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 №5

з дисципліни
«Алгоритми і структури даних»

Виконала:
студентка групи ІМ-41
Куц Анна Василівна
номер у списку групи: 11

Перевірив:
Сергієнко А.М.

Київ 2025

Постановка задачі

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

   Відмінність: коефіцієнт $k = 1.0 - n_3 * 0.01 - n_4 * 0.005 - 0.15$.

   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 $k = 1.0 - n_3 * 0.01 - n_4 * 0.005 - 0.15$, 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. Створити програму, яка виконує обхід напрямленого графа вшир (BFS) та вглиб (DFS).

   - у програмі виконання обходу відображати покроково, черговий крок виконувати за натисканням кнопки у вікні або на клавіатурі.

3. Під час обходу графа побудувати дерево обходу. У програмі дерево обходу виводити покроково у процесі виконання обходу графа. Це можна виконати одним із двох способів:

   - або виділяти іншим кольором ребра графа;
   - або будувати дерево обходу поряд із графом.

4. Зміну статусів вершин у процесі обходу продемонструвати зміною кольорів вершин, графічними позначками тощо, або ж у процесі обходу виводити протокол обходу у графічне вікно або в консоль.

5. Якщо після обходу графа лишилися невідвідані вершини, продовжувати обхід з невідвіданої вершини з найменшим номером, яка має щонайменше одну вихідну дугу.

Варіант №4111
$n_1n_2n_3n_4$ = 4111
k = 1.0 - n3 * 0.01 - n4 * 0.005 - 0.15 = 0.835
Розміщення колом. Кількість вершин = 10 + n_3 = 11
Оскільки, в обраній мові JavaScript не існує srand(seed), можна використати алгоритм генератора Парка Міллера для отримання псевдовипадкового числа. Алгоритм було створено мовою програмування JavaScript з використанням математичних бібліотек та вбудовану графічну бібліотеку <canvas>, для відображення використовувався HTML та CSS.

Текст програми
index. html

```
       <!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <title>Graph</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
<h1>Variant 4111🎀</h1>

<div class="button-container">

    <button id="btnDirected">Directed</button>
    <button id="btnBFS">BFS</button>
    <button id="btnDFS">DFS</button>
    <button id="btnNextStep">Next</button>
    <button id="btnReset">Reset</button>

</div>
<canvas id="canvas" width="800" height="800"></canvas>

<script src="script.js"></script>
</body>
```

styles.css

```
 body {
    font-family: "Times New Roman";
    display: flex;
    flex-direction: column;
    align-items: center;
    background: #ffe6f2;
}

canvas {
    border: 1px solid #ccc;
```

```
    background: #fff;
    margin-top: 1rem;
}

h1 {
    color: hotpink;
}

button {
    margin: 0.5rem;
    padding: 0.5rem 1rem;
    font-size: 16px;
    cursor: pointer;
    background: hotpink;
    border: none;
    border-radius: 10px;
    color: white;
    font-weight: bold;
    transition: background 0.2s, transform 0.2s;
}

button:hover {
    background: #ff69b4;
    transform: scale(1.05);
}
```

