ДОДАТОК А

```
package algorithms;

import imageUtils.ImageUtils;
import org.opencv.core.Mat;
import org.opencv.core.Point;
import org.opencv.core.Rect;
import org.opencv.core.Scalar;
import org.opencv.imgproc.Imgproc;

public class SpecialPointsFinder {

    private Mat image;

    public SpecialPointsFinder(Mat skeletImage) {
        this.image = skeletImage;
    }

    public Mat findSpecialPoints() {
        byte[][] imageMasByte = ImageUtils.getByteMas(image);
        for (int i = 1; i < imageMasByte.length - 1; i++)
            for (int j = 1; j < imageMasByte[0].length - 1; j++) {
                if (getSumPixelWithoutCenter(imageMasByte, i, j) == -5 &&
imageMasByte[i][j]==0) {
                    Rect r = new Rect(j, i, 5, 5);
                    Imgproc.rectangle(image, new Point(r.x, r.y), new Point(r.x +
r.width, r.y + r.height), new Scalar(0, 0, 255), 1);
                    System.out.println("Раздвоение");
```

```
                    }
                    if (getSumPixelWithoutCenter(imageMasByte, i, j) == -7 &&
imageMasByte[i][j]==0){
                        Rect r = new Rect(j-3, i-3, 5, 5);
                        Imgproc.rectangle(image, new Point(r.x, r.y), new Point(r.x +
r.width, r.y + r.height), new Scalar(0, 0, 255), 1);
                        System.out.println("Оканчание");
                        System.out.println();
                    }
                }
            return image;
        }

        private int getSumPixelWithoutCenter(byte[][] masByte, int i, int j) {
            int sumMasPixelsWithoutCenter = masByte[i - 1][j - 1] + masByte[i -
1][j] + masByte[i - 1][j + 1] + masByte[i][j - 1] +
                    masByte[i][j + 1] + masByte[i + 1][j - 1] + masByte[i + 1][j] +
masByte[i + 1][j + 1];
            return sumMasPixelsWithoutCenter;
        }
    }

    package algorithms;

    import imageUtils.ImageUtils;
    import org.opencv.core.Mat;

    import java.util.Arrays;
```

```
public class SkeleronByPatterNonStatic {

    private static final byte[] firstPattern = {-1,-1,0,0,-1,0};
    private static final byte[] secondPattern = {0,0,0,-1,-1,-1};
    private static final byte[] thirdPattern = {0,-1,0,0,-1,-1};
    private static final byte[] fourthPattern = {-1,-1,-1,0,0,0};
    private static final byte[] fifthPattern = {-1,-1,-1,0,0,0,0};
    private static final byte[] sixthPattern = {0,-1,0,0,-1,0,-1};
    private static final byte[] seventhPattern = {-1,0,-1,0,0,-1,0};
    private static final byte[] eighthPattern = {0,0,0,0,-1,-1,-1};

    private static final byte[] firstNoisePattern = {-1,-1,-1,-1,0,-1,-1,-1};
    private static final byte[] secondNoisePattern = {-1,-1,-1,-1,0,-1,-1,0,0};
    private static final byte[] thirdNoisePattern = {-1,-1,-1,0,0,-1,0,-1,-1};
    private static final byte[] fourthNoisePattern = {0,0,-1,-1,0,-1,-1,-1,-1};
    private static final byte[] fifthNoisePattern = {-1,-1,0,-1,0,0,-1,-1,-1};
    private static final byte[] sixthNoisePattern = {-1,-1,-1,-1,0,-1,0,0,-1};
    private static final byte[] seventhNoisePattern = {0,-1,-1,0,0,-1,-1,-1,-1};
    private static final byte[] eighthNoisePattern = {-1,0,0,-1,0,-1,-1,-1,-1};
    private static final byte[] ninethNoisePattern = {-1,-1,-1,-1,0,0,-1,-1,0};
    private static final byte[] tenthNoisePattern = {-1,-1,-1,-1,0,-1,0,0,0};
    private static final byte[] eleventhNoisePattern = {0,-1,-1,0,0,-1,0,-1,-1};
    private static final byte[] twelfthNoisePattern = {0,0,0,-1,0,-1,-1,-1,-1};
    private static final byte[] thirteenthNoisePattern = {-1,-1,0,-1,0,0,-1,-1,0};


    private Mat image;
    private byte[][] masByte;
```

```
public SkeleronByPatterNonStatic(Mat imageMat) {
    this.image = imageMat;
    this.masByte = ImageUtils.getByteMas(imageMat);
}

public byte[][] byteMasAfterSkeleton(){
    for (int i = 1; i<image.height()-1;i++)
        for (int j = 1; j<image.width()-1;j++){
            byte[] masFirstPattern = getMasForFirstPattern(masByte,i,j);
