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:

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

Перевірила:

Молчанова А. А.

Київ 2023

Постановка задачі

1. Представити напрямлений граф з заданими параметрами так само, як у лабораторній роботі №3. Відміна: матриця ***A*** 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)

- 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 – 0.15) * T)
```

Посилання на репозиторій з лабораторною роботою

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,
```

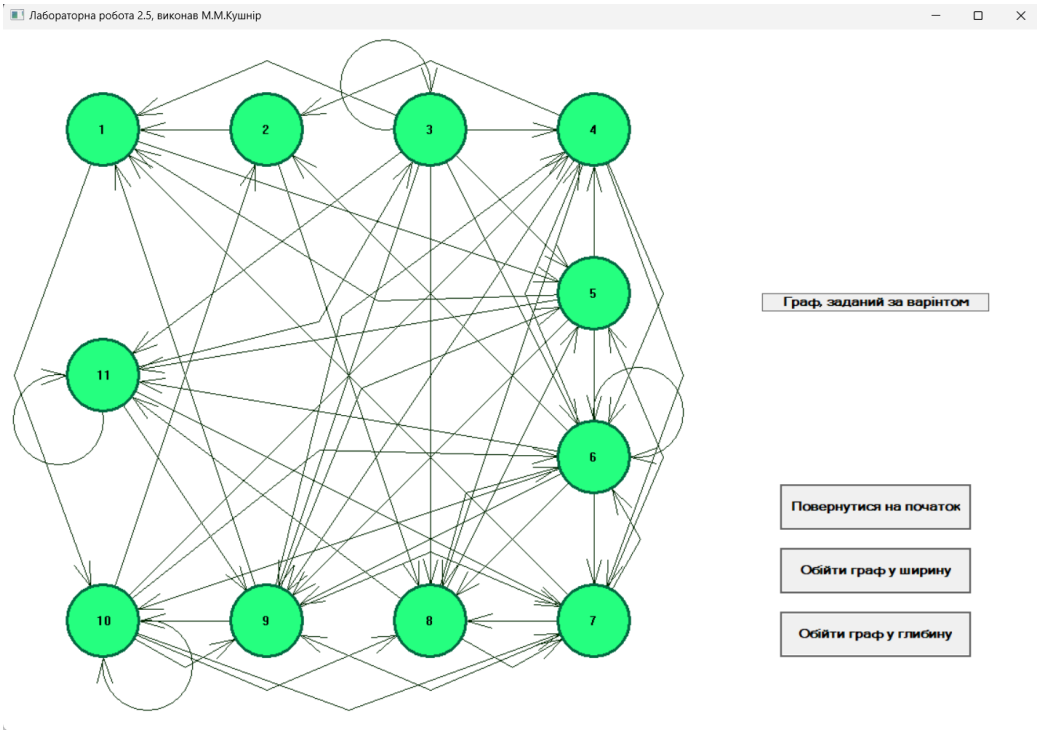
```
                        hdc);
                break;
            case TRAVERSAL_END:
                DrawGraph(N, tree_matrix, vertices_coords,
                          tree_e_pen, tree_v_brush, tree_v_pen, hdc);
                break;
            }
            break;
        }
        case DFS_ALGORITHM:
        {
            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));
```