script.js

```
document.addEventListener("DOMContentLoaded", () => {
    const canvas = document.getElementById("canvas");
    const ctx = canvas.getContext("2d");

    const seed = 4111;
    const n3 = 1;
    const n4 = 1;
    const n = 10 + n3;
    const k = 1.0 - n3 * 0.01 - n4 * 0.005 - 0.15;

    const w = canvas.width;
    const h = canvas.height;
    const RAD = 15;
    const centerX = w * 0.3;
    const centerY = h * 0.5;
    const radius = 180;

    let traversalQueue = [];
    let traversalStack = [];
    let visited = Array(n).fill("unvisited");
    let traversalTree = [];
    let traversalMode = null;
    let nodesInTree = new Set();
    let order = [];
```

```
    const COLORS = {
        edge: {
            normal: "#333",
            tree: "hotpink",
            highlight: "#FF5722"
        },
        node: {
            normal: "#fff",
            visited: "hotpink",
            discovered: "#FFC107",
            current: "#FF5722"
        }
    };

    function genRand(seed) {
        const MOD = 2147483647;
        let val = seed % MOD;
        return () => {
            val = (val * 16807) % MOD;
            return (val - 1) / MOD;
        };
    }

    const rand = genRand(seed);

    function genDirMatrix(k) {
        const raw = Array.from({length: n}, () =>
            Array.from({length: n}, () => rand() * 2)
        );
        const dir = raw.map(row => row.map(v => (v * k >= 1 ? 1 :
0)));

        for (let i = 0; i < n; i++) {
            for (let j = i + 1; j < n; j++) {
                if (dir[i][j] && dir[j][i]) {
                    if (rand() < 0.5) dir[i][j] = 0; else dir[j][i]
= 0;
                }
            }
        }
        return dir;
    }

    const positions = Array.from({length: n}, (_, i) => {
        const angle = (2 * Math.PI * i) / n - Math.PI / 2;
        return {
            x: centerX + radius * Math.cos(angle),
            y: centerY + radius * Math.sin(angle)
        };
    });

    function printMatrix(matrix, title) {
        console.log(`\n${title}:`);
```

```
        matrix.forEach(row => {
            console.log(row.join(" "));
        });
    }

    function distanceToLine(p1, p2, p) {
        const A = p.x - p1.x, B = p.y - p1.y;
        const C = p2.x - p1.x, D = p2.y - p1.y;
        const dot = A * C + B * D;
        const len2 = C * C + D * D;
        const param = dot / len2;
        const clamp = Math.max(0, Math.min(1, param));
        const xx = p1.x + clamp * C;
        const yy = p1.y + clamp * D;
        return Math.hypot(p.x - xx, p.y - yy);
    }

    function drawArrow(p1, p2, cp = null, color =
COLORS.edge.normal) {
        ctx.strokeStyle = color;
        ctx.fillStyle = color;
        let end = {...p2};
        let angle;

        if (cp) {
            const t = 0.85;
            const bx = (1 - t) ** 2 * p1.x + 2 * (1 - t) * t * cp.x
+ t ** 2 * p2.x;
            const by = (1 - t) ** 2 * p1.y + 2 * (1 - t) * t * cp.y
+ t ** 2 * p2.y;
            const dx = 2 * (1 - t) * (cp.x - p1.x) + 2 * t * (p2.x
- cp.x);
            const dy = 2 * (1 - t) * (cp.y - p1.y) + 2 * t * (p2.y
- cp.y);
            angle = Math.atan2(dy, dx);
            end = {x: bx, y: by};
        } else {
            const dx = p2.x - p1.x;
            const dy = p2.y - p1.y;
            angle = Math.atan2(dy, dx);
            end = {
                x: p2.x - RAD * Math.cos(angle),
                y: p2.y - RAD * Math.sin(angle)
            };
        }

        ctx.beginPath();
        ctx.moveTo(end.x, end.y);
        ctx.lineTo(end.x - 8 * Math.cos(angle - Math.PI / 8),
            end.y - 8 * Math.sin(angle - Math.PI / 8));
        ctx.lineTo(end.x - 8 * Math.cos(angle + Math.PI / 8),
            end.y - 8 * Math.sin(angle + Math.PI / 8));
        ctx.closePath();
        ctx.fill();
```

```
    }

    function drawSelfLoop(nodeX, nodeY, directed, color =
COLORS.edge.normal) {
        ctx.strokeStyle = color;
        ctx.fillStyle = color;
        const arcR = RAD * 0.85;
        const offset = RAD + 7.5;
        const dx = nodeX - centerX;
        const dy = nodeY - centerY;
        let theta = Math.atan2(dy, dx) * 180 / Math.PI;
        if (theta < 0) theta += 360;

        let cx, cy, start, end;
