```
package canny;

import java.awt.Color;
import java.awt.image.BufferedImage;
import java.io.IOException;

import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.mapreduce.Mapper;
import java.util.Arrays;

import javax.imageio.ImageIO;

public class CannyMapper extends Mapper<LongWritable, BufferedImage, LongWritable, Buffered
Image>{

        public void map(LongWritable key, BufferedImage value, Context context)
                        throws IOException, InterruptedException {

                System.out.println("map started");

                //create the detector
                CannyEdgeDetector detector = new CannyEdgeDetector();

                //adjust its parameters as desired
                detector.setLowThreshold(0.5f);
                detector.setHighThreshold(1f);

                //apply it to an image
                detector.setSourceImage(value);
                detector.process();

                System.out.println("Edge Detected chunk " + key.get());

                BufferedImage edges = detector.getEdgesImage();

                if(edges == null) {
                        System.out.println("edge detect made a null");
                }

                //context.write(key, edges);

                FileSystem dfs = FileSystem.get(context.getConfiguration());
                Path newimgpath = new Path(context.getWorkingDirectory(), context.getJobID(
).toString()+"/"+key.get());
                dfs.createNewFile(newimgpath);
                FSDataOutputStream ofs = dfs.create(newimgpath);
                ImageIO.write(edges, "jpg", ofs);
        }
}
```

```
package canny;


import java.awt.image.BufferedImage;
import java.util.Iterator;


import javax.imageio.ImageIO;
import javax.imageio.ImageReader;
import javax.imageio.stream.MemoryCacheImageInputStream;

import utils.*;

import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;

public class Canny {

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

                Configuration config = new Configuration();
                String[] otherArgs = new GenericOptionsParser(config,
                                args).getRemainingArgs();

                if (otherArgs.length != 2) {
                        System.err.println("Usage: Canny <in> <out>");
                        System.exit(2);
                }

                BufferedImage img = null;
                FileSystem dfs = FileSystem.get(config);
                Path dir = new Path(otherArgs[0]);
                FileStatus[] files = dfs.listStatus(dir);



                config.setInt("overlapPixel", 64);

                int overlapPixel = ImgRecordReader.overlapPixel;
                System.out.println(overlapPixel);

                Path filepath = null;
                for (FileStatus file: files) {
                        if (file.isDir()) continue;
                         filepath = file.getPath();

                        System.out.println(filepath);
                }

                Path outdir = new Path(otherArgs[1]);
                if (dfs.exists(outdir)) dfs.delete(outdir,true);
                Path workdir = dfs.getWorkingDirectory();

                Job job = new Job(config, "Canny Edge detection");
                job.setJarByClass(Canny.class);
                job.setMapperClass(CannyMapper.class);

                job.setReducerClass(CannyReducer.class);
```

```
		job.setInputFormatClass(InputFormatImg.class);

		job.setOutputKeyClass(LongWritable.class);
		job.setOutputValueClass(LongWritable.class);

		FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
		FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));


		boolean ret = job.waitForCompletion(true);
		String s = job.getTrackingURL();
		Path tmpdir = new Path(workdir, s.substring(s.indexOf("jobid")+6));

		int i = 0;
		Path iPath = new Path(tmpdir,""+i);
		int currX =0, currY = 0;
		int sizePixel = ImgRecordReader.sizePixel;
		int border = 16;

		FSDataInputStream filesys = null;
		MemoryCacheImageInputStream image = new MemoryCacheImageInputStream(dfs.ope
n(filepath));
		Iterator<ImageReader> readers = ImageIO.getImageReaders(image);
		ImageReader reader = (ImageReader) readers.next();
		reader.setInput(image);
		int imgwidth = 0, imgheight = 0;
		imgwidth = reader.getWidth(0);
		imgheight = reader.getHeight(0);


		img = new BufferedImage(imgwidth, imgheight, BufferedImage.TYPE_INT_RGB);

		if(imgwidth*imgheight <= sizePixel*sizePixel) {
			filesys = dfs.open(iPath);
			img = ImageIO.read(filesys);
			iPath = null;
		}
		while(iPath != null && dfs.exists(iPath)) {
			int x = currX, y = currY;
			currX += sizePixel;
			if (currX >= imgwidth) {
				currX = 0;
				currY += sizePixel;
			}

			filesys = dfs.open(iPath);
			BufferedImage window = ImageIO.read(filesys);
			int width = window.getWidth() - border*2;
			int height = window.getHeight() - border*2;

			img.setRGB(x+border, y+border, width, height,
					window.getRGB(border,border, width, height, null, 0
, width),
					0, width);
			filesys.close();
			i++;
			iPath = new Path(tmpdir,""+i);
		}
		Path newimgpath = new Path(outdir, filepath.getName());

		if (dfs.exists(newimgpath)) {
			dfs.delete(newimgpath,false);
		}

		dfs.createNewFile(newimgpath);
		FSDataOutputStream ofs = dfs.create(newimgpath);
		ImageIO.write(img, "JPG", ofs);
```

```
                ofs.close();
                dfs.delete(tmpdir, true);

                System.exit(ret ? 0 : 1);
        }
}
```

```
package canny;

import java.awt.image.BufferedImage;
import java.io.IOException;

import javax.imageio.ImageIO;

import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.mapreduce.Reducer;

public class CannyReducer extends Reducer<LongWritable, BufferedImage,LongWritable, LongWri
table> {

        public void reduce(LongWritable key, Iterable<BufferedImage> values, Context contex
t)
                        throws IOException, InterruptedException {

                FileSystem dfs = FileSystem.get(context.getConfiguration());
                Path newimgpath = new Path(context.getWorkingDirectory(), ""+key.get());
                dfs.createNewFile(newimgpath);
                FSDataOutputStream ofs = dfs.create(newimgpath);
                BufferedImage img = values.iterator().next();
                ImageIO.write(img, "JPG", ofs);
                context.write(key, new LongWritable(1));
        }

}
```

```
package canny;

import java.awt.image.BufferedImage;
import java.util.Arrays;

/**
 * <p><em>This software has been released into the public domain.
 * <strong>Please read the notes in this source file for additional information.
