CS214 Project 2021:
Animal spot detection using cellular automata

Last updated: March 1, 2021

Due dates

Phase	Date
Phase 1	10 May 2021
Phase 1 & 2	14 June 2021
Demo	18 June 2021

Project description

In this project, you must implement a program in Java that, given an image of a cheetah, recognise and count the number of spots on the cheetah. The project will be assessed in two separate phases: the first phase involves grey-scaling, noise removal and edge detection using cellular automata, while the second phase involves the counting of the spots, based on spot detection using masks of various sizes.

For the first part of the project, an article is provided – read it in detail!

Rough marking scheme

- 75% – Test cases.
- 10% – Code style & Javadoc comments.
- 15% – General impression / correctness of implementation (including bonus marks for effort exceeding the specification).

Please see the detailed marking sheet for more details. Note that a mark of 100% is the maximum you can obtain, regardless of bonus marks awarded.

Assessment

Note that all marking will be done using scripts. It is therefore necessary that your project can be compiled and run from the command line. If it does not compile and run from the command line, you will receive zero for your project.

The purpose of the project is to implement a solution using cellular automata and later masks, as described in the article and in the Specification section below. While there may be many alternate ways of solving this problem, the focus of this project is to do it in the prescribed manner. As such, **your project MUST use the cellular automaton approach in order to qualify for any marks, it must be written in Java, and you may not use any external image processing libraries (such as OpenCV)**.
When marking the project, various aspects and components will be tested separately. As such, the project must be able to run in different modes.

Your project submission must include a `Animal.java` file. To compile your program, we will run

```
javac Animal.java
```

To run the program, we will run

```
java Animal <mode> <filename> [parameters]
```

where `mode`, `filename` and `parameters` are as described in the Specification below.

The output specification must be followed EXACTLY to facilitate script marking. **Also ensure that nothing except for the expected output/error messages is printed to standard output/error**.

Specification

The goal of this project is to be able to take in an image of a cheetah, and get an approximate count of the number of spots in the image. Naturally, there is a fair amount of image processing before you can reach that point. Phase 1 is focused on preparing the image and doing edge detection, while phase 2 is focused on the spot counting.

This project is divided into two distinct phases, and four different modes:

Key	Mode	Phase
0	Grey-scaling	1
1	Noise reduction	1
2	Edge detection	1
3	Spot detection & Counting	2

For full details of the mark allocation for each mode and phase, please see the detailed marking sheet. More detail on what input the modes require is provided later.

With regards to the usage of libraries: you are allowed to use the classes provided by Sedgewick & Wayne (the Standard libraries such as `StdIn` or `Picture`) which you have seen in class, as well as the Java libraries such as `Image` and `BufferedImage` if you wish to be more adventurous. However, again, the cellular automata approach is required as described in this project specification, and no external image processing libraries may be used.

Note: All output images must be saved to a `PNG` extension. This is crucial to make sure that the images produce close enough output to the expected output.

Git repository

Everyone will be assigned a git repository for the project. This is where you will do everything related to the project. An email should be sent to your student number once you have been granted access to your repository.

A handy guide to follow to clone and set up your repository once received: Git Beginners. Make sure you follow it in detail.

Once your git repository has been set up, ensure that you follow the given directory structure in the root of your repository:

- `src`: the directory where all of your Java files will be contained. This includes `Animal.java` and any other files needed for your program to run
- `bin`: the directory where the compiled Java `.class` files go. **These may not be pushed to the repository – if you do, you will be penalised**
- `out`: the directory where you will save any output produced by your program (mainly the image files)
- `tests`: the directory containing some of the input test files, their output files and the commands to run to get the output files
- `testscript.py`: the marking script you can use to test your program

All of these directories/files must be present in your repository's main directory for test cases to be run.

Testing your own work

A marking script will be provided with your project repository. This script is the same script that will be used for the final evaluation. However, only a small subset of all test cases will be provided for each mode. This script can be manually run if you wish to check that your implementation will pass the provided few test cases. This is in no way an accurate description of your final mark (since it is only a small subset of the test cases). However, it is valuable to confirm that your input and output formats are correct, as well as that you followed the specifications properly.

A small subset of all the test cases will be provided for you to test your program. The directory containing these tests (that is, `tests`) will follow the following structure:

- `ed`: contains output files for edge detection
- `gs`: contains output files for grey-scaling
- `input`: contains the input files for some of the tests
- `nr`: contains output files for noise reduction
- `sd`: contains output files for spot detection

Within `ed`, `gs`, `nr` and `sd` there is a file called `values.in` which contains the information to run on the input to manually produce the output for your program, which you can then compare with the provided output files.