        if (theta >= 315 || theta < 45) {
            cx = nodeX + offset;
            cy = nodeY;
            start = -135;
            end = 135;
        } else if (theta < 135) {
            cx = nodeX;
            cy = nodeY + offset;
            start = 225;
            end = 135;
        } else if (theta < 225) {
            cx = nodeX - offset;
            cy = nodeY;
            start = 45;
            end = -45;
        } else {
            cx = nodeX;
            cy = nodeY - offset;
            start = 45;
            end = -45;
        }
        const s = start * Math.PI / 180;
        const e = end * Math.PI / 180;

        ctx.beginPath();
        ctx.arc(cx, cy, arcR, s, e, false);
        ctx.stroke();
        if (!directed) return;

        const ax = cx + arcR * Math.cos(e);
        const ay = cy + arcR * Math.sin(e);
        const ang = Math.atan2(nodeY - ay, nodeX - ax);
        const L = 0.55 * arcR;
        ctx.beginPath();
        ctx.moveTo(ax, ay);
        ctx.lineTo(ax - L * Math.cos(ang - Math.PI / 6), ay - L *
Math.sin(ang - Math.PI / 6));
        ctx.lineTo(ax - L * Math.cos(ang + Math.PI / 6), ay - L *
Math.sin(ang + Math.PI / 6));
        ctx.closePath();
```

```
        ctx.fill();
    }

    function updateNodesInTree() {
        nodesInTree.clear();
        traversalTree.forEach(edge => {
            nodesInTree.add(edge.from);
            nodesInTree.add(edge.to);
        });
    }

    function drawGraph(matrix, directed, highlightEdges = []) {
        updateNodesInTree();

        ctx.clearRect(0, 0, w, h);

        if (traversalTree.length) {
            drawTraversalTreeSeparate();
        }

        for (let i = 0; i < n; i++) {
            for (let j = 0; j < n; j++) {
                if (!matrix[i][j]) continue;
                if (!directed && j < i) continue;

                const isTreeEdge = traversalTree.some(e => e.from
=== i && e.to === j);
                const isHighlighted = highlightEdges.some(e =>
e.from === i && e.to === j);
                let edgeColor = COLORS.edge.normal;
                if (isTreeEdge) edgeColor = COLORS.edge.tree;
                if (isHighlighted) edgeColor =
COLORS.edge.highlight;

                const p1 = positions[i];
                const p2 = positions[j];

                if (i === j) {
                    drawSelfLoop(p1.x, p1.y, directed, edgeColor);
                    continue;
                }

                let curved = false, cp = null;
                for (let k2 = 0; k2 < n; k2++) {
                    if (k2 === i || k2 === j) continue;
                    if (distanceToLine(p1, p2, positions[k2]) < 25)
{
                        curved = true;
                        const mid = {x: (p1.x + p2.x) / 2, y: (p1.y
+ p2.y) / 2};
                        const perp = {x: -(p2.y - p1.y), y: p2.x -
p1.x};
                        const len = Math.hypot(perp.x, perp.y);
                        const sign = i < j ? 1 : -1;
```

```
                        cp = {
                            x: mid.x + sign * (perp.x / len) * 90,
                            y: mid.y + sign * (perp.y / len) * 90
                        };
                        break;
                    }
                }

                ctx.strokeStyle = edgeColor;
                ctx.beginPath();
                ctx.moveTo(p1.x, p1.y);
                curved ? ctx.quadraticCurveTo(cp.x, cp.y, p2.x,
p2.y)
                    : ctx.lineTo(p2.x, p2.y);
                ctx.stroke();

                if (directed) drawArrow(p1, p2, curved ? cp : null,
edgeColor);
            }
        }

        for (let i = 0; i < n; i++) {
            ctx.beginPath();

            const isInTree = nodesInTree.has(i);

            ctx.fillStyle = visited[i] === "current" ?
COLORS.node.current :
                visited[i] === "discovered" ?
COLORS.node.discovered :
                    (visited[i] === "visited" && isInTree) ?
COLORS.node.visited :
                        COLORS.node.normal;

            ctx.arc(positions[i].x, positions[i].y, RAD, 0, Math.PI
* 2);
            ctx.fill();

            if (visited[i] === "unvisited" || (visited[i] ===
"visited" && !isInTree)) {