 * </strong></em></p>
 *
 * <p>This class provides a configurable implementation of the Canny edge
 * detection algorithm. This classic algorithm has a number of shortcomings,
 * but remains an effective tool in many scenarios. <em>This class is designed
 * for single threaded use only.</em></p>
 *
 * <p>Sample usage:</p>
 *
 * <pre><code>
 * //create the detector
 * CannyEdgeDetector detector = new CannyEdgeDetector();
 * //adjust its parameters as desired
 * detector.setLowThreshold(0.5f);
 * detector.setHighThreshold(1f);
 * //apply it to an image
 * detector.setSourceImage(frame);
 * detector.process();
 * BufferedImage edges = detector.getEdgesImage();
 * </code></pre>
 *
 * <p>For a more complete understanding of this edge detector's parameters
 * consult an explanation of the algorithm.</p>
 *
 * @author Tom Gibara
 *
 */

public class CannyEdgeDetector {

	// statics

	private final static float GAUSSIAN_CUT_OFF = 0.005f;
	private final static float MAGNITUDE_SCALE = 100F;
	private final static float MAGNITUDE_LIMIT = 1000F;
	private final static int MAGNITUDE_MAX = (int) (MAGNITUDE_SCALE * MAGNITUDE_LIMIT);

	// fields

	private int height;
	private int width;
	private int picsize;
	private int[] data;
	private int[] magnitude;
	private BufferedImage sourceImage;
	private BufferedImage edgesImage;

	private float gaussianKernelRadius;
	private float lowThreshold;
	private float highThreshold;
	private int gaussianKernelWidth;
	private boolean contrastNormalized;

	private float[] xConv;
	private float[] yConv;
	private float[] xGradient;
	private float[] yGradient;

	// constructors
```

```
        /**
         * Constructs a new detector with default parameters.
         */

        public CannyEdgeDetector() {
                lowThreshold = 2.5f;
                highThreshold = 7.5f;
                gaussianKernelRadius = 2f;
                gaussianKernelWidth = 16;
                contrastNormalized = false;
        }

        // accessors

        /**
         * The image that provides the luminance data used by this detector to
         * generate edges.
         *
         * @return the source image, or null
         */

        public BufferedImage getSourceImage() {
                return sourceImage;
        }

        /**
         * Specifies the image that will provide the luminance data in which edges
         * will be detected. A source image must be set before the process method
         * is called.
         *
         * @param image a source of luminance data
         */

        public void setSourceImage(BufferedImage image) {
                sourceImage = image;
        }

        /**
         * Obtains an image containing the edges detected during the last call to
         * the process method. The buffered image is an opaque image of type
         * BufferedImage.TYPE_INT_ARGB in which edge pixels are white and all other
         * pixels are black.
         *
         * @return an image containing the detected edges, or null if the process
         * method has not yet been called.
         */

        public BufferedImage getEdgesImage() {
                return edgesImage;
        }

        /**
         * Sets the edges image. Calling this method will not change the operation
         * of the edge detector in any way. It is intended to provide a means by
         * which the memory referenced by the detector object may be reduced.
         *
         * @param edgesImage expected (though not required) to be null
         */

        public void setEdgesImage(BufferedImage edgesImage) {
                this.edgesImage = edgesImage;
        }

        /**
         * The low threshold for hysteresis. The default value is 2.5.
         *
         * @return the low hysteresis threshold
         */
```

```
	public float getLowThreshold() {
		return lowThreshold;
	}

	/**
	 * Sets the low threshold for hysteresis. Suitable values for this parameter
	 * must be determined experimentally for each application. It is nonsensical
	 * (though not prohibited) for this value to exceed the high threshold value.
	 *
	 * @param threshold a low hysteresis threshold
	 */

	public void setLowThreshold(float threshold) {
		if (threshold < 0) throw new IllegalArgumentException();
		lowThreshold = threshold;
	}

	/**
	 * The high threshold for hysteresis. The default value is 7.5.
	 *
	 * @return the high hysteresis threshold
	 */

	public float getHighThreshold() {
		return highThreshold;
	}

	/**
	 * Sets the high threshold for hysteresis. Suitable values for this
	 * parameter must be determined experimentally for each application. It is
	 * nonsensical (though not prohibited) for this value to be less than the
	 * low threshold value.
	 *
	 * @param threshold a high hysteresis threshold
	 */

	public void setHighThreshold(float threshold) {
		if (threshold < 0) throw new IllegalArgumentException();
		highThreshold = threshold;
	}

	/**
	 * The number of pixels across which the Gaussian kernel is applied.
	 * The default value is 16.
	 *
	 * @return the radius of the convolution operation in pixels
	 */

	public int getGaussianKernelWidth() {
		return gaussianKernelWidth;
	}

	/**
	 * The number of pixels across which the Gaussian kernel is applied.
	 * This implementation will reduce the radius if the contribution of pixel
	 * values is deemed negligable, so this is actually a maximum radius.
	 *
	 * @param gaussianKernelWidth a radius for the convolution operation in
	 * pixels, at least 2.
	 */

	public void setGaussianKernelWidth(int gaussianKernelWidth) {
		if (gaussianKernelWidth < 2) throw new IllegalArgumentException();
		this.gaussianKernelWidth = gaussianKernelWidth;
	}

	/**
```

```
         * The radius of the Gaussian convolution kernel used to smooth the source
         * image prior to gradient calculation. The default value is 16.
         *
         * @return the Gaussian kernel radius in pixels
         */

        public float getGaussianKernelRadius() {
                return gaussianKernelRadius;
        }

        /**
         * Sets the radius of the Gaussian convolution kernel used to smooth the
         * source image prior to gradient calculation.
         *
         * @return a Gaussian kernel radius in pixels, must exceed 0.1f.
         */

        public void setGaussianKernelRadius(float gaussianKernelRadius) {
                if (gaussianKernelRadius < 0.1f) throw new IllegalArgumentException();
                this.gaussianKernelRadius = gaussianKernelRadius;
        }

        /**
         * Whether the luminance data extracted from the source image is normalized
         * by linearizing its histogram prior to edge extraction. The default value
         * is false.
         *
         * @return whether the contrast is normalized
         */

        public boolean isContrastNormalized() {
                return contrastNormalized;
        }

        /**
         * Sets whether the contrast is normalized
         * @param contrastNormalized true if the contrast should be normalized,
         * false otherwise
         */

        public void setContrastNormalized(boolean contrastNormalized) {
                this.contrastNormalized = contrastNormalized;
        }

        // methods

        public void process() {
                width = sourceImage.getWidth();
                height = sourceImage.getHeight();
                picsize = width * height;
                initArrays();
                readLuminance();
                if (contrastNormalized) normalizeContrast();
                computeGradients(gaussianKernelRadius, gaussianKernelWidth);
                int low = Math.round(lowThreshold * MAGNITUDE_SCALE);
                int high = Math.round( highThreshold * MAGNITUDE_SCALE);
                performHysteresis(low, high);
                thresholdEdges();
                writeEdges(data);
        }

        // private utility methods

        private void initArrays() {
                if (data == null || picsize != data.length) {
                        data = new int[picsize];
                        magnitude = new int[picsize];
```

```
                        xConv = new float[picsize];
                        yConv = new float[picsize];
                        xGradient = new float[picsize];
                        yGradient = new float[picsize];
                }
        }

        //NOTE: The elements of the method below (specifically the technique for
        //non-maximal suppression and the technique for gradient computation)
        //are derived from an implementation posted in the following forum (with the
        //clear intent of others using the code):
        //  http://forum.java.sun.com/thread.jspa?threadID=546211&start=45&tstart=0
        //My code effectively mimics the algorithm exhibited above.
        //Since I don't know the providence of the code that was posted it is a
        //possibility (though I think a very remote one) that this code violates
        //someone's intellectual property rights. If this concerns you feel free to
        //contact me for an alternative, though less efficient, implementation.

