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 № 2.6

з дисципліни
«Алгоритми і структури даних»

Виконав:

студент групи ІМ-22
Кушнір Микола Миколайович
номер у списку групи: 13

Перевірила:

Молчанова А. А.

Київ 2023

Постановка задачі

1. Представити зважений ненапрямлений граф із заданими параметрами так само, як у лабораторній роботі №1. Відміна: матриця ***A*** за варіантом формується за командами:

```
A = mulmr((1.0 – n3 * 0.01 – n4 * 0.005 - 0.05) * T)
```

Матриця ваг ***W*** формується за наступним чином:

1) `Wt = roundm((randm(n, n) * 100) ¤ A);`

де `roundm` – це функція, що округляє кожен елемент матриці до найближчого цілого числа, символ «¤» – поелементне множення;

2) одержується матриця ***B***, у якій

$b_{ij} = 0$, якщо $w_{ij} = 0$,

$b_{ij} = 1$, якщо $w_{ij} > 0$, $b_{ij} \in \boldsymbol{B}$, $w_{ij} \in \boldsymbol{Wt}$;

3) одержується матриця ***C***, у якій

$c_{ij} = 1$, якщо b_{ij} != b_{ji},

та $c_{ij} = 0$ в іншому випадку;

4) одержується матриця ***D***, у якій

$d_{ij} = 1$, якщо $b_{ij} = b_{ji} = 1$,

та $d_{ij} = 0$ в інших випадках;

5) `Wt = (C + (D ¤ Tr)) ¤ Wt;`

де `Tr` – верхній трикутник матриці одиниць (без головної діагоналі),

+ – поелементна сума матриць;

6) одержується матриця ваг ***W*** шляхом симетризування матриці ***Wt***.

2. Створити програму для знаходження мінімального кістяка за алгоритмом Краскала при $\boldsymbol{n_4}$ – парному і за алгоритмом Пріма – при непарному. При цьому у програмі:

- графи представляти у вигляді динамічних списків, обхід графа, додавання, віднімання вершин, ребер виконувати як функції з вершинами відповідних списків;

- встановити функцію **halt** у точці додавання чергового ребра до кістяка,
- виводити зображення графа у графічному вікні перед кожною

зупинкою по функції **halt**.

3. Під час обходу графа побудувати дерево його кістяка. Вивести побудоване дерево у графічному вікні. При зображенні як графа, так і його кістяка, вказати ваги ребер.

Завдання для варіанту 13, (групи ІМ-22)

- n_1 = **2**;
- n_2 = **2**;
- n_3 = **1**;
- n_4 = **3**;

Число вершин n: 10 + **1** = **11**.

Розміщення вершин: **прямокутником (квадратом)**.

Формування матриці A:

```
srand(2 2 1 3);
T = randm(11, 11);
A = mulmr((1.0 - 1.0 *0.01 - 3.0 *0.005 - 0.05)*T)
```

Посилання на репозиторій з лабораторною роботою

Текст програми

Вміст файлу Configurations.h

```
/* N1N2 - номер групи, N3N4 - порядковий номер у списку групи */
#define N1  2
#define N2  2
#define N3  1
#define N4  3
/* Кількість рядків і стовпців матриць суміжності графів */
#define N (10 + N3)
/* Для позначення осей координат (позиції елементів записані у векторах) */
#define x   0
#define y   1
/* Значення, що використовується в обчисленнях */
#define SQRT_2  1.4142135624
```

```
#define VERTEX_RADIUS           40
#define LOOP_RADIUS             (5 * VERTEX_RADIUS / 4)
#define ONE_STEP_LENGTH         (9 * VERTEX_RADIUS / 2) /* Найменша відстань між
вершинами графа */
#define MAX_ONE_STEP_LENGTH     (3 * ONE_STEP_LENGTH / 2)
#define WINDOW_BORDER_OFFSET    (2 * LOOP_RADIUS + 10)

const wchar_t *vertices_names[] =
{
    L"1", L"2", L"3",
    L"4", L"5", L"6",
    L"7", L"8", L"9",
    L"10", L"11",
};

#define GRAPH_WIDTH     ((int)((double)(N - 5) * 0.5) * ONE_STEP_LENGTH)
#define GRAPH_HEIGHT    (3 * ONE_STEP_LENGTH)
/* Щоб граф коректно відобразився у вікні, його висота має бути сталою */

const int min_coords[] =
{
    WINDOW_BORDER_OFFSET,
    WINDOW_BORDER_OFFSET
};
const int max_coords[] =
{
    WINDOW_BORDER_OFFSET + GRAPH_WIDTH,
    WINDOW_BORDER_OFFSET + GRAPH_HEIGHT
};

const int graph_full_size[] =
{
    (2 * WINDOW_BORDER_OFFSET + GRAPH_WIDTH),
    (2 * WINDOW_BORDER_OFFSET + GRAPH_HEIGHT)
};

/* Визначають зміщення центру ребер для огинання вершин */
const double edge_center_offset_divs[] = { 5, 7 };

/* Визначає зсув числа, що означає вагу ребра від його центра */
const double weight_text_offset_div = 3.5;
```