                ctx.stroke();
            }

            const isDarkBackground = (visited[i] === "visited" &&
isInTree) ||
                visited[i] === "current";
            ctx.fillStyle = isDarkBackground ? "#fff" : "#000";

            ctx.font = "14px Times New Roman";
            ctx.textAlign = "center";
            ctx.textBaseline = "middle";
            ctx.fillText(String(i + 1), positions[i].x,
positions[i].y);
        }
```

```
    }

    function drawTraversalTreeSeparate() {
        const offsetX = w * 0.8;
        const spacingX = 60;
        const spacingY = 70;

        if (!traversalTree.length) return;
        const parent = {};
        traversalTree.forEach(e => parent[e.to] = e.from);
        const level = {};
        const roots = [...new Set(traversalTree.map(e =>
e.from).filter(v => !(v in parent)))];
        roots.forEach(r => level[r] = 0);
        const q = [...roots];
        while (q.length) {
            const v = q.shift();
            traversalTree.filter(e => e.from === v).forEach(e => {
                level[e.to] = level[v] + 1;
                q.push(e.to);
            });
        }
        const nodesByLevel = {};
        Object.entries(level).forEach(([node, lvl]) => {
            if (!nodesByLevel[lvl]) nodesByLevel[lvl] = [];
            nodesByLevel[lvl].push(+node);
        });
        const maxLevel = Math.max(...Object.values(level));
        const pos = {};
        Object.entries(nodesByLevel).forEach(([lvl, nodes]) => {
            const y = centerY - (maxLevel * spacingY / 2) +
spacingY * lvl;
            const totalW = (nodes.length - 1) * spacingX;
            nodes.forEach((node, idx) => {
                pos[node] = {
                    x: offsetX + idx * spacingX - totalW / 2,
                    y
                };
            });
        });
        ctx.strokeStyle = COLORS.edge.tree;
        traversalTree.forEach(e => {
            const from = pos[e.from];
            const to = pos[e.to];
            ctx.beginPath();
            ctx.moveTo(from.x, from.y);
            ctx.lineTo(to.x, to.y);
            ctx.stroke();
            drawArrow(from, to, null, COLORS.edge.tree);
        });
        ctx.font = "14px Times New Roman";
        Object.entries(pos).forEach(([node, p]) => {
            ctx.beginPath();
            ctx.fillStyle = COLORS.node.visited;
```

```
            ctx.arc(p.x, p.y, RAD, 0, Math.PI * 2);
            ctx.fill();
            ctx.fillStyle = "#fff";
            ctx.textAlign = "center";
            ctx.textBaseline = "middle";
            ctx.fillText(String(+node + 1), p.x, p.y);
        });
    }

    function createTreeMatrix() {
        const treeMatrix = Array.from({length: n}, () =>
Array(n).fill(0));
        traversalTree.forEach(edge => {
            treeMatrix[edge.from][edge.to] = 1;
        });
        return treeMatrix;
    }

    function logTraversalState() {
        if (traversalTree.length) {
            console.clear();
            printMatrix(dirMatrix, `Directed matrix of
${traversalMode}`);
            const start = traversalTree[0].from + 1;
            console.log(`\nStarting ${traversalMode} from vertex
${start}`);
            console.log("\nTree Edges:");
            traversalTree.forEach(e => {
                console.log(`${e.from + 1} -> ${e.to + 1}`);
            });
        }
    }

    function findStartVertex(matrix) {
        for (let i = 0; i < n; i++) if (matrix[i].some(v => v ===
1)) return i;
        return 0;
    }

    function resetTraversal() {
        traversalQueue = [];
        traversalStack = [];
        visited = Array(n).fill("unvisited");
        traversalTree = [];
        nodesInTree.clear();
        traversalMode = null;
        order = [];
        console.clear();
        console.log("Traversal reset");
        btnNextStep.disabled = true;
    }

    function initBFS(matrix) {
        resetTraversal();
```

```
        console.clear();
        traversalMode = "BFS";
        const start = findStartVertex(matrix);
        printMatrix(matrix, "Directed matrix of BFS");
        console.log(`\nStarting BFS from vertex ${start + 1}`);
        visited[start] = "discovered";
        traversalQueue.push(start);
