**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 № 2.5**

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 напрямлений граф з заданими параметрами так само, як у

лабораторній роботі №3. Відміна: матриця ***А*** за варіантом формується за

функцією:

A = mulmr(( 1.0 – n3 \*0.01 – n4 \*0.005 − 0.15) \* T);

***2.*** Створити програми для обходу в глибину та в ширину. Обхід

починати з вершини, яка має вихідні дуги. При цьому у програмі:

* встановити зупинку у точці призначення номеру черговій вершині за

допомогою повідомлення про натискання кнопки,

* виводити зображення графа у графічному вікні перед кожною

зупинкою.

***3.*** Під час обходу графа побудувати дерево обходу. Вивести побудоване

дерево у графічному вікні.

***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 − 0.15) \* T)

***Посилання***

***Текст програми***

***Вміст файлу BFS\_DFS.h***

#include "WorkWithQueueAndStack.h"  
  
int FindMaxSequenceNumber(int n, int \*sequence);  
int FindFirstVertex(int n, int \*\*graph\_matrix, int \*sequence);  
  
int PerformBFSStep(int n, int \*\*graph\_matrix, int \*sequence, queue \*Q, int \*\*tree\_matrix)  
{  
 int sequence\_number = FindMaxSequenceNumber(n, sequence);  
 if (IsQueueEmpty(Q))  
 {  
 int first\_vertex = FindFirstVertex(n, graph\_matrix, sequence);  
 if (first\_vertex != -1)  
 {  
 sequence\_number++;  
 sequence[first\_vertex] = sequence\_number;  
 Enqueue(Q, first\_vertex);  
 }  
 return -1;  
 }  
 int active\_vertex = Q->tail->data;  
 for (int i = 0; i < n; i++)  
 {  
 if (graph\_matrix[active\_vertex][i] == 1)  
 {  
 if (sequence[i] == 0)  
 {  
 sequence\_number++;  
 sequence[i] = sequence\_number;  
 tree\_matrix[active\_vertex][i] = 1;  
 Enqueue(Q, i);  
 return -1;  
 }  
 }  
 }  
 Dequeue(Q);  
 return active\_vertex;  
}  
  
int PerformDFSStep(int n, int \*\*graph\_matrix, int \*sequence, stack \*S, int \*\*tree\_matrix)  
{  
 int sequence\_number = FindMaxSequenceNumber(n, sequence);  
 if (IsStackEmpty(S))  
 {  
 int first\_vertex = FindFirstVertex(n, graph\_matrix, sequence);  
 if (first\_vertex != -1)  
 {  
 sequence\_number++;  
 sequence[first\_vertex] = sequence\_number;  
 Push(S, first\_vertex);  
 }  
 return -1;  
 }  
 int active\_vertex = S->head->data;  
 for (int i = 0; i < n; i++)  
 {  
 if (graph\_matrix[active\_vertex][i] == 1)  
 {  
 if (sequence[i] == 0)  
 {  
 sequence\_number++;  
 sequence[i] = sequence\_number;  
 tree\_matrix[active\_vertex][i] = 1;  
 Push(S, i);  
 return -1;  
 }  
 }  
 }  
 Pop(S);  
 return active\_vertex;  
}  
  
int FindMaxSequenceNumber(int n, int \*sequence)  
{  
 int max\_number = 0;  
 int i;  
 for (i = 0; i < n; i++)  
 if (sequence[i] > max\_number)  
 max\_number = sequence[i];  
 return max\_number;  
}  
  
int FindFirstVertex(int n, int \*\*graph\_matrix, int \*sequence)  
{  
 int i, j;  
 for (i = 0; i < n; i++)  
 if (sequence[i] == 0 )  
 for (j = 0; j < n; j++)  
 if (graph\_matrix[i][j] && (i != j))  
 return i;  
 for (i = 0; i < n; i++)  
 if (sequence[i] == 0 )  
 return i;  
 return -1;  
}  
  
int CheckTraversalState(int n, int \*sequence)  
{  
 int sequence\_nums\_sum = 0;  
 int max\_sequence\_nums\_sum = 0;  
 for (int i = 0; i < n; i++)  
 {  
 sequence\_nums\_sum += sequence[i];  
 max\_sequence\_nums\_sum += (i + 1);  
 }  
 if (sequence\_nums\_sum == 0)  
 return 1; */\* поточний стан обходу - TRAVERSAL\_START \*/* else if (sequence\_nums\_sum == max\_sequence\_nums\_sum)  
 return 3; */\* поточний стан обходу - TRAVERSAL\_END \*/* else  
 return 2; */\* поточний стан обходу - TRAVERSAL\_CONTINUATION \*/*}

***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 використовуються в обчисленнях \*/*#define **PI** 3.1415926536  
#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**)  
};  
  
const int vertex\_print\_offset[] = { 5, 8 };  
  
*/\* Визначають зміщення центру ребер для огинання вершин або вже намальованих ребер \*/*const double edge\_center\_offset\_dividers[] = { 1, 3.5, 4.7, 5.5, 7 };

***Вміст файлу DrawingDataSetter.h***

#include <math.h>  
#include "Configurations.h"  
#include "WorkWithMatrices.h"  
  
typedef struct DrawingData  
{  
 int edge\_type;  
 int start[2];  
 int center[2];  
 int end[2];  
 double angle;  
 int arrow\_end[2];  
} draw\_data;  
  
double Pow2(int value);  
double GetDistance(const int \*v1\_pos, const int \*v2\_pos);  
double ConvertDegreeToRad(double degree\_value);  
  
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(int v1, int v2, int \*\*coords, int drawn\_lines[**N**][**N**])  
{  
 draw\_data data;  
 data.edge\_type = 1;  
 data.start[**x**] = coords[v1][**x**];  
 data.start[**y**] = coords[v1][**y**];  
 data.end[**x**] = coords[v2][**x**];  
 data.end[**y**] = coords[v2][**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 index = (int) (GetDistance(data.start, data.end) / **ONE\_STEP\_LENGTH**);  
 int center\_offset[2] =  
 {  
 abs((int) (dy / edge\_center\_offset\_dividers[index])),  
 abs((int) (dx / edge\_center\_offset\_dividers[index]))  
 };  
 int is\_drawn = 0;  
 if (drawn\_lines[v1][v2] == 1 && drawn\_lines[v2][v1] == 1)  
 {  
 is\_drawn = 1;  
 data.center[**x**] += center\_offset[**x**];  
 data.center[**y**] += center\_offset[**y**];  
 }  
 else  
 drawn\_lines[v1][v2] = drawn\_lines[v2][v1] = 1;  
 int variable\_delta, static\_coord;  
 if (v1 == v2)  
 {  
 data.edge\_type = 2;  
 int loop\_offset\_direction[2] = { 0 };  
 int arrow\_direction[2] = { 0 };  
 if (data.center[**x**] > min\_coords[**x**] && data.center[**y**] == min\_coords[**y**])  
 {  
 --loop\_offset\_direction[**x**];  
 --loop\_offset\_direction[**y**];  
 --arrow\_direction[**y**];  
 data.angle = ConvertDegreeToRad(-87);  
 }  
 else if (data.center[**x**] == max\_coords[**x**] && data.center[**y**] > min\_coords[**y**])  
 {  
 ++loop\_offset\_direction[**x**];  
 --loop\_offset\_direction[**y**];  
 ++arrow\_direction[**x**];  
 data.angle = ConvertDegreeToRad(183);  
 }  
 else if (data.center[**x**] < max\_coords[**x**] && data.center[**y**] == max\_coords[**y**])  
 {  
 ++loop\_offset\_direction[**x**];  
 ++loop\_offset\_direction[**y**];  
 ++arrow\_direction[**y**];  
 data.angle = ConvertDegreeToRad(93);  
 }  
 else if (data.center[**x**] == min\_coords[**x**] && data.center[**y**] < max\_coords[**y**])  
 {  
 --loop\_offset\_direction[**x**];  
 ++loop\_offset\_direction[**y**];  
 --arrow\_direction[**x**];  
 data.angle = ConvertDegreeToRad(3);  
 }  
 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**];  
 data.arrow\_end[**x**] = data.end[**x**] + **VERTEX\_RADIUS** \* arrow\_direction[**x**];  
 data.arrow\_end[**y**] = data.end[**y**] + **VERTEX\_RADIUS** \* arrow\_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 ))  
 {  
 if (abs(variable\_delta) > **MAX\_ONE\_STEP\_LENGTH**)  
 {  
 if (data.start[static\_coord] == min\_coords[static\_coord])  
 {  
 if (!is\_drawn)  
 data.center[static\_coord] -= center\_offset[static\_coord];  
 }  
 else if (data.start[static\_coord] == max\_coords[static\_coord])  
 {  
 if (!is\_drawn)  
 data.center[static\_coord] += center\_offset[static\_coord];  
 else  
 data.center[static\_coord] -= 2 \* center\_offset[static\_coord];  
 }  
 }  
 }  
 else if (dx == dy && is\_drawn)  
 data.center[**x**] -= center\_offset[**x**];  
 else  
 {  
 if (is\_drawn)  
 {  
 int graph\_center[] =  
 {  
 ((min\_coords[**x**] + max\_coords[**x**]) / 2),  
 ((min\_coords[**y**] + max\_coords[**y**]) / 2)  
 };  
 int alternative\_center[] =  
 {  
 (data.center[**x**] - 2 \* center\_offset[**x**]),  
 (data.center[**y**] - 2 \* center\_offset[**y**])  
 };  
 if (GetDistance(data.center, graph\_center) >  
 GetDistance(alternative\_center, graph\_center))  
 {  
 data.center[**x**] = alternative\_center[**x**];  
 data.center[**y**] = alternative\_center[**y**];  
 }  
 }  
 }  
 int new\_dx = data.end[**x**] - data.center[**x**];  
 int new\_dy = data.end[**y**] - data.center[**y**];  
 double hypotenuse = GetDistance(data.center, data.end);  
 if (new\_dx >= 0 && new\_dy >= 0)  
 data.angle = acos(abs(new\_dx) / hypotenuse) \* -1;  
 else if (new\_dx >= 0 && new\_dy < 0)  
 data.angle = acos(abs(new\_dx) / hypotenuse);  
 else if (new\_dx < 0 && new\_dy >= 0)  
 data.angle = (**PI** - acos(abs(new\_dx) / hypotenuse)) \* -1;  
 else if (new\_dx < 0 && new\_dy < 0)  
 data.angle = **PI** - acos(abs(new\_dx) / hypotenuse);  
 data.arrow\_end[**x**] = data.end[**x**] - (int) round((double) **VERTEX\_RADIUS** \* cos(data.angle));  
 data.arrow\_end[**y**] = data.end[**y**] + (int) round((double) **VERTEX\_RADIUS** \* sin(data.angle));  
  
 return data;  
}  
  
double Pow2(int value)  
{  
 return (double)(value \* value);  
}  
  
double GetDistance(const int \*v1\_pos, const int \*v2\_pos)  
{  
 int a = v2\_pos[**x**] - v1\_pos[**x**];  
 int b = v2\_pos[**y**] - v1\_pos[**y**];  
 return sqrt(Pow2(a) + Pow2(b));  
}  
  
double ConvertDegreeToRad(double degree\_value)  
{  
 return **PI** \* degree\_value / 180.0;  
}