Вміст файлу DrawingDataSetter.h

```
#include <math.h>
#include "Configurations.h"
#include "WorkWithMatrices.h"
#include "WorkWithGraphList.h"

double Pow2(int value);

typedef struct DrawingData
{
    int edge_type;
    int start[2];
    int center[2];
    int end[2];
    int text_pos[2];
} draw_data;

int **SetVerticesCoords(int n)
```

```
{
    int **coords = Create2dIntArr(n, n);
    int current_pos[2] = { min_coords[x], min_coords[y] };
    for (int i = 0; i < n; i++)
    {
        coords[i][x] = current_pos[x];
        coords[i][y] = current_pos[y];
        if (current_pos[x] < max_coords[x] && current_pos[y] == min_coords[y])
            current_pos[x] += ONE_STEP_LENGTH;
        else if (current_pos[y] < max_coords[y] && current_pos[x] ==
max_coords[x])
            current_pos[y] += ONE_STEP_LENGTH;
        else if (current_pos[x] > min_coords[x] && current_pos[y] ==
max_coords[y])
            current_pos[x] -= ONE_STEP_LENGTH;
        else if (current_pos[y] > min_coords[y] && current_pos[x] ==
min_coords[x])
            current_pos[y] -= GRAPH_HEIGHT / (3 - n % 2);
        /*
         * Якщо к-сть вершин парна та більша ніж 10, то з
         * лівого боку буде розміщено дві вершини,
         * а якщо непарна - 1
         */
    }
    return coords;
}

draw_data SetEdgeDrawData(edge *e)
{
    draw_data data;
    data.edge_type = 1;
    data.start[x] = e->vertex1->coords[x];
    data.start[y] = e->vertex1->coords[y];
    data.end[x] = e->vertex2->coords[x];
    data.end[y] = e->vertex2->coords[y];
    data.center[x] = (data.start[x] + data.end[x]) / 2;
    data.center[y] = (data.start[y] + data.end[y]) / 2;
    int dx = data.end[x] - data.start[x];
    int dy = data.end[y] - data.start[y];
    int variable_delta, static_coord;
    if (e->vertex1 == e->vertex2)
    {
        data.edge_type = 2;
        int loop_offset_direction[2] = { 0 };
        if (data.center[x] > min_coords[x] && data.center[y] == min_coords[y])
        {
            --loop_offset_direction[x];
            --loop_offset_direction[y];
        }
        else if (data.center[x] == max_coords[x] && data.center[y] >
min_coords[y])
        {
            ++loop_offset_direction[x];
            --loop_offset_direction[y];
        }
        else if (data.center[x] < max_coords[x] && data.center[y] ==
max_coords[y])
        {
            ++loop_offset_direction[x];
            ++loop_offset_direction[y];
        }
        else if (data.center[x] == min_coords[x] && data.center[y] <
max_coords[y])
        {
```

```
            --loop_offset_direction[x];
            ++loop_offset_direction[y];
        }
        int loop_center_offset = (int) round(
                (VERTEX_RADIUS * SQRT_2 / 2 +
                 sqrt(Pow2(LOOP_RADIUS) - Pow2(VERTEX_RADIUS) / 2))
                / SQRT_2);
        data.center[x] += loop_center_offset * loop_offset_direction[x];
        data.center[y] += loop_center_offset * loop_offset_direction[y];
        return data;
    }
    else if ((dx == 0 ? (variable_delta = dy, static_coord = x, 1) : 0) ||
             (dy == 0 ? (variable_delta = dx, static_coord = y, 1) : 0))
    {
        int abs_variable_data = abs(variable_delta);
        if (abs_variable_data > MAX_ONE_STEP_LENGTH)
        {
            int edge_center_offset = (int)
            (abs_variable_data / edge_center_offset_divs[abs_variable_data /
ONE_STEP_LENGTH - 2]);
            /*
             * У масиві індекси починаються з 0, а мінімальна частка,
             * за якої слід надавати зміщення ребру - 2
             */
            if (data.start[static_coord] == min_coords[static_coord])
                data.center[static_coord] -= edge_center_offset;
            else if (data.start[static_coord] == max_coords[static_coord])
                data.center[static_coord] += edge_center_offset;
        }
        data.text_pos[x] = data.center[x];
        data.text_pos[y] = data.center[y];
    }
    else
    {
        int dx_text_offset = (int) ((data.end[x] - data.center[x]) /
weight_text_offset_div);
        int dy_text_offset = (int) ((data.end[y] - data.center[y]) /
weight_text_offset_div);
        if (e->vertex1->index < e->vertex2->index)
        {
            data.text_pos[x] = data.center[x] - dx_text_offset;
            data.text_pos[y] = data.center[y] - dy_text_offset;
        }
        else
        {
            data.text_pos[x] = data.center[x] + dx_text_offset;
            data.text_pos[y] = data.center[y] + dy_text_offset;
        }
    }
    return data;
}

double Pow2(int value)
{
    return (double)(value * value);
}
```