        drawGraph(matrix, true);
        btnNextStep.disabled = false;
    }

    function initDFS(matrix) {
        resetTraversal();
        console.clear();
        traversalMode = "DFS";
        const start = findStartVertex(matrix);
        printMatrix(matrix, "Directed matrix of DFS");
        console.log(`\nStarting DFS from vertex ${start + 1}`);
        visited[start] = "discovered";
        traversalStack.push(start);
        drawGraph(matrix, true);
        btnNextStep.disabled = false;
    }

    function performBFSStep(matrix) {
        if (!traversalQueue.length) {
            const next = visited.findIndex((v, idx) => v ===
"unvisited" && matrix[idx].some(x => x));
            if (next === -1) {
                btnNextStep.disabled = true;
                drawGraph(matrix, true);

                const treeMatrix = createTreeMatrix();
                printMatrix(treeMatrix, 'Tree matrix of BFS');
                console.log("\nOrder of BFS:", order.map(v => v +
1).join(" → "));
                return;
            }
            visited[next] = "discovered";
            traversalQueue.push(next);
            drawGraph(matrix, true);
            return;
        }

        const v = traversalQueue.shift();
        visited[v] = "current";

        const edges = [];
        for (let i = 0; i < n; i++) {
            if (matrix[v][i] && visited[i] === "unvisited") {
                visited[i] = "discovered";
                traversalQueue.push(i);
                traversalTree.push({from: v, to: i});
                edges.push({from: v, to: i});
```

```
            }
        }
        visited[v] = "visited";
        order.push(v);
        logTraversalState();
        drawGraph(matrix, true, edges);
    }

    function performDFSStep(matrix) {
        if (!traversalStack.length) {
            const next = visited.findIndex((v, idx) => v ===
"unvisited" && matrix[idx].some(x => x));
            if (next === -1) {
                btnNextStep.disabled = true;
                drawGraph(matrix, true);
                printMatrix(createTreeMatrix(), "Tree matrix of
DFS");
                console.log("\nOrder of DFS:", order.map(v => v +
1).join(" → "));
                return;
            }
            visited[next] = "discovered";
            traversalStack.push(next);
            drawGraph(matrix, true);
            return;
        }

        const v = traversalStack.pop();
        visited[v] = "current";

        for (let i = n - 1; i >= 0; i--) {
            if (matrix[v][i] && visited[i] === "unvisited") {
                traversalStack.push(v);
                visited[i] = "discovered";
                traversalStack.push(i);
                traversalTree.push({from: v, to: i});
                drawGraph(matrix, true, [{from: v, to: i}]);
                return;
            }
        }

        visited[v] = "visited";
        order.push(v);
        logTraversalState();
        drawGraph(matrix, true);
    }


    const dirMatrix = genDirMatrix(k);

    const btnDirected = document.getElementById("btnDirected");
    const btnBFS = document.getElementById("btnBFS");
    const btnDFS = document.getElementById("btnDFS");
    const btnNextStep = document.getElementById("btnNextStep");
```

```
    const btnReset = document.getElementById("btnReset");

    btnDirected.onclick = () => {
        console.clear();
        resetTraversal();
        printMatrix(dirMatrix, "Directed matrix (Adir)");
        drawGraph(dirMatrix, true);
        btnNextStep.disabled = true;
    };

    btnBFS.onclick = () => initBFS(dirMatrix);
    btnDFS.onclick = () => initDFS(dirMatrix);

    btnNextStep.onclick = () => {
        if (traversalMode === "BFS") {
            performBFSStep(dirMatrix);
        } else if (traversalMode === "DFS") {
            performDFSStep(dirMatrix);
        } else {
            console.log("Please select algorithm first (BFS or
DFS)");
            btnNextStep.disabled = true;
        }
    };

    btnReset.onclick = () => {
        resetTraversal();
        drawGraph(dirMatrix, true);
    };

    console.clear();
    printMatrix(dirMatrix, "Directed matrix (Adir)");
    drawGraph(dirMatrix, true);
    btnNextStep.disabled = true;
});
```