        private void computeGradients(float kernelRadius, int kernelWidth) {

                //generate the gaussian convolution masks
                float kernel[] = new float[kernelWidth];
                float diffKernel[] = new float[kernelWidth];
                int kwidth;
                for (kwidth = 0; kwidth < kernelWidth; kwidth++) {
                        float g1 = gaussian(kwidth, kernelRadius);
                        if (g1 <= GAUSSIAN_CUT_OFF && kwidth >= 2) break;
                        float g2 = gaussian(kwidth - 0.5f, kernelRadius);
                        float g3 = gaussian(kwidth + 0.5f, kernelRadius);
                        kernel[kwidth] = (g1 + g2 + g3) / 3f / (2f * (float) Math.PI * kern
elRadius * kernelRadius);
                        diffKernel[kwidth] = g3 - g2;
                }

                int initX = kwidth - 1;
                int maxX = width - (kwidth - 1);
                int initY = width * (kwidth - 1);
                int maxY = width * (height - (kwidth - 1));

                //perform convolution in x and y directions
                for (int x = initX; x < maxX; x++) {
                        for (int y = initY; y < maxY; y += width) {
                                int index = x + y;
                                float sumX = data[index] * kernel[0];
                                float sumY = sumX;
                                int xOffset = 1;
                                int yOffset = width;
                                for(; xOffset < kwidth ;) {
                                        sumY += kernel[xOffset] * (data[index - yOffset] +
data[index + yOffset]);
                                        sumX += kernel[xOffset] * (data[index - xOffset] +
data[index + xOffset]);
                                        yOffset += width;
                                        xOffset++;
                                }

                                yConv[index] = sumY;
                                xConv[index] = sumX;
                        }

                }

                for (int x = initX; x < maxX; x++) {
                        for (int y = initY; y < maxY; y += width) {
                                float sum = 0f;
                                int index = x + y;
                                for (int i = 1; i < kwidth; i++)
                                        sum += diffKernel[i] * (yConv[index - i] - yConv[in
```

```
dex + i]);

                                xGradient[index] = sum;
                        }

                }

                for (int x = kwidth; x < width - kwidth; x++) {
                        for (int y = initY; y < maxY; y += width) {
                                float sum = 0.0f;
                                int index = x + y;
                                int yOffset = width;
                                for (int i = 1; i < kwidth; i++) {
                                        sum += diffKernel[i] * (xConv[index - yOffset] - xC
onv[index + yOffset]);
                                        yOffset += width;
                                }

                                yGradient[index] = sum;
                        }

                }

                initX = kwidth;
                maxX = width - kwidth;
                initY = width * kwidth;
                maxY = width * (height - kwidth);
                for (int x = initX; x < maxX; x++) {
                        for (int y = initY; y < maxY; y += width) {
                                int index = x + y;
                                int indexN = index - width;
                                int indexS = index + width;
                                int indexW = index - 1;
                                int indexE = index + 1;
                                int indexNW = indexN - 1;
                                int indexNE = indexN + 1;
                                int indexSW = indexS - 1;
                                int indexSE = indexS + 1;

                                float xGrad = xGradient[index];
                                float yGrad = yGradient[index];
                                float gradMag = hypot(xGrad, yGrad);

                                //perform non-maximal supression
                                float nMag = hypot(xGradient[indexN], yGradient[indexN]);
                                float sMag = hypot(xGradient[indexS], yGradient[indexS]);
                                float wMag = hypot(xGradient[indexW], yGradient[indexW]);
                                float eMag = hypot(xGradient[indexE], yGradient[indexE]);
                                float neMag = hypot(xGradient[indexNE], yGradient[indexNE])
;
                                float seMag = hypot(xGradient[indexSE], yGradient[indexSE])
;
                                float swMag = hypot(xGradient[indexSW], yGradient[indexSW])
;
                                float nwMag = hypot(xGradient[indexNW], yGradient[indexNW])
;
                                float tmp;
                                /*
                                 * An explanation of what's happening here, for those who w
ant
                                 * to understand the source: This performs the "non-maximal
                                 * supression" phase of the Canny edge detection in which w
e
                                 * need to compare the gradient magnitude to that in the
                                 * direction of the gradient; only if the value is a local
                                 * maximum do we consider the point as an edge candidate.
                                 *
                                 * We need to break the comparison into a number of differe
```

```
nt
                                * cases depending on the gradient direction so that the
                                * appropriate values can be used. To avoid computing the
                                * gradient direction, we use two simple comparisons: first
 we
                                * check that the partial derivatives have the same sign (1
)
                                * and then we check which is larger (2). As a consequence,
 we
                                * have reduced the problem to one of four identical cases
that
                                * each test the central gradient magnitude against the val
ues at
                                * two points with 'identical support'; what this means is
that
                                * the geometry required to accurately interpolate the magn
itude
                                * of gradient function at those points has an identical
                                * geometry (upto right-angled-rotation/reflection).
                                *
                                * When comparing the central gradient to the two interpola
ted
                                * values, we avoid performing any divisions by multiplying
 both
                                * sides of each inequality by the greater of the two parti
al
                                * derivatives. The common comparand is stored in a tempora
ry
                                * variable (3) and reused in the mirror case (4).
                                *
                                */
                               if (xGrad * yGrad <= (float) 0 /*(1)*/
                                       ? Math.abs(xGrad) >= Math.abs(yGrad) /*(2)*/
                                               ? (tmp = Math.abs(xGrad * gradMag)) >= Math
.abs(yGrad * neMag - (xGrad + yGrad) * eMag) /*(3)*/
                                                       && tmp > Math.abs(yGrad * swMag - (
xGrad + yGrad) * wMag) /*(4)*/
                                               : (tmp = Math.abs(yGrad * gradMag)) >= Math
.abs(xGrad * neMag - (yGrad + xGrad) * nMag) /*(3)*/
                                                       && tmp > Math.abs(xGrad * swMag - (
yGrad + xGrad) * sMag) /*(4)*/
                                       : Math.abs(xGrad) >= Math.abs(yGrad) /*(2)*/
                                               ? (tmp = Math.abs(xGrad * gradMag)) >= Math
.abs(yGrad * seMag + (xGrad - yGrad) * eMag) /*(3)*/
                                                       && tmp > Math.abs(yGrad * nwMag + (
xGrad - yGrad) * wMag) /*(4)*/
                                               : (tmp = Math.abs(yGrad * gradMag)) >= Math
.abs(xGrad * seMag + (yGrad - xGrad) * sMag) /*(3)*/
                                                       && tmp > Math.abs(xGrad * nwMag + (
yGrad - xGrad) * nMag) /*(4)*/
                                       ) {
                                       magnitude[index] = gradMag >= MAGNITUDE_LIMIT ? MAG
NITUDE_MAX : (int) (MAGNITUDE_SCALE * gradMag);
                                       //NOTE: The orientation of the edge is not employed
 by this
                                       //implementation. It is a simple matter to compute
it at
                                       //this point as: Math.atan2(yGrad, xGrad);
                               } else {
                                       magnitude[index] = 0;
                               }
                       }
               }
       }

       //NOTE: It is quite feasible to replace the implementation of this method
       //with one which only loosely approximates the hypot function. I've tested
       //simple approximations such as Math.abs(x) + Math.abs(y) and they work fine.