Output naming convention

Each mode will require saving an image as output. See the table below for the output names required for each mode. This is essential as it is required by the marking script, so not following this means we cannot mark your submission. You simply need to add the extension to the end of the input file name.

All files must be saved in the `/out/` directory.

All images must be saved as a `.png` file.

key	Mode	Name extension
0	Grey-scale	_GS
1	Noise Reduction	_NR
2	Edge Detection	_ED
3	Spot Detection	_SD

Example

If you are given the input file `cheetah.png`, then the output for the grey-scale must be `cheetah_GS.png`, noise reduction `cheetah_NR.png`, the edge detection image `cheetah_ED.png` and the spot detection `cheetah_SD.png`.

Style

You are expected to follow the Google Java style guide. Your project will be run through a style checker, which will contribute to part of the mark. This style checker will be run every time you commit to your repository.

You will also be required to document your program using the JavaDoc style and format.

GUI

For demo purposes, you will be required to make a GUI. This is to show the output of your program to the demi marking your project. The format and look of the GUI is up to you, but at the very least have options to look at the following (given the original image):

- The grey-scaled version of the given image
- The image, but with noise removed
- The image, but with edge detection applied to it
- A image displaying the spots of the cheetah you detected, as well displaying the number of spots

Feel free to add in modes for these.

Note: The GUI does not specifically count marks, so do not spend much time on making a GUI.

Phase 1

Phase 1 focuses on manipulating the image to get it into a usable state for the spot detection. The process will be as follows: First, you will need to grey-scale the image. Once the image is grey-scaled, you will then use noise reduction on the image, and finally apply edge detection to find the outline of the image.

Note that each mode builds on the previous one, so in order to do edge detection, you MUST first grey-scale the image, then do noise reduction, then do edge detection. Likewise, in phase 2 it is assumed that your phase 1 is working perfectly.

Hint: Since all modes take an image as input, and require an image as output, your first priority should be to ensure you can open an image and save it again with a different name.

Mode 0: Grey-scaling

For this mode you will be provided with an input image which you must grey-scale. The grey-scale method to follow is the `weighted grey-scale method`, Where R, G and B represent the red, green and blue components of the pixel:

```
Greyscale = 0.299*R + 0.587*G + 0.114*B
```

Note that this is the grey-scaling method used in the marking scripts, so using other methods is not allowed.

Input

```
java Animal 0 <filename>
```

You can look at the provided `values.in` file for `gs` for a better understanding.

Output

The given image, but grey-scaled using the `weighted grey-scale method`. **Note** that you **must** follow the given naming convention, as well as use the chosen grey-scaling method.

Mode 1: Noise reduction

This section is based on the information given in the article, `PopoviciCAimageProcessing.pdf`, under "The CA Model for Filtering Digital Images". You will need to use the previous section as a building block for this section, that is, an image must be grey-scaled before noise reduction is applied to the image. The test cases will be based on this fact.

Be sure to follow this article in detail!

Input

```
java Animal 1 <filename>
```

You can look at the provided `values.in` file for `nr` for a better understanding.

Output

The given image with noise reduced in the image.