***Вміст файлу GraphPainter.h***

#include "DrawingDataSetter.h"  
  
void DrawEdgeLines(int \*start, int \*center, int \*end, HPEN e\_pen, HDC hdc);  
void DrawLoop(int \*center, HPEN e\_pen, HDC hdc);  
void DrawArrow(double angle, int \*arrow\_end, HPEN e\_pen, HDC hdc);  
void DrawVertex(int vertex, int \*\*coords, HBRUSH v\_brush, HPEN v\_pen, HDC hdc);  
  
void DrawGraph(int n,  
 int \*\*graph\_matrix,  
 int \*\*coords,  
 HPEN e\_pen,  
 HBRUSH v\_brush,  
 HPEN v\_pen,  
 HDC hdc)  
{  
 */\* Зображаємо ребра \*/* int drawn\_lines[**N**][**N**] = { 0 };  
 int i, j;  
 for (i = 0; i < n; i++)  
 {  
 for (j = 0; j < n; j++)  
 {  
 if (graph\_matrix[i][j] == 1)  
 {  
 draw\_data data = SetEdgeDrawData(i, j, coords, drawn\_lines);  
 switch (data.edge\_type)  
 {  
 case 1:  
 DrawEdgeLines(data.start, data.center, data.end, e\_pen, hdc);  
 break;  
 case 2:  
 DrawLoop(data.center, e\_pen, hdc);  
 break;  
 }  
 DrawArrow(data.angle, data.arrow\_end, e\_pen, hdc);  
 }  
 }  
 }  
 */\* Зображаємо вершини \*/* for (i = 0; i < n; i++)  
 DrawVertex(i, coords, v\_brush, v\_pen, hdc);  
}  
  
void DrawBFSStep(int \*\*graph\_matrix, int \*\*coords, queue \*Q,  
 HPEN tree\_e\_pen,  
 HBRUSH active\_v\_brush, HPEN active\_v\_pen,  
 HBRUSH visited\_v\_brush, HPEN visited\_v\_pen,  
 HDC hdc)  
{  
 int active\_vertex = Q->tail->data;  
 int visited\_vertex = Q->head->data;  
 int drawn\_lines[**N**][**N**] = { 0 };  
 if (active\_vertex > visited\_vertex)  
 if (graph\_matrix[visited\_vertex][active\_vertex])  
 drawn\_lines[visited\_vertex][active\_vertex] =  
 drawn\_lines[active\_vertex][visited\_vertex] = 1;  
 draw\_data data = SetEdgeDrawData(active\_vertex, visited\_vertex,  
 coords, drawn\_lines);  
 if (active\_vertex == visited\_vertex)  
 {  
 */\* Якщо вершина належить до нової компоненти, зображаємо лише її \*/* DrawVertex(active\_vertex,  
 coords,  
 active\_v\_brush, active\_v\_pen, hdc);  
 }  
 else  
 {  
 */\* Виділяємо ребро між активною та відвіданою вершиною \*/* DrawEdgeLines(data.start, data.center, data.end, tree\_e\_pen, hdc);  
 DrawArrow(data.angle, data.arrow\_end, tree\_e\_pen, hdc);  
 */\* Виділяємо активну вершину \*/* DrawVertex(active\_vertex,  
 coords,  
 active\_v\_brush, active\_v\_pen, hdc);  
 */\* Виділяємо щойно відвідану вершину \*/* DrawVertex(visited\_vertex,  
 coords,  
 visited\_v\_brush, visited\_v\_pen, hdc);  
 }  
}  
  
void DrawDFSStep(int \*\*graph\_matrix, int \*\*coords, stack \*S,  
 HPEN tree\_e\_pen,  
 HBRUSH active\_v\_brush, HPEN active\_v\_pen,  
 HBRUSH visited\_v\_brush, HPEN visited\_v\_pen,  
 HDC hdc)  
{  
 int active\_vertex = S->head->data;  
 if (S->head->next == **NULL**)  
 {  
 */\* Якщо вершина належить до нової компоненти, зображаємо лише її \*/* DrawVertex(active\_vertex,  
 coords,  
 active\_v\_brush, active\_v\_pen, hdc);  
 }  
 else  
 {  
 int previous\_active\_vertex = S->head->next->data;  
 int drawn\_lines[**N**][**N**] = { 0 };  
 if (previous\_active\_vertex > active\_vertex)  
 if (graph\_matrix[active\_vertex][previous\_active\_vertex])  
 drawn\_lines[active\_vertex][previous\_active\_vertex] =  
 drawn\_lines[previous\_active\_vertex][active\_vertex] = 1;  
 draw\_data data = SetEdgeDrawData(previous\_active\_vertex, active\_vertex,  
 coords, drawn\_lines);  
 */\* Виділяємо ребро між попередньою та поточною активними вершинами \*/* DrawEdgeLines(data.start, data.center, data.end, tree\_e\_pen, hdc);  
 DrawArrow(data.angle, data.arrow\_end, tree\_e\_pen, hdc);  
 */\* Повертаємо попередній активній вершині статус відвіданої \*/* DrawVertex(previous\_active\_vertex,  
 coords,  
 visited\_v\_brush, visited\_v\_pen, hdc);  
 */\* Виділяємо поточну активну вершину \*/* DrawVertex(active\_vertex,  
 coords,  
 active\_v\_brush, active\_v\_pen, hdc);  
 }  
}  
  
void DrawEdgeLines(int \*start, int \*center, int \*end, HPEN e\_pen, HDC hdc)  
{  
 SelectObject(hdc, e\_pen);  
 MoveToEx(hdc, start[**x**], start[**y**], **NULL**);  
 LineTo(hdc, center[**x**], center[**y**]);  
 MoveToEx(hdc, center[**x**], center[**y**], **NULL**);  
 LineTo(hdc, end[**x**], end[**y**]);  
}  
  
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 DrawArrow(double angle, int \*arrow\_end, HPEN e\_pen, HDC hdc)  
{  
 SelectObject(hdc, e\_pen);  
 double fi = **PI** - angle;  
 int leftLineEnd[2], rightLineEnd[2];  
 rightLineEnd[**x**] = arrow\_end[**x**] + (int) (30 \* cos(fi + 0.3));  
 rightLineEnd[**y**] = arrow\_end[**y**] + (int) (30 \* sin(fi + 0.3));  
 leftLineEnd[**x**] = arrow\_end[**x**] + (int) (30 \* cos(fi - 0.3));  
 leftLineEnd[**y**] = arrow\_end[**y**] + (int) (30 \* sin(fi - 0.3));  
 MoveToEx(hdc, leftLineEnd[**x**], leftLineEnd[**y**], **NULL**);  
 LineTo(hdc, arrow\_end[**x**], arrow\_end[**y**]);  
 LineTo(hdc, rightLineEnd[**x**], rightLineEnd[**y**]);  
}  
  
void DrawVertex(int vertex, int \*\*coords, HBRUSH v\_brush, HPEN v\_pen, HDC hdc)  
{  
 int left = (coords[vertex][**x**] - **VERTEX\_RADIUS**);  
 int top = (coords[vertex][**y**] - **VERTEX\_RADIUS**);  
 int right = (coords[vertex][**x**] + **VERTEX\_RADIUS**);  
 int bottom = (coords[vertex][**y**] + **VERTEX\_RADIUS**);  
 int print\_pos[2];  
 if (vertex > 8)  
 print\_pos[**x**] = coords[vertex][**x**] - (int)(1.5 \* vertex\_print\_offset[**x**]);  
 */\* Ці елементи складаються з двох цифр, тому зміщення має бути більшим \*/* else  
 print\_pos[**x**] = coords[vertex][**x**] - vertex\_print\_offset[**x**];  
 print\_pos[**y**] = coords[vertex][**y**] - vertex\_print\_offset[**y**];  
 SelectObject(hdc, v\_brush);  
 SelectObject(hdc, v\_pen);  
 Ellipse(hdc, left, top, right, bottom);  
 **TextOut**(hdc, print\_pos[**x**], print\_pos[**y**], vertices\_names[vertex], 2);  
}

***Вміст файлу PrimitiveTableOutput.h***

*/\* Типи лінії таблиці \*/*#define **FIRST\_LINE** 1  
#define **MIDDLE\_LINE** 2  
#define **LAST\_LINE** 3  
*/\* ASCII символи для зображення кожного типу лінії таблиці \*/*const int first\_line\_chars[] = {218, 196, 194, 191};  
const int middle\_line\_chars[] = {195, 196, 197, 180};  
const int last\_line\_chars[] = {192, 196, 193, 217};  
  
void PrintTableLine(int cols\_quantity, const int cols\_lengths[cols\_quantity], int line\_type)  
{  
 const int \*pointer;  
 switch (line\_type)  
 {  
 case **FIRST\_LINE** :  
 pointer = first\_line\_chars;  
 break;  
 case **MIDDLE\_LINE** :  
 pointer = middle\_line\_chars;  
 break;  
 case **LAST\_LINE** :  
 pointer = last\_line\_chars;  
 }  
 int symbols[4];  
 int i, j;  
 for (i = 0; i < 4; i++)  
 symbols[i] = pointer[i];  
 printf("%c", symbols[0]);  
 for (i = 0; i < cols\_quantity; i++)  
 {  
 for (j = 0; j < cols\_lengths[i]; j++)  
 printf("%c", symbols[1]);  
 if ((i + 1) != cols\_quantity)  
 printf("%c", symbols[2]);  
 }  
 printf("%c\n", symbols[3]);  
}

***Вміст файлу WorkWithMatrices.h***

#include <stdio.h>  
#include <stdlib.h>  
#include "PrimitiveTableOutput.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;  
}  
  
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 PrintSequence(int n, int \*sequence)  
{  
 int cols\_lengths[] = { 8, 17 };  
 PrintTableLine(2, cols\_lengths, 1);  
 printf("%c Vertex %c Sequence number %c\n", 179, 179, 179);  
 for (int i = 0; i < n; i++)  
 {  
 PrintTableLine(2, cols\_lengths, 2);  
 printf("%c %2d %c %2d %c\n", 179, (i + 1), 179, sequence[i], 179);  
 }  
 PrintTableLine(2, cols\_lengths, 3);  
}  
  
void PrintBooleanMatrix(int n, int \*\*matrix)  
{  
 int i, j;  
 int cols\_quantity = n;  
 int \*cols\_lengths = (int \*) calloc(cols\_quantity, sizeof(int));  
 printf(" %c", 179);  
 for (i = 0; i < n; i++)  
 {  
 cols\_lengths[i] = 3;  
 printf("%2d %c", (i + 1), 179);  
 }  
 printf("\n");  
 for (i = 0; i < n; i++)  
 {  
 printf(" %c%c%c", 196, 196, 196);  
 PrintTableLine(cols\_quantity, cols\_lengths, 2);  
 printf(" %2d %c", (i + 1), 179);  
 for (j = 0; j < n; j++)  
 printf("%2d %c", matrix[i][j], 179);  
 printf("\n");  
 }  
 printf(" %c%c%c", 196, 196, 196);  
 PrintTableLine(cols\_quantity, cols\_lengths, 3);  
 free(cols\_lengths);  
}  
  