```

```
        private float hypot(float x, float y) {
                return (float) Math.hypot(x, y);
        }

        private float gaussian(float x, float sigma) {
                return (float) Math.exp(-(x * x) / (2f * sigma * sigma));
        }

        private void performHysteresis(int low, int high) {
                //NOTE: this implementation reuses the data array to store both
                //luminance data from the image, and edge intensity from the processing.
                //This is done for memory efficiency, other implementations may wish
                //to separate these functions.
                Arrays.fill(data, 0);

                int offset = 0;
                for (int y = 0; y < height; y++) {
                        for (int x = 0; x < width; x++) {
                                if (data[offset] == 0 && magnitude[offset] >= high) {
                                        follow(x, y, offset, low);
                                }
                                offset++;
                        }
                }
        }

        private void follow(int x1, int y1, int i1, int threshold) {
                int x0 = x1 == 0 ? x1 : x1 - 1;
                int x2 = x1 == width - 1 ? x1 : x1 + 1;
                int y0 = y1 == 0 ? y1 : y1 - 1;
                int y2 = y1 == height -1 ? y1 : y1 + 1;

                data[i1] = magnitude[i1];
                for (int x = x0; x <= x2; x++) {
                        for (int y = y0; y <= y2; y++) {
                                int i2 = x + y * width;
                                if ((y != y1 || x != x1)
                                        && data[i2] == 0
                                        && magnitude[i2] >= threshold) {
                                        follow(x, y, i2, threshold);
                                        return;
                                }
                        }
                }
        }

        private void thresholdEdges() {
                for (int i = 0; i < picsize; i++) {
                        data[i] = data[i] > 0 ? -1 : 0xff000000;
                }
        }

        private int luminance(float r, float g, float b) {
                return Math.round(0.299f * r + 0.587f * g + 0.114f * b);
        }

        private void readLuminance() {
                int type = sourceImage.getType();
                if (type == BufferedImage.TYPE_INT_RGB || type == BufferedImage.TYPE_INT_AR
GB) {
                        int[] pixels = (int[]) sourceImage.getData().getDataElements(0, 0,
width, height, null);
                        for (int i = 0; i < picsize; i++) {
                                int p = pixels[i];
                                int r = (p & 0xff0000) >> 16;
                                int g = (p & 0xff00) >> 8;
                                int b = p & 0xff;
                                data[i] = luminance(r, g, b);
```

```
                        }
                } else if (type == BufferedImage.TYPE_BYTE_GRAY) {
                        byte[] pixels = (byte[]) sourceImage.getData().getDataElements(0, 0
, width, height, null);
                        for (int i = 0; i < picsize; i++) {
                                data[i] = (pixels[i] & 0xff);
                        }
                } else if (type == BufferedImage.TYPE_USHORT_GRAY) {
                        short[] pixels = (short[]) sourceImage.getData().getDataElements(0,
 0, width, height, null);
                        for (int i = 0; i < picsize; i++) {
                                data[i] = (pixels[i] & 0xffff) / 256;
                        }
                } else if (type == BufferedImage.TYPE_3BYTE_BGR) {
            byte[] pixels = (byte[]) sourceImage.getData().getDataElements(0, 0, width, hei
ght, null);
            int offset = 0;
            for (int i = 0; i < picsize; i++) {
                int b = pixels[offset++] & 0xff;
                int g = pixels[offset++] & 0xff;
                int r = pixels[offset++] & 0xff;
                data[i] = luminance(r, g, b);
            }
        } else {
                        throw new IllegalArgumentException("Unsupported image type: " + typ
e);
                }
        }

        private void normalizeContrast() {
                int[] histogram = new int[256];
                for (int i = 0; i < data.length; i++) {
                        histogram[data[i]]++;
                }
                int[] remap = new int[256];
                int sum = 0;
                int j = 0;
                for (int i = 0; i < histogram.length; i++) {
                        sum += histogram[i];
                        int target = sum*255/picsize;
                        for (int k = j+1; k <=target; k++) {
                                remap[k] = i;
                        }
                        j = target;
                }

                for (int i = 0; i < data.length; i++) {
                        data[i] = remap[data[i]];
                }
        }

        private void writeEdges(int pixels[]) {
                //NOTE: There is currently no mechanism for obtaining the edge data
                //in any other format other than an INT_ARGB type BufferedImage.
                //This may be easily remedied by providing alternative accessors.
                if (edgesImage == null) {
                        edgesImage = new BufferedImage(width, height, BufferedImage.TYPE_IN
T_ARGB);
                }
                edgesImage.getWritableTile(0, 0).setDataElements(0, 0, width, height, pixel
s);
        }

}
```

```
package equalize;

import java.awt.Color;
import java.awt.image.BufferedImage;
import java.io.IOException;

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Mapper.Context;

import utils.ArrayWritableLong;

public class HistoMapper extends Mapper<LongWritable, BufferedImage
, LongWritable , ArrayWritableLong>{

private final static LongWritable one = new LongWritable(1);

public void map(LongWritable key, BufferedImage value, Context context)
                throws IOException, InterruptedException {

        // Initialize histogram array
        LongWritable [] histogram = new LongWritable[256];
        for(int i = 0; i < histogram.length; i++){
                histogram[i] = new LongWritable();
        }

        for (int x = 0; x < value.getWidth(); x++) {
                for (int y = 0; y < value.getHeight(); y++) {
                        int rgb = value.getRGB(x,y);
                        int red = (rgb >> 16) & 0xFF;
                        int green = (rgb >> 8) & 0xFF;
                        int blue = rgb & 0xFF;
                        float hsb[] = new float[3];
                        Color.RGBtoHSB(red,green,blue,hsb);
                        int ind = (int) (255.0*hsb[2]);
                        histogram[ind].set(histogram[ind].get() + 1);
                }
        }

        context.write(one, new ArrayWritableLong(histogram));
}
}
```

```
package equalize;
import java.awt.Color;
import java.awt.image.BufferedImage;
import java.io.IOException;
import java.util.Iterator;
import java.util.StringTokenizer;

import javax.imageio.ImageIO;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

import utils.InputFormatImg;
import utils.ArrayWritableLong;

public class Equalize {

        @SuppressWarnings("deprecation")
        public static void main(String[] args) throws Exception {

                Configuration conf = new Configuration();
                String[] otherArgs = new GenericOptionsParser(conf,