            byte[] masSecondPattern = getMasForSecondPattern(masByte,i,j);
            byte[] masThirdPattern = getMasForThirdPattern(masByte,i,j);
            byte[] masFourthPattern = getMasForFourthPattern(masByte,i,j);
            byte[] masFifthPattern = getMasForFifthPattern(masByte,i,j);
            byte[] masSixthPattern = getMasForSixthPattern(masByte,i,j);
            byte[] masSeventhPattern = getMasForSeventhPattern(masByte,i,j);
            byte[] masEighthPattern = getMasForEighthPattern(masByte,i,j);

            if (Arrays.equals(masFirstPattern,firstPattern)){
                masByte[i][j] = -1;
            }
            if (Arrays.equals(masSecondPattern,secondPattern)){
                masByte[i][j] = -1;
            }
            if (Arrays.equals(masThirdPattern,thirdPattern)){
                masByte[i][j] = -1;
            }
```

```
            if (Arrays.equals(masFourthPattern,fourthPattern)){
                masByte[i][j] = -1;
            }
            if (Arrays.equals(masFifthPattern,fifthPattern)){
                masByte[i][j] = -1;
            }
            if (Arrays.equals(masSixthPattern,sixthPattern)){
                masByte[i][j] = -1;
            }
            if (Arrays.equals(masSeventhPattern,seventhPattern)){
                masByte[i][j] = -1;
            }
            if (Arrays.equals(masEighthPattern,eighthPattern)){
                masByte[i][j] = -1;
            }

        }

    return masByte;

}

public boolean checkForUnnecessaryPexel(){
    for (int i = 1; i<image.height()-1;i++)
        for (int j = 1; j<image.width()-1;j++){
            byte[] masFirstPattern = getMasForFirstPattern(masByte,i,j);
            byte[] masSecondPattern = getMasForSecondPattern(masByte,i,j);
            byte[] masThirdPattern = getMasForThirdPattern(masByte,i,j);
            byte[] masFourthPattern = getMasForFourthPattern(masByte,i,j);
```

```
byte[] masFifthPattern = getMasForFifthPattern(masByte,i,j);
byte[] masSixthPattern = getMasForSixthPattern(masByte,i,j);
byte[] masSeventhPattern = getMasForSeventhPattern(masByte,i,j);
byte[] masEighthPattern = getMasForEighthPattern(masByte,i,j);

if (Arrays.equals(masFirstPattern,firstPattern)){
   return true;
}
if (Arrays.equals(masSecondPattern,secondPattern)){
   return true;
}
if (Arrays.equals(masThirdPattern,thirdPattern)){
   return true;
}
if (Arrays.equals(masFourthPattern,fourthPattern)){
   return true;
}
if (Arrays.equals(masFifthPattern,fifthPattern)){
   return true;
}
if (Arrays.equals(masSixthPattern,sixthPattern)){
   return true;
}
if (Arrays.equals(masSeventhPattern,seventhPattern)){
   return true;
}
if (Arrays.equals(masEighthPattern,eighthPattern)){
   return true;
}
```

```
            }
        return false;
    }

    public byte[][] removeUnnecessaryPixel(){
        while (checkForUnnecessaryPexel()||checkForNoise()){
            removeNoise();
            byteMasAfterSkeleton();

        }
        return masByte;
    }

    private byte[] getMasForFirstPattern(byte[][] masByte,int i, int j){
        byte[] mas = {masByte[i-1][j],masByte[i-1][j+1],masByte[i][j-
1],masByte[i][j],masByte[i][j+1],masByte[i+1][j]};
        return mas;
    }

    private byte[] getMasForSecondPattern(byte[][] masByte,int i, int j){
        byte[] mas = {masByte[i-1][j],masByte[i][j-
1],masByte[i][j],masByte[i][j+1],masByte[i+1][j],masByte[i+1][j+1]};
        return mas;
    }

    private byte[] getMasForThirdPattern(byte[][] masByte,int i, int j){
        byte[] mas = {masByte[i-1][j],masByte[i][j-
1],masByte[i][j],masByte[i][j+1],masByte[i+1][j-1],masByte[i+1][j]};
        return mas;
```

```
        }

        private byte[] getMasForFourthPattern(byte[][] masByte,int i, int j){
            byte[] mas = {masByte[i-1][j-1],masByte[i-1][j],masByte[i][j-
1],masByte[i][j],masByte[i][j+1],masByte[i+1][j]};
            return mas;
        }

        private byte[] getMasForFifthPattern(byte[][] masByte,int i, int j){