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 WorkWithQueueAndStack.h***

typedef struct Node  
{  
 int data;  
 struct Node \*next;  
} node;  
  
*/\*\*\*\*\*\*\*\*\*\* Функції для роботи з чергою \*\*\*\*\*\*\*\*\*\*/*typedef struct Queue  
{  
 node \*head;  
 node \*tail;  
} queue;  
  
queue\* InitQueue()  
{  
 queue\* q = (queue \*) malloc(sizeof(queue));  
 q->tail = q->head = **NULL**;  
 return q;  
}  
  
int IsQueueEmpty(queue \*q)  
{  
 return (q->tail == **NULL**);  
}  
  
void Enqueue(queue \*q, int data)  
{  
 node \*newNode;  
 newNode = (node \*) malloc(sizeof(node));  
 newNode->data = data;  
 newNode->next = **NULL**;  
 if (IsQueueEmpty(q))  
 q->head = q->tail = newNode;  
 else  
 {  
 q->head->next = newNode;  
 q->head = newNode;  
 }  
}  
  
int Dequeue(queue \*q)  
{  
 if (!IsQueueEmpty(q))  
 {  
 node \*temp = q->tail;  
 q->tail = q->tail->next;  
 free(temp);  
 return 0;  
 }  
 q->head = **NULL**;  
 return -1;  
}  
  
void FreeQueue(queue \*q)  
{  
 int status\_of\_freeing = 0;  
 while (status\_of\_freeing != -1)  
 status\_of\_freeing = Dequeue(q);  
}  
  
*/\*\*\*\*\*\*\*\*\*\* Функції для роботи зі стеком \*\*\*\*\*\*\*\*\*\*/*typedef struct Stack  
{  
 node \*head;  
} stack;  
  
stack \*InitStack()  
{  
 stack \*s = (stack \*) malloc(sizeof(stack));  
 s->head = **NULL**;  
 return s;  
}  
  
int IsStackEmpty(stack \*s)  
{  
 return (s->head == **NULL**);  
}  
  
void Push(stack \*s, int data)  
{  
 node \*newNode;  
 newNode = (node \*) malloc(sizeof(node));  
 newNode->data = data;  
 newNode->next = s->head;  
 s->head = newNode;  
}  
  
int Pop(stack \*s)  
{  
 if (!IsStackEmpty(s))  
 {  
 node \*temp = s->head;  
 s->head = s->head->next;  
 free(temp);  
 return 0;  
 }  
 return -1;  
}  
  
void FreeStack(stack \*s)  
{  
 int status\_of\_freeing = 0;  
 while (status\_of\_freeing != -1)  
 status\_of\_freeing = Pop(s);  
}

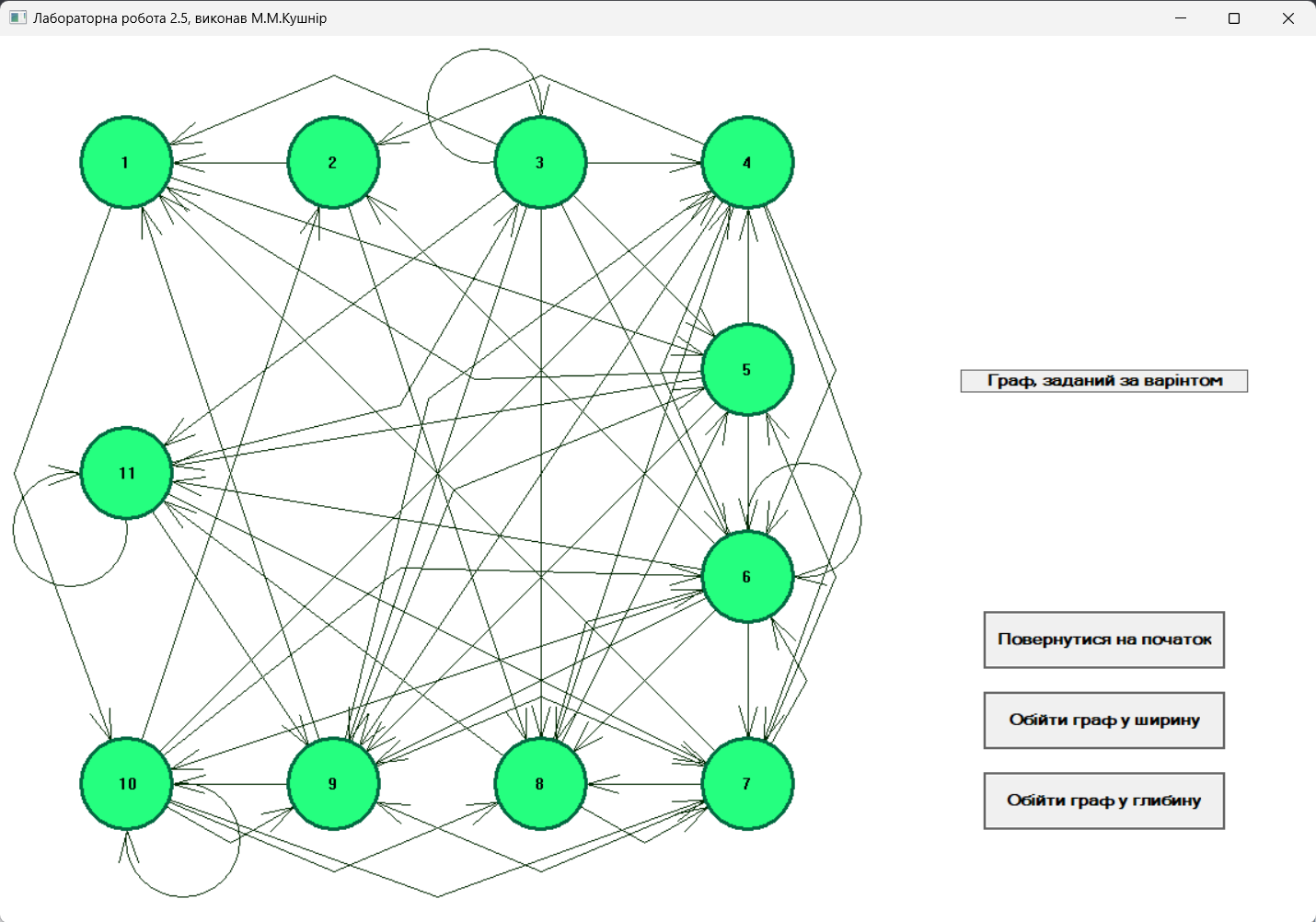
***Вміст файлу main.c***

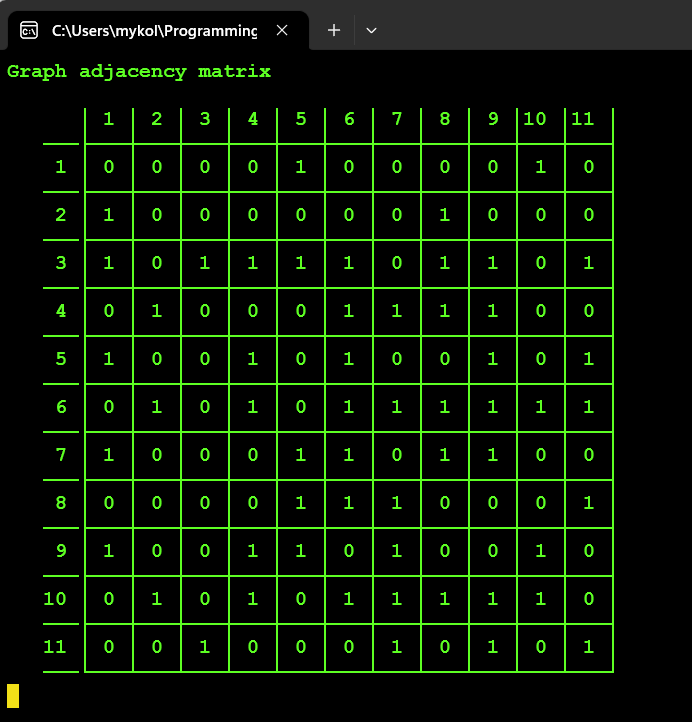
#ifndef **UNICODE**#define **UNICODE**#endif  
  
#include <windows.h>  
#include "BFS\_DFS.h"

#include "GraphPainter.h"  
  
#define **NO\_ALGORITHM** 0  
#define **BFS\_ALGORITHM** 1  
#define **DFS\_ALGORITHM** 2  
#define **NEXT\_STEP** 3  
int current\_algorithm = **NO\_ALGORITHM**;  
  
#define **TRAVERSAL\_START** 1  
#define **TRAVERSAL\_CONTINUATION** 2  
#define **TRAVERSAL\_END** 3  
int current\_traversal\_state = **TRAVERSAL\_END**;  
  
int \*\*vertices\_coords;  
int \*\*A;  
int \*\*tree\_matrix;  
queue \*Q;  
stack \*S;  
int sequence[**N**];  
int closed\_vertex = -1;  
  
HWND hNextStepButton = **NULL**;  
HWND hGreenLabel = **NULL**;  
HWND hBlueLabel = **NULL**;  
HWND hRedLabel = **NULL**;  
HWND hYellowLabel = **NULL**;  
HWND hProcessNameLabel;  
  
LRESULT **CALLBACK** WndProc(HWND, UINT, WPARAM, LPARAM);  
  
void CreateNextStepButton(HWND hWnd, int traversal\_state);  
void CreateColorTips(HWND hWnd);  
void CreateTip(HWND hWnd, int algorithm, int traversal\_state);  
  
int **WINAPI** WinMain(HINSTANCE hInstance,  
 HINSTANCE hPrevInstance,  
 LPSTR lpCmdLine,  
 int nCmdShow)  
{  
 WNDCLASS wndClass;  
 wndClass.lpszClassName = L"Лабораторна робота 2.5";  
 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 2.5",  
 L"Лабораторна робота 2.5, 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**);  
  
 int start\_pos[] = { (graph\_full\_size[**x**] + 95), 500 };  
 **CreateWindowW**(L"BUTTON", L"Повернутися на початок",  
 WS\_VISIBLE | WS\_CHILD | WS\_BORDER |  
 BS\_FLAT | BS\_MULTILINE | BS\_CENTER,  
 start\_pos[x], start\_pos[y], 210, 50,  
 hWnd,  
 (HMENU) NO\_ALGORITHM,  
 NULL, NULL);  
  
 **CreateWindowW**(L"BUTTON", L"Обійти граф у ширину",  
 WS\_VISIBLE | WS\_CHILD | WS\_BORDER |  
 BS\_FLAT | BS\_MULTILINE | BS\_CENTER,  
 start\_pos[x], (start\_pos[y] + 70), 210, 50,  
 hWnd,  
 (HMENU) BFS\_ALGORITHM,  
 NULL, NULL);  
  