                args).getRemainingArgs();
                if (otherArgs.length != 2) {
                        System.err.println("Usage: histeq <in> <out>");
                        System.exit(2);
                }
                BufferedImage img = null;
                FileSystem dfs = FileSystem.get(conf);
                Path dir = new Path(otherArgs[0]);
                FileStatus[] files = dfs.listStatus(dir);

                //String fname = null;
                conf.setInt("utils.imagerecordreader.overlapPixel", 0);
                Path fpath = null;

                for (FileStatus file: files) {
                        if (file.isDir()) continue;

                        fpath = file.getPath();
                        //fname = fpath.getName();
                        System.out.println(fpath);
                }

                Path outdir = new Path(otherArgs[1]);
                if (dfs.exists(outdir)) dfs.delete(outdir,true);

                Job job = new Job(conf, "Histogram equalization");

                job.setJarByClass(Equalize.class);
                job.setMapperClass(HistoMapper.class);
                job.setCombinerClass(HistoReducer.class);
                job.setReducerClass(HistoReducer.class);

                job.setInputFormatClass(InputFormatImg.class);
                job.setOutputKeyClass(LongWritable.class);
                job.setOutputValueClass(ArrayWritableLong.class);
```

```
                FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
                FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));

                boolean ret = job.waitForCompletion(true);

                Path reduceFile = new Path(outdir, "part-r-00000");
                FSDataInputStream fs = dfs.open(reduceFile);
                String str = null;
                float[] histogram = new float[256];
                str = fs.readLine();

                StringTokenizer tokenizer = new StringTokenizer(str);
                if (tokenizer.hasMoreTokens()) tokenizer.nextToken();

                for (int i = 0; i<histogram.length && tokenizer.hasMoreTokens(); i++)
                {
                        histogram[i] = Long.valueOf(tokenizer.nextToken()).longValue();
                        if (i > 0) histogram[i] += histogram[i-1];
                }

                Path imgpath = fpath;//new Path(dir, imgname);
                Path newimgpath = new Path(outdir, fpath.getName());
                if (dfs.exists(newimgpath)) dfs.delete(newimgpath,false);

                dfs.createNewFile(newimgpath);
                FSDataOutputStream ofs = dfs.create(newimgpath);

                fs = dfs.open(imgpath);
                img = ImageIO.read(fs);

                float pixelNum = img.getWidth()*img.getHeight();

                for (int x = 0; x < img.getWidth(); x++) {
                        for (int y = 0; y < img.getHeight(); y++) {
                                int rgb = img.getRGB(x,y);
                                int red = (rgb >> 16) & 0xFF;
                                int green = (rgb >> 8) & 0xFF;
                                int blue = rgb & 0xFF;
                                float hsb[] = new float[3];
                                Color.RGBtoHSB(red,green,blue,hsb);
                                int ind = (int) (255.0*hsb[2]);
                                int newrgb = Color.HSBtoRGB(hsb[0],hsb[1],histogram[ind]/pi
xelNum);
                                img.setRGB(x,y,newrgb);
                        }
                }

                ImageIO.write(img, "jpg", ofs);
                ofs.close();
                System.exit(ret ? 0 : 1);
        }
}
```

```
package equalize;

import java.io.IOException;
import java.util.Iterator;

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.Reducer.Context;

import utils.ArrayWritableLong;

public class HistoReducer extends Reducer<LongWritable,ArrayWritableLong
,LongWritable,ArrayWritableLong> {

public void reduce(LongWritable key, Iterable<ArrayWritableLong> values, Context context) t
hrows
    IOException, InterruptedException {

        // Initialize histogram array
        LongWritable [] histogram = new LongWritable[256];
        for(int i = 0; i < histogram.length; i++) {
                histogram[i] = new LongWritable();
        }

        // Sum the parts
        Iterator<ArrayWritableLong> it = values.iterator();
        while (it.hasNext()) {
                LongWritable[] part = (LongWritable[]) it.next().toArray();
                for(int i = 0; i < histogram.length; i++) {
                        histogram[i].set(histogram[i].get() + part[i].get());
                }
        }
        context.write(key, new ArrayWritableLong(histogram));
}
}
```

```
package sobel;

import java.awt.image.BufferedImage;
import java.io.IOException;

import javax.imageio.ImageIO;

import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.mapreduce.Reducer;

public class SobelReducer extends Reducer<LongWritable, BufferedImage,LongWritable, LongWri
table> {

        public void reduce(LongWritable key, Iterable<BufferedImage> values, Context contex
t)
                        throws IOException, InterruptedException {

                FileSystem filesys = FileSystem.get(context.getConfiguration());
                Path newimgpath = new Path(context.getWorkingDirectory(), ""+key.get());

                filesys.createNewFile(newimgpath);
                FSDataOutputStream ofs = filesys.create(newimgpath);

                BufferedImage img = values.iterator().next();

                ImageIO.write(img, "jpg", ofs);
                context.write(key, new LongWritable(1));
        }

}
```

```
package sobel;
import java.awt.event.*;
import java.awt.image.BufferedImage;
import java.io.*;

import javax.imageio.ImageIO;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class SobelEdgeDetector {

  public void process() throws IOException{

          int     i, j;
          double  Gx[][], Gy[][], G[][];


                BufferedImage inImg = getSourceImage();

                int imgWidth = inImg.getWidth();
                int imgHeight = inImg.getHeight();

                int[] pixelArr = new int[imgWidth * imgHeight];
                int[][] outArr = new int[imgWidth][imgHeight];

                inImg.getRaster().getPixels(0,0,imgWidth,imgHeight,pixelArr);

                int counter = 0;

                for(i = 0 ; i < imgWidth ; i++ )
                {
                for(j = 0 ; j < imgHeight ; j++ )
                        {

                                outArr[i][j] = pixelArr[counter];
                                counter = counter + 1;
                        }
                }


    Gx = new double[imgWidth][imgHeight];
    Gy = new double[imgWidth][imgHeight];
    G  = new double[imgWidth][imgHeight];

    for (i=0; i<imgWidth; i++) {
      for (j=0; j<imgHeight; j++) {
        if (i==0 || i==imgWidth-1 || j==0 || j==imgHeight-1)
          Gx[i][j] = Gy[i][j] = G[i][j] = 0; // Image boundary cleared
        else{
          Gx[i][j] = outArr[i+1][j-1] + 2*outArr[i+1][j] + outArr[i+1][j+1] -
          outArr[i-1][j-1] - 2*outArr[i-1][j] - outArr[i-1][j+1];
          Gy[i][j] = outArr[i-1][j+1] + 2*outArr[i][j+1] + outArr[i+1][j+1] -
          outArr[i-1][j-1] - 2*outArr[i][j-1] - outArr[i+1][j-1];
          G[i][j]  = Math.abs(Gx[i][j]) + Math.abs(Gy[i][j]);
        }
      }
    }
    counter = 0;
    for(int ii = 0 ; ii < imgWidth ; ii++ )
        {
        for(int jj = 0 ; jj < imgHeight ; jj++ )
                {