            byte[] mas = {masByte[i-1][j-1],masByte[i-1][j],masByte[i-
1][j+1],masByte[i][j-1],masByte[i][j],masByte[i][j+1],masByte[i+1][j]};
            return mas;
        }

        private byte[] getMasForSixthPattern(byte[][] masByte,int i, int j){
            byte[] mas = {masByte[i-1][j],masByte[i-1][j+1],masByte[i][j-
1],masByte[i][j],masByte[i][j+1],masByte[i+1][j],masByte[i+1][j+1]};
            return mas;
        }

        private byte[] getMasForSeventhPattern(byte[][] masByte,int i, int j){
            byte[] mas = {masByte[i-1][j-1],masByte[i-1][j],masByte[i][j-
1],masByte[i][j],masByte[i][j+1],masByte[i+1][j-1],masByte[i+1][j]};
            return mas;
        }

        private byte[] getMasForEighthPattern(byte[][] masByte,int i, int j){
            byte[] mas = {masByte[i-1][j],masByte[i][j-
1],masByte[i][j],masByte[i][j+1],masByte[i+1][j-
```

```
1],masByte[i+1][j],masByte[i+1][j+1]};
            return mas;
        }

        private boolean checkForNoise(){
            for (int i = 1; i<image.height()-1;i++)
                for (int j = 1; j<image.width()-1;j++){
                    byte[] masNoisePattern = {masByte[i-1][j-1],masByte[i-
1][j],masByte[i-1][j+1], masByte[i][j-1],masByte[i][j],masByte[i][j+1],
                            masByte[i+1][j-1],masByte[i+1][j],masByte[i+1][j+1]};
                    if (Arrays.equals(masNoisePattern,firstNoisePattern)){
                        return true;
                    }
                    if (Arrays.equals(masNoisePattern,secondNoisePattern)){
                        return true;
                    }
                    if (Arrays.equals(masNoisePattern,thirdNoisePattern)){
                        return true;
                    }
                    if (Arrays.equals(masNoisePattern,fourthNoisePattern)){
                        return true;
                    }
                    if (Arrays.equals(masNoisePattern,fifthNoisePattern)){
                        return true;
                    }
                    if (Arrays.equals(masNoisePattern,sixthNoisePattern)){
                        return true;
                    }
                    if (Arrays.equals(masNoisePattern,seventhNoisePattern)){
```

```
            return true;
        }
        if (Arrays.equals(masNoisePattern,eighthNoisePattern)){
            return true;
        }
        if (Arrays.equals(masNoisePattern,ninethNoisePattern)){
            return true;
        }
        if (Arrays.equals(masNoisePattern,tenthNoisePattern)){
            return true;
        }
        if (Arrays.equals(masNoisePattern,eleventhNoisePattern)){
            return true;
        }
        if (Arrays.equals(masNoisePattern,twelfthNoisePattern)){
            return true;
        }
        if (Arrays.equals(masNoisePattern,thirteenthNoisePattern)){
            return true;
        }
    }
  return false;
}


private void removeNoise(){
  for (int i = 1; i<masByte.length-1;i++)
    for (int j = 1; j<masByte[0].length-1;j++){
      byte[] masNoisePattern = {masByte[i-1][j-1],masByte[i-
```

```
1][j],masByte[i-1][j+1], masByte[i][j-1],masByte[i][j],masByte[i][j+1],
                masByte[i+1][j-1],masByte[i+1][j],masByte[i+1][j+1]};
            if (Arrays.equals(masNoisePattern,firstNoisePattern)){
               masByte[i][j]=-1;
            }
            if (Arrays.equals(masNoisePattern,secondNoisePattern)){
               masByte[i][j]=-1;
            }
            if (Arrays.equals(masNoisePattern,thirdNoisePattern)){
               masByte[i][j]=-1;
            }
            if (Arrays.equals(masNoisePattern,fourthNoisePattern)){
               masByte[i][j]=-1;
            }
            if (Arrays.equals(masNoisePattern,fifthNoisePattern)){
               masByte[i][j]=-1;
            }
            if (Arrays.equals(masNoisePattern,sixthNoisePattern)){
               masByte[i][j]=-1;
            }
            if (Arrays.equals(masNoisePattern,seventhNoisePattern)){