Вміст файлу GraphPainter.h

```
#include "DrawingDataSetter.h"

void DrawEdgeLines(draw_data data, int weight, HPEN e_pen, COLORREF num_color,
```

```
HDC hdc);
void DrawLoop(int *center, HPEN e_pen, HDC hdc);
void DrawVertex(vertex* v, HBRUSH v_brush, HPEN v_pen, COLORREF text_color, HDC
hdc);

void DrawGraph(graph *g,
               HPEN e_pen, COLORREF w_text_color,
               HBRUSH v_brush, HPEN v_pen,
               COLORREF v_text_color,
               HDC hdc)
{
    /* Зображаємо ребра */
    edge *current_e = g->edges->first_edge;
    int v1_index, v2_index;
    while (current_e != NULL)
    {
        v1_index = current_e->vertex1->index;
        v2_index = current_e->vertex2->index;
        if (!(v1_index > v2_index &&
              FindEdge(g, current_e->vertex2, current_e->vertex1) != NULL))
        {
            draw_data data = SetEdgeDrawData(current_e);
            switch (data.edge_type)
            {
            case 1:
                DrawEdgeLines(data, current_e->weight, e_pen, w_text_color,
hdc);
                break;
            case 2:
                DrawLoop(data.center, e_pen, hdc);
            }
        }
        current_e = current_e->next;
    }
    /* Зображаємо вершини */
    vertex *current_v = g->vertices->first_vertex;
    while (current_v != NULL)
    {
        DrawVertex(current_v, v_brush, v_pen, v_text_color, hdc);
        current_v = current_v->next;
    }
}

void DrawPAStep(graph *g,
                HPEN e_pen, COLORREF w_text_color,
                HBRUSH v_brush, HPEN v_pen,
                COLORREF v_text_color,
                HDC hdc)
{
    if (!IsEdgeSetEmpty(g))
    {
        draw_data data  = SetEdgeDrawData(g->edges->last_edge);
        DrawEdgeLines(data, g->edges->last_edge->weight, e_pen, w_text_color,
hdc);
        DrawVertex(g->edges->last_edge->vertex1, v_brush, v_pen, v_text_color,
hdc);
    }
    DrawVertex(g->vertices->last_vertex, v_brush, v_pen, v_text_color, hdc);
}

void DrawEdgeLines(draw_data data, int weight, HPEN e_pen, COLORREF num_color,
HDC hdc)
{
    SelectObject(hdc, e_pen);
```

```
    MoveToEx(hdc, data.start[x], data.start[y], NULL);
    LineTo(hdc, data.center[x], data.center[y]);
    MoveToEx(hdc, data.center[x], data.center[y], NULL);
    LineTo(hdc, data.end[x], data.end[y]);

    wchar_t output[5];
    swprintf(output, 5, L"%d", weight);
    int rect_sides_half[] = { 4, 6 };
    weight < 10 ? rect_sides_half[x] :
        weight < 100 ? (rect_sides_half[x] *= 2) :
            (rect_sides_half[x] *= 3);
    RECT rect;
    rect.left = data.text_pos[x] - rect_sides_half[x];
    rect.top = data.text_pos[y] - rect_sides_half[y];
    rect.right = data.text_pos[x] + rect_sides_half[x];
    rect.bottom = data.text_pos[y] + rect_sides_half[y];
    HBRUSH w_background = CreateSolidBrush(RGB(255, 255, 255));
    FillRect(hdc, &rect, w_background);
    DeleteObject(w_background);
    HFONT hFont = CreateFont(15,
                             0, 0, 0,
                             FW_BOLD, FALSE, FALSE, FALSE,
                             DEFAULT_CHARSET, OUT_OUTLINE_PRECIS,
                             CLIP_DEFAULT_PRECIS, CLEARTYPE_QUALITY,
                             VARIABLE_PITCH, TEXT("Times New Roman"));
    SelectObject(hdc, hFont);
    SetTextColor(hdc, num_color);
    DrawText(hdc, output, -1, &rect,
             DT_CENTER | DT_VCENTER | DT_SINGLELINE | DT_NOCLIP);
    DeleteObject(hFont);
}

void DrawLoop(int *center, HPEN e_pen, HDC hdc)
{
    SelectObject(hdc, e_pen);
    SelectObject(hdc, GetStockObject(NULL_BRUSH));
    Ellipse(hdc,
            (center[x] - LOOP_RADIUS),
            (center[y] - LOOP_RADIUS),
            (center[x] + LOOP_RADIUS),
            (center[y] + LOOP_RADIUS));
}

void DrawVertex(vertex *v, HBRUSH v_brush, HPEN v_pen, COLORREF text_color, HDC
hdc)
{
    RECT rect;
    rect.left = v->coords[x] - VERTEX_RADIUS;
    rect.top = v->coords[y] - VERTEX_RADIUS;
    rect.right = v->coords[x] + VERTEX_RADIUS;
    rect.bottom = v->coords[y] + VERTEX_RADIUS;
    SelectObject(hdc, v_brush);
    SelectObject(hdc, v_pen);
    Ellipse(hdc, rect.left, rect.top, rect.right, rect.bottom);
    HFONT hFont = CreateFont(40,0, 0, 0,
                             FW_BOLD, FALSE, FALSE, FALSE,
                             DEFAULT_CHARSET, OUT_OUTLINE_PRECIS,
                             CLIP_DEFAULT_PRECIS, CLEARTYPE_QUALITY,
                             VARIABLE_PITCH, TEXT("Segoe Script"));
    SelectObject(hdc, hFont);
    SetTextColor(hdc, text_color);
    DrawText(hdc, v->name, -1, &rect,
             DT_CENTER | DT_VCENTER | DT_SINGLELINE | DT_NOCLIP);
```

```
    DeleteObject(hFont);
}
```

Вміст файлу Prim'sAlgorithm.h

```
void PerformPrimAlgStep(graph *Gt, graph *G)
{
    if (IsVerticesSetEmpty(Gt))
    {
        CopyVertex(Gt, G, G->vertices->first_vertex);
        return;
    }
    edge *current_e = G->edges->first_edge, *lightest_e = NULL;
    while (current_e != NULL)
    {
        if (FindVertex(Gt, current_e->vertex1->index) != NULL &&
            FindVertex(Gt, current_e->vertex2->index) == NULL)
            if (lightest_e == NULL || current_e->weight < lightest_e->weight)
                lightest_e = current_e;
        current_e = current_e->next;
    }
    CopyVertex(Gt, G, lightest_e->vertex2);
    CopyEdge(Gt, G, lightest_e->vertex1, lightest_e->vertex2);
}
```