                //System.out.println(counter);

                        pixelArr[counter] = (int) G[ii][jj];
                        counter = counter + 1;
                }
```

```
        }

    BufferedImage outImg = new BufferedImage(imgWidth,imgHeight,BufferedImage.TYPE_BYTE_GRA
Y);
        outImg.getRaster().setPixels(0,0,imgWidth,imgHeight,pixelArr);
        setEdgesImage(outImg);

  }

  public SobelEdgeDetector() {

  }

  private BufferedImage sourceImage;
  private BufferedImage edgesImage;

  public BufferedImage getSourceImage() {
                return sourceImage;
        }

  public void setSourceImage(BufferedImage image) {
                sourceImage = image;
        }
  public BufferedImage getEdgesImage() {
                return edgesImage;
        }

        public void setEdgesImage(BufferedImage edgesImage) {
                this.edgesImage = edgesImage;
        }

}
```

```
package sobel;

import java.awt.Color;
import java.awt.image.BufferedImage;
import java.io.IOException;

import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.mapreduce.Mapper;
import java.util.Arrays;

import javax.imageio.ImageIO;

public class SobelMapper extends Mapper<LongWritable, BufferedImage, LongWritable, Buffered
Image>{

        public void map(LongWritable key, BufferedImage value, Context context)
                        throws IOException, InterruptedException {

                System.out.println("Map Phase started");

                //create the detector
                SobelEdgeDetector edgeDetector = new SobelEdgeDetector();

                //adjust its parameters as desired


                //apply it to an image
                edgeDetector.setSourceImage(value);
                edgeDetector.process();

                System.out.println("Edge Detected chunk " + key.get());

                BufferedImage edges = edgeDetector.getEdgesImage();

                if(edges == null) {
                        System.out.println("edge detect made a null");
                }

                //context.write(key, edges);

                FileSystem dfs = FileSystem.get(context.getConfiguration());
                Path newimgpath = new Path(context.getWorkingDirectory(), context.getJobID(
).toString()+"/"+key.get());
                dfs.createNewFile(newimgpath);
                FSDataOutputStream ofs = dfs.create(newimgpath);
                ImageIO.write(edges, "jpg", ofs);
        }
}
```

```
package sobel;


import java.awt.image.BufferedImage;
import java.util.Iterator;


import javax.imageio.ImageIO;
import javax.imageio.ImageReader;
import javax.imageio.stream.MemoryCacheImageInputStream;

import utils.*;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class SobelFilter {

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

                Configuration conf = new Configuration();
                String[] otherArgs = new GenericOptionsParser(conf,
                                args).getRemainingArgs();
                if (otherArgs.length != 2) {
                        System.err.println("Usage: SobelFilter <in> <out>");
                        System.exit(2);
                }

                BufferedImage img = null;
                FileSystem dfs = FileSystem.get(conf);
                Path dir = new Path(otherArgs[0]);
                FileStatus[] files = dfs.listStatus(dir);

                //String fname = null;

                conf.setInt("overlapPixel", 64);
                int overlapPixel = ImgRecordReader.overlapPixel;
                System.out.println(overlapPixel);
                Path fpath = null;
                for (FileStatus file: files) {
                        if (file.isDir()) continue;
                         fpath = file.getPath();

                        System.out.println(fpath);
                }

                Path outdir = new Path(otherArgs[1]);
                if (dfs.exists(outdir)) dfs.delete(outdir,true);
                Path workdir = dfs.getWorkingDirectory();

                Job job = new Job(conf, "Sobel Edge detection");
                job.setJarByClass(SobelFilter.class);
                job.setMapperClass(SobelMapper.class);

                job.setReducerClass(SobelReducer.class);

                job.setInputFormatClass(InputFormatImg.class);


                job.setOutputKeyClass(LongWritable.class);
```

```
		job.setOutputValueClass(LongWritable.class);

		FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
		FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));


		boolean ret = job.waitForCompletion(true);
		String s = job.getTrackingURL();
		Path tmpdir = new Path(workdir, s.substring(s.indexOf("jobid")+6));

		int i = 0;
		Path iPath = new Path(tmpdir,""+i);
		int currX =0, currY = 0;
		int sizePixel = ImgRecordReader.sizePixel;
		int border = 16;

		FSDataInputStream fs = null;
		MemoryCacheImageInputStream image = new MemoryCacheImageInputStream(dfs.ope
n(fpath));
		Iterator<ImageReader> readers = ImageIO.getImageReaders(image);
		ImageReader reader = (ImageReader) readers.next();
		reader.setInput(image);
		int imgwidth = 0, imgheight = 0;
		imgwidth = reader.getWidth(0);
		imgheight = reader.getHeight(0);


		img = new BufferedImage(imgwidth, imgheight, BufferedImage.TYPE_INT_RGB);
		if(imgwidth*imgheight <= sizePixel*sizePixel) {
			fs = dfs.open(iPath);
			img = ImageIO.read(fs);
			iPath = null;
		}
		while(iPath != null && dfs.exists(iPath)) {
			int x = currX, y = currY;
			currX += sizePixel;
			if (currX >= imgwidth) {
				currX = 0;
				currY += sizePixel;
			}

			fs = dfs.open(iPath);
			BufferedImage window = ImageIO.read(fs);
			int width = window.getWidth() - border*2;
			int height = window.getHeight() - border*2;

			img.setRGB(x+border, y+border, width, height,
					window.getRGB(border,border, width, height, null, 0
, width),
					0, width);
			fs.close();
			i++;
			iPath = new Path(tmpdir,""+i);
		}
		Path newimgpath = new Path(outdir, fpath.getName());

		if (dfs.exists(newimgpath)) dfs.delete(newimgpath,false);
		dfs.createNewFile(newimgpath);
		FSDataOutputStream ofs = dfs.create(newimgpath);
		ImageIO.write(img, "JPG", ofs);
		ofs.close();
		dfs.delete(tmpdir, true);
		System.exit(ret ? 0 : 1);
	}
}
```

```
package utils;
import java.io.IOException;
import java.util.Iterator;
import java.awt.Rectangle;
import java.awt.image.BufferedImage;
import javax.imageio.ImageIO;
import javax.imageio.ImageReadParam;
import javax.imageio.ImageReader;
import javax.imageio.stream.MemoryCacheImageInputStream;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

public class ImgRecordReader extends RecordReader<LongWritable, BufferedImage> {
        // Image information

        private String fileName = null;
        private ImageReader reader = null;

        private MemoryCacheImageInputStream  image = null;
        // Key/Value pair
        private LongWritable key = null;
        private BufferedImage value = null;

        // Configuration parameters
        // By default use percentage for splitting
        boolean splittusingPixel = false;

        // splits
        int totalXSplits = 0;
        int totalYSplits = 0;
        int loctX = 0;
        int loctY = 0;
        int currentSplit = 0;
        int imgwidth = 0;
        int imgheight = 0;


        public int getImgheight() {
                return imgheight;
        }

        static int overlapPercent = 0;
        static int sizePercent = 0;



        @Override
        public void close() throws IOException {
                //nothing here
        }