               masByte[i][j]=-1;
            }
            if (Arrays.equals(masNoisePattern,eighthNoisePattern)){
               masByte[i][j]=-1;
            }
            if (Arrays.equals(masNoisePattern,ninethNoisePattern)){
               masByte[i][j]=-1;
            }
```

```
            if (Arrays.equals(masNoisePattern,tenthNoisePattern)){
                masByte[i][j]=-1;
            }
            if (Arrays.equals(masNoisePattern,eleventhNoisePattern)){
                masByte[i][j]=-1;
            }
            if (Arrays.equals(masNoisePattern,twelfthNoisePattern)){
                masByte[i][j]=-1;
            }
            if (Arrays.equals(masNoisePattern,thirteenthNoisePattern)){
                masByte[i][j]=-1;
            }
        }
    }
package algorithms;

import imageUtils.ImageUtils;
import org.opencv.core.CvType;
import org.opencv.core.Mat;

import java.util.ArrayList;

public class GetNonVightPixel {

    private ArrayList<String> arrayWithBlackPixel;
    private ArrayList<Mat> masMat;
    private Mat resultImage;

    public GetNonVightPixel(ArrayList<Mat> masMat) {
```

```
            this.masMat = masMat;
            this.resultImage = new Mat(masMat.get(0).height(),
masMat.get(0).width(), CvType.CV_8U);
            arrayWithBlackPixel = new ArrayList<>();

        }

        public void getNonVightPixel() {
            for (int k = 0; k < masMat.size(); k++) {
                Mat image = masMat.get(k);
                image.convertTo(image, CvType.CV_8U);
                byte[][] pixelMas = ImageUtils.getByteMas(image);
                for (int i = 0; i < pixelMas.length; i++)
                    for (int j = 0; j < pixelMas[0].length; j++) {
                        if (!(pixelMas[i][j] == -1)) {
                            String coordinates = i + "," + j + "," + pixelMas[i][j];
                            arrayWithBlackPixel.add(coordinates);
                        }
                    }
            }
        }

        public void putBlackPixelIntoMas() {
            byte[][] resultByteMas = new
byte[resultImage.height()][resultImage.width()];
            for (int i = 0; i < resultImage.height(); i++)
                for (int j = 0; j < resultImage.width(); j++) {
                    resultByteMas[i][j] = -1;
                }
```

```
            for (int i = 0; i < this.arrayWithBlackPixel.size(); i++) {
                String str = this.arrayWithBlackPixel.get(i);
                String[] subStr;
                String delimeter = ",";
                subStr = str.split(delimeter);

                int x = Integer.parseInt(subStr[0]);
                int y = Integer.parseInt(subStr[1]);
                byte value = Byte.parseByte(subStr[2]);
                if (resultByteMas[x][y] < value)
                    resultByteMas[x][y] = value;

            }
            byte[] oneDimentionalMas =
ImageUtils.convertToTwoDimensionalMas(resultByteMas);
            resultImage.put(0, 0, oneDimentionalMas);
        }


        public Mat getResultImage() {
            return resultImage;
        }
    }

    package algorithms;

    import imageUtils.ImageUtils;
    import org.opencv.core.Core;
    import org.opencv.core.CvType;
```

```
import org.opencv.core.Mat;
import org.opencv.core.Size;
import org.opencv.imgproc.Imgproc;

import javax.imageio.ImageIO;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class GaborFilter {

    private ArrayList<Mat> matList;

    public GaborFilter(){
        matList = new ArrayList<>();
    }

    public Mat getGabor(Mat image, Size kSize, double sigma, double theta,
double lambda, double gamma, double psi){
        image.convertTo(image,CvType.CV_32F);

        Mat gabor = new Mat (image.width(), image.height(), CvType.CV_32F);

        Mat kernel1 = Imgproc.getGaborKernel(kSize, sigma, theta, lambda,
gamma, psi, CvType.CV_32F);

        Imgproc.filter2D(image, gabor, -1, kernel1);
```

```
