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 No4

з дисципліни

«Алгоритми і структури даних»

Виконав:
студент групи ІМ-33
Козарезов Кирил Олександрович
номер у списку групи: 12

Перевірила:
Молчанова А. А.

Київ 2024

Постановка задачі

1. Представити напрямлений та ненапрямлений графи із заданими параметрами так само, як у лабораторній роботі №3.

   Відмінність: коефіцієнт $k = 1.0 - n_3 * 0.01 - n_4 * 0.01 - 0.3$.

   Отже, матриця суміжності A_{dir} напрямленого графа за варіантом формується таким чином:

   1) встановлюється параметр (seed) генератора випадкових чисел, рівне номеру варіанту $n_1 n_2 n_3 n_4$;
   2) матриця розміром $n \cdot n$ заповнюється згенерованими випадковими числами в діапазоні $[0, 2.0)$;
   3) обчислюється коефіцієнт $k = 1.0 - n_3 * 0.01 - n_4 * 0.01 - 0.3$, кожен елемент матриці множиться на коефіцієнт k;
   4) елементи матриці округлюються: 0 — якщо елемент менший за 1.0, 1 — якщо елемент більший або дорівнює 1.0.

2. Обчислити:

   1) степені вершин напрямленого і ненапрямленого графів;
   2) напівстепені виходу та заходу напрямленого графа;
   3) чи є граф однорідним (регулярним), і якщо так, вказати степінь однорідності графа;

4) перелік висячих та ізольованих вершин.

Результати вивести у графічне вікно, консоль або файл.

3. ***Змінити матрицю*** A_{dir}, коефіцієнт $k = 1.0 - n_3 * 0.005 - n_4 * 0.005 - 0.27$.

4. Для нового орграфа обчислити:

   1) півстепені вершин;
   2) всі шляхи довжини 2 і 3;
   3) матрицю досяжності;
   4) матрицю сильної зв'язності;
   5) перелік компонент сильної зв'язності;
   6) граф конденсації.

   Результати вивести у графічне вікно, в консоль або файл.

   Шляхи довжиною 2 і 3 слід шукати за матрицями A^2 і A^3, відповідно. Як результат вивести перелік шляхів, включно з усіма проміжними вершинами, через які проходить шлях.

   Матрицю досяжності та компоненти сильної зв'язності слід шукати за допомогою операції транзитивного замикання. У переліку компонент слід вказати, які вершини належать до кожної компоненти.

   Граф конденсації вивести у графічне вікно.

При проєктуванні програми ***слід врахувати наступне***:

1) мова програмування обирається студентом самостійно;

2) графічне зображення усіх графів має формуватися програмою з тими ж вимогами, як у ЛР №3;

3) всі графи, включно із графом конденсації, обов'язково зображувати у графічному вікні;

4) типи та структури даних для внутрішнього представлення всіх даних у програмі слід вибрати самостійно;

5) обчислення перелічених у завданні результатів має виконуватися розробленою програмою (не вручну і не сторонніми засобами);

6) матриці, переліки степенів та маршрутів тощо можна виводити в графічне вікно або консоль — на розсуд студента;

7) у переліку знайдених шляхів треба вказувати не лише початок та кінець шляху, але й усі проміжні вершини, через які він проходить (наприклад, 1 – 5 – 3 – 2).