        public int getImgwidth() {
                return imgwidth;
        }

        @Override
        public LongWritable getCurrentKey() throws
            IOException, InterruptedException {
                return key;
        }
```

```
	@Override
	public BufferedImage getCurrentValue() throws
	    IOException, InterruptedException {
	        return value;
	}
	public static int overlapPixel = 0;

	@Override
	public float getProgress()
			throws IOException, InterruptedException {

		if((float)(totalXSplits * totalYSplits) == 0) {
			return 0;
		}
		float retval = (float)currentSplit / (float)(totalXSplits * totalYSplits);

		if(retval > 1) {
			return 0;
		}
		return retval;
	}
	public static int sizePixel = 1000;
	@Override
	public void initialize(InputSplit genericSplit, TaskAttemptContext context) throws
IOException, InterruptedException {
		// Get file split

		FileSplit chunk = (FileSplit) genericSplit;
		Configuration conf = context.getConfiguration();
		// Ensure that value is not negative
		overlapPixel = conf.getInt("overlapPixel", 0);
		if(overlapPixel < 0){
			overlapPixel = 0;
		}

		// Open the file
		Path file = chunk.getPath();
		FileSystem fs = file.getFileSystem(conf);
		FSDataInputStream fileIn = fs.open(chunk.getPath());
		image = new MemoryCacheImageInputStream(fileIn);

		// Get filename to use as key
		fileName = chunk.getPath().getName().toString();

		Iterator<ImageReader> imgrdr = ImageIO.getImageReaders(image);
		reader = (ImageReader) imgrdr.next();
		reader.setInput(image);
		imgwidth = reader.getWidth(0);
		imgheight = reader.getHeight(0);
		findTotalSplits();

	}

	@Override
	public boolean nextKeyValue() throws IOException, InterruptedException {
		if (loctY < imgheight && fileName != null) {
			key = new LongWritable(currentSplit);//new Text(fileName);

			if(imgwidth*imgheight <= sizePixel*sizePixel) {
				Rectangle rect = new Rectangle(0, 0, imgwidth, imgheight);
				ImageReadParam irp = new ImageReadParam();
				irp.setSourceRegion(rect);
				value = reader.read(0,irp);
				loctY = imgheight;
			}
			else {
				value = getChunk();
```

```
                        }
                        currentSplit += 1;
                        return true;
                }
                return false;
        }

        private BufferedImage getChunk(){
                int x = loctX, y = loctY;
                loctX += sizePixel;

                if (loctX >= imgwidth) {
                        loctX = 0;
                        loctY += sizePixel;
                }
                int width = Math.min(sizePixel + overlapPixel, imgwidth-x);
                int height = Math.min(sizePixel + overlapPixel, imgheight-y);

                Rectangle rect = new Rectangle(x,y,width,height);
                ImageReadParam irp = new ImageReadParam();
                irp.setSourceRegion(rect);

                try {
                        return reader.read(0,irp);
                }
                catch (IOException e) {
                        e.printStackTrace();
                        return null;
                }
        }

        private void findTotalSplits(){
                try {
                        totalXSplits = (int)Math.ceil(reader.getWidth(0) / Math.min(sizePix
el, reader.getWidth(0)));
                        totalYSplits = (int)Math.ceil(reader.getHeight(0) / Math.min(sizePi
xel, reader.getHeight(0)));
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }

}
```

```
package utils;
import org.apache.hadoop.io.ArrayWritable;
import org.apache.hadoop.io.LongWritable;

public class ArrayWritableLong extends ArrayWritable {
        //Helper class to write a set of values in an array together

        public ArrayWritableLong() {
                super(LongWritable.class);
        }

        public ArrayWritableLong(LongWritable[] values) {
                super(LongWritable.class, values);
        }

        @Override
        public String toString() {
                String [] strings = toStrings();
                String str = "";
                for (int i = 0; i < strings.length; i++) {
                        str +=  strings[i] + " ";
                }
                return str;
        }
}
```

```
package utils;
import java.io.IOException;
import java.awt.image.BufferedImage;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.JobContext;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;


public class InputFormatImg extends FileInputFormat<LongWritable, BufferedImage> {

        @Override
        public RecordReader<LongWritable, BufferedImage> createRecordReader(InputSplit spli
t,
                TaskAttemptContext context) throws IOException,
                InterruptedException {
                        return new ImgRecordReader();
                }

        @Override
        protected boolean isSplitable(JobContext context, Path file) {
                return false;
        }
}
```

```
package gaussian_blur;

import java.awt.Color;
import java.awt.image.BufferedImage;
import java.io.IOException;

import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.mapreduce.Mapper;
import java.util.Arrays;

import javax.imageio.ImageIO;

public class GaussianMapper extends Mapper<LongWritable, BufferedImage, LongWritable, Buffe
redImage>{

        public void map(LongWritable key, BufferedImage value, Context context)
                        throws IOException, InterruptedException {

                System.out.println("map started");

                //create the detector
                GaussianBlur filter = new GaussianBlur();

                //adjust its parameters as desired


                //apply it to an image
                filter.setSourceImage(value);
                filter.process();

                System.out.println("Edge Detected chunk " + key.get());

                BufferedImage edges = filter.getEdgesImage();

                if(edges == null) {
                        System.out.println("edge detect made a null");
                }

                //context.write(key, edges);

                FileSystem dfs = FileSystem.get(context.getConfiguration());
                Path newimgpath = new Path(context.getWorkingDirectory(), context.getJobID(
).toString()+"/"+key.get());
                dfs.createNewFile(newimgpath);
                FSDataOutputStream ofs = dfs.create(newimgpath);
                ImageIO.write(edges, "jpg", ofs);
        }
}
```

```
//This will be done like Canny. End of Story
package gaussian_blur;


import java.awt.image.BufferedImage;
import java.util.Iterator;


import javax.imageio.ImageIO;
import javax.imageio.ImageReader;
import javax.imageio.stream.MemoryCacheImageInputStream;

import utils.*;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class GaussianMain {

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

                Configuration conf = new Configuration();

                String[] otherArgs = new GenericOptionsParser(conf,
                                args).getRemainingArgs();

                if (otherArgs.length != 2) {
                        System.err.println("Usage: SobelFilter <in> <out>");
                        System.exit(2);
                }

                BufferedImage img = null;
                FileSystem dfs = FileSystem.get(conf);
                Path dir = new Path(otherArgs[0]);
                FileStatus[] files = dfs.listStatus(dir);

                //String fname = null;

                conf.setInt("overlapPixel", 64);
                int overlapPixel = ImgRecordReader.overlapPixel;
