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 № 2.4

з дисципліни
«Алгоритми і структури даних»

Виконав:

студент групи ІМ-22
Коваленко Михайло Володимирович
номер у списку групи: 11

Перевірила:

Молчанова А. А.

Київ 2023

Лабораторна робота 4

Характеристики та зв'язність графа.

Мета лабораторної роботи

Метою лабораторної роботи No4. «Характеристики та зв'язність графа» є дослідити характеристики графів та навчитись визначати їх на конкретних прикладах, вивчення методу транзитивного замикання.

Постановка задачі

1. Представити напрямлений граф з заданими параметрами так само, як у лабораторній роботі No3.

Відміна: матриця A напрямленого графа за варіантом формується за функціями:

srand(п1 п2 п3 п4);

T = randm(n,n);

A = mulmr((1.0 – n3*0.01 – n4*0.01 − 0.3)*T);

Перетворити граф у ненапрямлений.

2. Визначити степені вершин напрямленого і ненапрямленого графів. Програма на екран виводить степені усіх вершин ненапрямленого графу і напівстепені виходу та заходу напрямленого графу. Визначити, чи граф є однорідним та якщо так, то вказати степінь однорідності графу.

3. Визначити всі висячі та ізольовані вершини. Програма на екран виводить перелік усіх висячих та ізольованих вершин графу.

4. Змінити матрицю графу за функцією

A = mulmr((1.0 – n3*0.005 – n4*0.005 − 0.27)*T);

Створити програму для обчислення наступних результатів:

1) матриця суміжності;

2) півстепені вузлів;

3) всі шляхи довжини 2 і 3;

4) матриця досяжності;

5) компоненти сильної зв’язності;

6) матриця зв’язності;

7) граф конденсації.

Шляхи довжиною 2 і 3 слід шукати за матрицями A2 і A3, відповідно. Матриця досяжності та компоненти сильної зв’язності слід шукати за допомогою операції транзитивного замикання.

Варіант: 11

Число вершин: 11

Розміщення колом

srand(2211);

A = mulmr((1.0 – 1*0.01 – 1*0.01 − 0.3)*T);

A = mulmr((1.0 – 1*0.005 – 1*0.005 − 0.27)*T);

Текст програми

Для коректного запуску файлів рекомендую використовувати gcc в консолі і вводити такі команди:

- Для запуску графічного вікна:
  gcc .\functions.c .\window.c -mwindows -o window.exe
  .\window.exe
- Для запуску консольної частини:
  gcc .\functions.c .\console.c -o console.exe
  .\console.exe

window.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <math.h>
#include "props.h"
#define verts_amount 11
#define IDC_BUTTON1 1
#define IDC_BUTTON2 2
#define IDC_BUTTON3 3
#define IDC_BUTTON4 4

