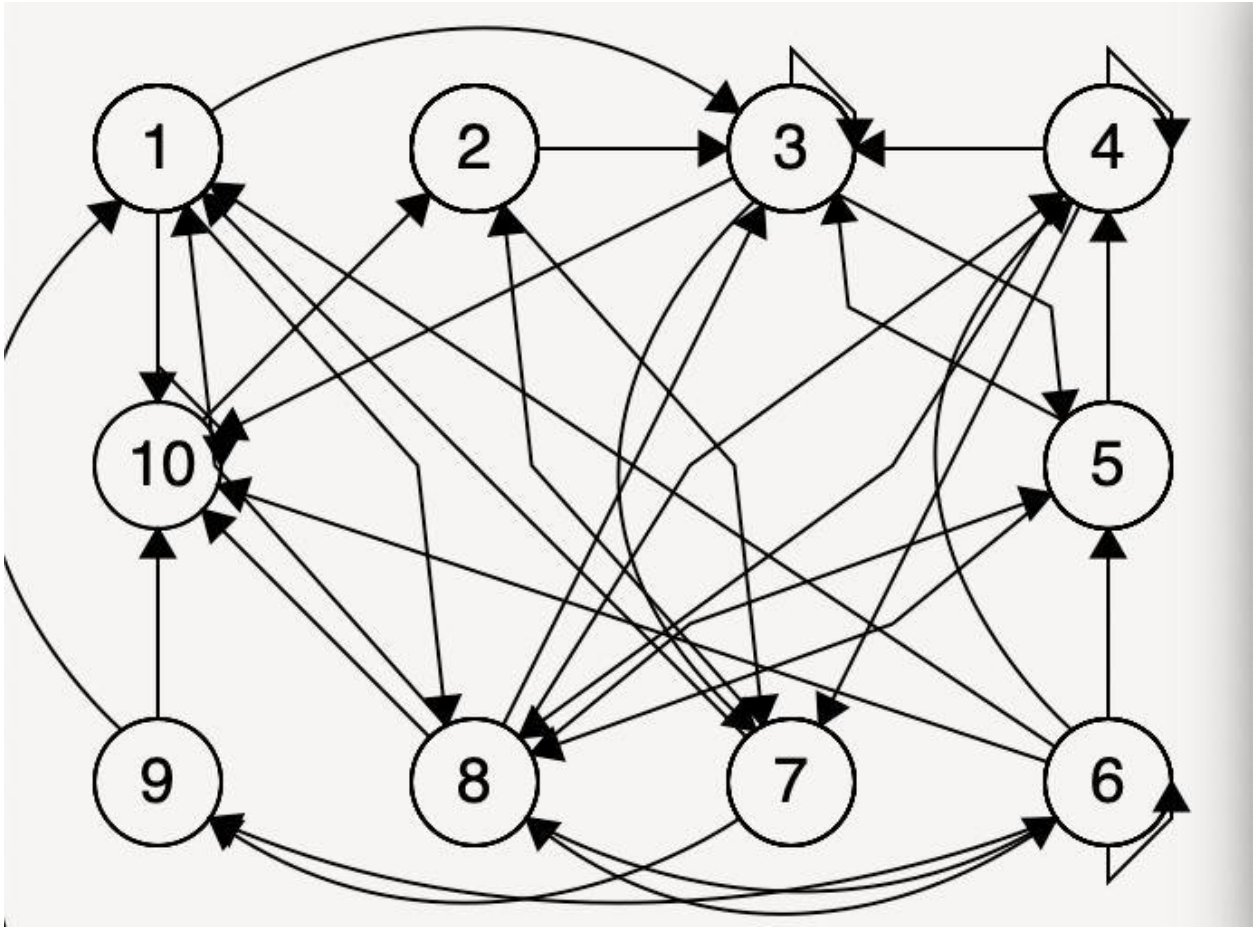


```
Print directed graph or not? (0 - no, any other - yes)
1
Index of node connections to show? (input index of node, '-1' - to show all)
-1
0 0 1 0 0 0 0 1 0 1
0 0 1 0 0 0 1 0 0 0
0 0 1 0 1 0 1 0 0 1
0 0 1 1 0 0 1 1 0 0
0 0 1 1 0 0 0 1 0 0
1 0 0 1 1 1 0 1 1 1
1 1 0 0 0 0 0 0 1 0
1 0 1 1 1 1 0 0 0 1
1 0 0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0 0 1
```

Скріншоти напрямленого і ненапрямленого графів, які побудовані за варіантом




1
2
3
4
10
5
9
8
7
6


1
2
3
4
10
5
9
8
7
6