 **CreateWindowW**(L"BUTTON", L"Обійти граф у глибину",  
 WS\_VISIBLE | WS\_CHILD | WS\_BORDER |  
 BS\_FLAT | BS\_MULTILINE | BS\_CENTER,  
 start\_pos[x], (start\_pos[y] + 140), 210, 50,  
 hWnd,  
 (HMENU) DFS\_ALGORITHM,  
 NULL, NULL);  
  
 CreateTip(hWnd, **NO\_ALGORITHM**, **TRAVERSAL\_END**);  
  
 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 HBRUSH graph\_v\_brush;  
 static HPEN graph\_v\_pen;  
 static HBRUSH active\_v\_brush;  
 static HPEN active\_v\_pen;  
 static HBRUSH visited\_v\_brush;  
 static HPEN visited\_v\_pen;  
 static HBRUSH tree\_v\_brush;  
 static HPEN tree\_v\_pen;  
 static HPEN tree\_e\_pen;  
  
 HDC hdc;  
 PAINTSTRUCT ps;  
 switch (message)  
 {  
 case **WM\_COMMAND**:  
 {  
 switch (wParam)  
 {  
 case **NO\_ALGORITHM**:  
 {  
 current\_algorithm = **NO\_ALGORITHM**;  
 current\_traversal\_state = **TRAVERSAL\_END**;  
 RedrawWindow(hWnd, **NULL**, **NULL**,  
 **RDW\_ERASE** | **RDW\_INVALIDATE**);  
 if (hNextStepButton != **NULL**)  
 {  
 DestroyWindow(hNextStepButton);  
 hNextStepButton = **NULL**;  
 }  
 if (hGreenLabel != **NULL**)  
 {  
 DestroyWindow(hNextStepButton);  
 hNextStepButton = **NULL**;  
 DestroyWindow(hGreenLabel);  
 hGreenLabel = **NULL**;  
 DestroyWindow(hBlueLabel);  
 hBlueLabel = **NULL**;  
 DestroyWindow(hRedLabel);  
 hRedLabel = **NULL**;  
 DestroyWindow(hYellowLabel);  
 hYellowLabel = **NULL**;  
 }  
 DestroyWindow(hProcessNameLabel);  
 CreateTip(hWnd, **NO\_ALGORITHM**, **TRAVERSAL\_END**);  
 break;  
 }  
 case **BFS\_ALGORITHM**:  
 case **DFS\_ALGORITHM**:  
 {  
 if (wParam == **BFS\_ALGORITHM**)  
 {  
 current\_algorithm = **BFS\_ALGORITHM**;  
 FreeQueue(Q);  
 DestroyWindow(hProcessNameLabel);  
 CreateTip(hWnd, **BFS\_ALGORITHM**, **TRAVERSAL\_START**);  
 }  
 else  
 {  
 current\_algorithm = **DFS\_ALGORITHM**;  
 FreeStack(S);  
 DestroyWindow(hProcessNameLabel);  
 CreateTip(hWnd, **DFS\_ALGORITHM**, **TRAVERSAL\_START**);  
 }  
 current\_traversal\_state = **TRAVERSAL\_START**;  
 RedrawWindow(hWnd, **NULL**, **NULL**,  
 **RDW\_ERASE** | **RDW\_INVALIDATE**);  
 if (hNextStepButton != **NULL**)  
 DestroyWindow(hNextStepButton);  
 CreateNextStepButton(hWnd, **TRAVERSAL\_START**);  
 if (hGreenLabel == **NULL**)  
 CreateColorTips(hWnd);  
 int i, j;  
 for (i = 0; i < **N**; i++)  
 {  
 sequence[i] = 0;  
 for (j = 0; j < **N**; j++)  
 tree\_matrix[i][j] = 0;  
 }  
 system("cls");  
 PrintSequence(**N**, sequence);  
 break;  
 }  
 case **NEXT\_STEP**:  
 {  
 switch (current\_traversal\_state)  
 {  
 case **TRAVERSAL\_START**:  
 DestroyWindow(hNextStepButton);  
 CreateNextStepButton(hWnd, **TRAVERSAL\_CONTINUATION**);  
 current\_traversal\_state = **TRAVERSAL\_CONTINUATION**;  
 case **TRAVERSAL\_CONTINUATION**:  
 {  
 int is\_empty;  
 if (current\_algorithm == **BFS\_ALGORITHM**)  
 {  
 closed\_vertex = PerformBFSStep(**N**, A, sequence, Q, tree\_matrix);  
 is\_empty = IsQueueEmpty(Q);  
 }  
 else  
 {  
 closed\_vertex = PerformDFSStep(**N**, A, sequence, S, tree\_matrix);  
 is\_empty = IsStackEmpty(S);  
 }  
 system("cls");  
 PrintSequence(**N**, sequence);  
 InvalidateRect(hWnd, **NULL**, **FALSE**);  
 UpdateWindow(hWnd);  
 if (CheckTraversalState(**N**, sequence) == **TRAVERSAL\_END** && is\_empty)  
 {  
 DestroyWindow(hNextStepButton);  
 CreateNextStepButton(hWnd, **TRAVERSAL\_END**);  
 current\_traversal\_state = **TRAVERSAL\_END**;  
 }  
 break;  
 }  
 case **TRAVERSAL\_END**:  
 system("cls");  
 printf("Traversal tree matrix\n\n");  
 PrintBooleanMatrix(**N**, tree\_matrix);  
 DestroyWindow(hProcessNameLabel);  
 if (current\_algorithm == **BFS\_ALGORITHM**)  
 CreateTip(hWnd, **BFS\_ALGORITHM**, **TRAVERSAL\_END**);  
 else  
 CreateTip(hWnd, **DFS\_ALGORITHM**, **TRAVERSAL\_END**);  
 DestroyWindow(hNextStepButton);  
 hNextStepButton = **NULL**;  
 DestroyWindow(hGreenLabel);  
 hGreenLabel = **NULL**;  
 DestroyWindow(hBlueLabel);  
 hBlueLabel = **NULL**;  
 DestroyWindow(hRedLabel);  
 hRedLabel = **NULL**;  
 DestroyWindow(hYellowLabel);  
 hYellowLabel = **NULL**;  
 RedrawWindow(hWnd, **NULL**, **NULL**,  
 **RDW\_ERASE** | **RDW\_INVALIDATE**);  
 }  
 }  
 }  
 }  
 case **WM\_PAINT**:  
 {  
 hdc = BeginPaint(hWnd, &ps);  
 SetBkMode(hdc, **TRANSPARENT**);  
 switch (current\_algorithm)  
 {  
 case **NO\_ALGORITHM**:  
 {  
 system("cls");  
 printf("Graph adjacency matrix\n\n");  
 PrintBooleanMatrix(**N**, A);  
 DrawGraph(**N**, A, vertices\_coords,  
 graph\_e\_pen, graph\_v\_brush, graph\_v\_pen, hdc);  
 break;  
 }  
 case **BFS\_ALGORITHM**:  
 {  
 switch (current\_traversal\_state)  
 {  
 case **TRAVERSAL\_START**:  
 DrawGraph(**N**, A, vertices\_coords,  
 graph\_e\_pen, graph\_v\_brush, graph\_v\_pen, hdc);  
 break;  
 case **TRAVERSAL\_CONTINUATION**:  
 if (closed\_vertex != -1)  
 {  
 */\* Виділяємо останню закриту вершину \*/* DrawVertex(closed\_vertex,  
 vertices\_coords,  
 tree\_v\_brush, tree\_v\_pen, hdc);  
 */\* Позначаємо першу у черзі відвідану вершину як активну \*/* if (!IsQueueEmpty(Q))  
 DrawVertex(Q->tail->data,  
 vertices\_coords,  
 active\_v\_brush, active\_v\_pen, hdc);  
 }  
 else  
 DrawBFSStep(A, vertices\_coords, Q,  
 tree\_e\_pen, active\_v\_brush, active\_v\_pen,  
 visited\_v\_brush, visited\_v\_pen,  
 hdc);  
 break;  
 case **TRAVERSAL\_END**:  
 DrawGraph(**N**, tree\_matrix, vertices\_coords,  
 tree\_e\_pen, tree\_v\_brush, tree\_v\_pen, hdc);  
 break;  
 }  
 break;  
 }  
 case **DFS\_ALGORITHM**:  
 {  
 switch (current\_traversal\_state)  
 {  
 case **TRAVERSAL\_START**:  
 DrawGraph(**N**, A, vertices\_coords,  
 graph\_e\_pen, graph\_v\_brush, graph\_v\_pen, hdc);  
 break;  
 case **TRAVERSAL\_CONTINUATION**:  
 if (closed\_vertex != -1)  
 {  
 */\* Виділяємо останню закриту вершину \*/* DrawVertex(closed\_vertex,  
 vertices\_coords,  
 tree\_v\_brush, tree\_v\_pen, hdc);  
 */\* Позначаємо першу у стеку відвідану вершину як активну \*/* if (!IsStackEmpty(S))  
 DrawVertex(S->head->data,  
 vertices\_coords,  
 active\_v\_brush, active\_v\_pen, hdc);  
 }  
 else  
 DrawDFSStep(A, vertices\_coords, S,  
 tree\_e\_pen,  
 active\_v\_brush, active\_v\_pen,  
 visited\_v\_brush, visited\_v\_pen,  
 hdc);  
 break;  
 case **TRAVERSAL\_END**:  
 DrawGraph(**N**, tree\_matrix, vertices\_coords,  
 tree\_e\_pen, tree\_v\_brush, tree\_v\_pen, hdc);  
 break;  
 }  
 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**:  
 {  
 graph\_e\_pen = CreatePen(**PS\_SOLID**, 1, **RGB**(0, 38, 0));  
 graph\_v\_brush = CreateSolidBrush(**RGB**(37, 255, 127));  
 graph\_v\_pen = CreatePen(**PS\_SOLID**, 3, **RGB**(3, 104, 65));  
 active\_v\_brush = CreateSolidBrush(**RGB**(233, 210, 39));  
 active\_v\_pen = CreatePen(**PS\_SOLID**, 3, **RGB**(228, 114, 0));  
 visited\_v\_brush = CreateSolidBrush(**RGB**(95, 141, 225));  
 visited\_v\_pen = CreatePen(**PS\_SOLID**, 3, **RGB**(7, 0, 186));  
 tree\_e\_pen = CreatePen(**PS\_SOLID**, 2, **RGB**(242, 0, 0));  
 tree\_v\_brush = CreateSolidBrush(**RGB**(233, 99, 98));  
 tree\_v\_pen = CreatePen(**PS\_SOLID**, 3, **RGB**(143, 1, 24));  
  
 vertices\_coords = SetVerticesCoords(**N**);  
 srand(**N1** \* 1000 + **N2** \* 100 + **N3** \* 10 + **N4**);  
 double \*\*T = randm(**N**, **N**);  
 A = mulmr(**N**, **N**, T, (1.0 - **N3** \* 0.01 - **N4** \* 0.005 - 0.15));  
 FreeDouble2dArr(**N**, T);  
 tree\_matrix = Create2dIntArr(**N**, **N**);  
 Q = InitQueue();  
 S = InitStack();  
 break;  
 }  
 case **WM\_DESTROY**:  
 {  
 DeleteObject(graph\_e\_pen);  
 DeleteObject(graph\_v\_pen);  
 DeleteObject(graph\_v\_brush);  
 DeleteObject(active\_v\_brush);  
 DeleteObject(active\_v\_pen);  
 DeleteObject(visited\_v\_brush);  
 DeleteObject(visited\_v\_pen);  
 DeleteObject(tree\_e\_pen);  
 DeleteObject(tree\_v\_brush);  
 DeleteObject(tree\_v\_pen);  
  