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

```
char ProgName[] = "Lab 4 Mykhailo Kovalenko";

struct coordinates {
    double nx[verts_amount];
    double ny[verts_amount];
    double loopX[verts_amount];
    double loopY[verts_amount];
};

void arrow(double fi, double px, double py, HDC hdc) {
    double lx, ly, rx, ry;
    lx = px + 15 * cos(fi + 0.3);
    rx = px + 15 * cos(fi - 0.3);
    ly = py + 15 * sin(fi + 0.3);
    ry = py + 15 * sin(fi - 0.3);
    MoveToEx(hdc, lx, ly, NULL);
    LineTo(hdc, px, py);
    LineTo(hdc, rx, ry);
}

void drawUndirectedGraph(int centerX, int centerY, int radiusOfGraph,
int radiusOfVertex, int radiusOfLoop, double angle,
                         struct coordinates coordinates, double
**matrix,
                         HPEN KPen, HPEN GPen, HDC hdc) {
    for (int i = 0; i < verts_amount; ++i) {

        for (int j = 0; j < verts_amount; ++j) {
            MoveToEx(hdc, coordinates.nx[i], coordinates.ny[i],
NULL);

            if (matrix[i][j] == 1) {
                if (i == j) {
                    SelectObject(hdc, GPen);
                    Ellipse(hdc, coordinates.loopX[i] - radiusOfLoop,
coordinates.loopY[i] - radiusOfLoop,
                            coordinates.loopX[i] + radiusOfLoop,
coordinates.loopY[i] + radiusOfLoop);
                    SelectObject(hdc, KPen);
                    //MoveToEx(hdc, coordinates.loopX[i],
coordinates.loopY[i] + radiusOfLoop, NULL);
                } else {
                    LineTo(hdc, coordinates.nx[j],
coordinates.ny[j]);
                }
            }
        }
    }
```

```
}

void drawArch(int startX, int startY, int finishX, int finishY, int
archInterval, HDC hdc) {
    XFORM transformedMatrix;
    XFORM initialMatrix;
    GetWorldTransform(hdc, &initialMatrix);

    double angle = atan2(finishY - startY, finishX - startX) - M_PI /
2.0;
    transformedMatrix.eM11 = (FLOAT) cos(angle);
    transformedMatrix.eM12 = (FLOAT) sin(angle);
    transformedMatrix.eM21 = (FLOAT) (-sin(angle));
    transformedMatrix.eM22 = (FLOAT) cos(angle);
    transformedMatrix.eDx = (FLOAT) startX;
    transformedMatrix.eDy = (FLOAT) startY;
    SetWorldTransform(hdc, &transformedMatrix);

    const double archWidth = 0.75;
    double length = sqrt((finishX - startX) * (finishX - startX) +
(finishY - startY) * (finishY - startY));
    double radiusOfVertex = 15.0;
    double semiMinorAxis = archWidth * length;
    double semiMajorAxis = length / 2;
    double vertexAreaSquared = semiMajorAxis * semiMajorAxis *
radiusOfVertex * radiusOfVertex;
    double semiAxesSquared = semiMinorAxis * semiMinorAxis *
semiMajorAxis * semiMajorAxis;
    double ellipseStartY = semiMajorAxis;
    double radius = semiMinorAxis * semiMinorAxis * ellipseStartY *
ellipseStartY;
    double distance = semiMinorAxis * semiMinorAxis * radiusOfVertex
* radiusOfVertex;
    double semiMinorAxisPow = pow(semiMinorAxis, 4);
    double crossing = semiMajorAxis * sqrt(vertexAreaSquared -
semiAxesSquared + radius - distance + semiMinorAxisPow);
    double semiMinorAxisSquaredEllipseStartY = semiMinorAxis *
semiMinorAxis * ellipseStartY;
    double denominator = -semiMajorAxis * semiMajorAxis +
semiMinorAxis * semiMinorAxis;
    double contactYRightTop = (semiMinorAxisSquaredEllipseStartY -
crossing) / denominator;
    double contactXRightTop = sqrt(radiusOfVertex * radiusOfVertex -
contactYRightTop * contactYRightTop);
    double contactYBottom = length - contactYRightTop;
    double contactXLeftBottom = -contactXRightTop;

    if (archInterval <= verts_amount / 2) {
        Arc(hdc, -archWidth * length, length, archWidth * length, 0,
```

```
0, 0, 0, length);
        double angleOfArrow = -atan2(length - contactYBottom,
contactXLeftBottom) + 0.3 / 3;
        arrow(angleOfArrow, contactXLeftBottom, contactYBottom, hdc);
    } else {
        Arc(hdc, -archWidth * length, length, archWidth * length, 0,
0, length, 0, 0);
        double angleOfArrow = -atan2(length - contactYBottom, -
contactXLeftBottom) - 0.3 / 3;
        arrow(angleOfArrow, -contactXLeftBottom, contactYBottom,
hdc);
    }
    SetWorldTransform(hdc, &initialMatrix);
}

void drawDirectedGraph(int n, int centerX, int centerY, int
radiusOfGraph, int radiusOfVertex, int radiusOfLoop, double angle,
                       struct coordinates coordinates, double
**matrix,
                       HPEN KPen, HPEN GPen, HDC hdc) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            MoveToEx(hdc, coordinates.nx[i], coordinates.ny[i],
NULL);
            if ((j >= i && matrix[i][j] == 1) || (j <= i &&
matrix[i][j] == 1 && matrix[j][i] == 0)) {
                if (i == j) {
                    SelectObject(hdc, GPen);
                    Ellipse(hdc, coordinates.loopX[i] - radiusOfLoop,
coordinates.loopY[i] - radiusOfLoop,
                            coordinates.loopX[i] + radiusOfLoop,
coordinates.loopY[i] + radiusOfLoop);
                    double triangleHeight = sqrt(3) * radiusOfVertex
/ 2.;
                    double radiusOfContact = radiusOfGraph +
radiusOfLoop / 2.;
                    double distance = sqrt(radiusOfContact *
radiusOfContact + triangleHeight * triangleHeight);
                    double angleToContactVertex =
atan2(coordinates.ny[i] - centerY, coordinates.nx[i] - centerX);
                    double loopAngle = atan2(triangleHeight,
radiusOfContact);
                    double contactCoordX = centerX + distance *
cos(angleToContactVertex + loopAngle);
                    double contactCoordY = centerY + distance *
sin(angleToContactVertex + loopAngle);
                    double curveAngle = angleToContactVertex + 0.3 /
2.;
                    arrow(curveAngle, contactCoordX, contactCoordY,
```

```
hdc);
                    SelectObject(hdc, KPen);
                } else {
                    LineTo(hdc, coordinates.nx[j],
coordinates.ny[j]);
                    double line_angle = atan2(coordinates.ny[i] -
coordinates.ny[j], coordinates.nx[i] - coordinates.nx[j]);
                    arrow(line_angle, coordinates.nx[j] +
radiusOfVertex * cos(line_angle),
                          coordinates.ny[j] + radiusOfVertex *
sin(line_angle), hdc);
                }
            } else if (j < i && matrix[i][j] == 1 && matrix[j][i] ==
1) {
                drawArch(coordinates.nx[i], coordinates.ny[i],
coordinates.nx[j], coordinates.ny[j], fabs(i - j), hdc);
            }

        }
    }
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpszCmdLine, int nCmdShow) {
    WNDCLASS w;
    w.lpszClassName = ProgName;
    w.hInstance = hInstance;
    w.lpfnWndProc = WndProc;
    w.hCursor = LoadCursor(NULL, IDC_ARROW);
    w.hIcon = 0;
    w.lpszMenuName = 0;
    w.hbrBackground = WHITE_BRUSH;
    w.style = CS_HREDRAW | CS_VREDRAW;
    w.cbClsExtra = 0;
    w.cbWndExtra = 0;
    if (!RegisterClass(&w)) {
        return 0;
    }
    HWND hWnd;
    MSG lpMsg;
    hWnd = CreateWindow(ProgName,
                        (LPCSTR) "Lab3 (Mykhailo Kovalenko IM-22)",
                        WS_OVERLAPPEDWINDOW,
                        100,
                        100,
                        950,
                        800,
                        (HWND) NULL,
                        (HMENU) NULL,
```

```
                        (HINSTANCE) hInstance,
                        (HINSTANCE) NULL);
    ShowWindow(hWnd, nCmdShow);
    while (GetMessage(&lpMsg, hWnd, 0, 0)) {
        TranslateMessage(&lpMsg);
        DispatchMessage(&lpMsg);
    }