Результати тестування програм

```
Directed matrix (Adir):
0 1 0 0 0 0 0 1 0 0 0
0 0 0 1 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 1 0 0 0 0
0 1 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 1 0 0
0 0 0 0 0 0 0 0 0 0 0
1 1 1 0 0 0 0 1 0 0 0
1 1 1 1 0 1 0 1 1 0 0
0 0 1 1 1 0 1 1 0 0 0
```

Згенерована матриця суміжності

```
Tree matrix of DFS:
0 1 0 0 0 0 0 1 0 0 0
0 0 0 1 0 0 1 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 0 0 0
0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 1 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 0 0 0 0 0
```

Матриця суміжності дерева обходу вглиб(DFS)

```
Tree matrix of BFS:
0 1 0 0 0 0 0 1 0 0 0
0 0 0 1 0 0 1 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 1 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 1 0 0
0 0 0 0 0 0 0 0 0 0 0
```

Матриця суміжності дерева обходу вшир(BFS)

```
Order of DFS: 8 → 3 → 9 → 7 → 5 → 4 → 2 → 1 → 11 → 6 → 10
```

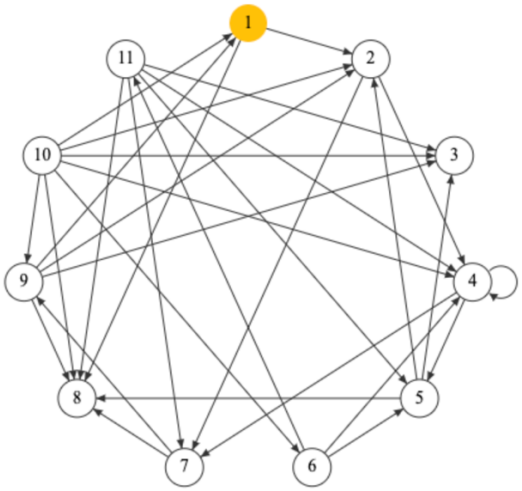
Список номерів вершин після обходу вглиб(DFS)

```
Order of BFS: 1 → 2 → 8 → 4 → 7 → 5 → 9 → 3 → 6 → 11 → 10
```

Список номерів вершин після обходу вшир(BFS)

Variant 4111🎀

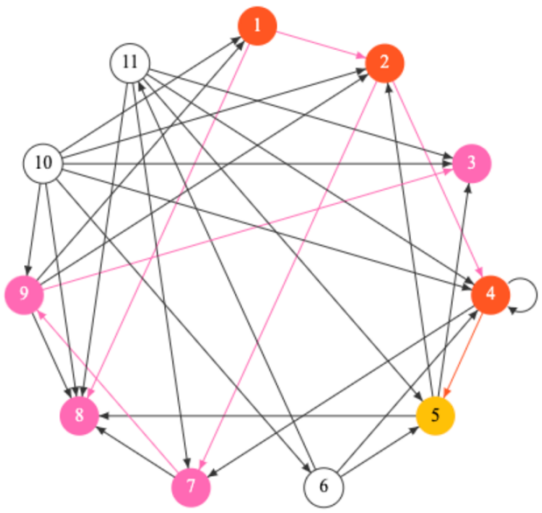
Directed | BFS | DFS | Next | Reset

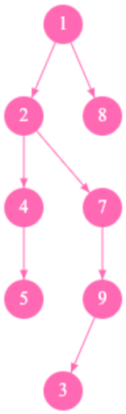



```
Directed matrix of DFS:
0 1 0 0 0 0 0 1 0 0 0
0 0 0 1 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 1 0 0 0 0
0 1 1 0 0 0 0 1 0 0 0
0 0 0 1 1 0 0 0 0 0 1
0 0 0 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 0 0 0 0 1 0 0 0
1 1 1 1 0 1 0 1 1 0 0
0 0 1 1 1 0 1 1 0 0 0

Starting DFS from vertex 1
```