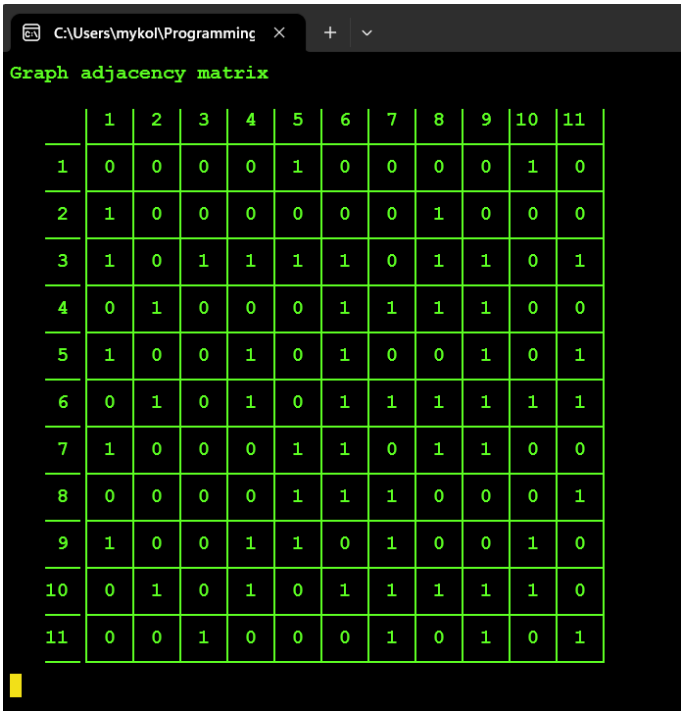
```
            tree_e_pen =        CreatePen(PS_SOLID, 2, RGB(242, 0, 0));
            tree_v_brush =      CreateSolidBrush(RGB(233, 99, 98));
            tree_v_pen =        CreatePen(PS_SOLID, 3, RGB(143, 1, 24));

            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

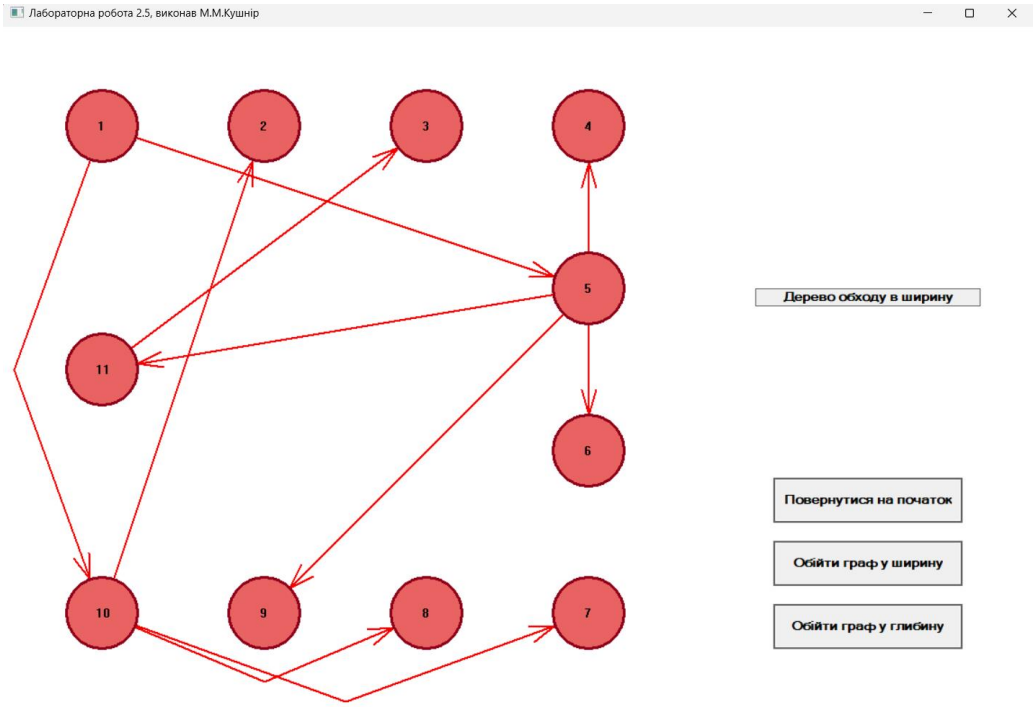
Напрямлений граф та його згенерована матриця суміжності





	1	2	3	4	5	6	7	8	9	10	11
1	0	0	0	0	1	0	0	0	0	1	0
2	1	0	0	0	0	0	0	1	0	0	0
3	1	0	1	1	1	1	0	1	1	0	1
4	0	1	0	0	0	1	1	1	1	0	0
5	1	0	0	1	0	1	0	0	1	0	1
6	0	1	0	1	0	1	1	1	1	1	1
7	1	0	0	0	1	1	0	1	1	0	0
8	0	0	0	0	1	1	1	0	0	0	1