Вміст файлу WorkWithGraphList.h

```
typedef struct VertexData
{
    int index;
    wchar_t *name;
    int coords[2];
    struct VertexData *next;
} vertex;

typedef struct EdgeData
{
    vertex *vertex1;
    vertex *vertex2;
    int weight;
    struct EdgeData *next;
} edge;

typedef struct SetOfVertices
{
    vertex *first_vertex;
    vertex *last_vertex;
} v_set;

typedef struct SetOfEdges
{
    edge *first_edge;
    edge *last_edge;
} e_set;

typedef struct GraphList
{
    v_set *vertices;
    e_set *edges;
} graph;
```

```
/****************************************** ПЕРЕЛІК ФУНКЦІЙ
*************************************/
graph  *InitGraph           ();
void    RefreshGraph        (graph *g);
void    FreeGraph           (graph *g);
void    PrintGraph          (graph *g);
/***** Функції для роботи з множиною вершин
*************************************************/
int     IsVerticesSetEmpty  (graph *g);
int     IsVerticesSetFull   (int n, graph *g);
vertex *FindVertex          (graph *g, int v_index);
void    AddVertex           (graph *g, int v_index, wchar_t *v_name, const int
*v_coords);
void    CopyVertex          (graph *to, graph *from, vertex *v);
/***** Функції для роботи з множиною ребер
*************************************************/
int     IsEdgeSetEmpty      (graph *g);
edge   *FindEdge            (graph *g, vertex *v1, vertex *v2);
void    AddEdge             (graph *g, vertex *v1_index, vertex *v2_index, int
weight);
void    CopyEdge            (graph *to, graph *from, vertex *v1, vertex *v2);
/*****************************************************************************
*****************/

graph *ConvertDataToGraphList(int n,
                              wchar_t **v_names,
                              int **v_coords,
                              int **matrix_uA,
                              int **matrix_W)
{
    graph *g = InitGraph();
    int i, j;
    for(i = 0; i < n; i++)
        AddVertex(g, i, v_names[i], v_coords[i]);
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (matrix_uA[i][j] == 1)
                AddEdge(g,
                        FindVertex(g, i),
                        FindVertex(g, j),
                        matrix_W[i][j]);
    return g;
}

graph *InitGraph()
{
    graph *g = (graph *) malloc(sizeof(graph));
    g->vertices = (v_set *) malloc(sizeof (v_set));
    g->edges = (e_set *) malloc(sizeof(e_set));

    g->vertices->first_vertex = g->vertices->last_vertex = NULL;
    g->edges->first_edge = g->edges->last_edge = NULL;

    return g;
}

int IsVerticesSetEmpty(graph *g)
{
    return (g->vertices->first_vertex == NULL);
}

int IsVerticesSetFull(int n, graph *g)
{
```

```
    int max_indices_sum = 0;
    int current_indices_sum = 0;
    for (int i = 0; i < n; i++)
        max_indices_sum += i;
    vertex *current_v = g->vertices->first_vertex;
    while (current_v != NULL)
    {
        current_indices_sum += current_v->index;
        current_v = current_v->next;
    }
    return (current_indices_sum == max_indices_sum ? 1 : 0);
}

vertex *FindVertex(graph *g, int v_index)
{
    vertex *current_v = g->vertices->first_vertex;
    while(current_v != NULL)
    {
        if (current_v->index == v_index)
            return current_v;
        else
            current_v = current_v->next;
    }
    return NULL;
}

void AddVertex(graph *g, int v_index, wchar_t *v_name, const int *v_coords)
{
    vertex *v = (vertex *) malloc(sizeof(vertex));
    v->index = v_index;
    v->name = v_name;
    v->coords[x] = v_coords[x];
    v->coords[y] = v_coords[y];
    v->next = NULL;
    if (!IsVerticesSetEmpty(g))
    {
        g->vertices->last_vertex->next = v;
        g->vertices->last_vertex = v;
    }
    else
        g->vertices->last_vertex = g->vertices->first_vertex = v;
}

void CopyVertex(graph *to, graph *from, vertex *v)
{
    vertex *copied_v = (vertex *) malloc(sizeof(vertex));
    vertex *original_v = FindVertex(from, v->index);
    copied_v->index = original_v->index;
    copied_v->name = original_v->name;
    copied_v->coords[x] = original_v->coords[x];
    copied_v->coords[y] = original_v->coords[y];
    copied_v->next = NULL;
    if (!IsVerticesSetEmpty(to))
    {
        to->vertices->last_vertex->next = copied_v;
        to->vertices->last_vertex = copied_v;
    }
    else
        to->vertices->last_vertex = to->vertices->first_vertex = copied_v;
}

int IsEdgeSetEmpty(graph *g)
{
    return (g->edges->first_edge == NULL);
```

```
}

edge *FindEdge(graph *g, vertex *v1, vertex *v2)
{
    edge *current_e = g->edges->first_edge;
    while (current_e != NULL)
    {
        if (current_e->vertex1->index == v1->index &&
            current_e->vertex2->index == v2->index)
            return current_e;
        else
            current_e = current_e->next;
    }
    return NULL;
}

void AddEdge(graph *g, vertex *v1, vertex *v2, int weight)
{
    edge *e = (edge *) malloc(sizeof(edge));
    e->vertex1 = v1;
    e->vertex2 = v2;
    e->weight = weight;
    e->next = NULL;
    if (!IsEdgeSetEmpty(g))
    {
        g->edges->last_edge->next = e;
        g->edges->last_edge = e;
    }
    else
        g->edges->last_edge = g->edges->first_edge = e;
}

void CopyEdge(graph *to, graph *from, vertex *v1, vertex *v2)
{
    edge *copied_e = (edge *) malloc(sizeof(edge));
    edge *original_e = FindEdge(from, v1, v2);
    copied_e->vertex1 = FindVertex(to, v1->index);
    copied_e->vertex2 = FindVertex(to, v2->index);
    copied_e->weight = original_e->weight;
    copied_e->next = NULL;
    if (!IsEdgeSetEmpty(to))
    {
        to->edges->last_edge->next = copied_e;
        to->edges->last_edge = copied_e;
    }
    else
        to->edges->last_edge = to->edges->first_edge = copied_e;
}

void RefreshGraph(graph *g)
{
    vertex *temp_v;
    while (!IsVerticesSetEmpty(g))
    {
        temp_v = g->vertices->first_vertex;
        g->vertices->first_vertex = g->vertices->first_vertex->next;
        free(temp_v);
    }
    edge *temp_e;
    while (!IsEdgeSetEmpty(g))
    {
        temp_e = g->edges->first_edge;
        g->edges->first_edge = g->edges->first_edge->next;
        free(temp_e);
```

```
    }
}

void FreeGraph(graph *g)
{
    RefreshGraph(g);
    free(g->vertices);
    free(g->edges);
    free(g);
    g = NULL;
}

void PrintGraph(graph *g)
{
    printf("{");
    /* Друкуємо множину вершин */
    if (IsVerticesSetEmpty(g))
        printf("%c", 155);
    else
    {
        printf("{");
        vertex *current_v = g->vertices->first_vertex;
        while (current_v->next != NULL)
        {
            wprintf(L"%ls, ", current_v->name);
            current_v = current_v->next;
        }
        wprintf(L"%ls", current_v->name);
        printf("},\n");
    }
    /* Друкуємо множину ребер */
    if (IsEdgeSetEmpty(g))
        printf(" %c ",155);
    else
    {
        edge *current_e = g->edges->first_edge;
        while (current_e->next != NULL)
        {
            wprintf(L" (%ls, %ls) ",
                    current_e->vertex1->name,
                    current_e->vertex2->name);
            printf("[ weight = %3d ],\n", current_e->weight);
            current_e = current_e->next;
        }
        wprintf(L" (%ls, %ls) ",
                current_e->vertex1->name,
                current_e->vertex2->name);
        printf("[ weight = %3d ] ", current_e->weight);
    }
    /* у таблиці ASCII ((char) 155) - це щось схоже на символ порожньої множини
:) */
    printf("}\n");
}
```