 FreeInt2dArr(**N**, vertices\_coords);  
 FreeInt2dArr(**N**, A);  
 FreeInt2dArr(**N**, tree\_matrix);  
 free(Q);  
 free(S);  
 PostQuitMessage(0);  
 break;  
 }  
 default :  
 return **DefWindowProc**(hWnd, message, wParam, lParam);  
 }  
}  
  
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 обходу";  
 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), 400, 250, 60,  
 hWnd,  
 (HMENU) NEXT\_STEP,  
 NULL, NULL);  
}  
  
void CreateColorTips(HWND hWnd)  
{  
 int start\_pos[] = { (graph\_full\_size[**x**] + 75), 100 };  
 hGreenLabel = **CreateWindowW**(L"STATIC", L" Зелений - нові вершини",  
 WS\_VISIBLE | WS\_CHILD | WS\_BORDER,  
 start\_pos[x], start\_pos[y], 250, 20,  
 hWnd,  
 NULL, NULL, NULL);  
 hBlueLabel = **CreateWindowW**(L"STATIC", L" Синій - відвідані вершини",  
 WS\_VISIBLE | WS\_CHILD | WS\_BORDER,  
 start\_pos[x], (start\_pos[y] + 30), 250, 20,  
 hWnd,  
 NULL, NULL, NULL);  
 hRedLabel = **CreateWindowW**(L"STATIC", L" Червоний - закриті вершини",  
 WS\_VISIBLE | WS\_CHILD | WS\_BORDER,  
 start\_pos[x], (start\_pos[y] + 60), 250, 20,  
 hWnd,  
 NULL, NULL, NULL);  
 hYellowLabel = **CreateWindowW**(L"STATIC", L" Жовтий - активна вершина",  
 WS\_VISIBLE | WS\_CHILD | WS\_BORDER,  
 start\_pos[x], (start\_pos[y] + 90), 250, 20,  
 hWnd,  
 NULL, NULL, NULL);  
}  
  
void CreateTip(HWND hWnd, int algorithm, int traversal\_state)  
{  
 wchar\_t \*no\_algorithm = L"Граф, заданий за варінтом";  
 wchar\_t \*bfs\_traversal = L"Обхід графа в ширину";  
 wchar\_t \*dfs\_traversal = L"Обхід графа в глибину";  
 wchar\_t \*bfs\_tree = L"Дерево обходу в ширину";  
 wchar\_t \*dfs\_tree = L"Дерево обходу в глибину";  
 wchar\_t \*tip\_text;  
 switch (algorithm)  
 {  
 case **NO\_ALGORITHM**:  
 tip\_text = no\_algorithm;  
 break;  
 case **BFS\_ALGORITHM**:  
 if (traversal\_state == **TRAVERSAL\_END**)  
 tip\_text = bfs\_tree;  
 else  
 tip\_text = bfs\_traversal;  
 break;  
 case **DFS\_ALGORITHM**:  
 if (traversal\_state == **TRAVERSAL\_END**)  
 tip\_text = dfs\_tree;  
 else  
 tip\_text = dfs\_traversal;  
 }  
 hProcessNameLabel = **CreateWindowW**(L"STATIC", tip\_text,  
 WS\_VISIBLE | WS\_CHILD | WS\_BORDER | SS\_CENTER,  
 (graph\_full\_size[x] + 75), 290, 250, 20,  
 hWnd,  
 NULL, NULL, NULL);  
}

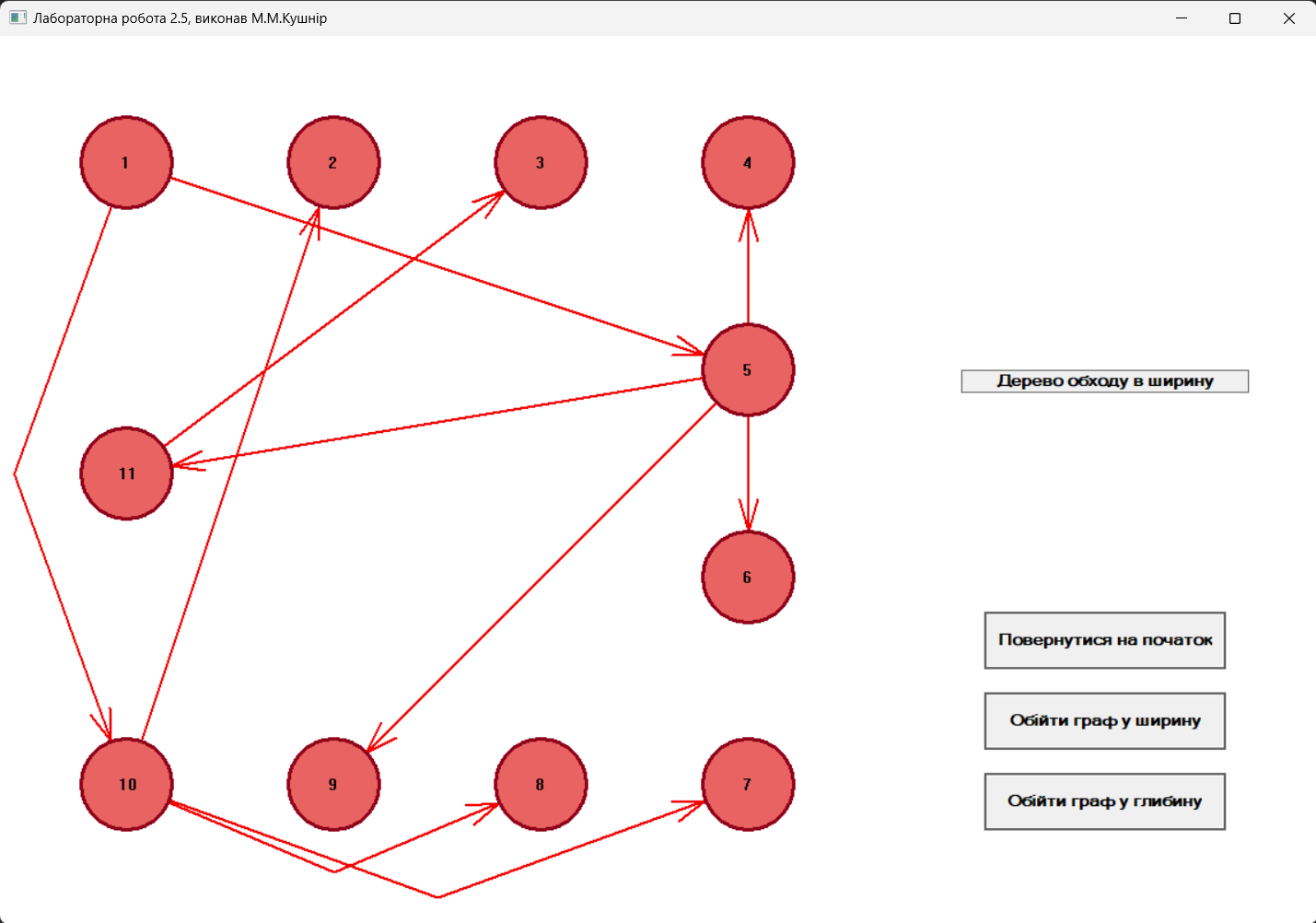
***Результати тестування програми***

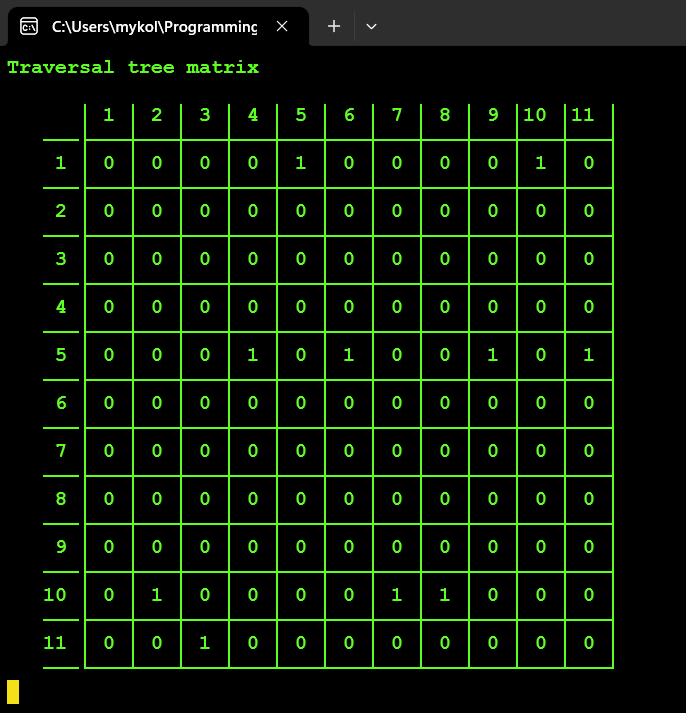
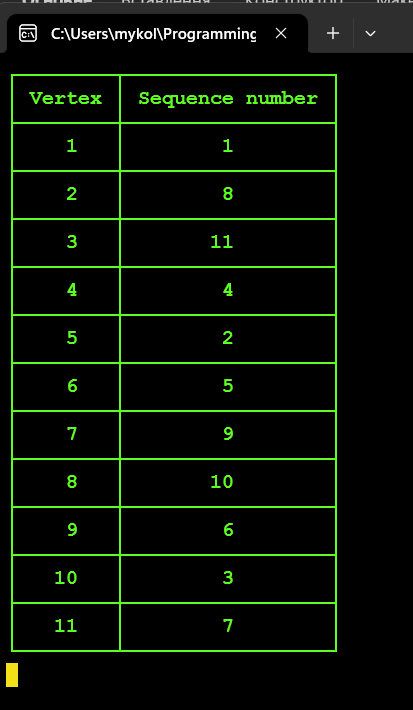
***Напрямлений граф та його згенерована матриця суміжності***