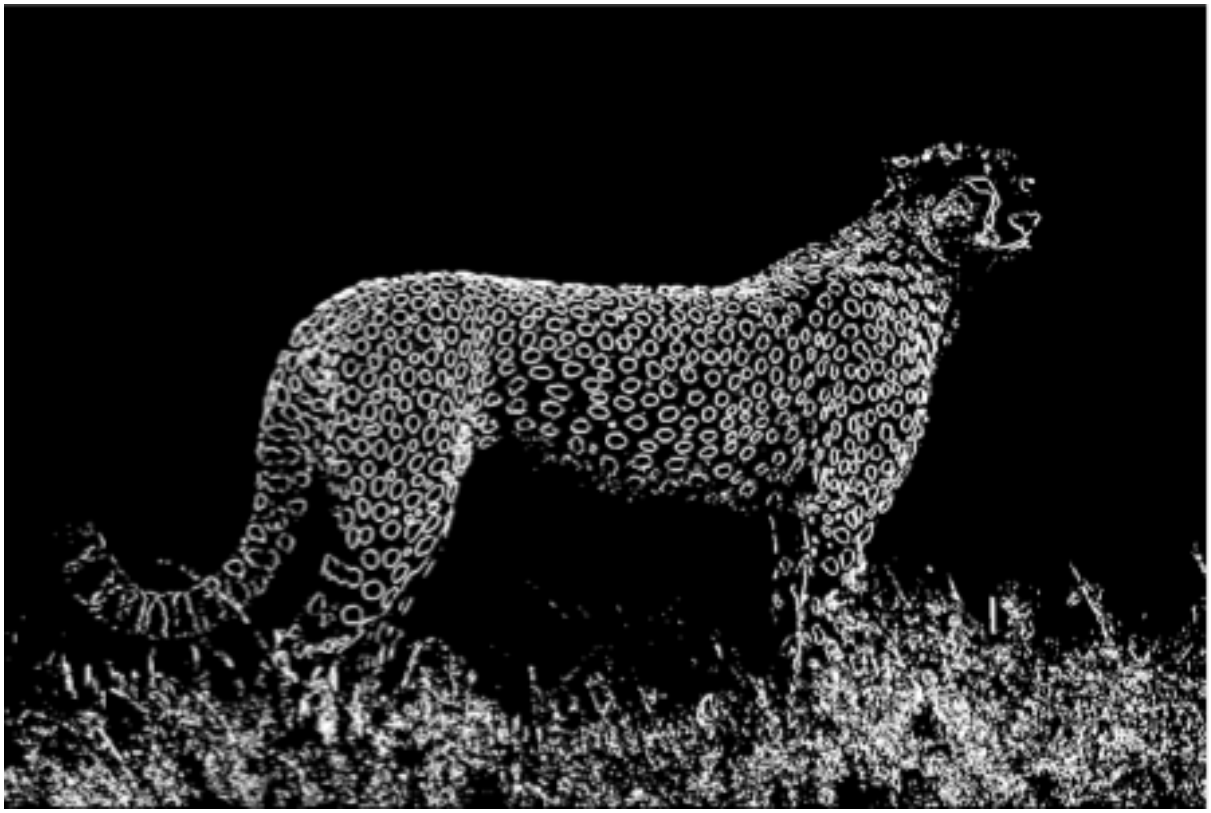
Note: Due to the nature of image processing, some tiny variation is likely to occur. Thus the output images do not need to be exactly the same as the given noise reduced images. Instead, when compared to the output images, it must be within a certain (very high) percentage similarity. Use the provided test cases as a guide: if your implementation can pass those, then you should be fine. However, if it cannot pass the given test cases, you should assume that your code is incorrect.

Mode 2: Edge detection

This section is based on the information given in the article, `PopoviciCAimageProcessing.pdf`, under "The CA Model for Border Detection in Digital Images". Here, the current pixel will be set to either *0* or s depending on the given conditions. The interpretation we want you to follow is:

- **For edges**: WHITE
- **For non-edges**: BLACK

A typical output after edge detection might look like this:



Input

```
java Animal 2 <filename> <epsilon>
```

The epsilon value provided corresponds to the epsilon value used in Popovici's article. This will be an integer between 0 and 255 (corresponding to a grey-scale value). Each test case will provide an epsilon value. You can look at the provided `values.in` file for `ed` as well as the provided test scripts output for some examples.

Output

After applying the edge detection to the image, a black and white image will be returned. This image will indicate where all the edges are for the given image. This is crucial for the second phase of the project.

Error handling

Some error handling will be required. That is, a few test cases will have invalid input parameters and an error message must be sent to standard **error** (not standard output). The errors below are listed from highest priority to lowest priority – that is, if the mode is a string and not an integer, then `invalid argument type` must be displayed, and not `invalid mode`.

Here is the list of errors that must be catered for as well as their expected outputs:

Invalid number of arguments

If the incorrect number of arguments is given (for example, mode with no file; or using edge detection with no epsilon), then the following error must be displayed:

```
ERROR: invalid number of arguments
```

This error message must be prioritised over any other error message.

Invalid argument type

If an argument is of the wrong TYPE (for example, a mode of "abc" is given, or an epsilon of "hello"), then this error must be displayed:

```
ERROR: invalid argument type
```

Invalid mode

If the mode is not one of the modes mentioned in the table above, then display the error message:

```
ERROR: invalid mode
```

Invalid epsilon

Since epsilon is a value based on the RGB value (0 to 255), any value outside this range is invalid. Then the error message

```
ERROR: invalid epsilon
```

must be displayed.

Invalid file

If the file name is given, but the file cannot be opened or does not exist, the following error must be displayed:

```
ERROR: invalid or missing file
```

Phase 2

Now that you have an edge outline of the animal, the final part of the project is to analyse this "edge-map" to find as many spots as possible. To find the spots, you should create a small image of a spot, called a mask. Then you must "move" the mask over the image, checking each time whether the mask and the current image area that it is covering, are similar. This phase will consist of only one mode.

Mode 3: Spot detection

The spot detection in this project must be implemented by using masks of various sizes. This consists of two parts: the creation of the masks, and then moving the masks over the image.

