**Homework 5 (2-9/11/2023)**  
  
**Exercise**  
  
M servers are subject to attacks during a period of time T (for instance 1 year).  
Subdivide the interval T in N subinterval of size T/N and in each of this suppose that  
an attack can occur with probability λ T/N.  
Simulate the attacks to the M servers and represent each of them with a line which  
makes jumps of 1 at each attack event.  
  
Using the same objects ("movable/resizable rectangle", histogram, etc.) of the previous homework 3  
draw vertically on the line chart the 2 histograms representing the distribution of the number  
of attacks at the end of the period and one internal istant for comparison.  
  
Study what happens asymptotically, for N large, and a number of systems M a sufficient to give shape to  
a simulated distribution. Make some personal considerations about the shape and the average of the distributions that you see.

JS CODE

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Security Score Animation</title>

</head>

<body>

<canvas id="scoreCanvas" width="1920" height="1080" style="border: 2px solid black;"></canvas>

<script>

const canvas = document.getElementById("scoreCanvas");

const ctx = canvas.getContext("2d");

const lambda = 450;

const M = 500;

const T = 1;

const N = 500;

const numIntervals = N;

const intervalSize = T / numIntervals;

const intervals = [];

class Interval {

    constructor(upper, lower, count) {

        this.upper = upper;

        this.lower = lower;

        this.count = count;

    }

}

class Rectangle {

    constructor(x, y, width, height) {

        this.x = x;

        this.y = y;

        this.width = width;

        this.height = height;

    }

    left() {

        return this.x

    }

    top() {

        return this.y

    }

    right() {

        return this.x + this.width

    }

    bottom() {

        return this.y + this.height

    }

    aspectRatio() {

        return this.width / this.height || 1

    }

    drawRectangle(ctx, lineColor, lineWidth, lineDash) {

        ctx.save();

        ctx.beginPath();

        ctx.rect(this.x, this.y, this.width, this.height);

        ctx.strokeStyle = lineColor;

        ctx.lineWidth = lineWidth;

        ctx.setLineDash(lineDash);

        ctx.stroke();

        ctx.restore()

    }

}

const viewRect = new Rectangle(10, 10, 1000, 300);

const rects = [viewRect];

class Prof2dUtilities {

    static transformX(x, min\_x, rangeL\_x, left, width) {

        return left + width \* (x - min\_x) / rangeL\_x;

    }

    static transformY(y, min\_y, rangeL\_y, top, height) {

        return top + height - (height \* (y - min\_y) / rangeL\_y);

    }

}

class Histogram {

    static verticalHistoFromIntervals(ctx, intervals, y\_min, y\_range, viewRect, strokeStyle, lineWidth, fillStyle) {

        let maxcount = 0;

        for (const interval of intervals) {

            maxcount = Math.max(maxcount, interval.count);

        }

        for (const interval of intervals) {

            let x\_rect = viewRect.x;

            let width\_rect = viewRect.width \* interval.count / maxcount;

            let y\_rect\_top = Prof2dUtilities.transformY(interval.upper, y\_min, y\_range, viewRect.y, viewRect.height);

            let y\_rect\_bottom = Prof2dUtilities.transformY(interval.lower, y\_min, y\_range, viewRect.y, viewRect.height);

            let height\_rect = y\_rect\_bottom - y\_rect\_top;

            let rectInterval = new Rectangle(x\_rect, y\_rect\_top, width\_rect, height\_rect);

            ctx.rect(rectInterval.x, rectInterval.y, rectInterval.width, rectInterval.height);

            const gradient = ctx.createLinearGradient(rectInterval.x, rectInterval.y, rectInterval.x, rectInterval.y + rectInterval.height);

            gradient.addColorStop(0, 'black');

            gradient.addColorStop(0.25, fillStyle);

            gradient.addColorStop(0.5, 'white');

            gradient.addColorStop(0.75, fillStyle);

            gradient.addColorStop(1, 'black');

            ctx.fillStyle = gradient;

            ctx.fillRect(rectInterval.x, rectInterval.y, rectInterval.width, rectInterval.height);

        }

    }

}

function createMouse(element) {

    var mouse = {

        x: 0,

        y: 0,

        button1: false,

        button2: false,

        button3: false,

        over: false,

    };

    function mouseEvent(event) {

        var bounds = element.getBoundingClientRect();

        mouse.x = event.clientX - bounds.left;

        mouse.y = event.clientY - bounds.top;

        if (event.type === "mousedown") {

            mouse["button" + event.which] = true;

        } else if (event.type === "mouseup") {

            mouse["button" + event.which] = false;

        } else if (event.type === "mouseover") {

            mouse.over = true;

        } else if (event.type === "mouseout") {

            mouse.over = false;

            mouse.button1 = false;

            mouse.button2 = false;

            mouse.button3 = false;

        }

        event.preventDefault();

    }

    var events = "mousemove,mousedown,mouseup,mouseout,mouseover".split(',');

    events.forEach(eventType => element.addEventListener(eventType, mouseEvent));

    mouse.remove = function () {

    events.forEach(eventType => element.removeEventListener(eventType, mouseEvent));

    }

    return mouse;

}

const mouse = createMouse(canvas);

let isResizing = false;

let resizeHandle = null;

let isDragging = false;

function update() {

    if (mouse.button1) {

        if (!isResizing) {

            isResizing = true;

            for (let i = 0; i < rects.length; i++) {

                if (

                    mouse.x >= rects[i].x &&

                    mouse.x <= rects[i].x + rects[i].width &&

                    mouse.y >= rects[i].y &&

                    mouse.y <= rects[i].y + rects[i].height) {

                    resizeHandle = {

                        startX: mouse.x,

                        startY: mouse.y,

                        initialWidth: rects[i].width,

                        initialHeight: rects[i].height,

                        rectIndex: i,

                    };

                    break;