Текст програми:

```
const variant = 3312;
const variantString = variant.toString();
const n1 = parseInt(variantString[0]);
const n2 = parseInt(variantString[1]);
const n3 = parseInt(variantString[2]);
const n4 = parseInt(variantString[3]);

const numTops = n3 + 10;
let coefficient = 1 - n3 * 0.01 - n4 * 0.01 - 0.3;

let matrixDirected = [];
let matrixUndirected = [];

// Генерація напрямленої матриці суміжності
for (let i = 0; i < numTops; i++) {
    let row = [];
    for (let j = 0; j < numTops; j++) {
        let elem = (Math.random() * 2) * coefficient;
        row.push(Math.floor(elem));
    }
    matrixDirected.push(row);
}

// Генерація ненапрямленої матриці суміжності
for (let i = 0; i < numTops; i++) {
    let row = [];
    for (let j = 0; j < numTops; j++) {
        row.push(matrixDirected[i][j] || matrixDirected[j][i]);
    }
    matrixUndirected.push(row);
}

// Обчислення степенів вершин для напрямленого графа
let inDegrees = new Array(numTops).fill(0);
let outDegrees = new Array(numTops).fill(0);
for (let i = 0; i < numTops; i++) {
    for (let j = 0; j < numTops; j++) {
        outDegrees[i] += matrixDirected[i][j];
        inDegrees[j] += matrixDirected[i][j];
    }
}

// Обчислення степенів вершин для ненапрямленого графа
let degrees = new Array(numTops).fill(0);
for (let i = 0; i < numTops; i++) {
    for (let j = 0; j < numTops; j++) {
        degrees[i] += matrixUndirected[i][j];
    }
}

console.log("In-degrees (Directed):", inDegrees);
console.log("Out-degrees (Directed):", outDegrees);
console.log("Degrees (Undirected):", degrees);

// Перевірка на однорідність (регулярність)
const isDirectedRegular = inDegrees.every(val => val === inDegrees[0]) &&
outDegrees.every(val => val === outDegrees[0]);
const isUndirectedRegular = degrees.every(val => val === degrees[0]);
let directedRegularityDegree = isDirectedRegular ? inDegrees[0] : null;
let undirectedRegularityDegree = isUndirectedRegular ? degrees[0] : null;

console.log("Is Directed Graph Regular:", isDirectedRegular, "with degree:",
directedRegularityDegree);
console.log("Is Undirected Graph Regular:", isUndirectedRegular, "with degree:",
undirectedRegularityDegree);
```

```
// Визначення висячих та ізольованих вершин
let hangingVerticesDirected = [];
let isolatedVerticesDirected = [];
let hangingVerticesUndirected = [];
let isolatedVerticesUndirected = [];

for (let i = 0; i < numTops; i++) {
    if (inDegrees[i] + outDegrees[i] === 1) {
        hangingVerticesDirected.push(i + 1);
    }
    if (inDegrees[i] === 0 && outDegrees[i] === 0) {
        isolatedVerticesDirected.push(i + 1);
    }
    if (degrees[i] === 1) {
        hangingVerticesUndirected.push(i + 1);
    }
    if (degrees[i] === 0) {
        isolatedVerticesUndirected.push(i + 1);
    }
}

console.log("Hanging vertices (Directed):", hangingVerticesDirected);
console.log("Isolated vertices (Directed):", isolatedVerticesDirected);
console.log("Hanging vertices (Undirected):", hangingVerticesUndirected);
console.log("Isolated vertices (Undirected):", isolatedVerticesUndirected);
console.log("Directed matrix:");
console.log(matrixDirected);
console.log("Undirected matrix:");
console.log(matrixUndirected);



const width = 1000;
const height = 1000;
const vertexRadius = 30;
const arrOfNode = [];
const arrOfNode2=[];
const arrOfThreeNodes = [
    { x: -200, y: 0 },   // Coordinates for AC
    { x: 0, y: -400 },   // Coordinates for BC
    { x: 200, y: 0 }     // Coordinates for AB
];
const arrOfThreeNodesArrow=[
    { x: -200, y: 0 },   // Coordinates for AC
    { x: 0, y: -400 },   // Coordinates for BC
    { x: 200, y: 0 }
];



//Undirected graph
const canvas = document.getElementById('graphCanvas');
const ctx = canvas.getContext('2d');

canvas.width = width;
canvas.height = height;

function transformCoordinateArea() {
    ctx.translate(width / 2, height / 1.25);
}

function drawCircle(x, y, num, radius) {
    ctx.beginPath();
    ctx.arc(x, y, radius, 0, Math.PI * 2);
    ctx.fillStyle = 'purple';
    ctx.fill();
```

```
    ctx.closePath();

    ctx.fillStyle = 'white';
    ctx.font = 'bold 20px Arial';
    ctx.textAlign = 'center';
    ctx.textBaseline = 'middle';

    ctx.fillText(num, x, y);
}

transformCoordinateArea();

function createVertex() {
    const lengthAB = Math.sqrt(((-200 - 200) ** 2) + (0 ** 2));
    const lengthBC = Math.sqrt(((200) ** 2) + (0 - (-400)) ** 2);
    const lengthAC = lengthBC;

    const cosAngle = 200 / lengthAC;
    const sinAngle = Math.sqrt(1 - cosAngle ** 2);

    const dXAC = (lengthAC * cosAngle) / 4;
    const dYAC = (lengthAC * sinAngle) / 4;

    const dXBC = (lengthAC * cosAngle) / 4;
    const dYBC = (lengthAC * sinAngle) / 4;

    const dXAB = lengthAB / 3;

    let XforAC = -200;
    let YforAC = 0;
    let XforBC = 0;
    let YforBC = -400;
    let XforAB = 200;

    for (let i = 0; i < 4; i++) {
        arrOfNode.push({ x: XforAC, y: YforAC });
        drawCircle(XforAC, YforAC, String.fromCharCode(65 + i), vertexRadius);
// A, B, C, D
        XforAC += dXAC;
        YforAC -= dYAC;
    }

    for (let i = 0; i < 4; i++) {
        arrOfNode.push({ x: XforBC, y: YforBC });
        drawCircle(XforBC, YforBC, String.fromCharCode(69 + i), vertexRadius);
// E, F, G, H
        XforBC += dXBC;
        YforBC += dYBC;
    }

    for (let i = 0; i < 3; i++) {
        arrOfNode.push({ x: XforAB, y: 0 });
        drawCircle(XforAB, 0, String.fromCharCode(73 + i), vertexRadius); // I,
J, K
        XforAB -= dXAB;
    }
}

createVertex();

function drawEdges() {
    for (let i = 0; i < numTops; i++) {
        for (let j = i; j < numTops; j++) {
            if (matrixUndirected[i][j] === 1) {
                if (i === j) {
                    drawSelfLoop(arrOfNode[j]);
```

```
                } else {
                    chooseLine(arrOfNode[i].x, arrOfNode[i].y, arrOfNode[j].x,
arrOfNode[j].y);
                }
            }
        }
    }
}

function drawSelfLoop(coordinate) {
    ctx.beginPath();
    ctx.arc(coordinate.x - 45, coordinate.y, 20, Math.PI / 6, (Math.PI * 11) /
6);
    ctx.stroke();
    ctx.closePath();
}

function drawArcLine(start, end, bendAngle = Math.PI / 8) {
    const midX = (start.x + end.x) / 2;
    const midY = (start.y + end.y) / 2;

    let controlX, controlY;

    if (start.x !== end.x && start.y !== end.y) {
        controlX = midX + Math.cos(bendAngle) * (midY - start.y);
        controlY = midY + Math.sin(bendAngle) * (midX - start.x);
    } else if (start.x === end.x) {
        controlX = midX + 100;
        controlY = midY;
    } else {
        controlX = midX;
        controlY = midY + 100;
    }

    ctx.beginPath();
    ctx.moveTo(start.x, start.y);
    ctx.quadraticCurveTo(controlX, controlY, end.x, end.y);
    ctx.stroke();
    ctx.closePath();
}

function chooseLine(x1, y1, x2, y2) {
    if (checkFunction({ x: x1, y: y1 }, { x: x2, y: y2 })) {
        drawArcLine({ x: x1, y: y1 }, { x: x2, y: y2 });
    } else {
        ctx.beginPath();
        ctx.moveTo(x1, y1);
        ctx.lineTo(x2, y2);
        ctx.stroke();
        ctx.closePath();
    }
}

function checkFunction(start, end) {
    const [A, B, C] = arrOfThreeNodes;

    function isPointOnLineSegment(P, A, B) {
        const crossProduct = (P.y - A.y) * (B.x - A.x) - (P.x - A.x) * (B.y -
A.y);
        if (Math.abs(crossProduct) !== 0) return false;

        const dotProduct = (P.x - A.x) * (B.x - A.x) + (P.y - A.y) * (B.y -
A.y);
        if (dotProduct < 0) return false;

        const squaredLengthBA = (B.x - A.x) ** 2 + (B.y - A.y) ** 2;
```