Variant 4111🎀

Directed | BFS | DFS | Next | Reset

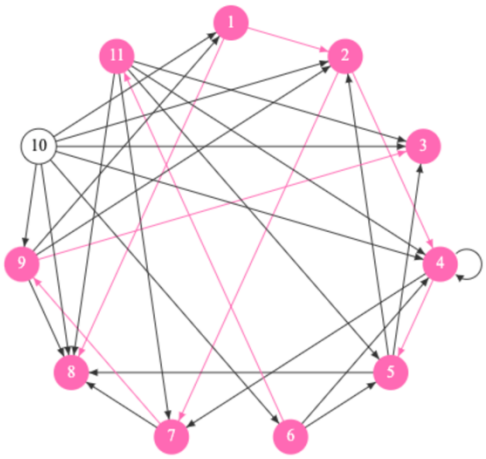


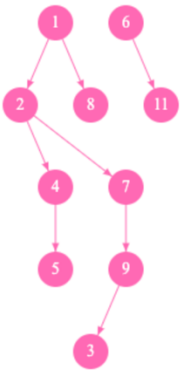



```
Directed matrix of DFS:
0 1 0 0 0 0 0 1 0 0 0
0 0 0 1 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 1 0 0 0 0
0 1 1 0 0 0 0 1 0 0 0
0 0 0 1 1 0 0 0 0 0 1
0 0 0 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 0 0 0 0 1 0 0 0
1 1 1 1 0 1 0 1 1 0 0
0 0 1 1 1 0 1 1 0 0 0

Starting DFS from vertex 1

Tree Edges:
1 -> 8
1 -> 2
2 -> 7
7 -> 9
9 -> 3
```

Variant 4111🎀

Directed | BFS | DFS | Next | Reset



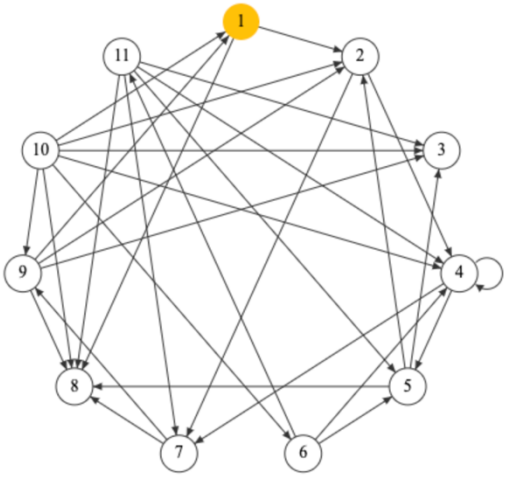



```
Starting DFS from vertex 1

Tree Edges:
1 -> 8
1 -> 2
2 -> 7
7 -> 9
9 -> 3
2 -> 4
4 -> 5
6 -> 11

Tree matrix of DFS:
0 1 0 0 0 0 0 1 0 0 0
0 0 0 1 0 0 1 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 0 0 0
0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 1 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 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0

Order of DFS: 8 → 3 → 9 → 7 → 5 → 4 → 2 → 1 → 11 → 6 → 10
```

DFS

Variant 4111🎀

Directed | BFS | DFS | Next | Reset

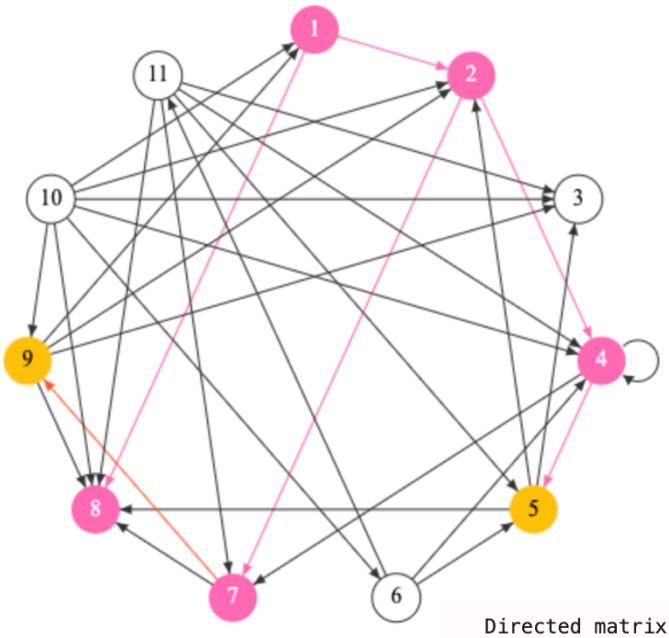



```
Directed matrix of BFS:
0 1 0 0 0 0 0 1 0 0 0
0 0 0 1 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 1 0 0 0 0
0 1 1 0 0 0 0 1 0 0 0
0 0 0 1 1 0 0 0 0 0 1
0 0 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0
1 1 1 0 0 0 0 1 0 0 0
1 1 1 1 0 1 0 1 1 0 0
0 0 1 1 1 0 1 1 0 0 0

Starting BFS from vertex 1
```