Mask creation

Since the creation of the masks themselves is somewhat of an art, we have given you some pseudo-code for the creation of a mask:

```
create_spot_mask(radius, width, delta) returns 2D integer array:
    xx, yy, circle, donut, mask are all 2D arrays
    for i from 0 to 2*radius+1:
        for j from 0 to 2*radius+1:
            xx[i][j] = i
            yy[i][j] = j
    circle = (xx - radius)^2 + (yy - radius)^2
    donut = (circle < ((radius-delta)^2 + width)) &
            (circle > ((radius-delta)^2 - width))
    mask = 255 where donut is true, else 0 where it is false
    return mask
```

Feel free to optimise the mask pseudocode. However, do make sure to confirm that your results are still the same as with the original pseudo-code implementation.

Mask application

Now that you have a mask of a fixed size, you will need to iterate over the image, and for each pixel, decide if the rectangular area created around that pixel is "close enough" to the required mask to be considered a spot. If it is, save it. Repeat this for all pixels in the image.

Basic pseudo-code for this idea could be:

```
find_spots(edges, spots, mask):
    for every x value in the edges width:
        for every y value in the edges height:
            Create an image block of the width and height of the mask, with
            this pixel being the center of the square.
            copy the values from the edges map at these coordinates
            into this image block.

            Find the max pixel value in this block.
            if max is 0, continue to next y coordinate.

            for every pixel in this block, normalise it
                (i.e. set its new value to its current value * 255.0/max)

            Then, for every pixel in the image block:
                take the absolute value of the difference between
                the pixel's value and the pixel at the same coordinates
                in the mask. Sum all these differences together.

            if the sum of differences is less than the diff for this radius,
            then it is a spot and add this image block to the spots image,
            and increase the count by 1 - BUT ONLY IF this spot was not
            already counted and added to the spots image prior

    return spots, and the count
```

It is very important that you do not re-count spots.

Hint: This pseudo-code deserves careful consideration, as it does not necessarily translate into working code if you are not careful. See if you can spot the potential run-time error.

However, different spots in the same image – and especially in different images – will vary. To combat this, you will be required to create multiple masks of various sizes and repeat this procedure for each of them, and saving all spots found across all runs. To keep the results similar amongst all different implementations (so that marking is possible), we will provide a list of all configurations. You MUST implement all of them. Two of the input parameters will specify upper and lower bounds for the radius, and you will be expected to use masks of all radii between those sizes.

In the interest of speed, we will limit this to a minimum radius of four, and a maximum radius of 11. Running these masks (radii 4 to 11) over a larger image is slow, so don't be alarmed if it takes approximately 5-10 seconds to process. We have provided all the details for each mask in the table below. These are the configurations that will be used by the marking scripts:

radius	width	delta	difference
4	6	0	4800
5	9	1	6625
6	12	1	11000
7	15	1	15000
8	18	1	19000
9	21	1	23000
10	24	2	28000
11	27	2	35000

Once the spots have been found and counted, you will be expected to save an image of the spots discovered, as well as print out the total number of spots to the standard output. Once again, this will be marked by approximation due to the nature of the project, so as long as you can pass the given subset of the test cases, you are on the right track.
It is very important that you follow the implementation as described in the specification closely. If you wish to experiment in order to show off something interesting in the demo, go for it – but make sure that you can toggle it on and off. If you wish to experiment with the differences or parameters to show a better implementation for the demo, then please make sure that you can easily disable it for the final submission (and enable it again for the demo).

Input

```
java Animal 3 <filename> <epsilon> <lower_limit> <upper_limit>
```

The epsilon value is used for the edge detection, and follows the same limits as in mode 2. The lower limit is the smallest mask radius to begin with, and the upper limit is the maximum mask radius to end with. For example,

```
java Animal 3 cheetah.png 50 4 7
```

will create masks of radius 4, 5, 6 and 7 as per the specifications in the table above, and will run over the image using the mask of radius 4 to find spots, then mask of radius 5, and so on, until all four masks have been used.

Output

There are two expected outputs. The first is the image containing all spots discovered using the masks, named in accordance with the table above (`InputName_SD.png`).

The second is the total number of spots found displayed to the standard output of the terminal.

Example

If the file `cheetah5.png` was used as input, with epsilon 50 and radius 4 to 11:

```
java Animal 3 cheetah5.png 50 4 11
```

and 20 spots were found, then

```
20
```

would be displayed to the standard output, and the image of spots found would be saved to the file `cheetah5_SD.png` in the `out` directory.

You can look at the provided `values.in` file in the `sd` directory, as well as the provided test scripts output, for additional examples.