    return (lpMsg.wParam);
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT messg, WPARAM wParam, LPARAM
lParam) {
    HDC hdc;
    PAINTSTRUCT ps;
    HWND Button_directed;
    HWND Button_undirected;
    HWND Button_modified;
    int state = 0;
    switch (messg) {
        case WM_CREATE: {
            Button_directed = CreateWindow(
                    (LPCSTR) "BUTTON",
                    (LPCSTR) "Directed",
                    WS_TABSTOP | WS_VISIBLE | WS_CHILD |
BS_DEFPUSHBUTTON,
                    700,
                    20,
                    160,
                    50,
                    hWnd,
                    (HMENU) IDC_BUTTON1,
                    (HINSTANCE) GetWindowLongPtr(hWnd,
GWLP_HINSTANCE),
                    NULL);
            Button_undirected = CreateWindow(
                    (LPCSTR) "BUTTON",
                    (LPCSTR) "Undirected",
                    WS_TABSTOP | WS_VISIBLE | WS_CHILD |
BS_DEFPUSHBUTTON,
                    700,
                    100,
                    160,
                    50,
                    hWnd,
                    (HMENU) IDC_BUTTON2,
                    (HINSTANCE) GetWindowLongPtr(hWnd,
GWLP_HINSTANCE),
                    NULL);
            Button_modified = CreateWindow(
```

```
                    (LPCSTR) "BUTTON",
                    (LPCSTR) "Modified",
                    WS_TABSTOP | WS_VISIBLE | WS_CHILD |
BS_DEFPUSHBUTTON,
                    700,
                    180,
                    160,
                    50,
                    hWnd,
                    (HMENU) IDC_BUTTON3,
                    (HINSTANCE) GetWindowLongPtr(hWnd,
GWLP_HINSTANCE),
                    NULL);
            Button_modified = CreateWindow(
                    (LPCSTR) "BUTTON",
                    (LPCSTR) "Condensation",
                    WS_TABSTOP | WS_VISIBLE | WS_CHILD |
BS_DEFPUSHBUTTON,
                    700,
                    260,
                    160,
                    50,
                    hWnd,
                    (HMENU) IDC_BUTTON4,
                    (HINSTANCE) GetWindowLongPtr(hWnd,
GWLP_HINSTANCE),
                    NULL);
            return 0;
        }
        case WM_COMMAND: {
            switch (LOWORD(wParam)) {

                case IDC_BUTTON1:
                    state = 0;
                    InvalidateRect(hWnd, NULL, FALSE);
                    break;

                case IDC_BUTTON2:
                    state = 1;
                    InvalidateRect(hWnd, NULL, FALSE);
                    break;
                case IDC_BUTTON3:
                    state = 2;
                    InvalidateRect(hWnd, NULL, FALSE);
                    break;
                case IDC_BUTTON4:
                    state = 3;
                    InvalidateRect(hWnd, NULL, FALSE);
                    break;
```

```
            }
        }
        case WM_PAINT :
            hdc = BeginPaint(hWnd, &ps);
            SetGraphicsMode(hdc, GM_ADVANCED);
            HPEN BPen = CreatePen(PS_SOLID, 2, RGB(50, 0, 255));
            HPEN KPen = CreatePen(PS_SOLID, 1, RGB(20, 20, 5));
            HPEN GPen = CreatePen(PS_SOLID, 2, RGB(0, 255, 0));
            HPEN NoPen = CreatePen(PS_NULL, 0, RGB(0, 0, 0));
            SelectObject(hdc, NoPen);
            Rectangle(hdc, 0, 0, 670, 700);
            char *nn[verts_amount] = {"1", "2", "3", "4", "5", "6",
"7", "8", "9", "10\0", "11\0"};
            struct coordinates coordinates;
            double circleRadius = 200;
            double vertexRadius = circleRadius / 11;
            double loopRadius = vertexRadius;
            double dtx = vertexRadius / 2.5;
            double circleCenterX = 370;
            double circleCenterY = 360;
            double angleAlpha = 2.0 * M_PI / (double) verts_amount;
            for (int i = 0; i < verts_amount; i++) {

                double sinAlpha = sin(angleAlpha * (double) i);
                double cosAlpha = cos(angleAlpha * (double) i);
                coordinates.nx[i] = circleCenterX + circleRadius *
sinAlpha;
                coordinates.ny[i] = circleCenterY - circleRadius *
cosAlpha;
                coordinates.loopX[i] = circleCenterX + (circleRadius
+ loopRadius) * sinAlpha;
                coordinates.loopY[i] = circleCenterY - (circleRadius
+ loopRadius) * cosAlpha;

            }
            double **T = randm(verts_amount);
            double coefficient = 1.0 - 0.01 - 0.01 - 0.3;
            double **A = mulmr(coefficient, T, verts_amount);
            double **R = randm(verts_amount);
            double **C = symmetricalMatrix(mulmr(coefficient, R,
verts_amount), verts_amount);
            double** K = randm(verts_amount);
            double modifiedCoefficient = 1.0 - 0.005 - 0.005 - 0.27;
            double** D = mulmr(modifiedCoefficient, K, verts_amount);
            double **condensationMatrix =
condensationMatrixWithCoefficient(modifiedCoefficient);
            double **matrix =
generateAdjacencyMatrixFromStrongComponents(condensationMatrix);
            double **reachabilityMatrix = getReachabilityMatrix(D);
```

```
            double **connectivityMatrix =
getStrongConnectivityMatrix(reachabilityMatrix);
            int amount = getStrongComponents(connectivityMatrix);
            printf("%d", amount);
            SelectObject(hdc, GetStockObject(HOLLOW_BRUSH));
            SelectObject(hdc, KPen);
            if (state == 0) {
                drawDirectedGraph(verts_amount,circleCenterX,
circleCenterY, circleRadius, vertexRadius, loopRadius, angleAlpha,
                                  coordinates, A, KPen, GPen, hdc);
            }
            if (state == 1){
                drawUndirectedGraph(circleCenterX, circleCenterY,
circleRadius, vertexRadius, loopRadius, angleAlpha,
                                    coordinates, C, KPen, GPen, hdc);
            }
            if (state == 2) {
                drawDirectedGraph(verts_amount,circleCenterX,
circleCenterY, circleRadius, vertexRadius, loopRadius, angleAlpha,
                                    coordinates, D, KPen, GPen, hdc);
            }
            if (state == 3) {
                drawDirectedGraph(amount,circleCenterX,
circleCenterY, circleRadius, vertexRadius, loopRadius, angleAlpha,
                                    coordinates, matrix, KPen, GPen,
hdc);
            }

            SelectObject(hdc, BPen);
            SelectObject(hdc, GetStockObject(DC_BRUSH));
            SetDCBrushColor(hdc, RGB(204, 204, 255));
            SetBkMode(hdc, TRANSPARENT);
            int length = state == 3 ? amount : verts_amount;

            for (int i = 0; i < length; ++i) {
                Ellipse(hdc, coordinates.nx[i] - vertexRadius,
coordinates.ny[i] - vertexRadius,
                        coordinates.nx[i] + vertexRadius,
coordinates.ny[i] + vertexRadius);
                TextOut(hdc, coordinates.nx[i] - dtx,
coordinates.ny[i] - vertexRadius / 2, nn[i], 2);
            }
            EndPaint(hWnd, &ps);
            freeMatrix(A, verts_amount);
            freeMatrix(C, verts_amount);
            freeMatrix(D, verts_amount);
            freeMatrix(matrix, verts_amount);
            freeMatrix(condensationMatrix, verts_amount);
            freeMatrix(connectivityMatrix, verts_amount);
```

```
            freeMatrix(reachabilityMatrix, verts_amount);
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return (DefWindowProc(hWnd, messg, wParam, lParam));
    }
    return 0;
}
```