9	1	0	0	1	1	0	1	0	0	1	0
10	0	1	0	1	0	1	1	1	1	1	0
11	0	0	1	0	0	0	1	0	1	0	1

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



C:\Users\mykol\Programming

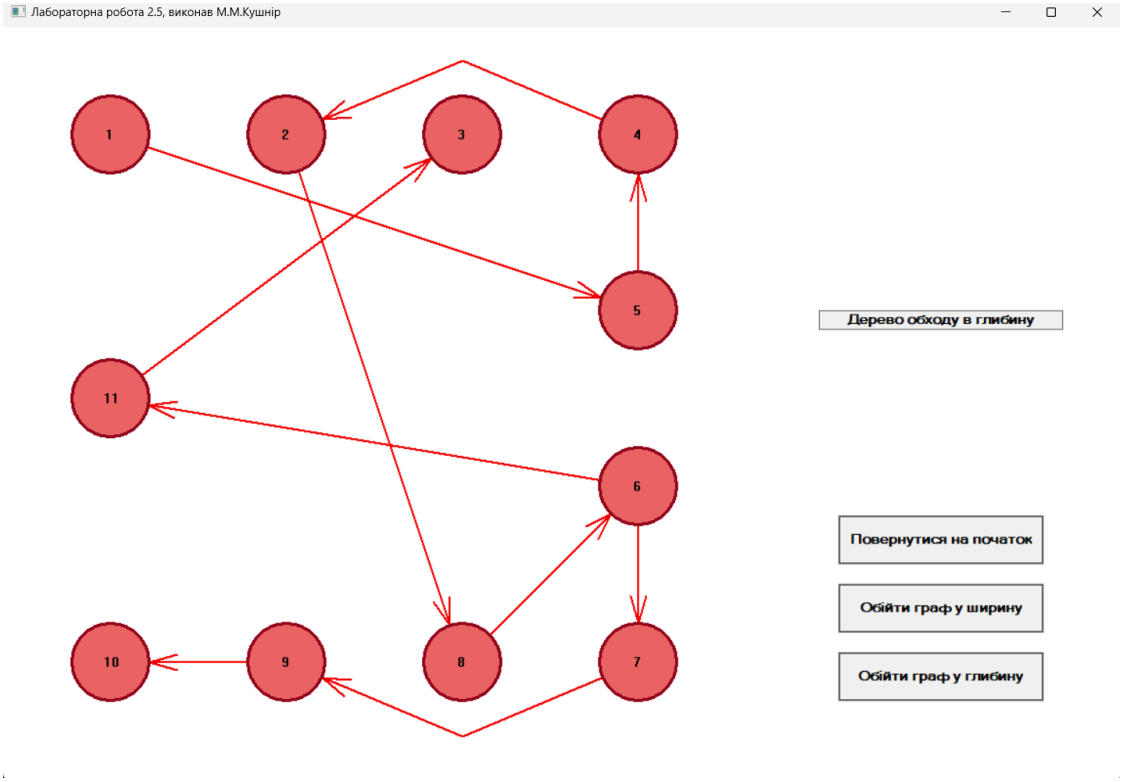
Traversal tree matrix

	1	2	3	4	5	6	7	8	9	10	11
1	0	0	0	0	1	0	0	0	0	1	0
2	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	1	0	1	0	0	1	0	1
6	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0
10	0	1	0	0	0	0	1	1	0	0	0
11	0	0	1	0	0	0	0	0	0	0	0

C:\Users\mykol\Programming

Vertex	Sequence number
1	1
2	8
3	11
4	4
5	2
6	5
7	9
8	10
9	6
10	3
11	7

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



C:\Users\mykol\Programming

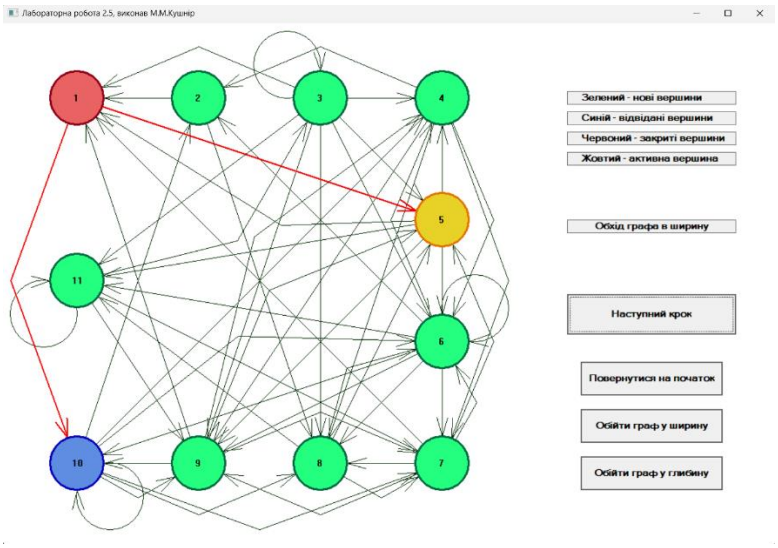