>

Variant 4111🎀

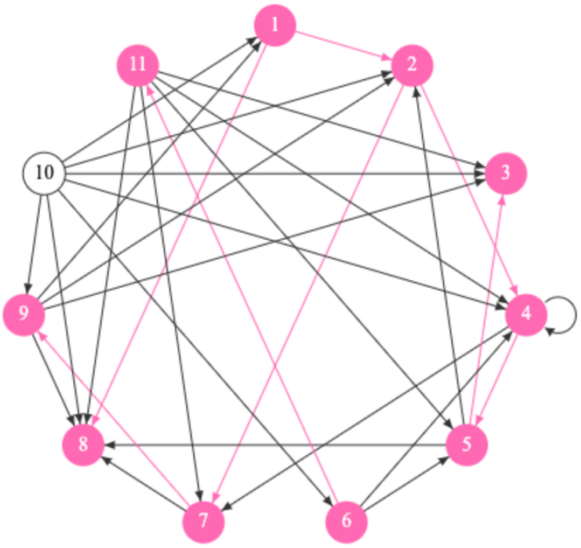
Directed | BFS | DFS | Next | Reset

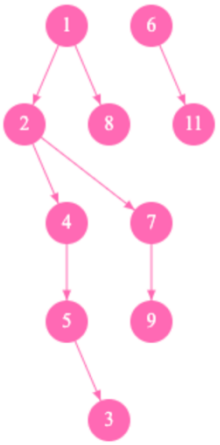



```
Directed matrix of BFS:
0 1 0 0 0 0 0 1 0 0 0
0 0 0 1 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 1 0 0 0 0
0 1 1 0 0 0 0 1 0 0 0
0 0 0 1 1 0 0 0 0 0 1
0 0 0 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0
1 1 1 0 0 0 0 1 0 0 0
1 1 1 1 0 1 0 1 1 0 0
0 0 1 1 1 0 1 1 0 0 0

Starting BFS from vertex 1

Tree Edges:
1 -> 2
1 -> 8
2 -> 4
2 -> 7
4 -> 5
7 -> 9
```

Variant 4111🎀

Directed | BFS | DFS | Next | Reset






```
Starting BFS from vertex 1

Tree Edges:
1 -> 2
1 -> 8
2 -> 4
2 -> 7
4 -> 5
7 -> 9
5 -> 3
6 -> 11

Tree matrix of BFS:
0 1 0 0 0 0 0 1 0 0 0
0 0 0 1 0 0 1 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 1 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 1 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0

Order of BFS: 1 → 2 → 8 → 4 → 7 → 5 → 9 → 3 → 6 → 11 → 10
```

BFS

Висновок:
Під час виконання лабораторної роботи №5 “Обхід графа” я вивчила метод дослідження графа за допомогою обходу його вершин в глибину та в ширину. Користувач має користувацькі інтерфейси як “Directed”, “BFS”, “DFS”, “Next”, “Reset”. При натисканні на “Directed” виводиться граф і матриця в консоль, згенерована заданим seed-числом. Код реалізує BFS і DFS через покрокові оновлення станів вершин (“unvisited”, “discovered”, “current”, “visited”) і дві структури — чергу для BFS і стек для DFS. У BFS стартова вершина стає “discovered” і додається в чергу, а потім із черги по черзі витягуються вузли, їхні невідвідані сусіди відкриваються, додаються в чергу, ребра записуються в traversalTree, і потім вузол стає “visited”. У DFS стартова вершина штовхається в стек, з якого витягуються вершини, знаходяться перші невідвідані сусіди (у зворотному порядку для коректної імітації рекурсії), штовхаються назад у стек для повернення, а після вичерпання сусідів вузол позначається “visited”. Після кожного кроку викликається drawGraph, щоб оновити візуалізацію на канві та логи в консолі. У консоль додається список оновлених вершин. А кнопка “Next” після натискання однієї з кнопок DFS чи BFS допомагає прослідкувати як малюється дерево. Я вирішила зробити і на графі, й паралельно малюючи дерево збоку, аби наочно показати, як працюють задані алгоритми. Оскільки при поточному параметрі k і seed у кожної вершини фактично лише один вихідний сусід, обидва алгоритми формують єдину лінійну гілку обходу без розгалужень, тож traversalTree виходить ідентичним для BFS і для DFS.