console.c

```
#include <stdio.h>
#include <stdlib.h>
#define verts_amount 11
#include "props.h"

void typeMatrix(double **matrix) {
    const int number = verts_amount;
    for (int i = 0; i < number; i++) {
        for (int j = 0; j < number; j++) {
            printf("%.0lf  ", matrix[i][j]);
        }
        printf("\n");
    }
}

void printComponents(double **matrix, int number) {
    int n = verts_amount;
    int componentCount = 1;
    for (int i = 0; i < n; i++) {
        int isNewComponent = 1;
        for (int j = 0; j < n; j++) {
            if(matrix[i][j]) {
                if (isNewComponent) printf("Component %d: [  ",
componentCount);
                printf("%d  ", j + 1);
                isNewComponent = 0;
            }
        }
        if (!isNewComponent) {
            componentCount++;
            printf("]\n");
        }
    }
}

void printDegrees(int *degrees) {
    const int number = verts_amount;
```

```
    printf("{  ");
    for (int i = 0; i < number; i++) {
        printf("%d  ", degrees[i]);
    }
    printf("}\n");
}

int getInterimVertsLength2(double** modifiedMatrix, int startPoint,
int endPoint){
    int interimVert;
    for (int i = 0; i < verts_amount; ++i) {
        for (int j = 0; j < verts_amount; ++j) {
            if (modifiedMatrix[startPoint][j] == 1 &&
modifiedMatrix[j][endPoint] == 1)
                interimVert = j;
        }
    }
    return ++interimVert;
}

int* getInterimVertsLength3(double** modifiedMatrix, double**
squaredMatrix, int startPoint, int endPoint){
    int* interimVerts = calloc(2, sizeof(int));
    for (int i = 0; i < verts_amount; ++i) {
        for (int j = 0; j < verts_amount; ++j) {
            if (modifiedMatrix[startPoint][j] == 1 &&
squaredMatrix[j][endPoint] >= 1){
                interimVerts[0] = j;
                for (int k = 0; k < verts_amount; ++k) {
                    if (modifiedMatrix[interimVerts[0]][k] == 1 &&
modifiedMatrix[k][endPoint]){
                        interimVerts[1] = k;
                    }
                }
            }
        }
    }
    interimVerts[0] += 1;
    interimVerts[1] += 1;
    return interimVerts;
}

void printPathwaysLength2(double** pathMatrix, double**
modifiedMatrix) {
    const int numbers = verts_amount;
    for (int i = 0; i < numbers; i++) {
        for (int j = 0; j < numbers; j++) {
            if (*(*(pathMatrix + i) + j) != 0) {
                int interimVert =
```

```
getInterimVertsLength2(modifiedMatrix, i, j);
                printf("%d -> %d -> %d; ", i + 1, interimVert, j +
1);
            }
        }
        printf("\n");
    }
}

void printPathwaysLength3(double** pathMatrix, double**
modifiedMatrix, double** squaredMatrix) {
    const int numbers = verts_amount;
    for (int i = 0; i < numbers; i++) {
        for (int j = 0; j < numbers; j++) {
            if (*(*(pathMatrix + i) + j) != 0) {
                int* interimVerts =
getInterimVertsLength3(modifiedMatrix, squaredMatrix, i, j);
                printf("%d -> %d -> %d -> %d; ", i + 1,
interimVerts[0], interimVerts[1], j + 1);
                free(interimVerts);
            }
        }
        printf("\n");
    }
}

void printVertices(int* verticesNumber) {
    printf("{ ");
    for (int i = 0; verticesNumber[i] != 0; ++i) {
        printf("%d ", verticesNumber[i]);
    }
    printf("}\n");
}

void directedGraphInfo() {
    double **T = randm(verts_amount);
    double coefficient = 1.0 - 0.01 - 0.01 - 0.3;
    double **A = mulmr(coefficient, T, verts_amount);
    int* entry = halfDegreeEntry(A);
    int* exit = halfDegreeExit(A);
    int* summedDegrees = summarizeHalfDegrees(exit, entry);

    printf("\n\nDirected Graph \n");
    printf("\n\tInitial matrix\n");
    typeMatrix(A);

    printf("Exit degree : ");
    printDegrees(exit);
```

```
    printf("Entry degree : ");
    printDegrees(entry);

    if(isUniform(summedDegrees)) {
        printf("%d\n", summedDegrees[0]);
    } else {
        printf("\tThe graph is not uniform ");
    }

    printf("\nIsolated vertices : ");

    for (int i = 0; i < verts_amount; ++i) {
        if (summedDegrees[i] == 0) printf("# %d ", i+1);
    }

    printf("\nTerminal vertices : ");
    for (int i = 0; i < verts_amount; ++i) {
        if (summedDegrees[i] == 1) printf("# %d ", i+1);
    }
    freeMatrix(A, verts_amount);

    free(entry);
    free(exit);

    free(summedDegrees);
}

void undirectedGraphInfo() {
    double coefficient = 1.0 - 0.01 - 0.01 - 0.3;
    double **R = randm(verts_amount);
    double **C = symmetricalMatrix(mulmr(coefficient, R,
verts_amount), verts_amount);
    int* degree = graphDegrees(C);

    printf("\n\nUndirected Graph \n");
    printf("\n\tMatrix for Undirected Graph\n");
    typeMatrix(C);

    printf("Undirected graph degrees : ");
    printDegrees(degree);

    if(isUniform(degree)) {
        printf("%d\n", degree[0]);
    } else {
        printf("\tThe graph is not uniform ");
    }

    printf("\nIsolated vertices : ");
    for (int i = 0; i < verts_amount; ++i) {
```

```
        if (degree[i] == 0) printf("# %d ", i+1);
    }

    printf("\nTerminal vertices : ");
    for (int i = 0; i < verts_amount; ++i) {
        if (degree[i] == 1) printf("# %d ", i+1);
    }

    freeMatrix(C, verts_amount);
    free(degree);
}

void modifiedGraphInfo() {
    double** K = randm(verts_amount);
    double modifiedCoefficient = 1.0 - 0.005 - 0.005 - 0.27;
    double** D = mulmr(modifiedCoefficient, K, verts_amount);
    int* entry = halfDegreeEntry(D);
    int* exit = halfDegreeExit(D);
    int* summedDegrees = summarizeHalfDegrees(exit, entry);
    double** squaredMatrix = multiplyMatrices(D,D);
    double** cubedMatrix = multiplyMatrices(squaredMatrix, D);
    double **reachabilityMatrix = getReachabilityMatrix(D);
    double **connectivityMatrix =
getStrongConnectivityMatrix(reachabilityMatrix);

    double** strongComponents =
findStrongComponents(connectivityMatrix);
    printf("\n\nModified Graph \n");
    printf("\n\tMatrix for Modified Graph\n");
    typeMatrix(D);

    printf("Exit degree : ");
    printDegrees(exit);

    printf("Entry degree : ");
    printDegrees(entry);

    if(isUniform(summedDegrees)) {
        printf("%d\n", summedDegrees[0]);
    } else {
        printf("\tThe graph is not uniform ");
    }

    printf("\nIsolated vertices : ");
    for (int i = 0; i < verts_amount; ++i) {
        if (summedDegrees[i] == 0) printf("# %d ", i+1);
    }

    printf("\nTerminal vertices : ");
```

```
    for (int i = 0; i < verts_amount; ++i) {
        if (summedDegrees[i] == 1) printf("# %d ", i+1);
    }

    printf("\n\nMatrix squared : 2\n");
    typeMatrix(squaredMatrix);

    printf("\nPathways with length : 2\n");
    printPathwaysLength2(squaredMatrix, D);

    printf("\nMatrix cubed : 3\n");
    typeMatrix(cubedMatrix);

    printf("\nPathways with length : 3\n");
    printPathwaysLength3(cubedMatrix, D, squaredMatrix);

    printf("\nReachability Matrix of Mod graph\n");
    typeMatrix(reachabilityMatrix);

    printf("\nConnected Matrix of Mod graph\n");
    typeMatrix(connectivityMatrix);

    printf("\nStrongly Connected Components of Mod Graph\n");
    printComponents(strongComponents, verts_amount);

    printf("\nMatrix of Condensation Graph\n");
    condensationMatrix(strongComponents);

    freeMatrix(D, verts_amount);
    freeMatrix(squaredMatrix, verts_amount);
    freeMatrix(cubedMatrix, verts_amount);
    freeMatrix(reachabilityMatrix, verts_amount);
    freeMatrix(connectivityMatrix, verts_amount);
    freeMatrix(strongComponents, verts_amount);
    free(summedDegrees);
    free(entry);
    free(exit);
}

int main() {
    directedGraphInfo();
    undirectedGraphInfo();
    modifiedGraphInfo();
}
```