Вміст файлу WorkWithMatrices.h

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

/***** Допоміжні функції
*****************************************************************************
*************/
double   RandInRange                        (double min, double max);
```

```
int    **Create2dIntArr                    (int rows, int cols);
void     FreeInt2dArr                      (int rows, int **arr);
void     FreeDouble2dArr                   (int rows, double **arr);
/*****************************************************************************
************************************/

double **randm(int n1, int n2)
{
    double **matrix_T = (double **) malloc(sizeof(double *) * n1);
    int i, j;
    for (i = 0; i < n1; i++)
    {
        matrix_T[i] = (double *) malloc(sizeof(double) * n2);
        for (j = 0; j < n2; j++)
           matrix_T[i][j] = RandInRange(0.0, 2.0);
    }
    return matrix_T;
}

int **mulmr(int n1, int n2, double **matrix_T, double coefficient)
{
    int **matrix_A = Create2dIntArr(n1, n2);
    int i, j;
    for (i = 0; i < n1; i++)
    {
        for (j = 0; j < n2; j++)
            matrix_A[i][j] = (int) (matrix_T[i][j] * coefficient);
    }
    return matrix_A;
}

int roundm(double value)
{
    int int_part = (int) value;
    return ((value - int_part) >= 0.5 ? (int_part + 1) : int_part);
}

int **GetWtMatrix(int n, int **matrix_A)
{
    int **matrix_Wt = Create2dIntArr(n, n);
    double **matrix_T = randm(n, n);
    int i, j;
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
            matrix_Wt[i][j] = roundm((matrix_T[i][j] * 100) * matrix_A[i][j]);
    }
    FreeDouble2dArr(n, matrix_T);
    return matrix_Wt;
}

int **SymmetrizeMatrix(int n, int **matrix)
{
    int **symmetric_matrix = Create2dIntArr(n, n);
    int i, j;
    for (i = 0; i < n; i++)
    {
        for (j = i; j < n; j++)
            if (matrix[i][j] > 0 || matrix[j][i] > 0)
                symmetric_matrix[i][j] = symmetric_matrix[j][i] =
                        (matrix[i][j] > matrix[j][i] ? matrix[i][j] :
matrix[j][i]);
    }
    return symmetric_matrix;
```

```
}

int **GetMatrixOfWeights(int n, int **matrix_A)
{
    int **matrix_W;
    /* Формування матриці Wt - пункт 1) */
    int **matrix_Wt = GetWtMatrix(n, matrix_A);

    int **matrix_B = Create2dIntArr(n, n);
    int **matrix_C = Create2dIntArr(n, n);
    int **matrix_D = Create2dIntArr(n, n);

    int i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            /* Формування матриці B - пункт 2) */
            if (matrix_Wt[i][j] > 0)
                matrix_B[i][j] = 1;
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            /* Формування матриці C - пункт 3) */
            if (matrix_B[i][j] != matrix_B[j][i])
                matrix_C[i][j] = 1;
            /* Формування матриці D - пункт 4) */
            if (matrix_B[i][j] == 1 && matrix_B[j][i] == 1)
                matrix_D[i][j] = 1;
            /* Множення матриці D на верхній трикутник матриці одиниць Tr */
            if (j <= i)
                matrix_D[i][j] = 0;
            /* Переприсвоювання значень матриці Wt - пункт 5) */
            matrix_Wt[i][j] = (matrix_C[i][j] + matrix_D[i][j]) *
matrix_Wt[i][j];
        }
    }
    /* Формування матриці ваг W шляхом симетризування матриці Wt - пункт 6) */
    matrix_W = SymmetrizeMatrix(n, matrix_Wt);