            return gabor;
        }

        public ArrayList<Mat> getGaborWithDifferentTheta(Mat image,Size kSize,
double sigma, double theta, double lambda, double gamma, double psi) throws
IOException {

            Mat resultImage = getGabor(image, kSize, sigma, 0, lambda, gamma,
psi);
            matList.add(resultImage);
            File first = new File("imagefirst.jpg");
            ImageIO.write(ImageUtils.matToBufferedImage(resultImage), "jpg",
first);
            for (double i = theta; i<= 180; i += theta){
               Mat tempImage = getGabor(image, kSize, sigma, i, lambda, gamma,
psi);
               matList.add(tempImage);
               File outputFile = new File("image"+i+".jpg");
               ImageIO.write(ImageUtils.matToBufferedImage(tempImage), "jpg",
outputFile);
            }
        return matList;
        }

        public List<Mat> getMatList(){
            return matList;
        }
    }
```

```
import algorithms.GetNonVightPixel;
import algorithms.SkeleronByPatterNonStatic;
import algorithms.SpecialPointsFinder;
import imageUtils.ImageUtils;
import org.opencv.core.*;
import org.opencv.imgcodecs.Imgcodecs;
import org.opencv.imgproc.Imgproc;

import javax.imageio.ImageIO;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class Test5 {

    static {
        System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
    }

    public static void main(String[] args) throws IOException {

        GaborFilter gaborFilter = new GaborFilter();
        Mat myImg = Imgcodecs.imread("src/main/resources/test11.png", 0);

        Size kSize = new Size(11,11);

        double lambda = 6.75;
        double sigma = 1.55;
```

```
double gamma = 0.55;
double psi =  0;

ArrayList<Mat> arrayGaborMat =
gaborFilter.getGaborWithDifferentTheta(myImg,kSize, sigma, 45, lambda, gamma,
psi);

GetNonVightPixel getNonVightPixel = new
GetNonVightPixel(arrayGaborMat);
getNonVightPixel.getNonVightPixel();
getNonVightPixel.putBlackPixelIntoMas();

Mat resultMat = getNonVightPixel.getResultImage();
File gaborFinal = new File("GaborFinal.jpg");
ImageIO.write(ImageUtils.matToBufferedImage(resultMat), "jpg",
gaborFinal);

ImageUtils.showBufImage(ImageUtils.matToBufferedImage(resultMat));


Mat binaryImg = new Mat();


Imgproc.adaptiveThreshold(resultMat,binaryImg,255,Imgproc.ADAPTIVE_THRES
H_MEAN_C, Imgproc.THRESH_BINARY,11,0.5);

ImageUtils.showBufImage(ImageUtils.matToBufferedImage(binaryImg));
File binary = new File("BinaryFinal.jpg");
ImageIO.write(ImageUtils.matToBufferedImage(binaryImg), "jpg",
```

```
binary);

        SkeleronByPatterNonStatic skeleronByPatterNonStatic = new
SkeleronByPatterNonStatic(binaryImg);
        byte[][] mas99 = skeleronByPatterNonStatic.removeUnnecessaryPixel();
        byte[] mas24 = ImageUtils.convertToTwoDimensionalMas(mas99);
        binaryImg.put(0,0,mas24);

ImageUtils.showBufImage(ImageUtils.matToBufferedImage(binaryImg));
        File outputFile1 = new File("Scelet.jpg");
        ImageIO.write(ImageUtils.matToBufferedImage(binaryImg), "jpg",
outputFile1);
        SpecialPointsFinder specialPointsFinder = new
SpecialPointsFinder(binaryImg);
        Mat imageWithSpecialPoints = specialPointsFinder.findSpecialPoints();

ImageUtils.showBufImage(ImageUtils.matToBufferedImage(imageWithSpecialPoint
s));
        File outputFile = new File("Finalimage"+".jpg");
        ImageIO.write(ImageUtils.matToBufferedImage(binaryImg), "jpg",
outputFile);
        /**
         * List of special  points
         */
        List<SpecialPoint> listOfBranches =
specialPointsFinder.getListOfSpecialPointsBranches();
        List<SpecialPoint> listOfEnds =
specialPointsFinder.getListOfSpecialPointsEnds();
```

```
            /**
             * Filter duplicated branches points
             */
            List<SpecialPoint> branchesWithoutDuplicates = new
SpecialPointProcessor().removeDuplicatesBranches(listOfBranches);

            System.out.println(branchesWithoutDuplicates.size());