```
        if (dotProduct > squaredLengthBA) return false;

        return true;
    }

    function isOnSameEdge(P1, P2, A, B) {
        return isPointOnLineSegment(P1, A, B) && isPointOnLineSegment(P2, A, B);
    }

    return (
        isOnSameEdge(start, end, A, B) ||
        isOnSameEdge(start, end, B, C) ||
        isOnSameEdge(start, end, C, A)
    );
}

drawEdges();
console.log(arrOfNode);

//Directed graph
let canvasArrow = document.getElementById('graphCanvasWithArrows');
let ctxArrow = canvasArrow.getContext('2d');
ctxArrow.strokeStyle='black';

const widthArrow = 1000;
const heightArrow = 1000;
const vertexRadiusArrow = 15;

canvasArrow.width = widthArrow;
canvasArrow.height = heightArrow;

ctxArrow.arc(widthArrow / 2, heightArrow / 2, vertexRadiusArrow, 0, Math.PI *
2);
ctxArrow.fillStyle = "green";

function drawCircleArrow(x, y, num, radius) {
    ctxArrow.beginPath();
    ctxArrow.arc(x, y, radius, 0, Math.PI * 2);
    ctxArrow.fillStyle = 'green';
    ctxArrow.fill();
    ctxArrow.closePath();

    ctxArrow.fillStyle = 'white';
    ctxArrow.font = 'bold 20px Arial';
    ctxArrow.textAlign = 'center';
    ctxArrow.textBaseline = 'middle';

    ctxArrow.fillText(num, x, y);
}
function transformCoordinateAreaArrow() {
    ctxArrow.translate(widthArrow / 2, heightArrow / 1.25);

}
transformCoordinateAreaArrow();
function createVertexArrow() {
    const lengthABArrow = Math.sqrt(((-200 - 200) ** 2) + (0 ** 2));
    const lengthBCArrow = Math.sqrt(((200) ** 2) + (0 - (-400)) ** 2);
    const lengthACArrow = lengthBCArrow;

    const cosAngle = 200 / lengthACArrow;
    const sinAngle = Math.sqrt(1 - cosAngle ** 2);

    // Gap between which the circles (vertices) will be drawn
    const dXACArrow = (lengthACArrow * cosAngle) / 4;
    const dYACArrow = (lengthACArrow * sinAngle) / 4;
    const dXBCArrow = (lengthACArrow * cosAngle) / 4;
```

```
    const dYBCArrow = (lengthACArrow * sinAngle) / 4;
    const dXABArrow = lengthABArrow / 3;

    let XforACArrow = -200;
    let YforACArrow = 0;
    let XforBCArrow = 0;
    let YforBCArrow = -400;
    let XforABArrow = 200;
    // Draw vertices A, B, C
    for (let i = 0; i < 4; i++) {
        arrOfNode2.push({ x: XforACArrow, y: YforACArrow });
        drawCircleArrow(XforACArrow, YforACArrow, String.fromCharCode(65 + i),
30); // A, B, C, D
        XforACArrow += dXACArrow;
        YforACArrow -= dYACArrow;
    }
    // Draw vertices E, F, G
    for (let i = 0; i < 4; i++) {
        arrOfNode2.push({ x: XforBCArrow, y: YforBCArrow });
        drawCircleArrow(XforBCArrow, YforBCArrow, String.fromCharCode(69 + i),
30); // E, F, G, H
        XforBCArrow += dXBCArrow;
        YforBCArrow += dYBCArrow;
    }
    // Draw vertices I, J, K
    for (let i = 0; i < 3; i++) {
        arrOfNode2.push({ x: XforABArrow, y: 0 });
        drawCircleArrow(XforABArrow, 0, String.fromCharCode(73 + i), 30); // I,
J, K
        XforABArrow -= dXABArrow;
    }
}

createVertexArrow();

function drawEdgesArrow() {
    for (let i = 0; i < 11; i++) {
        for (let j = i; j < 11; j++) {
            if (matrixDirected[i][j] === 1 && i===j) {
                drawSelfLoopArrow(arrOfNode2[j]);
            } else if(matrixDirected[i][j] === 1) {
                chooseLineArrow(arrOfNode2[i].x, arrOfNode2[i].y,
arrOfNode2[j].x, arrOfNode2[j].y);
            }

        }
    }
}


function drawSelfLoopArrow(coordinateArrow) {
    const arrowheadSize=10
    ctxArrow.beginPath();
    ctxArrow.arc(coordinateArrow.x - 45, coordinateArrow.y, 20, Math.PI / 6,
(Math.PI * 11) / 6);
    ctxArrow.stroke();
    ctxArrow.closePath();



    ctxArrow.save();
    ctxArrow.translate(coordinateArrow.x-34+ Math.cos((Math.PI * 11) / 6) * (20
- 10), coordinateArrow.y + Math.sin((Math.PI * 11) / 6) * (20 - 10));
    ctxArrow.rotate(45);
    ctxArrow.beginPath();
    ctxArrow.moveTo(0, 0);
    ctxArrow.lineTo(-arrowheadSize, arrowheadSize / 2);
```

```
    ctxArrow.lineTo(-arrowheadSize, -arrowheadSize / 2);
    ctxArrow.closePath();
    ctxArrow.fill();
    ctxArrow.restore();
    ctxArrow.fillStyle = 'black';
    ctxArrow.beginPath();
    ctxArrow.arc(coordinateArrow.x-45, coordinateArrow.y , 20, Math.PI / 6,
Math.PI * 11 / 6);
    ctxArrow.stroke();
    ctxArrow.closePath();
}

function drawEdgeLineArrow(start, end, bendAngle = Math.PI / 8, arrowDistance =
20) {
    const arrowSize = 15;

    const midXArrow = (start.x + end.x) / 2;
    const midYArrow = (start.y + end.y) / 2;

    let controlX, controlY;

    if (start.x !== end.x && start.y !== end.y) {
        controlX = midXArrow + Math.cos(bendAngle) * (midYArrow - start.y);
        controlY = midYArrow + Math.sin(bendAngle) * (midXArrow - start.x);
    } else if (start.x === end.x) {
        controlX = midXArrow + 100;
        controlY = midYArrow;
    } else {
        controlX = midXArrow;
        controlY = midYArrow + 100;
    }


    ctxArrow.beginPath();
    ctxArrow.moveTo(start.x, start.y);
    ctxArrow.quadraticCurveTo(controlX, controlY, end.x, end.y);
    ctxArrow.stroke();
    ctxArrow.closePath();


    const angle = Math.atan2(end.y - controlY, end.x - controlX);


    const newEndX = end.x - arrowDistance * Math.cos(angle);
    const newEndY = end.y - arrowDistance * Math.sin(angle);


    ctxArrow.save();
    ctxArrow.translate(newEndX, newEndY);
    ctxArrow.rotate(angle);
    ctxArrow.fillStyle = 'black';
    ctxArrow.beginPath();
    ctxArrow.moveTo(0, 0);
    ctxArrow.lineTo(-arrowSize, arrowSize / 2);
    ctxArrow.lineTo(-arrowSize, -arrowSize / 2);
    ctxArrow.closePath();
    ctxArrow.fill();
    ctxArrow.restore();
}

function chooseLineArrow(x1, y1, x2, y2) {
    if (checkFunctionArrow({ x: x1, y: y1 }, { x: x2, y: y2 })) {
        drawEdgeLineArrow({ x: x1, y: y1 }, { x: x2, y: y2 });
    } else {
        ctxArrow.beginPath();
        ctxArrow.moveTo(x1, y1);
```

```
        ctxArrow.lineTo(x2, y2);
        ctxArrow.stroke();
        ctxArrow.closePath();


        const angle = Math.atan2(y2 - y1, x2 - x1);


        const arrowSize = 12;
        const gapX = Math.cos(angle) * 10;
        const gapY = Math.sin(angle) * 10;


        ctxArrow.beginPath();
        ctxArrow.fillStyle = 'black';
        ctxArrow.moveTo(x2 - gapX, y2 - gapY);
        ctxArrow.lineTo(
            x2 - arrowSize * Math.cos(angle - Math.PI / 6) - gapX,
            y2 - arrowSize * Math.sin(angle - Math.PI / 6) - gapY
        );
        ctxArrow.lineTo(
            x2 - arrowSize * Math.cos(angle + Math.PI / 6) - gapX,
            y2 - arrowSize * Math.sin(angle + Math.PI / 6) - gapY
        );
        ctxArrow.closePath();
        ctxArrow.fill();
    }
}

function checkFunctionArrow(start, end) {
    const [A, B, C] = arrOfThreeNodesArrow;

    function isPointOnLineSegmentArrow(P, A, B) {
        const crossProductArrow = (P.y - A.y) * (B.x - A.x) - (P.x - A.x) * (B.y
- A.y);
        if (Math.abs(crossProductArrow) !== 0) return false;

        const dotProductArrow = (P.x - A.x) * (B.x - A.x) + (P.y - A.y) * (B.y -
A.y);
        if (dotProductArrow < 0) return false;

        const squaredLengthBA = (B.x - A.x) ** 2 + (B.y - A.y) ** 2;
        if (dotProductArrow > squaredLengthBA) return false;

        return true;
    }

    function isOnSameEdgeArrow(P1, P2, A, B) {
        return isPointOnLineSegmentArrow(P1, A, B) &&
isPointOnLineSegmentArrow(P2, A, B);
    }

    return (
        isOnSameEdgeArrow(start, end, A, B) ||
        isOnSameEdgeArrow(start, end, B, C) ||
        isOnSameEdgeArrow(start, end, C, A)
    );
}
drawEdgesArrow();
```