    FreeInt2dArr(n, matrix_Wt);
    FreeInt2dArr(n, matrix_B);
    FreeInt2dArr(n, matrix_C);
    FreeInt2dArr(n, matrix_D);
    return matrix_W;
}

double RandInRange(double min, double max)
{
    double random = (double) rand() / RAND_MAX;
    double range = max - min;
    return min + range * random;
}

int **Create2dIntArr(int rows, int cols)
{
    int **arr = (int **) malloc(sizeof(int *) * rows);
    for (int i = 0; i < rows; i++)
        arr[i] = (int *) calloc(cols, sizeof(int));
    return arr;
}

void FreeInt2dArr(int rows, int **arr)
{
    for (int i = 0; i < rows; i++)
```

```
        free(arr[i]);
    free(arr);
}

void FreeDouble2dArr(int rows, double **arr)
{
    for (int i = 0; i < rows; i++)
        free(arr[i]);
    free(arr);
}
```

Вміст файлу main.c

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include "GraphPainter.h"
#include "Prim'sAlgorithm.h"

#define GO_TO_START 0
#define NEXT_STEP   1

#define TRAVERSAL_START         1
#define TRAVERSAL_CONTINUATION  2
#define TRAVERSAL_END           3
int current_traversal_state = TRAVERSAL_START;

graph *G;
graph *Gt;

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

void CreateGoToStartButton(HWND hWnd);

void CreateNextStepButton(HWND hWnd, int traversal_state);
HWND hNextStepButton;

void CreateTip(HWND hWnd, int traversal_state);
HWND hTip;

int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow)
{
    WNDCLASS wndClass;
    wndClass.lpszClassName = L"Лабораторна робота 2.6";
    wndClass.hInstance = hInstance;
    wndClass.lpfnWndProc = WndProc;
    wndClass.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndClass.hIcon = 0;
    wndClass.lpszMenuName = 0;
    wndClass.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
    wndClass.style = CS_HREDRAW | CS_VREDRAW;
    wndClass.cbClsExtra = 0;
    wndClass.cbWndExtra = 0;
    if (!RegisterClass(&wndClass)) return 0;
    HWND hWnd;
    MSG lpMsg;
    hWnd = CreateWindowExW(0, L"Лабораторна робота 2.6",
```

```
                                L"Лабораторна робота 2.6, виконав М.М.Кушнір",
                                WS_OVERLAPPEDWINDOW,
                                0, 0,
                                (graph_full_size[x] + 400),
                                (graph_full_size[y] + 50),
                                (HWND) NULL,
                                (HMENU) NULL,
                                (HINSTANCE) hInstance,
                                (HINSTANCE) NULL);
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);
    int GetMessageRes;
    while ((GetMessageRes = GetMessage(&lpMsg, hWnd, 0, 0)) != 0)
    {
        if (GetMessageRes == -1)
            return lpMsg.wParam;
        else
        {
            TranslateMessage(&lpMsg);
            DispatchMessage(&lpMsg);
        }
    }
}

LRESULT CALLBACK WndProc(HWND hWnd,
                         UINT message,
                         WPARAM wParam,
                         LPARAM lParam)
{
    static HPEN     graph_e_pen;
    static COLORREF graph_w_text_color;
    static HBRUSH   graph_v_brush;
    static HPEN     graph_v_pen;
    static COLORREF graph_v_text_color;
    static HPEN     tree_e_pen;
    static COLORREF tree_w_text_color;
    static HBRUSH   tree_v_brush;
    static HPEN     tree_v_pen;
    static COLORREF tree_v_text_color;

    HDC hdc;
    PAINTSTRUCT ps;
    switch (message)
    {
    case WM_COMMAND:
    {
        switch (wParam)
        {
        case GO_TO_START:
        {
            current_traversal_state = TRAVERSAL_START;
            RedrawWindow(hWnd, NULL, NULL,
                         RDW_ERASE | RDW_INVALIDATE);
            if (hNextStepButton != NULL)
                DestroyWindow(hNextStepButton);
            CreateNextStepButton(hWnd, TRAVERSAL_START);
            DestroyWindow(hTip);
            CreateTip(hWnd, TRAVERSAL_START);
            break;
        }
        case NEXT_STEP:
        {
            switch (current_traversal_state)
            {
```