functions.c:

```
#define verts_amount 11
#include <stdlib.h>
```

```c
#include <stdio.h>
#include "props.h"

double **randm(int n) {
    srand(2211);
    double **matrix = (double **) malloc(sizeof(double *) * n);
    for (int i = 0; i < n; i++) {
        matrix[i] = (double *) malloc(sizeof(double) * n);
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            matrix[i][j] = (double) (rand() * 2.0) / (double)
RAND_MAX;
        }
    }
    return matrix;
}

double **mulmr(double coef, double **matrix, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            matrix[i][j] *= coef;
            matrix[i][j] = matrix[i][j] < 1 ? 0 : 1;
        }
    }
    return matrix;
}

double **symmetricalMatrix(double **matrix, int n) {
    double **symmetrical = (double **) malloc(n * sizeof(double *));
    for (int i = 0; i < n; ++i) {
        symmetrical[i] = (double *) malloc(n * sizeof(double));
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            symmetrical[i][j] = matrix[i][j];
        }
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (symmetrical[i][j] != symmetrical[j][i]) {
                symmetrical[i][j] = 1;
                symmetrical[j][i] = 1;
            }
        }
    }
    return symmetrical;
}
```

```
void freeMatrix(double **matrix, int n) {
    for (int i = 0; i < n; ++i) {
        free(matrix[i]);
    }
    free(matrix);
}

int* graphDegrees(double** matrix) {
    const int number = verts_amount;
    int* vertexDegree;
    vertexDegree = malloc(number * sizeof(int));
    int vertexDegreeCounter;
    for (int i = 0; i < number; ++i) {
        vertexDegreeCounter = 0;
        for (int j = 0; j < number; ++j) {
            if(matrix[i][j] && i == j) vertexDegreeCounter += 2;
            else if(matrix[i][j]) vertexDegreeCounter++;
        }
        vertexDegree[i] = vertexDegreeCounter;
    }
    return vertexDegree;
}

int* halfDegreeEntry(double** matrix) {
    const int number = verts_amount;
    int* vertexDegree;
    vertexDegree = malloc(number * sizeof(int));
    int vertexDegreeCounter;
    for (int j = 0; j < number; ++j) {
        vertexDegreeCounter = 0;
        for (int i = 0; i < number; ++i) {
            if(matrix[i][j] ) vertexDegreeCounter++;
        }
        vertexDegree[j] = vertexDegreeCounter;
    }
    return vertexDegree;
}

int* halfDegreeExit(double** matrix) {
    const int number = verts_amount;
    int* vertexDegree;
    vertexDegree = malloc(number * sizeof(int));
    int vertexDegreeCounter;
    for (int i = 0; i < number; ++i) {
        vertexDegreeCounter = 0;
        for (int j = 0; j < number; ++j) {
            if(matrix[i][j]) vertexDegreeCounter++;
        }
        vertexDegree[i] = vertexDegreeCounter;
```

```
    }
    return vertexDegree;
}

int* summarizeHalfDegrees(const int* exit, const int* entry) {
    const int number = verts_amount;
    int* vertexDegree = malloc(number * sizeof(int));
    for (int i = 0; i < number; ++i) {
        vertexDegree[i] = exit[i] + entry[i];
    }

    return vertexDegree;
}

int isUniform(const int* degreesArray) {
    const int number = verts_amount;
    int firstDegree = degreesArray[0];
    for (int i = 1; i < number; ++i) {
        if(degreesArray[i] != firstDegree) return 0;
        firstDegree = degreesArray[i];
    }
    return 1;
}

double** summarizeMatrices(double** AMatrix, double** BMatrix) {
    const int number = verts_amount;
    double **summedMatrix = (double **) malloc(sizeof(double *) *
number);
    for (int i = 0; i < number; i++) {
        summedMatrix[i] = (double *) malloc(sizeof(double) * number);
        for (int j = 0; j < number; ++j) {
            summedMatrix[i][j] = AMatrix[i][j] + BMatrix[i][j];
        }
    }
    return summedMatrix;
}

double** multiplyMatrices(double** AMatrix, double** BMatrix) {
    const int number = verts_amount;
    double **multipliedMatrix = (double **) malloc(sizeof(double *) *
number);
    for (int i = 0; i < number; ++i) {
        multipliedMatrix[i] = (double *) malloc(sizeof(double) *
number);
        for (int j = 0; j < number; ++j) {
            multipliedMatrix[i][j] = 0;
            for (int e = 0; e < number; ++e) {
                multipliedMatrix[i][j] += AMatrix[i][e] *
BMatrix[e][j];
```

```
            }
        }
    }
    return multipliedMatrix;
}

double** copyMatrix(double** matrix) {
    const int number = verts_amount;
    double **copiedMatrix = (double **) malloc(sizeof(double *) *
number);
    for (int i = 0; i < number; ++i) {
        copiedMatrix[i] = (double *) malloc(sizeof(double) * number);
        for (int j = 0; j < number; ++j) {
            copiedMatrix[i][j] = matrix[i][j];
        }
    }
    return copiedMatrix;
}

void booleanConversion(double** matrix) {
    const int number = verts_amount;
    for (int i = 0; i < number; ++i) {
        for (int j = 0; j < number; ++j) {
            if (matrix[i][j]) matrix[i][j] = 1;
        }
    }
}

double **directedMatrix(double K) {
    double **T = randm(verts_amount);
    double **A = mulmr(K, T, verts_amount);
    return A;
}

double** getReachabilityMatrix(double** matrix) {
    const int number = verts_amount;
    double **copy = copyMatrix(matrix);
    double **sum = copy;
    double **prev = copy;
    double **tempPrev, **tempSum;
    for (int i = 1; i < number - 1; i++) {
        tempPrev = multiplyMatrices(prev, matrix);
        tempSum = summarizeMatrices(sum, tempPrev);
        freeMatrix(sum,number);
        freeMatrix(prev,number);
        prev = tempPrev;
        sum = tempSum;
    }
    for (int i = 0; i < number; i++) {
```

```
        sum[i][i] += 1;
    }
    freeMatrix(prev,number);
    booleanConversion(sum);
    return sum;
}

void depthFirstSearch(double** connectivityMatrix, int startVertex,
double* component, int* visited) {
    const int number = verts_amount;
    visited[startVertex] = 1;
    component[startVertex] = 1;
    for (int nearbyVertex = 0; nearbyVertex < number; ++nearbyVertex)
{
        if(!visited[nearbyVertex] &&
connectivityMatrix[startVertex][nearbyVertex]) {
            depthFirstSearch(connectivityMatrix, nearbyVertex,
component, visited);
        }
    }
}

double** transposeMatrix(double** matrix,  int number) {
    number = verts_amount;
    double **transposedMatrix = malloc(number * sizeof(double*));
    for (int i = 0; i < number; ++i) {
        transposedMatrix[i] = malloc(number * sizeof(double ));
        for (int j = 0; j < number; ++j) {
            transposedMatrix[i][j] = matrix[j][i];
        }
    }
    return transposedMatrix;

}

int countNonZeroEntries(double **matrix) {
    int numVertices = verts_amount;
    int count = 0;

    for (int row = 0; row < numVertices; row++) {
        for (int col = 0; col < numVertices; col++) {
            if (matrix[row][col]) {
                count++;
                row++;
            }
        }
    }
    return count;
}
```

```
double** findStrongComponents(double** strongMatrix) {
    const int number = verts_amount;
    int* visitedVertex = calloc(number, sizeof(int));
    double** connectedComponents = calloc(number, sizeof(double *));
    for (int i = 0; i < number; i++) {
        connectedComponents[i] = calloc(number, sizeof(double));
    }

    for (int i = 0; i < number; ++i) {
        if(!visitedVertex[i]) {
            depthFirstSearch(strongMatrix, i, connectedComponents[i],
visitedVertex);
        }
    }

    free(visitedVertex);
    return connectedComponents;
}

double** getStrongConnectivityMatrix(double **reachabilityMatrix) {
    const int number = verts_amount;
    double** transposedMatrix = transposeMatrix(reachabilityMatrix,
number);
    double **strongConnectivityMatrix = malloc(number*
sizeof(double*));
    for (int i = 0; i < number; i++) {
        strongConnectivityMatrix[i] = malloc(number* sizeof(double));
        for (int j = 0; j < number; j++) {
            strongConnectivityMatrix[i][j] = reachabilityMatrix[i][j]
* transposedMatrix[i][j];
        }
    }
    freeMatrix(transposedMatrix, number);
    return strongConnectivityMatrix;
}

void condensationMatrix(double** strongComponents) {
    int numComponents = countNonZeroEntries(strongComponents);

    double **adjacencyMatrix = calloc(numComponents,
sizeof(double*));
    for (int i = 0; i < numComponents; i++) {
        adjacencyMatrix[i] = calloc(numComponents, sizeof(double));
    }

    int position = 1;
    for (int i = 0; i < verts_amount; ++i) {
        if(!strongComponents[0][i]) {
```

```
            adjacencyMatrix[0][position] = 1;
            position++;
        }
    }

    for (int i = 0; i < numComponents; i++) {
        for (int j = 0; j < numComponents; j++) {
            printf("%.0lf  ", adjacencyMatrix[i][j]);
        }
        printf("\n");
    }

    freeMatrix(adjacencyMatrix, verts_amount);

}
double **generateAdjacencyMatrixFromStrongComponents(double
**components) {
    const int number = verts_amount;
    double **matrix = calloc(number, sizeof(size_t*));
    for (int i = 0; i < number; i++) {
        matrix[i] = calloc(number, sizeof(double));
    }

    for(int i = 0; i < number; i++) {
        if (!components[0][i]) matrix[1][i+1] = 1;
    }

    return matrix;
}

void dfss(double** strongMatrix, int vertex, int* visitedVertex) {
    visitedVertex[vertex] = 1;

    for (int i = 0; i < verts_amount; ++i) {
        if (strongMatrix[vertex][i] && !visitedVertex[i]) {
            dfss(strongMatrix, i, visitedVertex);
        }
    }
}

double **condensationMatrixWithCoefficient(double K) {
    double **matrix = directedMatrix(K);
    double **reachability = getReachabilityMatrix(matrix);
    double **connectivity =
getStrongConnectivityMatrix(reachability);
    double **components = findStrongComponents(connectivity);

    freeMatrix(matrix, verts_amount);
    freeMatrix(reachability, verts_amount);
```

```
    freeMatrix(connectivity, verts_amount);
    return components;
}

int getStrongComponents(double** strongMatrix) {
    const int number = verts_amount;
    int* visitedVertex = calloc(number, sizeof(int));
    int count = 0;

    for (int i = 0; i < number; ++i) {
        if (!visitedVertex[i]) {
            dfss(strongMatrix, i,  visitedVertex);
            count++;
        }
    }

    free(visitedVertex);
    return count;
}
```