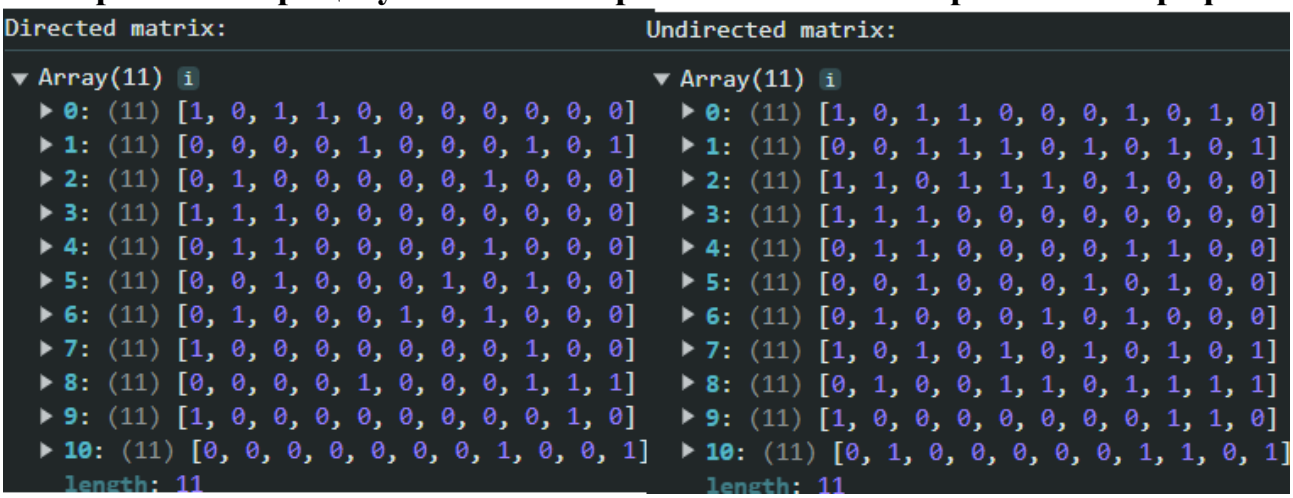
```
matrixDirected=[];
coefficient=1 - n3 * 0.005 - n4 * 0.005 - 0.27;
for (let i = 0; i < numTops; i++) {
    let row = [];
    for (let j = 0; j < numTops; j++) {
        let elem = (Math.random() * 2) * coefficient;
        row.push(Math.floor(elem));
    }
    matrixDirected.push(row);
}

console.log("Directed matrix 2:");
console.log(matrixDirected);
canvasArrow = document.getElementById('graphCanvasWithArrows2');
ctxArrow = canvasArrow.getContext('2d');
ctxArrow.strokeStyle='black';
transformCoordinateAreaArrow();
createVertexArrow();
drawEdgesArrow();


inDegrees= new Array(numTops).fill(0);
outDegrees= new Array(numTops).fill(0);
for (let i = 0; i < numTops; i++) {
    for (let j = 0; j < numTops; j++) {
        outDegrees[i] += matrixDirected[i][j];
        inDegrees[j] += matrixDirected[i][j];
    }
}

console.log("In-degrees (Directed):", inDegrees);
console.log("Out-degrees (Directed):", outDegrees);

// Знаходження шляхів довжини 2 та 3
let connectLength2 = [];
let connectLength3 = [];
for (let i = 0; i < numTops; i++) {
    for (let j = 0; j < numTops; j++) {
        if (matrixDirected[i][j] === 1 && i !== j) {
            connectLength2.push(`${i + 1} - ${j + 1}`);
            for (let q = 0; q < numTops; q++) {
                if (matrixDirected[j][q] === 1 && q !== i && q !== j) {
                    connectLength3.push(`${i + 1} - ${j + 1} - ${q + 1}`);
                }
            }
        }
    }
}
console.log(`Connection length equal 2:(${connectLength2})`);
console.log(`Connection length equal 3:(${connectLength3})`);

// Функція перевірки доступності вершини
const checkAccess = (i, j, visited = new Set()) => {
    if (matrixDirected[i][j] === 1) return true;
    visited.add(i);
    return [...Array(numTops).keys()].some(q =>
        matrixDirected[i][q] === 1 && !visited.has(q) && checkAccess(q, j,
visited));
};

// Створення матриці досяжності
let matrixAccess = Array.from({ length: numTops }, (_, i) =>
    Array.from({ length: numTops }, (_, j) => checkAccess(i, j) ? 1 : 0));

console.log('Access matrix:');
console.log(matrixAccess);
```

```
// Функція для друку матриці
function printMatrix(matrix) {
    matrix.forEach(row => console.log(row.join(' ')));
}

// Знаходження матриці сильної зв'язності та компонент сильної зв'язності
const findStrongConnectivity = (matrix) => {
    let stack = [];
    let visited = new Array(numTops).fill(false);
    const transposedMatrix = matrix[0].map((_, colIndex) => matrix.map(row =>
row[colIndex]));

    const fillOrder = (v) => {
        visited[v] = true;
        matrix[v].forEach((edge, i) => !visited[i] && edge && fillOrder(i));
        stack.push(v);
    };

    const dfs = (v, component) => {
        visited[v] = true;
        component.push(v);
        transposedMatrix[v].forEach((edge, i) => !visited[i] && edge && dfs(i,
component));
    };

    for (let i = 0; i < numTops; i++) {
        if (!visited[i]) fillOrder(i);
    }

    visited.fill(false);
    let components = [];
    while (stack.length) {
        let v = stack.pop();
        if (!visited[v]) {
            let component = [];
            dfs(v, component);
            components.push(component);
        }
    }

    let connectivityMatrix = Array.from({ length: numTops }, () =>
Array(numTops).fill(0));
    components.forEach(component => {
        component.forEach(v => component.forEach(n => matrix[v][n] &&
(connectivityMatrix[v][n] = 1)));
    });

    return { connectivityMatrix, components };
};

let { connectivityMatrix, components } = findStrongConnectivity(matrixDirected);
console.log("\nMatrix of strong connection:");
printMatrix(connectivityMatrix);

console.log("\nStrongly connected components:");
components.forEach((component, index) => {
    console.log(`Component ${index + 1}: ${component.join(', ')}`);
});

const canvasCond=document.getElementById("graphCanvasCondensation");
const ctxCond=canvasCond.getContext('2d');
ctxCond.strokeStyle = 'blue';
ctxCond.lineWidth = 2;
canvasCond.width = width;
canvasCond.height = height;
```