Traversal tree matrix

	1	2	3	4	5	6	7	8	9	10	11
1	0	0	0	0	1	0	0	0	0	0	0
2	0	0	0	0	0	0	0	1	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	1	0	0	0	0	0	0	0	0	0
5	0	0	0	1	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0	1
7	0	0	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	1	0
10	0	0	0	0	0	0	0	0	0	0	0
11	0	0	1	0	0	0	0	0	0	0	0

C:\Users\mykol\Programming

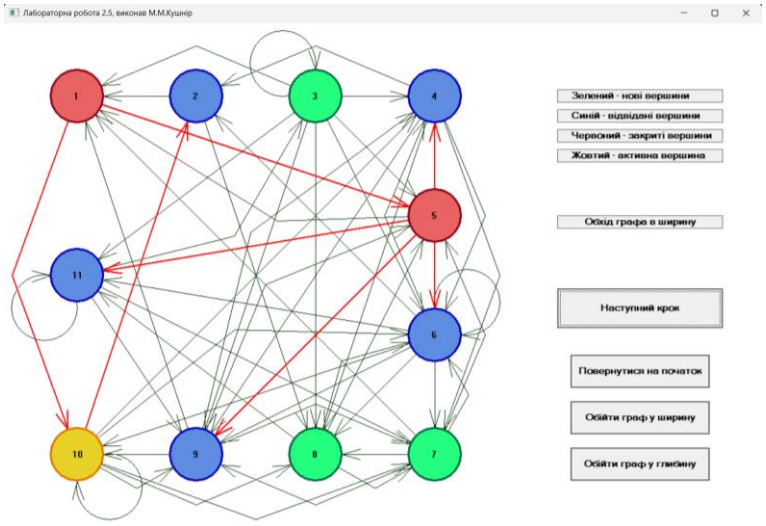
Vertex	Sequence number
1	1
2	4
3	11
4	3
5	2
6	6
7	7
8	5
9	8
10	9
11	10

Процес обходу в ширину



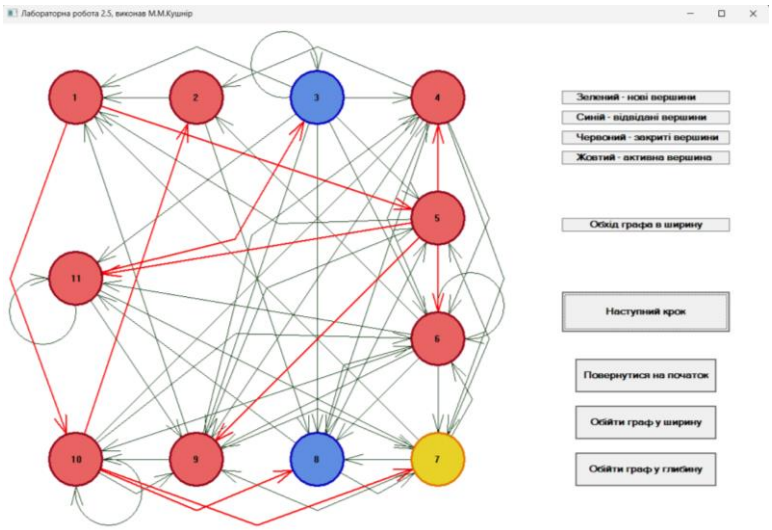
C:\Users\mykol\Programming

Vertex	Sequence number
1	1
2	0
3	0
4	0
5	2
6	0
7	0
8	0
9	0
10	3
