**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: Перевірила:

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

Київ 2023

***Постановка задачі***

***1.*** Представити зважений ненапрямлений граф із заданими параметрами

так само, як у лабораторній роботі №1. Відміна: матриця ***А*** 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

*bij* = 0, якщо *wij* = 0,

*bij* = 1, якщо *wij* > 0, *bij* ∈ ***B***, *wij* ∈ ***Wt***;

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

*cij* = 1, якщо *bij* != *bji*,

та *cij* = 0 в іншому випадку;

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

*dij* = 1, якщо *bij* = *bji* = 1,

та *dij* = 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 ***n4*** – 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)***

* *n1* = **2**;
* *n2* = **2**;
* *n3* = **1**;
* *n4* = **3**;

Число вершин *n*: 10 + **1** = **11**.

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

Формування матриці ***А***:

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***](https://github.com/kushnirko/asd-labs/tree/main/semester-2/lab-2.6)

***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***

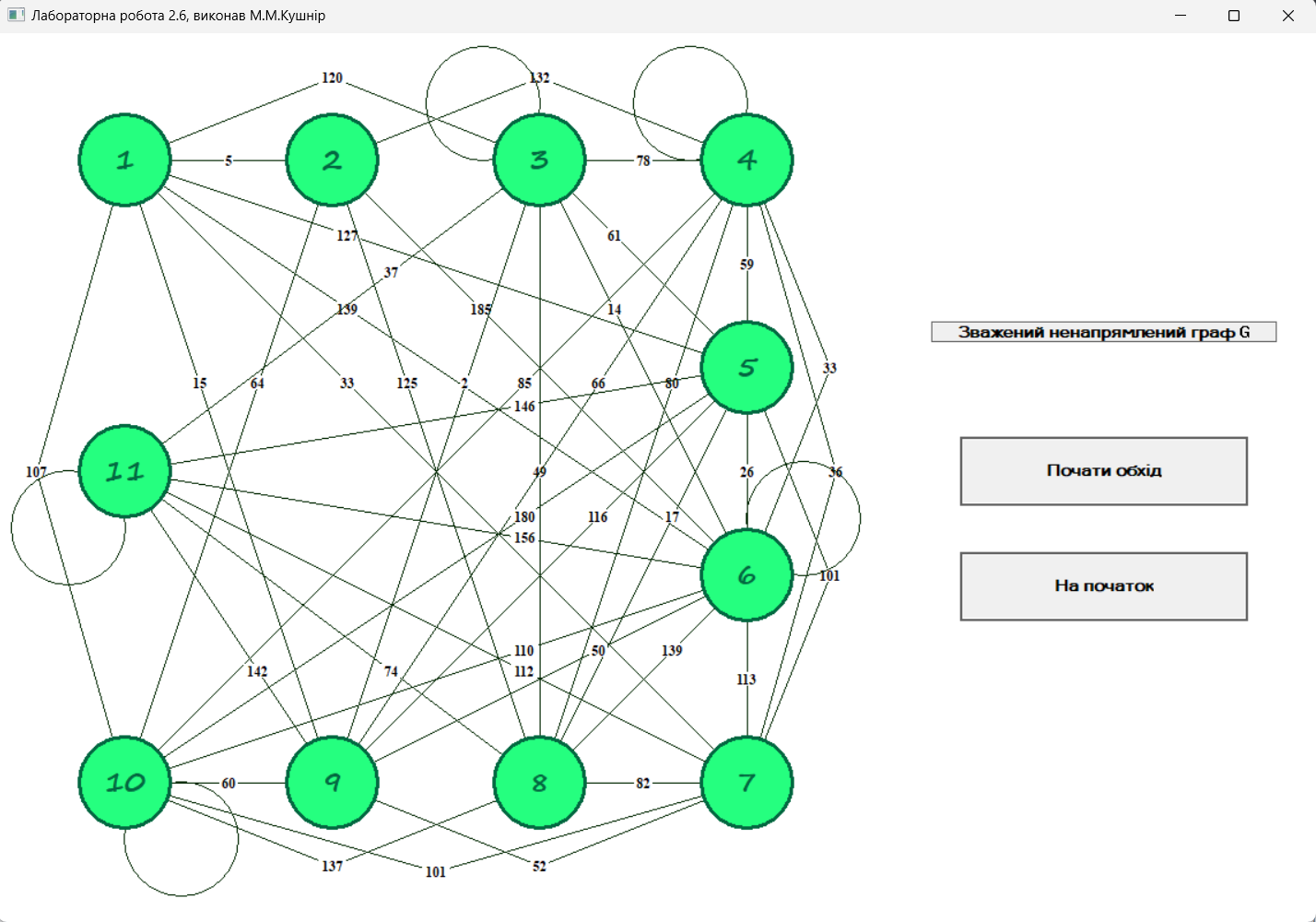
#ifndef **UNICODE**#define **UNICODE**#endif  
  