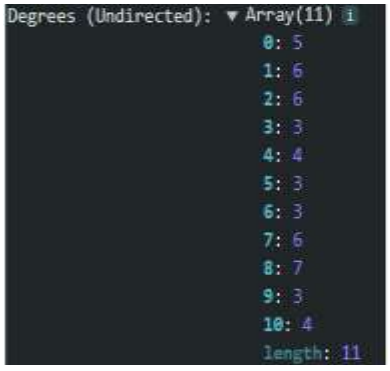
```
function transformCoordinateAreaCond() {
    ctxCond.translate(width/ 2, height/ 1.25);

}
transformCoordinateAreaCond();

function buildCondensationGraph(matrix, components) {
    const condensationGraph = [];
    const componentMap = new Map(); // Відображення між вершинами графа і
компонентами

    // Створення вершин для кожної компоненти
    components.forEach((component, index) => {
        const componentVertex = `C${index + 1}`;
        condensationGraph.push([]);
        componentMap.set(componentVertex, component);
    });

    // Додавання ребер між компонентами
    components.forEach((component1, i) => {
        components.forEach((component2, j) => {
            if (i !== j) {
                const isConnected = component1.some(vertex1 =>
                    component2.some(vertex2 => matrix[vertex1][vertex2])
                );
                // Додавання ребра, якщо компоненти зв'язані
                if (isConnected) {
                    condensationGraph[i].push(j);
                }
            }
        });
    });

    return { graph: condensationGraph, componentMap };
}


// Використання функції побудови графа конденсації
let condensationGraphData = buildCondensationGraph(matrixDirected, components);
let condensationGraph = condensationGraphData.graph;
let componentMap = condensationGraphData.componentMap;


console.log("\nCondensation graph:");
// Вивід зв'язків між компонентами
condensationGraph.forEach((neighbors, index) => {
    const componentVertex = `C${index + 1}`;
    const componentVertices = componentMap.get(componentVertex);
    const neighborComponents = neighbors.map(neighborIndex => `C${neighborIndex
+ 1}`);
    console.log(`${componentVertex}: ${componentVertices.join(', ')} ->
${neighborComponents.join(', ')}`);
});

// Малювання ребер між компонентами
const drawEdgesCond = (ctx, nodes) => (neighbors, index) => {
    const startPoint = nodes[index];
    neighbors.forEach(neighborIndex => {
        const endPoint = nodes[neighborIndex];
        ctx.beginPath();
        ctx.moveTo(startPoint.x, startPoint.y);
        ctx.lineTo(endPoint.x, endPoint.y);
        ctx.stroke();
    });
};
```

```
condensationGraph.forEach(drawEdgesCond(ctxCond, arrOfNode2));

// Малювання вершин компонент
const drawVertices = (ctx, nodes) => (_, index) => {
    const componentVertex = `C${index + 1}`;
    const point = nodes[index];
    ctx.fillStyle = 'pink';
    ctx.strokeStyle = 'black';
    ctx.lineWidth = 1;
    ctx.beginPath();
    ctx.arc(point.x, point.y, 20, 0, Math.PI * 2);
    ctx.fill();
    ctx.stroke();
    ctx.closePath();
    ctx.fillStyle = 'white';
    ctx.font = '16px Arial';
    ctx.textAlign = 'center';
    ctx.textBaseline = 'middle';
    ctx.fillText(componentVertex, point.x, point.y);
    ctx.fillStyle = 'pink';
};

condensationGraph.forEach(drawVertices(ctxCond, arrOfNode2));
```