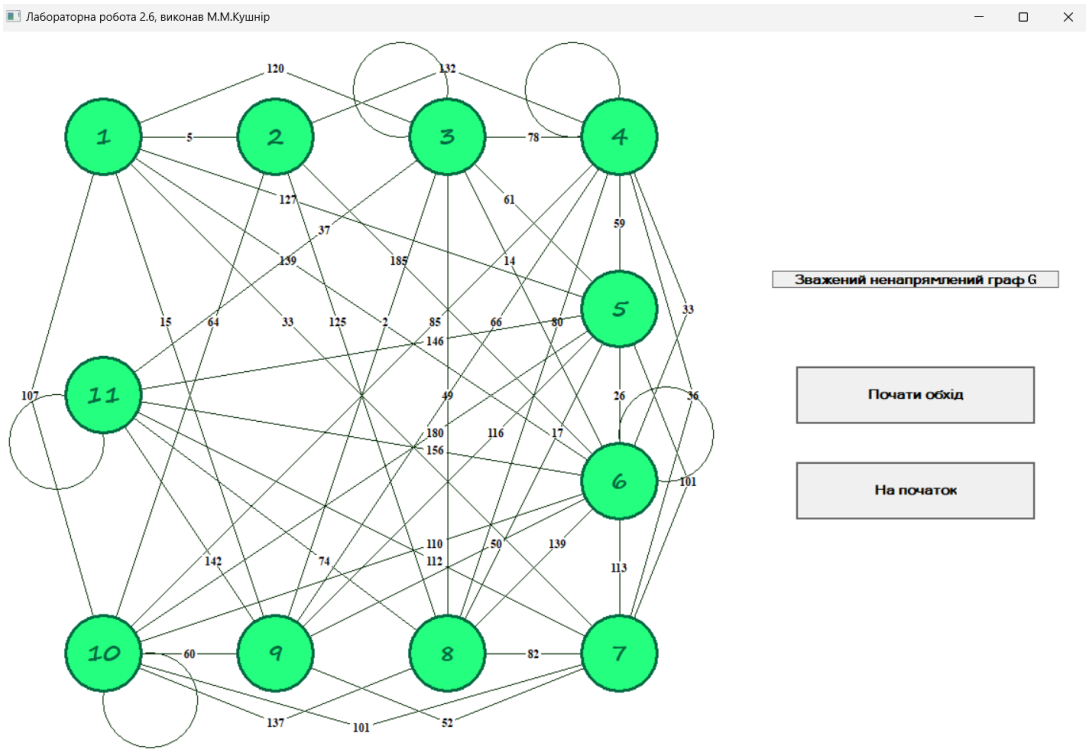
```
                case TRAVERSAL_START:
                    DestroyWindow(hNextStepButton);
                    CreateNextStepButton(hWnd, TRAVERSAL_CONTINUATION);
                    DestroyWindow(hTip);
                    CreateTip(hWnd, TRAVERSAL_CONTINUATION);
                    current_traversal_state = TRAVERSAL_CONTINUATION;
                    RefreshGraph(Gt);
                case TRAVERSAL_CONTINUATION:
                    PerformPrimAlgStep(Gt, G);
                    InvalidateRect(hWnd, NULL, TRUE);
                    UpdateWindow(hWnd);
                    if (IsVerticesSetFull(N, Gt))
                    {
                        DestroyWindow(hNextStepButton);
                        CreateNextStepButton(hWnd, TRAVERSAL_END);
                        current_traversal_state = TRAVERSAL_END;
                    }
                    break;
                case TRAVERSAL_END:
                    DestroyWindow(hTip);
                    CreateTip(hWnd, TRAVERSAL_END);
                    RedrawWindow(hWnd, NULL, NULL,
                                 RDW_ERASE | RDW_INVALIDATE);
                    DestroyWindow(hNextStepButton);
                    hNextStepButton = NULL;
                }
            }
            }
        }
        case WM_PAINT:
        {
            hdc = BeginPaint(hWnd, &ps);
            SetBkMode(hdc, TRANSPARENT);
            switch (current_traversal_state)
            {
            case TRAVERSAL_START:
                system("cls");
                printf("Weighted graph [ G ]:\n\n");
                PrintGraph(G);
                DrawGraph(G,
                          graph_e_pen, graph_w_text_color,
                          graph_v_brush, graph_v_pen, graph_v_text_color,
                          hdc);
                break;
            case TRAVERSAL_CONTINUATION:
                system("cls");
                printf("Current spanning tree [ Gt ]:\n\n");
                PrintGraph(Gt);
                DrawPAStep(Gt, tree_e_pen, tree_w_text_color,
                           tree_v_brush, tree_v_pen, tree_v_text_color,
                           hdc);
                break;
            case TRAVERSAL_END:
                system("cls");
                printf("Minimum spanning tree [ Gt ] of the weighted graph [ G
]:\n\n");
                PrintGraph(Gt);
                DrawGraph(Gt, tree_e_pen, tree_w_text_color,
                          tree_v_brush, tree_v_pen, tree_v_text_color,
                          hdc);
                break;
            }
            EndPaint(hWnd, &ps);
            break;
```

```
    }
    case WM_ERASEBKGND:
    {
        if (current_traversal_state == TRAVERSAL_CONTINUATION)
            return 1;
        else
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    case WM_CREATE:
    {
        CreateGoToStartButton(hWnd);
        CreateNextStepButton(hWnd, TRAVERSAL_START);
        CreateTip(hWnd, TRAVERSAL_START);

        graph_e_pen = CreatePen(PS_SOLID, 1, RGB(0, 38, 0));
        graph_w_text_color = RGB(0, 0, 0);
        graph_v_brush = CreateSolidBrush(RGB(37, 255, 127));
        graph_v_pen = CreatePen(PS_SOLID, 3, RGB(3, 104, 65));
        graph_v_text_color = RGB(3, 104, 65);
        tree_e_pen = CreatePen(PS_SOLID, 3, RGB(242, 0, 0));
        tree_w_text_color = RGB(242, 0, 0);
        tree_v_brush = CreateSolidBrush(RGB(233, 99, 98));
        tree_v_pen = CreatePen(PS_SOLID, 3, RGB(143, 1, 24));
        tree_v_text_color = RGB(255, 255, 255);

        int **vertices_coords = SetVerticesCoords(N);

        srand(N1 * 1000 + N2 * 100 + N3 * 10 + N4);
        double **T = randm(N, N);
        int **A = mulmr(N, N, T, (1.0 - N3 * 0.01 - N4 * 0.005 - 0.05));
        int **uA = SymmetrizeMatrix(N, A);
        int **W = GetMatrixOfWeights(N, A);

        G = ConvertDataToGraphList(N, vertices_names, vertices_coords, uA, W);
        Gt = InitGraph();

        FreeDouble2dArr(N, T);
        FreeInt2dArr(N, A);
        FreeInt2dArr(N, uA);
        FreeInt2dArr(N, W);
        break;
    }
    case WM_DESTROY:
    {
        DeleteObject(graph_e_pen);
        DeleteObject(graph_v_pen);
        DeleteObject(graph_v_brush);
        DeleteObject(tree_e_pen);
        DeleteObject(tree_v_brush);
        DeleteObject(tree_v_pen);
        FreeGraph(G);
        FreeGraph(Gt);
        PostQuitMessage(0);
        break;
    }
    default :
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
}

void CreateGoToStartButton(HWND hWnd)
{
    CreateWindowW(L"BUTTON", L"На початок",
                  WS_VISIBLE | WS_CHILD | WS_BORDER |
```