                System.out.println(overlapPixel);
                Path fpath = null;

                for (FileStatus file: files) {
                        if (file.isDir()) continue;
                         fpath = file.getPath();

                        System.out.println(fpath);
                }

                Path outdir = new Path(otherArgs[1]);
                if (dfs.exists(outdir)) dfs.delete(outdir,true);
                Path workdir = dfs.getWorkingDirectory();

                Job job = new Job(conf, "Sobel Edge detection");
                job.setJarByClass(GaussianMain.class);
                job.setMapperClass(GaussianMapper.class);

                job.setReducerClass(GaussianReducer.class);
```

```
		job.setInputFormatClass(InputFormatImg.class);


		job.setOutputKeyClass(LongWritable.class);
		job.setOutputValueClass(LongWritable.class);

		FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
		FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));


		boolean ret = job.waitForCompletion(true);
		String s = job.getTrackingURL();
		Path tmpdir = new Path(workdir, s.substring(s.indexOf("jobid")+6));

		int i = 0;
		Path iPath = new Path(tmpdir,""+i);
		int currX =0, currY = 0;
		int sizePixel = ImgRecordReader.sizePixel;
		int border = 16;

		FSDataInputStream fs = null;
		MemoryCacheImageInputStream image = new MemoryCacheImageInputStream(dfs.ope
n(fpath));
		Iterator<ImageReader> readers = ImageIO.getImageReaders(image);
		ImageReader reader = (ImageReader) readers.next();
		reader.setInput(image);
		int imgwidth = 0, imgheight = 0;
		imgwidth = reader.getWidth(0);
		imgheight = reader.getHeight(0);


		img = new BufferedImage(imgwidth, imgheight, BufferedImage.TYPE_INT_RGB);
		if(imgwidth*imgheight <= sizePixel*sizePixel) {
			fs = dfs.open(iPath);
			img = ImageIO.read(fs);
			iPath = null;
		}
		while(iPath != null && dfs.exists(iPath)) {
			int x = currX, y = currY;
			currX += sizePixel;
			if (currX >= imgwidth) {
				currX = 0;
				currY += sizePixel;
			}

			fs = dfs.open(iPath);
			BufferedImage window = ImageIO.read(fs);
			int width = window.getWidth() - border*2;
			int height = window.getHeight() - border*2;

			img.setRGB(x+border, y+border, width, height,
					window.getRGB(border,border, width, height, null, 0
, width),
					0, width);
			fs.close();
			i++;
			iPath = new Path(tmpdir,""+i);
		}
		Path newimgpath = new Path(outdir, fpath.getName());

		if (dfs.exists(newimgpath)) dfs.delete(newimgpath,false);
		dfs.createNewFile(newimgpath);
		FSDataOutputStream ofs = dfs.create(newimgpath);
		ImageIO.write(img, "jpg", ofs);
		ofs.close();
		dfs.delete(tmpdir, true);
		System.exit(ret ? 0 : 1);
	}
```

```
}
```

```
package gaussian_blur;

import java.awt.image.BufferedImage;
import java.io.IOException;

import javax.imageio.ImageIO;

import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.mapreduce.Reducer;

public class GaussianReducer extends Reducer<LongWritable, BufferedImage,LongWritable, Long
Writable> {

        public void reduce(LongWritable key, Iterable<BufferedImage> values, Context contex
t)
                        throws IOException, InterruptedException {

                FileSystem dfs = FileSystem.get(context.getConfiguration());
                Path newimgpath = new Path(context.getWorkingDirectory(), ""+key.get());
                dfs.createNewFile(newimgpath);
                FSDataOutputStream ofs = dfs.create(newimgpath);
                BufferedImage img = values.iterator().next();
                ImageIO.write(img, "jpg", ofs);
                context.write(key, new LongWritable(1));
        }

}
```

```
package gaussian_blur;

import java.awt.Color;
import java.awt.image.BufferedImage;

/**
 * @author Aish
 */

public class GaussianBlur {

        private double sigma;
        private BufferedImage sourceImage;
          private BufferedImage filtredImage;

          public BufferedImage getSourceImage() {
                        return sourceImage;
                }

          public void setSourceImage(BufferedImage image) {
                        sourceImage = image;
                }
          public BufferedImage getEdgesImage() {
                        return filtredImage;
                }

                public void setEdgesImage(BufferedImage edgesImage) {
                        this.filtredImage = edgesImage;
                }


    public void gaussianBlur(BufferedImage image,double sigma) {

        int height = image.getHeight(null);
        int width = image.getWidth(null);

        BufferedImage tempImage = new BufferedImage(width, height,
                BufferedImage.TYPE_INT_RGB);

        BufferedImage filteredImage = new BufferedImage(width, height,
                BufferedImage.TYPE_INT_RGB);

        //--->>
        int n = (int) (6 * sigma + 1);

        double[] window = new double[n];
        double s2 = 2 * sigma * sigma;

        window[(n - 1) / 2] = 1;
        for (int i = 0; i < (n - 1) / 2; i++) {
            window[i] = Math.exp((double) (-i * i) / (double) s2);
            window[n - i - 1] = window[i];
        }

        //--->>
        for (int i = 0; i < width; i++) {
            for (int j = 0; j < height; j++) {
                double sum = 0;
                double[] colorRgbArray = new double[]{0, 0, 0};
                for (int k = 0; k < window.length; k++) {
                    int l = i + k - (n - 1) / 2;
                    if (l >= 0 && l < width) {
                        Color imageColor = new Color(image.getRGB(l, j));
                        colorRgbArray[0] = colorRgbArray[0] + imageColor.getRed() * window[
k];
                        colorRgbArray[1] = colorRgbArray[1] + imageColor.getGreen() * windo
w[k];
```

```
                        colorRgbArray[2] = colorRgbArray[2] + imageColor.getBlue() * window
[k];
                        sum += window[k];
                    }
                }
                for (int t = 0; t < 3; t++) {
                    colorRgbArray[t] = colorRgbArray[t] / sum;
                }
                Color tmpColor = new Color((int) colorRgbArray[0], (int) colorRgbArray[1],
(int) colorRgbArray[2]);
                tempImage.setRGB(i, j, tmpColor.getRGB());
            }
        }

        //--->>
        for (int i = 0; i < width; i++) {
            for (int j = 0; j < height; j++) {

                double sum = 0;
                double[] colorRgbArray = new double[]{0, 0, 0};

                for (int k = 0; k < window.length; k++) {
                    int l = j + k - (n - 1) / 2;
                    if (l >= 0 && l < height) {
                        Color imageColor = new Color(tempImage.getRGB(i, l));
                        colorRgbArray[0] = colorRgbArray[0] + imageColor.getRed() * window[
k];
                        colorRgbArray[1] = colorRgbArray[1] + imageColor.getGreen() * windo
w[k];
                        colorRgbArray[2] = colorRgbArray[2] + imageColor.getBlue() * window
[k];
                        sum += window[k];
                    }
                }

                for (int t = 0; t < 3; t++) {
                    colorRgbArray[t] = colorRgbArray[t] / sum;
                }

                Color tmpColor = new Color((int) colorRgbArray[0], (int) colorRgbArray[1],
(int) colorRgbArray[2]);
                filteredImage.setRGB(i, j, tmpColor.getRGB());
            }
        }

        //return filteredImage;
        setEdgesImage(filteredImage);
    }

    public void process() {
        this.gaussianBlur(sourceImage, sigma);
    }
}
```