University of Cape Town
Department of Computer Science

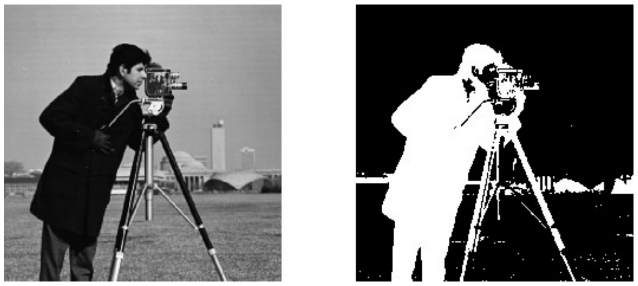
CSC3022F

Assignment 3 - Image processing with connected components

March 2025

In the area of image processing, we often threshold the intensity of pixels in a greyscale image to produce a new "binary" image which has foreground (intensity=255) pixels corresponding to the object(s) of interest (inputs >= threshold value), and background pixels (intensity=0), corresponding to image parts which are not of interest (lower than threshold value input). This serves as the first stage to more complex processing to allow more accurate detection of objects and allows us to reason about object sizes and to reject erroneous object detections in an image (or video).

For this assignment, you are given a greyscale input image (PGM) and asked to provide an image processing class that can extract all the connected components for the image, given some user-supplied threshold. A connected component is a collection of image pixels which 'touch' each other. There are two categories of connectedness — but here we are concerned only with 4-connected pixels. These are foreground pixels which are connected to each other if they have a foreground neighbour to the N, E, S or W (any of these cases means they are connected). An approach to extracting these is provided later in the description. In the PGM image below, a particular intensity threshold has resulted in the binary image on the right. *Note that here we have (for illustration purposes) kept values BELOW the threshold – in your case, always keep values >= threshold.* You can see that most of the retained pixels are connected in one large component that includes the cameraman and the tripod but also includes other features like background buildings. There are also larger separate components and many (low pixel count) 'noise' components. A single isolated foreground pixel is still considered a component — however, see the full problem description for how we might change this.



In addition to finding all connected components, you will need to create a class *ConnectedComponent*, that stores all the information needed to represent a component. You will also need to use a **container** to store smart pointers to each *ConnectedComponent* and package this as a private member of the core class, *PGMimageProcessor*. You will instantiate/construct a **single** *PGMimageProcessor* instance in your driver file (on the stack — so be careful about how much memory is grabs from the stack), and there should be public methods provided to invoke the necessary functionality to build, manipulate and output these connected components. In particular, you will need to provide functionality to iterate (with a **container iterator**) through your collection of detected components and write them back out to produce a PGM image with only the selected components present (as pixels values = 255 and all other values set to pixel value=0) .

Throughout, you should keep RAII design principles in mind and make sure to provide the 'Big 6' for construction, destruction, copying and moving of all relevant class types. If your use of automatic variables means that you do not need any special behaviour, indicate this by setting the method to 'default' in the class declaration.

1 Core requirements

The requirements for this assignment are as follows:

Create a class, *PGMimageProcessor*, that encapsulates the functionality you need to read in a PGM image and apply a connected component analysis to that image. This class will need a sensible destructor and a constructor which takes in at least the input file name, and must implement the 'Big 6' (see above about use of 'default'). It should also contain (at least) the following public methods:

```
/* process the input image to extract all the connected components,
   based on the supplied threshold (0...255) and excluding any components
   of less than the minValidSize. The final number of components that
   you store in your container (after discarding undersized one)
   must be returned.
*/

int extractComponents(unsigned char   threshold,  int  minValidSize);

/* iterate  - with an iterator - though your container of connected
   components and filter out (remove) all the components which do not
   obey the size criteria, [minSize, maxSize] passed as
   arguments. The number remaining after this operation should be
   returned. NOTE: minSize need not be the minValidSize above – you can
   choose to keep ANY range of sizes after initial processing. These are
   logically distinct operations and a user of your API may want this.
*/

int   filterComponentsBySize(int  minSize,  int  maxSize);

/* create a new PGM file which contains all available components
   (255=component pixel, 0 otherwise) and write this to outFileName as a
   valid PGM. the return value indicates success of operation
*/
```

```
bool writeComponents(const std::string & outFileName);

// return   number of components
int getComponentCount(void) const;


// return number of pixels in largest component

int getLargestSize(void) const;

// return number of pixels in smallest component
int getSmallestSize(void) const;


/* print the data for a component to std::cout
   see ConnectedComponent class;
   print out to std::cout: component ID, number of pixels
*/

void printComponentData(const ConnectedComponent & theComponent) const;
```

NOTE: You must use an STL container to store the ConnectedComponents**. This can be unordered (std::vector or std::list) or something with an ordering (this makes things harder though)**

In addition to the *PGMimageProcessor* class, you must also define a 'helper' class — *ConnectedComponent* — to encapsulates information about a connected component. This must also obey the rules for RAII and implement 'the big 6'. The information stored in the component should be

1. int — the number of pixels in the component
2. int — a unique integer identifier for a component (you can just assign integers starting at 0).
3. a way to store the pixels in that component (as (x,y) pairs). At the very least, you can store a std::vector< std::pair<int,int> > of coordinates.

   See http://www.cplusplus.com/reference/utility/pair/pair/

   This is not terribly efficient, but has the virtue of simplicity. Depending on how you determine the components, other more efficient methods may become apparent.

The *ConnectedComponent* class should have its own implementation file and header file and this header file should also be included in *PGMimageProcessor.h* You can write appropriate public methods to expose the data in this class to other classes.