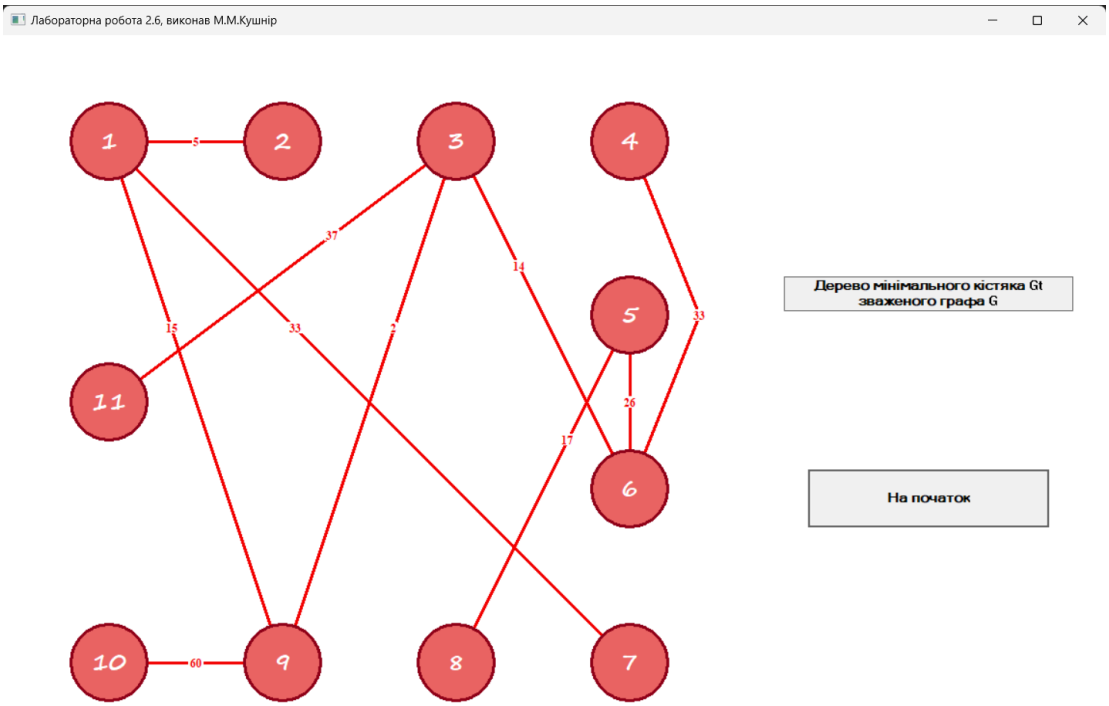
```
                    BS_FLAT | BS_MULTILINE | BS_CENTER,
                    (graph_full_size[x] + 75), 450, 250, 60,
                    hWnd,
                    (HMENU) GO_TO_START,
                    NULL, NULL);
}

void CreateNextStepButton(HWND hWnd, int traversal_state)
{
    wchar_t *start_traversal = L"Почати обхід";
    wchar_t *next_step = L"Наступний крок";
    wchar_t *draw_tree = L"Намалювати дерево кістяка";
    wchar_t *button_text;
    switch (traversal_state)
    {
    case TRAVERSAL_START:
        button_text = start_traversal;
        break;
    case TRAVERSAL_CONTINUATION:
        button_text = next_step;
        break;
    case TRAVERSAL_END:
        button_text = draw_tree;
    }
    hNextStepButton = CreateWindowW(L"BUTTON", button_text,
                                    WS_VISIBLE | WS_CHILD | WS_BORDER |
                                    BS_FLAT | BS_MULTILINE | BS_CENTER,
                                    (graph_full_size[x] + 75), 350, 250, 60,
                                    hWnd,
                                    (HMENU) NEXT_STEP,
                                    NULL, NULL);
}

void CreateTip(HWND hWnd, int traversal_state)
{
    wchar_t *graph_text = L"Зважений ненапрямлений граф G";
    wchar_t *traversal_text = L"Пошук мінімального кістяка за алгоритмом Пріма";
    wchar_t *tree_text = L"Дерево мінімального кістяка Gt зваженого графа G";
    wchar_t *tip_text;
    int lines_quantity;
    switch (traversal_state)
    {
    case TRAVERSAL_START:
        tip_text = graph_text;
        lines_quantity = 1;
        break;
    case TRAVERSAL_CONTINUATION:
        tip_text = traversal_text;
        lines_quantity = 2;
        break;
    case TRAVERSAL_END:
        tip_text = tree_text;
        lines_quantity = 2;
    }
    hTip = CreateWindowW(L"STATIC", tip_text,
                    WS_VISIBLE | WS_CHILD | WS_BORDER | SS_CENTER,
                    (graph_full_size[x] + 50), 250, 300, (18 * lines_quantity),
                    hWnd,
                    NULL, NULL, NULL);
}
```

Результати тестування програми

Зважений ненапрямлений граф



Дерево мінімального кістяка



Процес пошуку мінімального кістяка