props.h:

```
#ifndef LAB_2_4_PROPS_H
#define LAB_2_4_PROPS_H

double **randm(int n);
double **mulmr(double coef, double **matrix, int n);
int* halfDegreeEntry(double** matrix);
int* halfDegreeExit(double** matrix);
int* summarizeHalfDegrees(const int* exit, const int* entry);
int isUniform(const int* degreesArray);
void freeMatrix(double **matrix, int n);
double **symmetricalMatrix(double **matrix, int n);
int* graphDegrees(double** matrix);
double** multiplyMatrices(double** AMatrix, double** BMatrix);
double** getStrongConnectivityMatrix(double **reachabilityMatrix);
double** getReachabilityMatrix(double** matrix);
double** findStrongComponents(double** strongMatrix);
void condensationMatrix(double** strongComponents);
double **generateAdjacencyMatrixFromStrongComponents(double
**components);
double **condensationMatrixWithCoefficient(double K);
int getStrongComponents(double** strongMatrix);
#endif
```

Результати виконання

Результати для напрямленого та ненапрямленого графів з підстепенями входу та виходу (для напрямленого) та степенями (для ненапрямленого), перевіркою на однорідність і на ізольовані й висячі вершини:

```
Directed Graph

        Initial matrix
0  0  0  0  0  0  0  0  0  0  0
1  1  0  0  0  1  0  0  1  1  0
1  0  1  0  0  1  0  0  0  0  0
1  1  0  0  0  0  0  1  0  0  0
0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  1  0  0
0  0  0  0  0  0  0  0  0  0  0
0  1  0  1  0  1  0  0  0  1  0
0  0  0  1  0  0  0  0  0  0  1
1  0  0  0  0  0  1  0  0  0  0
0  0  0  1  0  1  1  0  0  0  0
Exit degree : {  0  5  3  3  0  1  0  4  2  2  3  }
Entry degree : {  4  3  1  3  0  4  2  1  2  2  1  }
        The graph is not uniform
Isolated vertices : # 5
Terminal vertices :

Undirected Graph

        Matrix for Undirected Graph
0  1  1  1  0  0  0  0  0  1  0
1  1  0  1  0  1  0  1  1  1  0
1  0  1  0  0  1  0  0  0  0  0
1  1  0  0  0  0  0  1  1  0  1
0  0  0  0  0  0  0  0  0  0  0
0  1  1  0  0  0  0  1  1  0  1
0  0  0  0  0  0  0  0  0  1  1
0  1  0  1  0  1  0  0  0  1  0
0  1  0  1  0  1  0  0  0  0  1
1  1  0  0  0  0  1  1  0  0  0
0  0  0  1  0  1  1  0  1  0  0
Undirected graph degrees : {  4  8  4  5  0  5  2  4  4  4  4  }
        The graph is not uniform
Isolated vertices : # 5
Terminal vertices :
```

Для модифікованого графу з підстепенями входу та виходу, перевіркою на однорідність і на ізольовані й висячі вершини:

```
Modified Graph

        Matrix for Modified Graph
0  0  0  1  0  0  0  0  0  0  0
1  1  0  0  0  1  0  0  1  1  0
1  0  1  0  0  1  0  0  0  0  0
1  1  0  0  0  0  0  1  1  0  0
0  0  0  0  0  0  0  0  1  0  0
0  0  0  0  0  0  0  0  1  0  0
0  0  0  0  0  0  0  0  0  0  0
0  1  0  1  0  1  0  0  0  1  0
0  0  0  1  0  0  0  0  0  0  1
1  0  0  0  0  0  1  0  0  1  0
0  0  0  1  0  1  1  0  0  0  0
Exit degree : {  1  5  3  4  1  1  0  4  2  3  3  }
Entry degree : {  4  3  1  4  0  4  2  1  4  3  1  }
        The graph is not uniform
Isolated vertices :
Terminal vertices : # 5
```