******

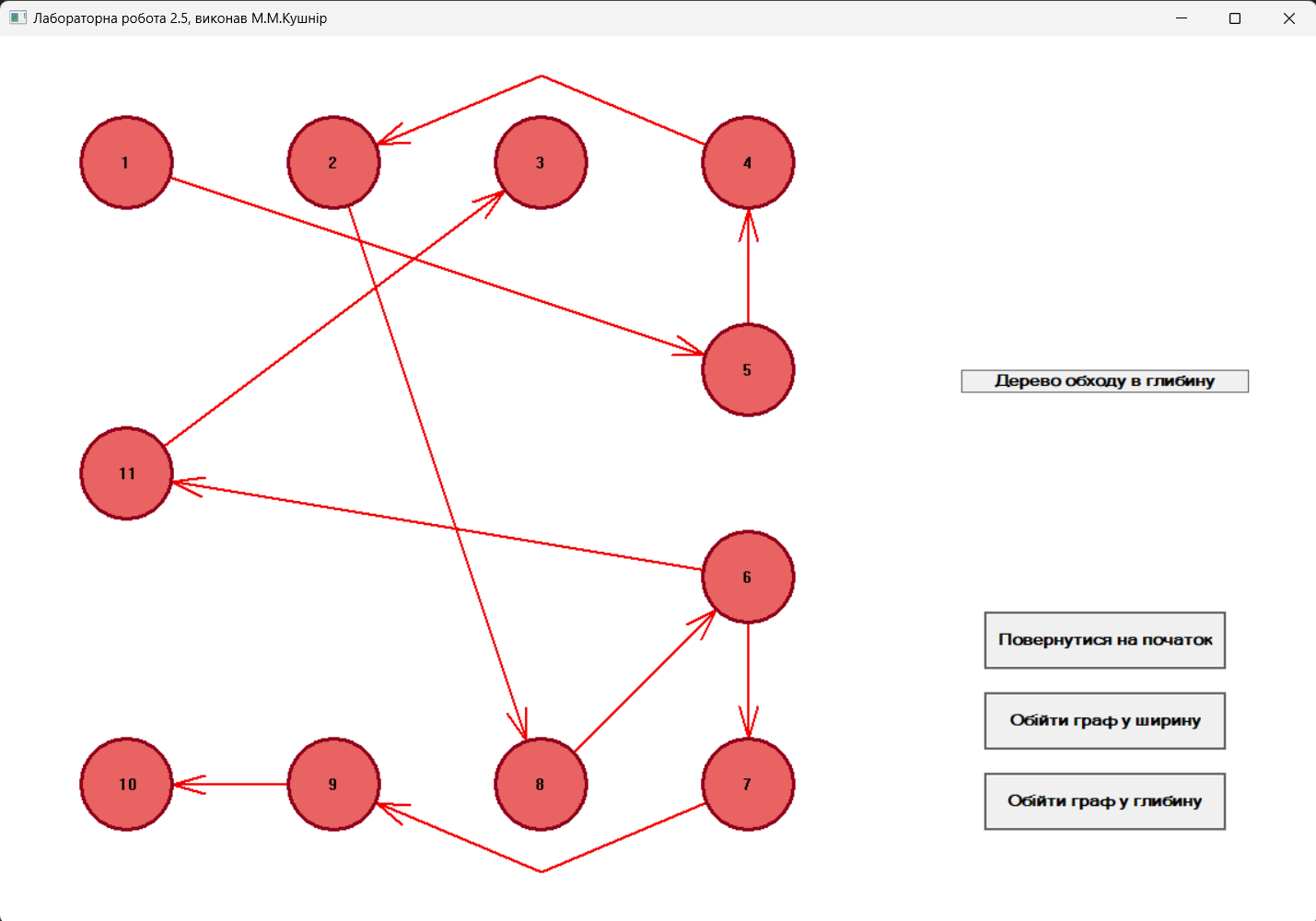
******

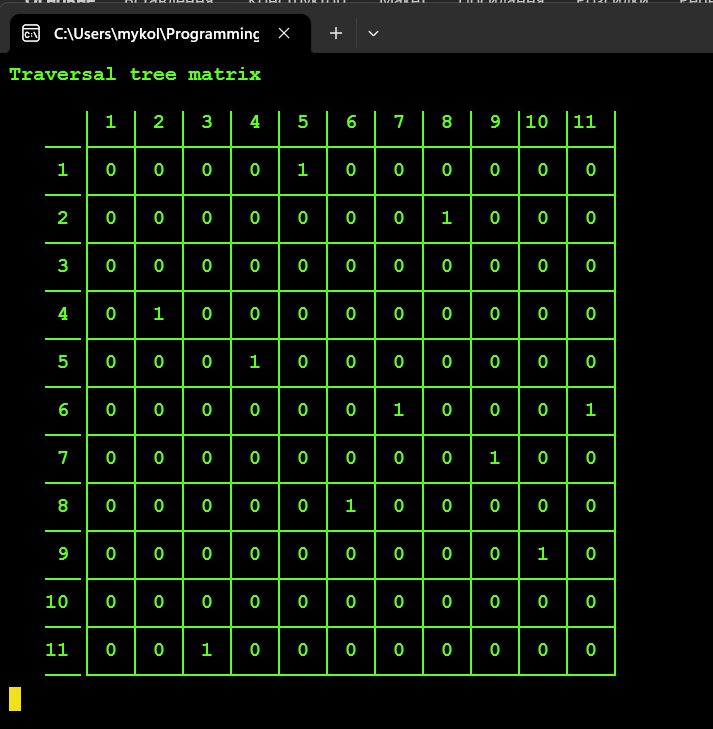
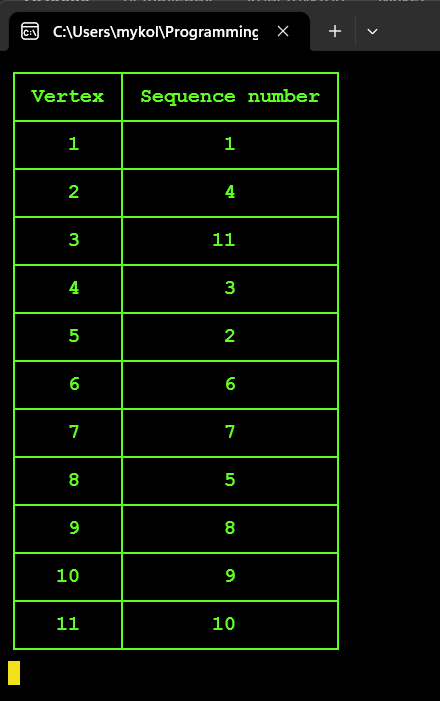
***Дерево обходу в ширину, його матриця суміжності та матриця відповідності вершин і одержаної нумерації***



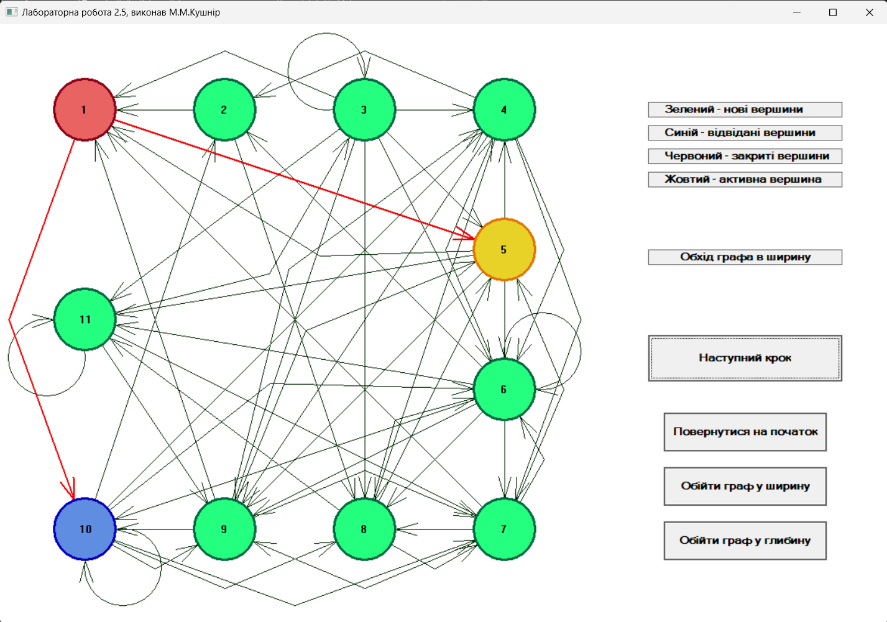
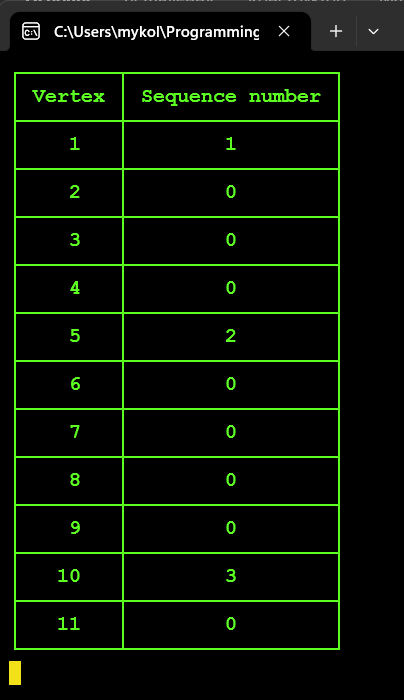
 