                }

            }

        }

    } else {

        isResizing = false;

        resizeHandle = null;

    }

    if (isResizing) {

        const rectIndex = resizeHandle.rectIndex;

        const rect = rects[rectIndex];

        const width = resizeHandle.initialWidth + (mouse.x - resizeHandle.startX);

        const height = resizeHandle.initialHeight + (mouse.y - resizeHandle.startY);

        rect.width = width

        rect.height = height;

        ctx.clearRect(0, 0, canvas.width, canvas.height);

        viewRect.drawRectangle(ctx, "black", 2, [1, 1]);

        drawLabels(viewRect, y\_range);

        drawScores(securityTrajectories, currentInterval, Math.abs(y\_min), y\_range, viewRect);

        if (currentInterval >= Math.floor(N / 2)) {

            printHistogram(Math.floor(N / 2));

        }

        if (currentInterval >= N) {

            printHistogram(N);

        }

    }

}

setInterval(update, 1000 / 60);

for (let interval = 0; interval < numIntervals; interval++) {

    const lower = interval \* intervalSize;

    const upper = (interval + 1) \* intervalSize;

    intervals.push(new Interval(lower, upper));

}

const securityTrajectories = [];

for (let system = 0; system < M; system++) {

    const scores = [];

    let score = 0;

    for (let interval = 0; interval < N; interval++) {

        const outcome = Math.random() <= lambda \* T / numIntervals ? 1 : 0;

        score += outcome;

        scores.push(score);

    }

    securityTrajectories.push(scores);

}

function drawScores(data, frame, y\_min, y\_range, viewRect) {

    for (let system = 0; system < M; system++) {

        const currentTrajectory = data[system];

        const x\_start = viewRect.left();

        const x\_step = viewRect.width / N;

        const y\_start = viewRect.bottom()-viewRect.height;

        let x = x\_start;

        let y = Prof2dUtilities.transformY(currentTrajectory[0], y\_min, y\_range, y\_start, viewRect.height);

        for (let i = 0; i <= frame; i++) {

            const outcome = currentTrajectory[i];

            ctx.strokeStyle = "black";

            ctx.beginPath();

            ctx.moveTo(x, y);

            x += x\_step;

            y = Prof2dUtilities.transformY(outcome, y\_min, y\_range, y\_start, viewRect.height);

            ctx.lineTo(x, y);

            ctx.stroke();

        }

    }

}

function drawLabels(rect, yrange) {

    const ctx = canvas.getContext("2d");

    const xAxisLabelY = rect.bottom() + 10;

    const yAxisLabelX = rect.left() - 10;

    const labelPadding = 10;

    const yrangeHalf = yrange / 2;

    const yStep = Math.floor(yrange / 4);

    const xStep = Math.floor(numIntervals / 5);

for (let i = 0; i <= numIntervals; i += xStep) {

        const x = rect.left() + i \* (rect.width / numIntervals);

        const label = i.toString();

        ctx.fillStyle = "black";

        ctx.fillText(label, x - labelPadding, xAxisLabelY);

    }

for (let i = 0; i <= yrange; i += yStep) {

const y = rect.top() + i \* (rect.height / yrange);

const label = Math.floor(yrange - i).toString();

ctx.fillStyle = "black";

ctx.fillText(label, yAxisLabelX, y);

}

    ctx.stroke();

}

function calculateIntervals(data, systemIndex, numIntervals) {

    const systemData = data.map(subArray => subArray[systemIndex - 1]);

    const minValue = Math.min(...systemData);

    const maxValue = Math.max(...systemData);

    const intervalSize = 1;

    const intervals = [];

    for (let i = 0; i < numIntervals; i++) {

        const lower = minValue + i \* intervalSize;

        const upper = lower + intervalSize;

        const count = systemData.filter((value) => value >= lower && value < upper).length;

        const interval = new Interval(upper, lower, count);

        intervals.push(interval);

    }

    intervals.sort((a, b) => a.upper - b.upper);

    return intervals;

}

function printHistogram(numAttack) {

    const intervals = calculateIntervals(securityTrajectories, numAttack, M + 2);

    let histoHeight = viewRect.height;

    let y\_offset = (viewRect.height - histoHeight) \* 0.5;

    const histoRect1 = new Rectangle(Prof2dUtilities.transformX(numAttack, 0, N, viewRect.x, viewRect.width), viewRect.y + y\_offset, 150, histoHeight);

    Histogram.verticalHistoFromIntervals(ctx, intervals, y\_min, y\_range, histoRect1, "yellow", 3, "yellow");

}

let currentInterval = 1;

let allTrajectories = securityTrajectories.flat();

let y\_max = Math.max(...allTrajectories);

let y\_range = y\_max;

y\_min = 0;

function animate() {

    if (currentInterval <= N) {

        ctx.clearRect(0, 0, canvas.width, canvas.height);

        viewRect.drawRectangle(ctx, "black", 2, [1, 1]);

        drawLabels(viewRect, y\_range);

        drawScores(securityTrajectories, currentInterval, Math.abs(y\_min), y\_range, viewRect);

        if (currentInterval >= Math.floor(N / 2)) {

            printHistogram(Math.floor(N / 2));

        }

        if (currentInterval >= N) {

            printHistogram(N);

        }

        currentInterval=currentInterval++;

        if (currentInterval <= N) {

            setTimeout(animate, frameDuration);

        }

    } else {

        currentInterval = 1;

        animate();

    }

}

const frameDuration = 10;

animate();

</script>

</body>

</html>