Квадратна модифікована матриця і шляхи довжиною 2:

```
Matrix squared : 2
1  1  0  0  0  0  0  1  1  0  0
2  1  0  2  0  1  1  0  2  2  1
1  0  1  1  0  1  0  0  1  0  0
1  2  0  3  0  2  0  0  1  2  1
0  0  0  1  0  0  0  0  0  0  1
0  0  0  1  0  0  0  0  0  0  1
0  0  0  0  0  0  0  0  0  0  0
3  2  0  0  0  1  1  1  3  2  0
1  1  0  1  0  1  1  1  1  0  0
1  0  0  1  0  0  1  0  0  1  0
1  1  0  0  0  0  0  1  2  0  0

Pathways with length : 2
1 -> 4 -> 1; 1 -> 4 -> 2; 1 -> 4 -> 8; 1 -> 4 -> 9;
2 -> 10 -> 1; 2 -> 2 -> 2; 2 -> 9 -> 4; 2 -> 2 -> 6; 2 -> 10 -> 7; 2 -> 6 -> 9; 2 -> 10 -> 10; 2 -> 9 -> 11;
3 -> 3 -> 1; 3 -> 3 -> 3; 3 -> 1 -> 4; 3 -> 3 -> 6; 3 -> 6 -> 9;
4 -> 2 -> 1; 4 -> 8 -> 2; 4 -> 9 -> 4; 4 -> 8 -> 6; 4 -> 2 -> 9; 4 -> 8 -> 10; 4 -> 9 -> 11;
5 -> 9 -> 4; 5 -> 9 -> 11;
6 -> 9 -> 4; 6 -> 9 -> 11;

8 -> 10 -> 1; 8 -> 4 -> 2; 8 -> 2 -> 6; 8 -> 10 -> 7; 8 -> 4 -> 8; 8 -> 6 -> 9; 8 -> 10 -> 10;
9 -> 4 -> 1; 9 -> 4 -> 2; 9 -> 11 -> 4; 9 -> 11 -> 6; 9 -> 11 -> 7; 9 -> 4 -> 8; 9 -> 4 -> 9;
10 -> 10 -> 1; 10 -> 1 -> 4; 10 -> 10 -> 7; 10 -> 10 -> 10;
11 -> 4 -> 1; 11 -> 4 -> 2; 11 -> 4 -> 8; 11 -> 6 -> 9;
```

Кубічна модифікована матриця і шляхи довжиною 3:

```
Matrix cubed : 3
1  2  0  3  0  2  0  0  1  2  1
5  3  0  5  0  2  3  2  4  3  2
2  1  1  2  0  1  0  1  2  0  1
7  5  0  3  0  3  3  3  7  4  1
1  1  0  1  0  1  1  1  1  0  0
1  1  0  1  0  1  1  1  1  0  0
0  0  0  0  0  0  0  0  0  0  0
4  3  0  7  0  3  2  0  3  5  3
2  3  0  3  0  2  0  1  3  2  1
2  1  0  1  0  0  1  1  1  1  0
1  2  0  4  0  2  0  0  1  2  2
```

```
Pathways with length : 3
1 -> 4 -> 2 -> 1; 1 -> 4 -> 8 -> 2; 1 -> 4 -> 9 -> 4; 1 -> 4 -> 8 -> 6; 1 -> 4 -> 2 -> 9; 1 -> 4 -> 8 -> 10; 1
-> 4 -> 9 -> 11;
2 -> 10 -> 10 -> 1; 2 -> 9 -> 4 -> 2; 2 -> 10 -> 1 -> 4; 2 -> 9 -> 11 -> 6; 2 -> 10 -> 10 -> 7; 2 -> 9 -> 4 ->
8; 2 -> 9 -> 4 -> 9; 2 -> 10 -> 10 -> 10; 2 -> 6 -> 9 -> 11;
3 -> 3 -> 3 -> 1; 3 -> 1 -> 4 -> 2; 3 -> 3 -> 3 -> 3; 3 -> 6 -> 9 -> 4; 3 -> 3 -> 3 -> 6; 3 -> 1 -> 4 -> 8; 3 -
> 3 -> 6 -> 9; 3 -> 6 -> 9 -> 11;
4 -> 9 -> 4 -> 1; 4 -> 9 -> 4 -> 2; 4 -> 9 -> 11 -> 4; 4 -> 9 -> 11 -> 6; 4 -> 9 -> 11 -> 7; 4 -> 9 -> 4 -> 8;
4 -> 9 -> 4 -> 9; 4 -> 8 -> 10 -> 10; 4 -> 2 -> 9 -> 11;
5 -> 9 -> 4 -> 1; 5 -> 9 -> 4 -> 2; 5 -> 9 -> 11 -> 4; 5 -> 9 -> 11 -> 6; 5 -> 9 -> 11 -> 7; 5 -> 9 -> 4 -> 8;
5 -> 9 -> 4 -> 9;
6 -> 9 -> 4 -> 1; 6 -> 9 -> 4 -> 2; 6 -> 9 -> 11 -> 4; 6 -> 9 -> 11 -> 6; 6 -> 9 -> 11 -> 7; 6 -> 9 -> 4 -> 8;
6 -> 9 -> 4 -> 9;

8 -> 10 -> 10 -> 1; 8 -> 4 -> 8 -> 2; 8 -> 10 -> 1 -> 4; 8 -> 4 -> 8 -> 6; 8 -> 10 -> 10 -> 7; 8 -> 4 -> 2 -> 9
; 8 -> 10 -> 10 -> 10; 8 -> 6 -> 9 -> 11;
9 -> 11 -> 4 -> 1; 9 -> 11 -> 4 -> 2; 9 -> 4 -> 9 -> 4; 9 -> 4 -> 8 -> 6; 9 -> 11 -> 4 -> 8; 9 -> 11 -> 6 -> 9;
 9 -> 4 -> 8 -> 10; 9 -> 4 -> 9 -> 11;
10 -> 10 -> 10 -> 1; 10 -> 1 -> 4 -> 2; 10 -> 10 -> 1 -> 4; 10 -> 10 -> 10 -> 7; 10 -> 1 -> 4 -> 8; 10 -> 1 ->
4 -> 9; 10 -> 10 -> 10 -> 10;
11 -> 4 -> 2 -> 1; 11 -> 4 -> 8 -> 2; 11 -> 6 -> 9 -> 4; 11 -> 4 -> 8 -> 6; 11 -> 4 -> 2 -> 9; 11 -> 4 -> 8 ->
10; 11 -> 6 -> 9 -> 11;
```

Матриця досяжності:

```
Reachability Matrix of Mod graph
1  1  0  1  0  1  1  1  1  1  1
1  1  0  1  0  1  1  1  1  1  1
1  1  1  1  0  1  1  1  1  1  1
1  1  0  1  0  1  1  1  1  1  1
1  1  0  1  1  1  1  1  1  1  1
1  1  0  1  0  1  1  1  1  1  1
0  0  0  0  0  0  1  0  0  0  0
1  1  0  1  0  1  1  1  1  1  1
1  1  0  1  0  1  1  1  1  1  1
1  1  0  1  0  1  1  1  1  1  1
1  1  0  1  0  1  1  1  1  1  1
```

Матриця зв’язності:

```
Connected Matrix of Mod graph
1  1  0  1  0  1  0  1  1  1  1
1  1  0  1  0  1  0  1  1  1  1
0  0  1  0  0  0  0  0  0  0  0
1  1  0  1  0  1  0  1  1  1  1
0  0  0  0  1  0  0  0  0  0  0
1  1  0  1  0  1  0  1  1  1  1
0  0  0  0  0  0  1  0  0  0  0
1  1  0  1  0  1  0  1  1  1  1
1  1  0  1  0  1  0  1  1  1  1
1  1  0  1  0  1  0  1  1  1  1
1  1  0  1  0  1  0  1  1  1  1
```

Компоненти сильної зв’язності:

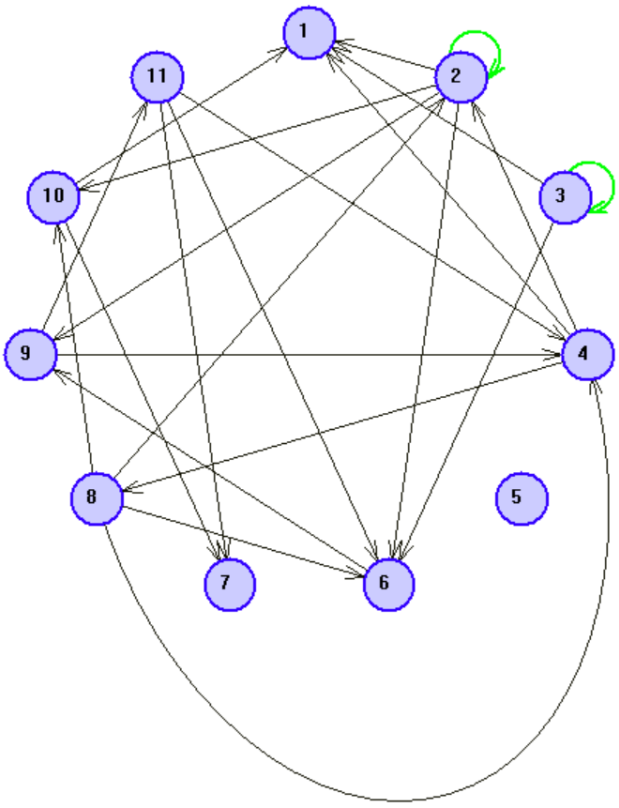
```
Strongly Connected Components of Mod Graph
Component 1: [  1  2  4  6  8  9  10  11  ]
Component 2: [  3  ]
Component 3: [  5  ]
Component 4: [  7  ]
```

Матриця графа конденсації

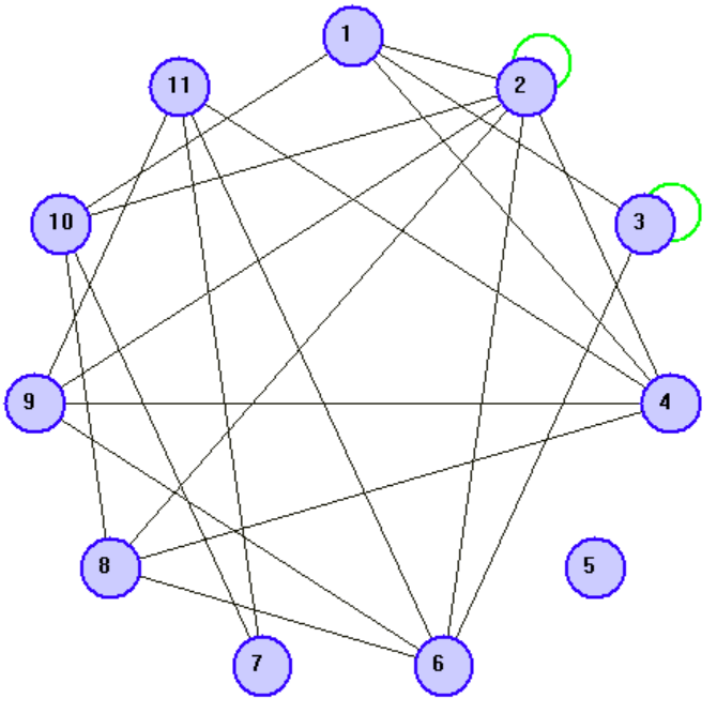
```
Matrix of Condensation Graph
0  1  1  1
0  0  0  0
0  0  0  0
0  0  0  0
```

Графічні представлення графів

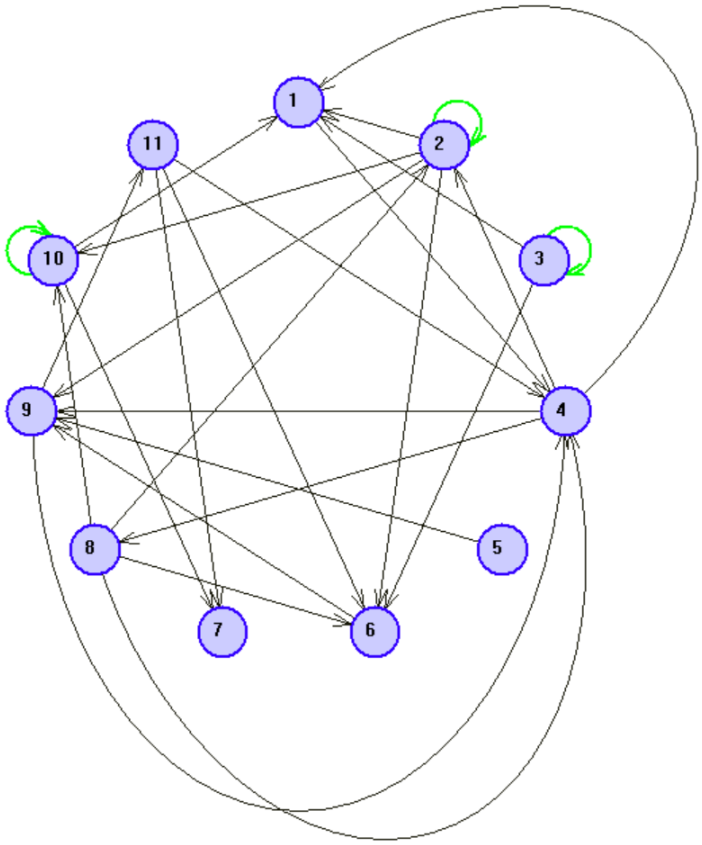
Напрямлений:



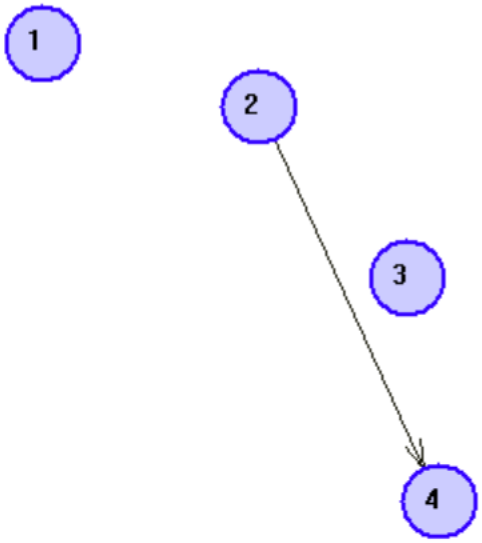
Ненапрямлений:



Модифікований:



Граф конденсації:



Висновок

Виконавши лабораторну роботу, я дослідив характеристики графів та навчився визначати їх на конкретних прикладах, вивчив метод транзитивного замикання, ознайомився з алгоритмами обходу графа і використав пошук вглибину для знаходження компонентів сильної зв’язності. Я обрав саме цей алгоритм, адже він простіший в реалізації та більш зрозумілий, завдяки рекурсивній структурі. Також покращив розуміння і навички роботи з динамічною пам’яттю (виділенням і очищенням).