**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: Перевірила:

студент групи ІМ-13 Молчанова А. А.

Аненко Іван Ігорович

номер у списку групи: 2

Київ 2022

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

**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 С**

**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;

//координати вершини

typedef struct node\_pos

{

double x;

double y;

}

node\_pos\_t;

//кількість вершин графа на конжній стороні поля

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); //промальовуємо з'єднання

double dx = start\_x - end\_x;

double dy = start\_y - end\_y;

double length = sqrt(dx \* dx + dy \* dy); //довжина лінії

double ratio = ARROW\_LENGTH / length; //відношення довжини боку стрілки до довжини лінії

cairo\_new\_path(cr); //створюємо новий шлях малювання задля того, щоб не замальовувалися інші елементи графу

double x\_first = end\_x + ratio \* (dx \* cos(ARROW\_ANGLE) + dy \* sin(ARROW\_ANGLE)); //розраховуємо за формулою координати кінців стрілки

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); //якщо ми з'єднуємо вершину з серединною, то малюємо зігнуту лінію

}

}

}

}

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)

{

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++) {

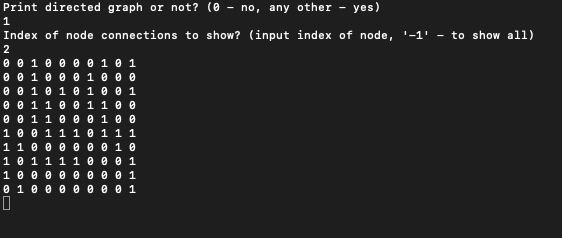
free(matrix[i]);

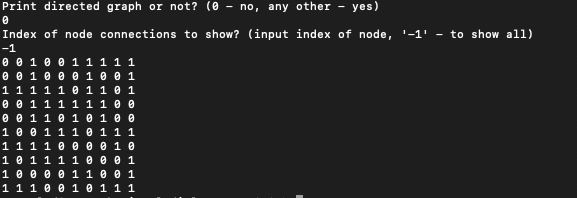
}

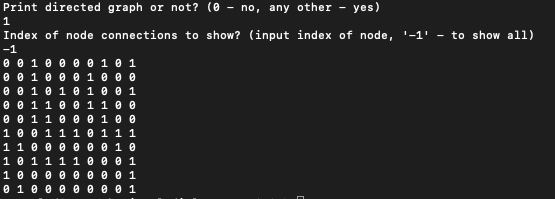
free(matrix);

}

**Згенеровані матриці суміжності напямленого і ненапрямленого графів**







**Скріншоти напрямленого і ненапрямленого графів, які побудовані за варіантом**