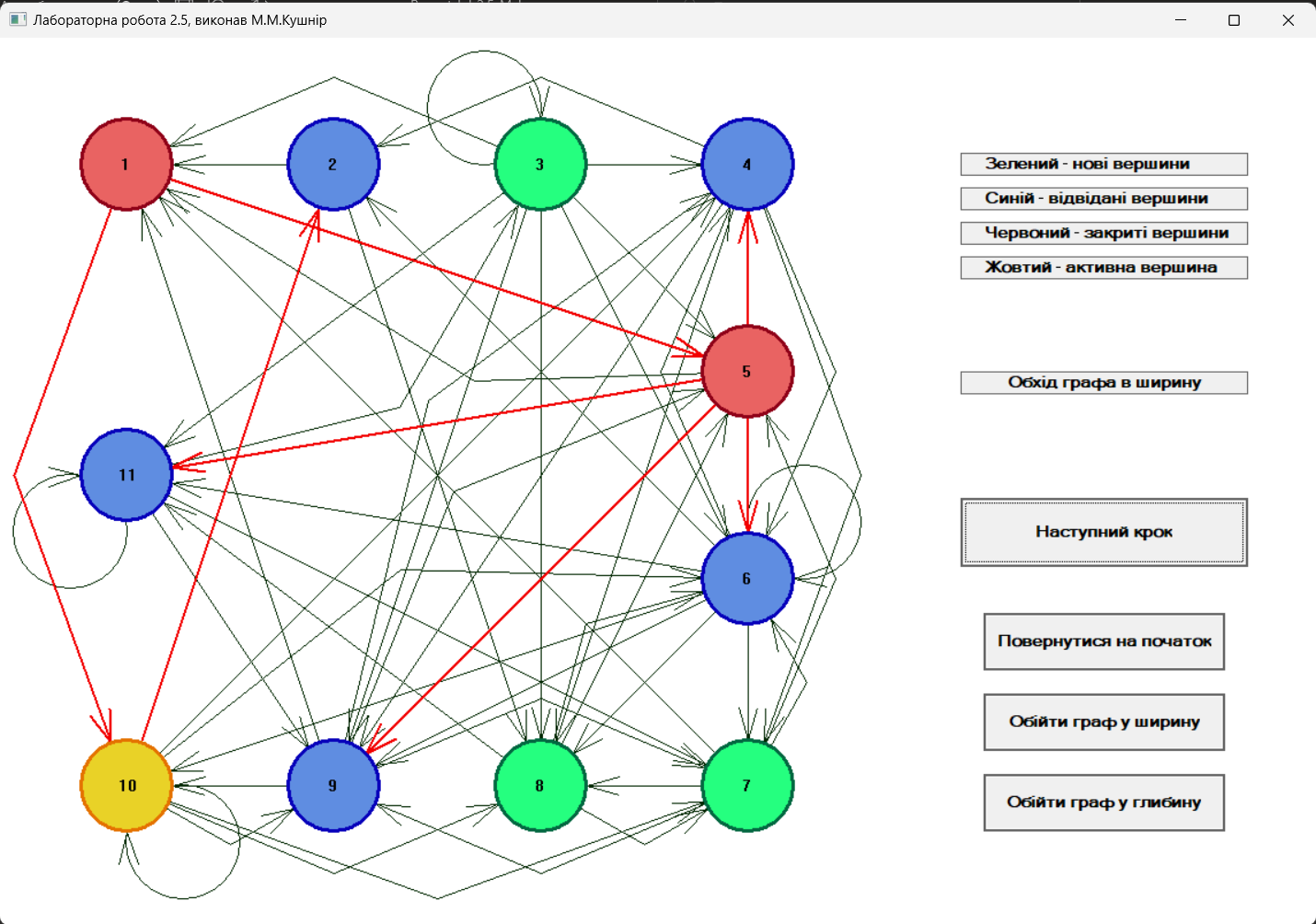
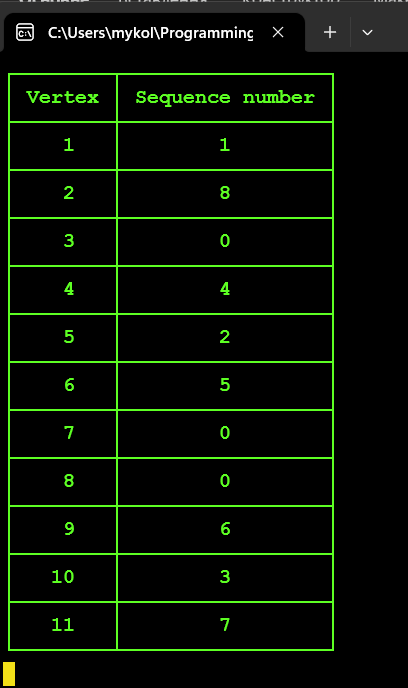
***Дерево обходу в глибину, його матриця суміжності та матриця відповідності вершин і одержаної нумерації***

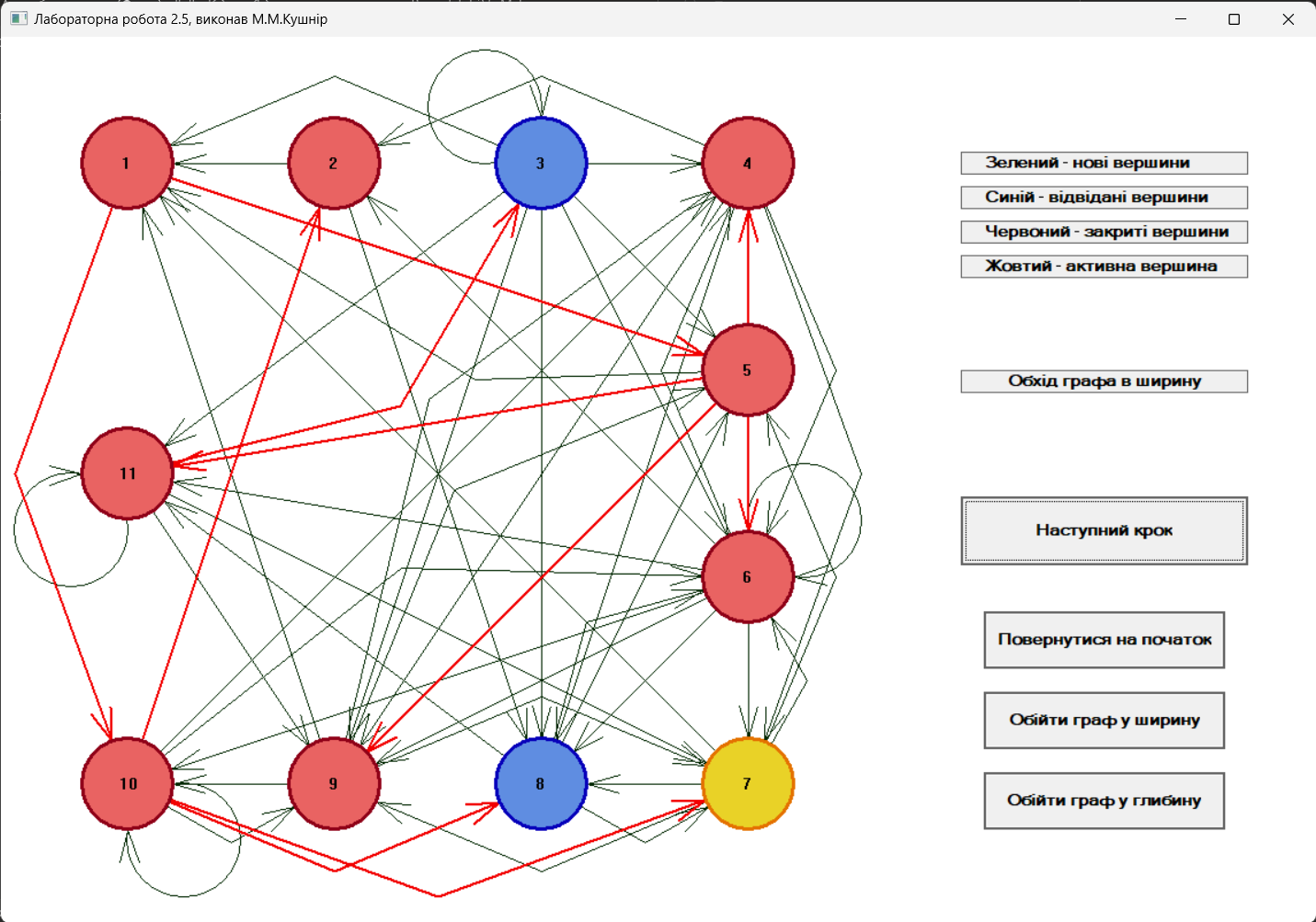
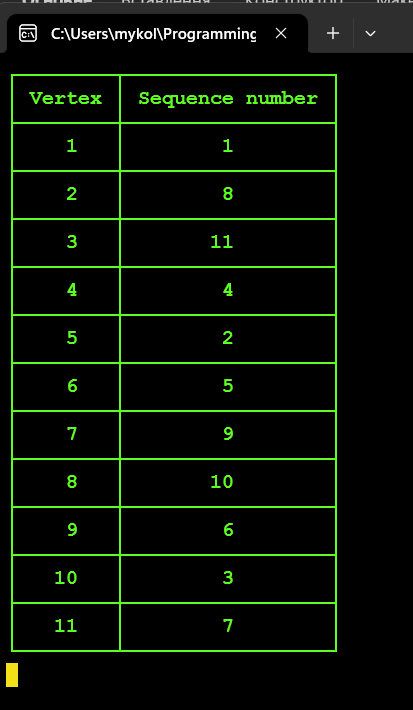
******

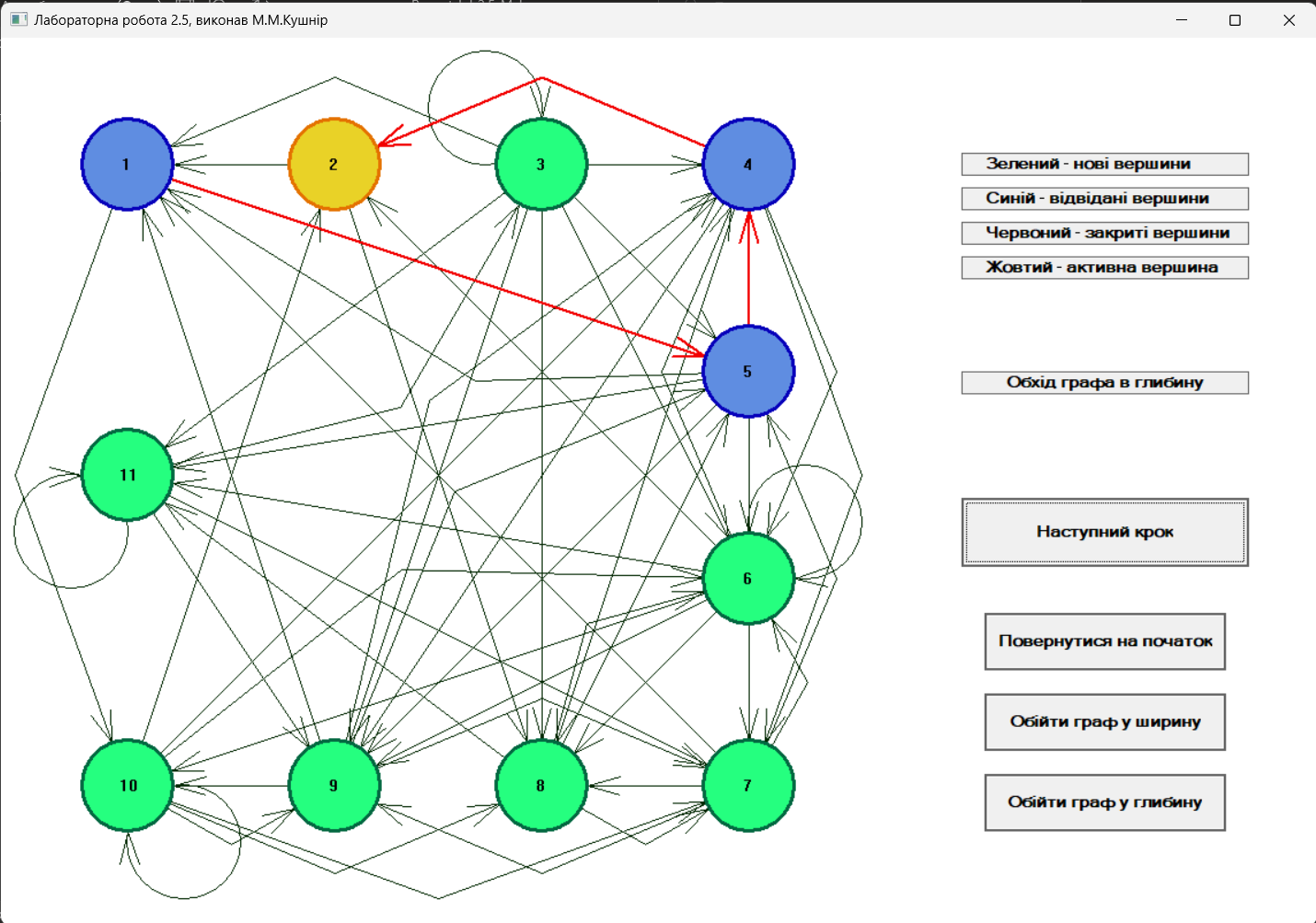
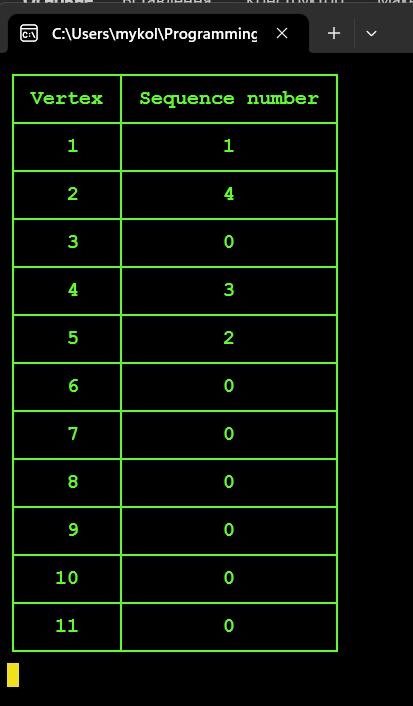
***Процес обходу в ширину***