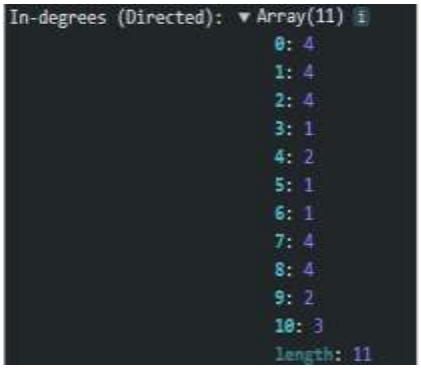
Згенеровані матриці суміжності напрямленого та ненапрямленого графів



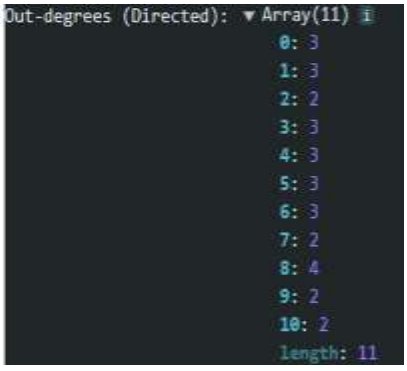
Перелік степенів, півстепенів, результат перевірки на однорідність, переліки висячих та ізольованих вершин.



Степені



Півстепені входу



Півстепені виходу

```
Is Directed Graph Regular: false with degree: null
Is Undirected Graph Regular: false with degree: null
```

Перевірка на однорідність

Матриця модифікованого графа

```
Directed matrix 2:
▼ Array(11) i
  ▶ 0: (11) [1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0]
  ▶ 1: (11) [1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0]
  ▶ 2: (11) [0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0]
  ▶ 3: (11) [0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0]
  ▶ 4: (11) [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
  ▶ 5: (11) [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
  ▶ 6: (11) [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
  ▶ 7: (11) [0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0]
  ▶ 8: (11) [0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0]
  ▶ 9: (11) [1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0]
  ▶ 10: (11) [0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1]
    length: 11
```

Connection length equal 2:(1 - 3,1 - 4,1 - 5,1 - 6,1 - 8,2 - 1,2 - 3,2 - 9,3 - 5,3 - 9,3 - 10,4 - 3,4 - 5,4 - 7,4 - 8,4 - 9,4 - 10,5 - 2,6 - 4,8 - 2,8 - 4,8 - 9,9 - 3,9 - 7,10 - 1,10 - 4,10 - 5,10 - 6,10 - 7,10 - 8,11 - 5,11 - 8) – **Шляхи довжиною 2**