            /**
             * List with all points
             */
            List<SpecialPoint> listWithAllPoints = new ArrayList<SpecialPoint>();
            listWithAllPoints.addAll(listOfEnds);
            listWithAllPoints.addAll(branchesWithoutDuplicates);

            /**
             * List with counted distances
             */
            List<SpecialPoint> pointsWithDistances = new
SpecialPointDistanceFinder().findDistancesBetweenPoints(listWithAllPoints);

            List<SpecialPoint> listOfPointsWithCountedDistances = new
SpecialPointDistanceFinder().sortDistances(pointsWithDistances);


System.out.println(listOfPointsWithCountedDistances.get(0).getListOdDistances());

System.out.println(listOfPointsWithCountedDistances.get(1).getListOdDistances());
        }
```

```
        }
class SpecialPointDistanceFinder {


    fun findDistancesBetweenPoints(list: List<SpecialPoint>): List<SpecialPoint> {
        return list.mapIndexed { index, specialPoint ->
            val newSpecialPoint = SpecialPoint(specialPoint.xCoordinate,
specialPoint.yCoordinate, specialPoint.pointType)
            list.forEachIndexed { forEachIndex, forEachSpecialPoint ->
                if (index == forEachIndex) return@forEachIndexed
                else newSpecialPoint.listOdDistances[findDistance(specialPoint,
forEachSpecialPoint)] = forEachSpecialPoint
            }
            newSpecialPoint
        }

    }

    fun sortDistances(list: List<SpecialPoint>): List<SpecialPoint> {
        return list.map { specialPoint ->
            SpecialPoint(specialPoint.xCoordinate, specialPoint.yCoordinate,
specialPoint.pointType, specialPoint.listOdDistances.toSortedMap())
        }
    }

    private fun findDistance(firstPoint: SpecialPoint, secondPoint: SpecialPoint):
Double {
        val resolveX = (secondPoint.xCoordinate - firstPoint.xCoordinate).pow(2)
        val resolveY = (secondPoint.yCoordinate - firstPoint.yCoordinate).pow(2)
```

```
        val resolveSumOfCoordinate = resolveX.plus(resolveY)
        return sqrt(resolveSumOfCoordinate)
    }
}
class SpecialPointProcessor {

    fun removeUnnecessaryPoints(listOfSpecialPoints: List<SpecialPoint>):
List<SpecialPoint> {
        val listOfBranches = listOfSpecialPoints.filter { it.pointType ==
PointType.BRANCH }
        val s = HashSet<SpecialPoint>()
println(listOfBranches[25]==listOfBranches[25])
        s.addAll(listOfBranches)
        println("--------")
        println(s.size)
        println("----------")
        return listOfSpecialPoints.filter { it.pointType == PointType.BRANCH
}.toHashSet().toList()
    }

    fun removeDuplicatesBranches(list: List<SpecialPoint>): List<SpecialPoint> {
        val listOfBranches = list.filter { it.pointType == PointType.BRANCH }
        // Set set1 = new LinkedHashSet(list);
        val set = TreeSet(Comparator<SpecialPoint> { o1, o2 ->
            if (o1!!.xCoordinate in (o2!!.xCoordinate.minus(1)..o2.xCoordinate.plus(1))
&& o1.yCoordinate in (o2.yCoordinate.minus(1)..o2.yCoordinate.plus(1)))
                0
            else -1
        })
```

```
    set.addAll(listOfBranches)
    return ArrayList(set)
}

fun showAllBranches(listOfSpecialPoints: List<SpecialPoint>) {
    listOfSpecialPoints
            .filter { it.pointType == PointType.BRANCH }
            .forEach { println("${it.xCoordinate} , ${it.yCoordinate}") }

}
```

ВІДОМІСТЬ АТЕСТАЦІЙНОЇ РОБОТИ

Позначення	Найменування	Дод. відомості
	Текстові документи	
1	Пояснювальна записка	97 с.
2	Презентаційний матеріал	22 с.
	Інші документи	с.
3	Роздруківки програм	22 с.
4	Рецензія	2 с.
5	Відгук керівника	1 с.

					Дослідження сучасних методів аналізу папілярних візерунків та їх застосувань			
Змін	**Арк.**	**Номер докум.**	**Підп.**	**Дата**				
Розроб.		**Кононенко Н.В.**			**(Тема роботи) Відомість атестаційної роботи**		**Аркуш**	**Аркушів**
Перевір.		**Кобзєв В.Г.**						
Н. контр.		**Сидоров М.В.**				**ХНУРЕ кафедра ПМ**		
Затв.		**Тевяшев А.Д.**						