11	0



C:\Users\mykol\Programming

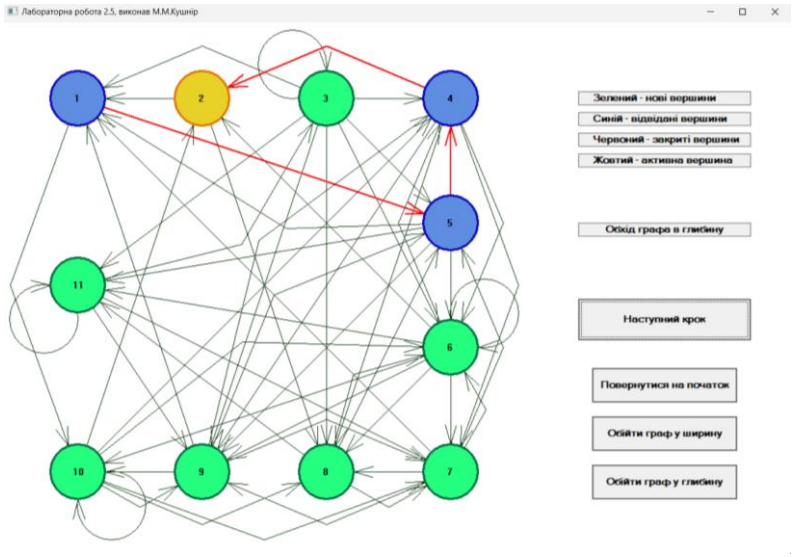
Vertex	Sequence number
1	1
2	8
3	0
4	4
5	2
6	5
7	0
8	0
9	6
10	3
11	7



C:\Users\mykol\Programming

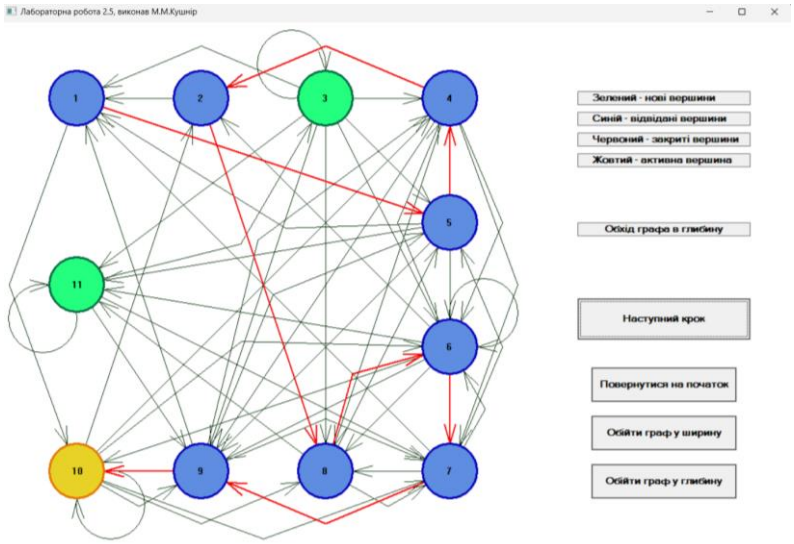
Vertex	Sequence number
1	1
2	8
3	11
4	4
5	2
6	5
7	9
8	10
9	6
10	3
11	7

Процес обходу в глибину



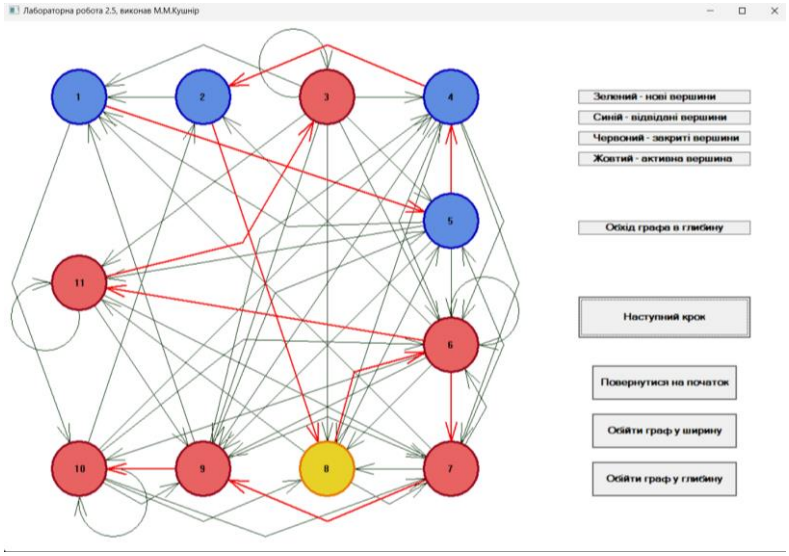
C:\Users\mykol\Programming

Vertex	Sequence number
1	1
2	4
3	0
4	3
5	2
6	0
7	0
8	0
9	0
10	0
11	0



C:\Users\mykol\Programming

Vertex	Sequence number
1	1
2	4
3	0
4	3
5	2
6	6
7	7
8	5
9	8
10	9
11	0



C:\Users\mykol\Programming

Vertex	Sequence number
1	1
2	4
3	11
4	3
5	2
6	6
7	7
8	5
9	8
10	9
11	10