Connection length equal 3:(1 - 3 - 5,1 - 3 - 9,1 - 3 - 10,1 - 4 - 3,1 - 4 - 5,1 - 4 - 7,1 - 4 - 8,1 - 4 - 9,1 - 4 - 10,1 - 5 - 2,1 - 6 - 4,1 - 8 - 2,1 - 8 - 4,1 - 8 - 9,2 - 1 - 3,2 - 1 - 4,2 - 1 - 5,2 - 1 - 6,2 - 1 - 8,2 - 3 - 5,2 - 3 - 9,2 - 3 - 10,2 - 9 - 3,2 - 9 - 7,3 - 5 - 2,3 - 9 - 7,3 - 10 - 1,3 - 10 - 4,3 - 10 - 5,3 - 10 - 6,3 - 10 - 7,3 - 10 - 8,4 - 3 - 5,4 - 3 - 9,4 - 3 - 10,4 - 5 - 2,4 - 8 - 2,4 - 8 - 9,4 - 9 - 3,4 - 9 - 7,4 - 10 - 1,4 - 10 - 5,4 - 10 - 6,4 - 10 - 7,4 - 10 - 8,5 - 2 - 1,5 - 2 - 3,5 - 2 - 9,6 - 4 - 3,6 - 4 - 5,6 - 4 - 7,6 - 4 - 8,6 - 4 - 9,6 - 4 - 10,8 - 2 - 1,8 - 2 - 3,8 - 2 - 9,8 - 4 - 3,8 - 4 - 5,8 - 4 - 7,8 - 4 - 9,8 - 4 - 10,8 - 9 - 3,8 - 9 - 7,9 - 3 - 5,9 - 3 - 10,10 - 1 - 3,10 - 1 - 4,10 - 1 - 5,10 - 1 - 6,10 - 1 - 8,10 - 4 - 3,10 - 4 - 5,10 - 4 - 7,10 - 4 - 8,10 - 4 - 9,10 - 5 - 2,10 - 6 - 4,10 - 8 - 2,10 - 8 - 4,10 - 8 - 9,11 - 5 - 2,11 - 8 - 2,11 - 8 - 4,11 - 8 - 9) – **Шляхи довжиною 3**

```
Access matrix:
▼ Array(11) i
  ▶ 0: (11) [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
  ▶ 1: (11) [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
  ▶ 2: (11) [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
  ▶ 3: (11) [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
  ▶ 4: (11) [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
  ▶ 5: (11) [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
  ▶ 6: (11) [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
  ▶ 7: (11) [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
  ▶ 8: (11) [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
  ▶ 9: (11) [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
  ▶ 10: (11) [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
    length: 11
```

Матриця доступності

```
Matrix of strong connection:
1 0 1 1 1 1 0 1 0 0 0
1 1 1 0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0 1 1 0
0 0 1 0 1 0 0 1 1 1 0
0 1 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 0 0 0 0
0 1 0 1 0 0 0 1 1 0 0
0 0 1 0 0 0 0 0 0 0 0
1 0 0 1 1 1 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 1
```

Матриця сильної зв’язності

```
In-degrees (Directed): ▼ Array(11) i
                         0: 3
                         1: 3
                         2: 4
                         3: 4
                         4: 5
                         5: 2
                         6: 3
                         7: 5
                         8: 4
                         9: 2
                         10: 1
                         length: 11
```

Півстпені входу

```
Out-degrees (Directed): ▼ Array(11) i
                         0: 6
                         1: 4
                         2: 3
                         3: 6
                         4: 1
                         5: 1
                         6: 0
                         7: 4
                         8: 2
                         9: 6
                         10: 3
                         length: 11
```

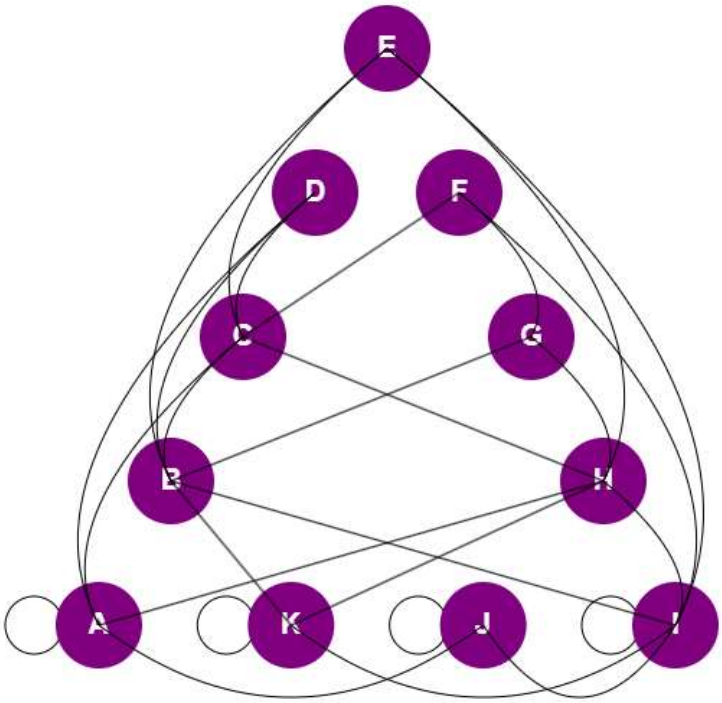
Півстепені виходу

```
Strongly connected components:
Component 1: 10
Component 2: 0, 1, 4, 2, 3, 5, 9, 7, 8
Component 3: 6
```