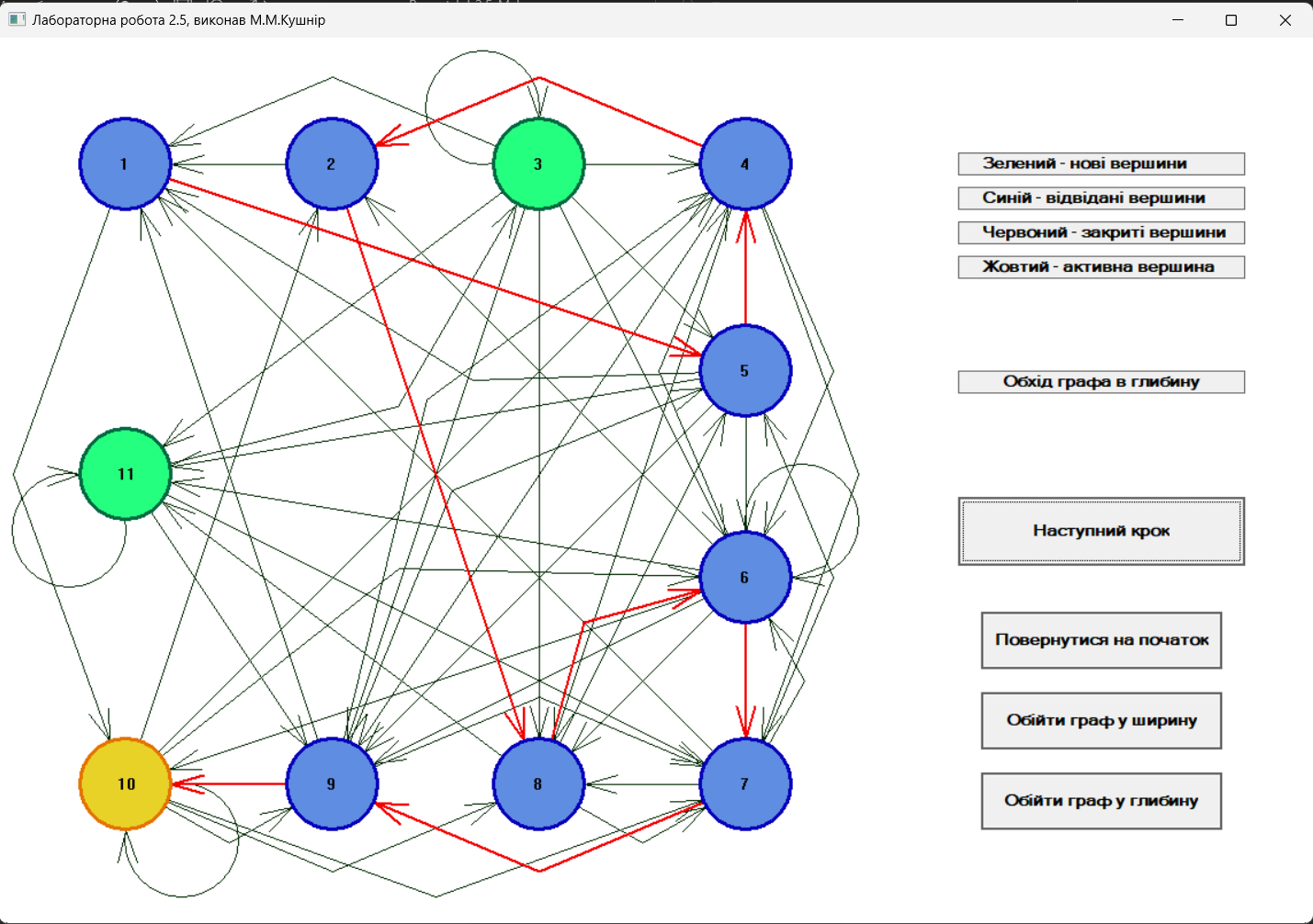
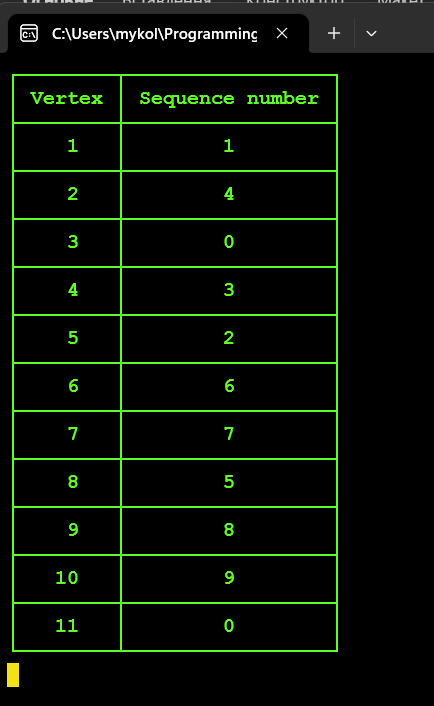
****** ******

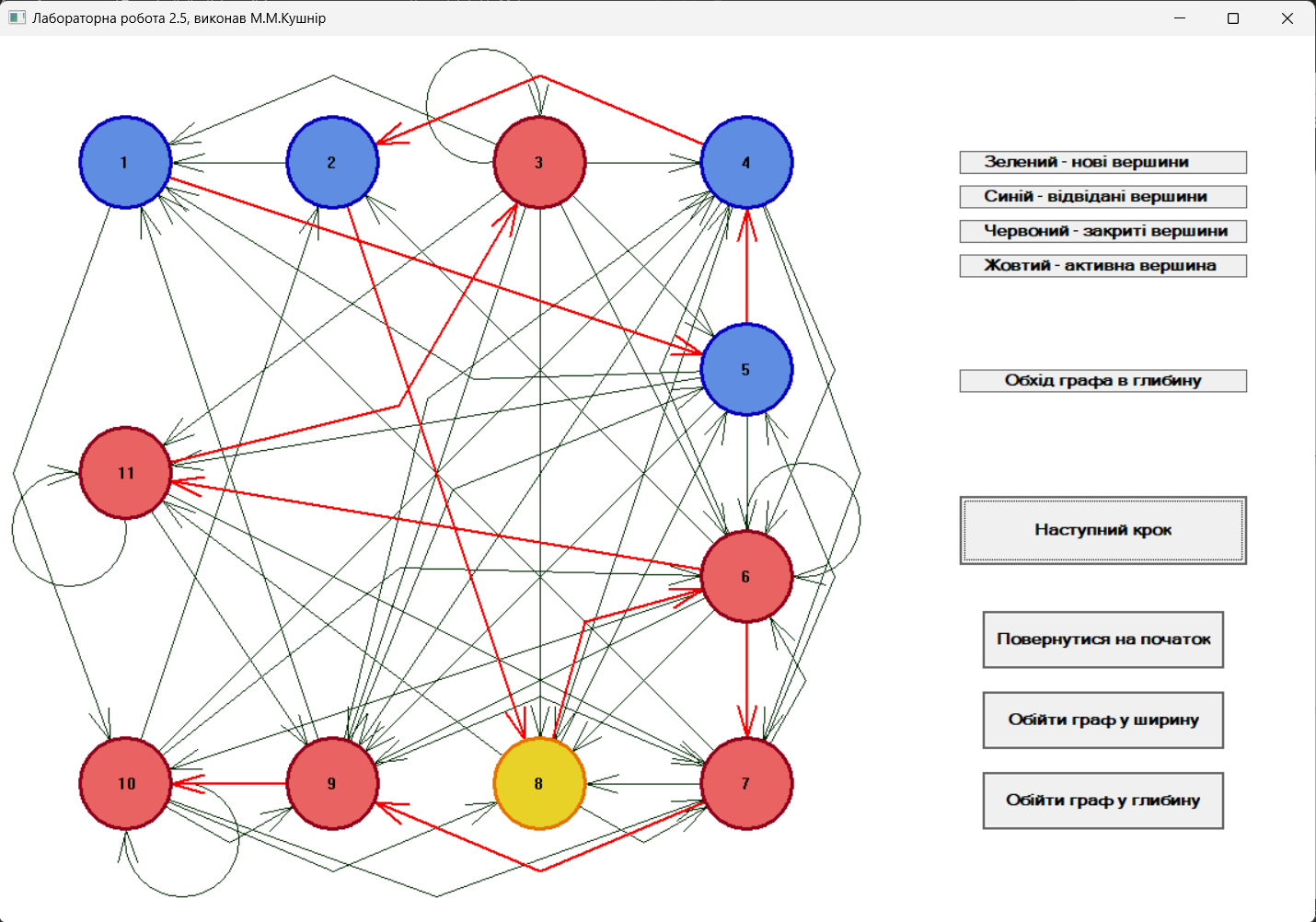
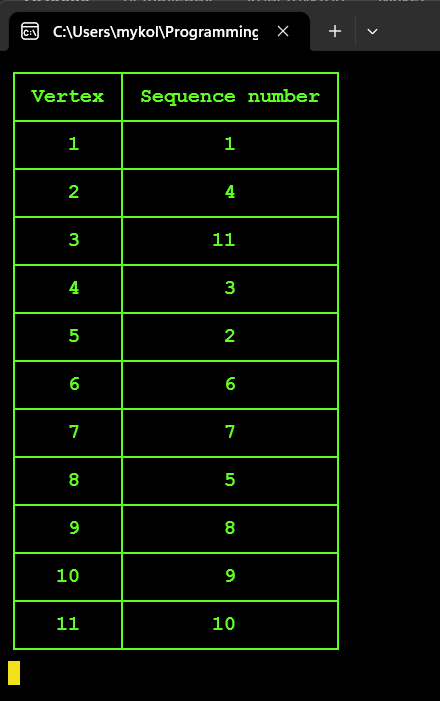
****** ******

****** 

***Процес обходу в глибину***

****** ******

****** ******

****** ******