#include <windows.h>  
#include "GraphPainter.h"  
#include "Prim'sAlgorithm.h"  
  
#define **GO\_TO\_START** 0  
#define **NEXT\_STEP** 1  
  
#define **TRAVERSAL\_START** 1  
#define **TRAVERSAL\_CONTINUATION** 2  
#define **TRAVERSAL\_END** 3  
int 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));

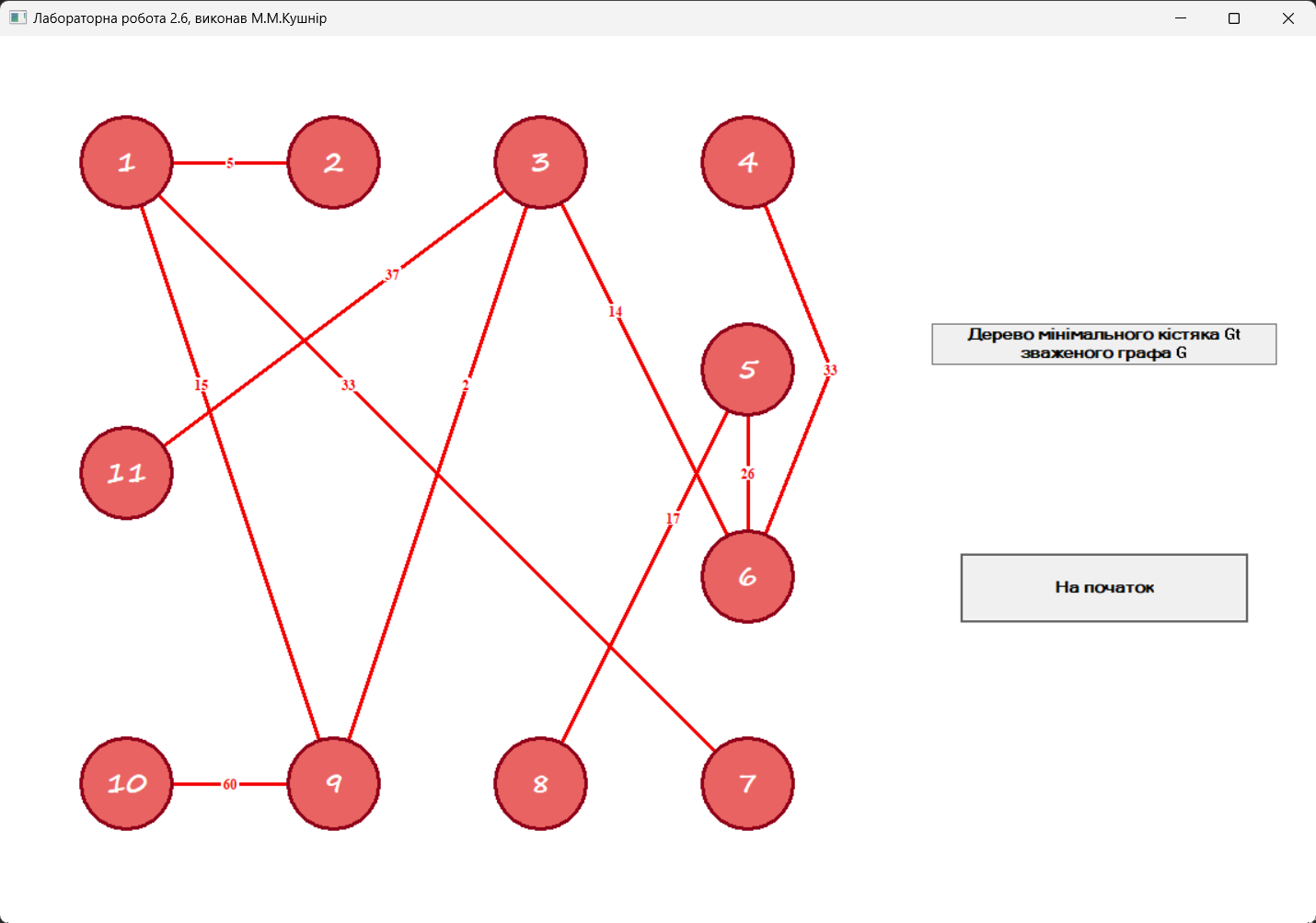
tree\_w\_text\_color = **RGB**(242, 0, 0);  
 tree\_v\_brush = CreateSolidBrush(**RGB**(233, 99, 98));  
 tree\_v\_pen = CreatePen(**PS\_SOLID**, 3, **RGB**(143, 1, 24));  
 tree\_v\_text\_color = **RGB**(255, 255, 255);  
  
 int \*\*vertices\_coords = SetVerticesCoords(**N**);  
  
 srand(**N1** \* 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***