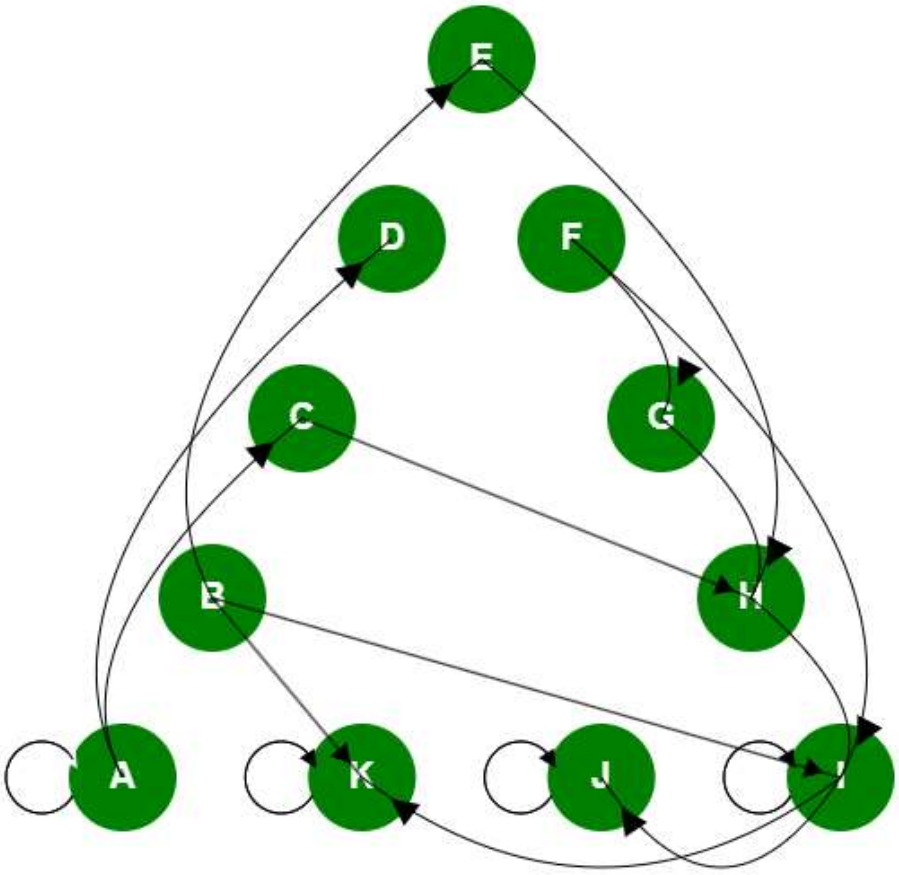
Компоненти сильної зв’язності

```
Condensation graph:
C1: 10 -> C2
C2: 0, 1, 4, 2, 3, 5, 9, 7, 8 -> C3
C3: 6 ->
```

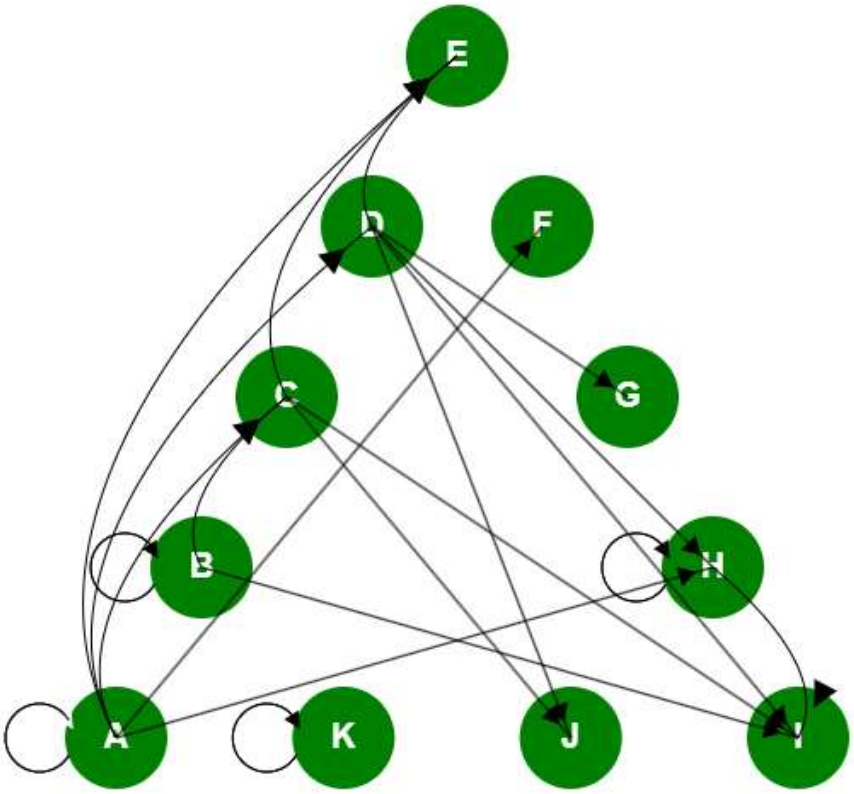
Граф конденсації



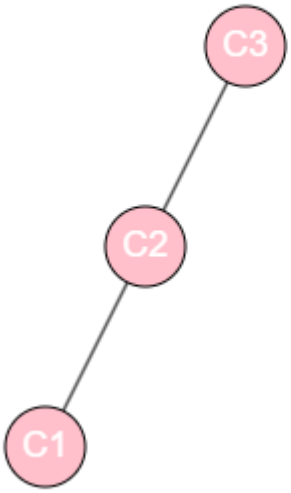
Ненапрямлений граф



Напрямлений граф



Модифікований граф



Граф конденсації