Remember that for move semantics to work effectively, the class types you define must be 'movable'. This is particularly important when you use a C++ container of smart pointers, since a new class instance will often need to be 'moved' into a container element by (move) assignment.

Command line invocation

Assuming your executable is called *findcomp*, it must support the following command line options:

```
findcomp [options] <inputPGMfile>
```

where <inputPGMfile> is a valid PGM image and valid options are:

-m <int> set the minimum size for valid components when creating connected components

[default = 1]

-f <int> <int> (-s min max) ; set the minimum and maximum components sizes to keep after filtering

-t <int> set the threshold for component detection (default=128, limit to [0. . . 255])

-p print out all the component data (entries do not have to be sorted) as well as the total component number and the sizes of the smallest and largest components.

-w <string> (-s outNamePGM) write out all retained components (as foreground/background pixels) to a new PGM file.

Demonstrate that all these options work in main() by providing code to invoke them.

Component Extraction

There are multiple ways to do this. The easiest is to start at the top left pixel of the image and scan along the image rows until you hit a foreground pixel. At that point you can start a Breadth First Search (BFS) to determine all 4-connected foreground pixels that are attached to this (this is effectively a 'floodfill' algorithm). Essentially, you check all possible 4-connected neighbours (N/S/E/W) and if a neighbour is 255, and not yet processed, add it to your component, pushing it's non-tested N/S/E/W neighbour coordinates onto a queue (initially empty). You need to take care to set the current foreground pixel added to your component to 0 in the thresholded image, so you don't revisit it. The BFS continues, popping off candidate pixel coordinates from the queue, and expanding/testing those, until the queue has been exhausted. Then you can continue scanning from where you initially started building the component, looking for the starting 'seed' for next component. Each new component should increase the component identifier (just start at 0). You can optimize the process described above, but it should work well enough as is. **Once you have extracted the components, you should delete the memory used to hold the PGM input image, UNLESS you re-use the memory to create the output image.**

Unit Testing your C++ Code

You must provide unit tests - for the methods listed above - in your code. The unit tests should be specified in a separate file and compiled using a separate rule in your Makefile (e.g. make tests). Unit testing will require that you separate out different steps in your algorithm into separate testable units of work, so that they can be tested individually. Therefore you should avoid writing very long methods that do many things at once. Consider breaking your implementation logic into smaller private methods used by that class.

You should use the catch unit testing framework to test your code.

Some more information can be found at
https://github.com/philsquared/Catch/blob/master/docs/tutorial.md.

This uses a newer split cpp/hpp implementation of Catch. So to make life easier just use the catch.cpp handed out in Tutorial 4.

Failing to implement unit tests or implementing tests that fail to varying degrees will impose a penalty of between 0% and 10% on your final mark. The marks awarded for various components also reflect tests to show that these work as expected.

Note that to test images component detection, we have provided example image with components counts. However, if you wish to create your own test images you can certainly do this (using say the Gimp image editor, and thresholding input images). You could look to test that more complex component shapes are also extracted not only simple shapes like squares, circles etc. Some image editors have pixel counting tools that can count the pixels matching a colour (histogram). Thus, after thesholding an image you could recolour each component (GIMP has a tools to do this: select by colour (where you click) and then choose a foreground fill colour and fill with this) and then extract each components pixel count. In GIMP, selection by colour will allow you to select a component by clicking on it, and the colors → info → histogram tool will display how many pixels it contains. We have provided a test image which shows an input, a thresholded version, a recoloured version and a file of pixel counts.

Completing all the above core work correctly will enable you to score up to 90%. To achieve a higher mark, you should tackle the mastery work outlined below.

2 Mastery work

The remaining 10% will require extending the program as follows (these will require additional command line flags which should be reported in your README):

Template support for RAW PPM input and output — 5% Using templated code, change your image class to work for both PGM and PPM images. A PPM (portable pixmap) is a colour image with a similar header to a PGM (the URL provided in Assignment 2 explains more). A RAW (binary) PPM image header starts with P6 and otherwise has the same format as PGM. The main difference is that the each pixel now has THREE bytes of data — Red, Green, and Blue — each of which is an unsigned byte and represents how much of that colour primary is present (0 means it's off, and 255 means its at maximum strength). Note that the templating is specifically to get a colour image in and write one out. When your are doing processing to detect components, including thresholding etc, you should convert the colour pixel values to greyscale and work with those. The conversion equation you should use is $I = 0.299 * R + 0.587 * G + 0.114 * B$, where (R, G, B) are the channel intensities for your colour pixel. The image data is arranged as before, row by row, but each pixel now occupies 3 bytes (three unsigned char) rather than one unsigned char.

Draw bounding boxes for components — 5% Add another flag -b <PPMimagename> to produce an output PPM image which is the original image with colour boxes drawn over it to show where each retained component is in the input image. Note that if the input image is a PGM, you can convert it to colour PPM by simply making $R = G = B = I$, where I is the greyscale intensity for a pixel. The box colour should be something with generally good visibility, like Red = (255, 0, 0) or Green = (0, 255, 0). Also note that to compute the bounding box you simply find the min/max (x, y) pixel range within a component's set of pixels. Drawing the box requires overwriting pixels that define a 1 pixel box which ranges from (x_0, y_0) to (x_1, y_1), your box starting and ending pixel values. This does not require general line drawing, only requiring horizontal and vertical lines to be drawn, which can be done in one or two lines of code.

Please Note:

1. A working Makefile must be submitted. If the tutor cannot compile your program on nightmare.cs by typing make, you will only receive **50%** of your final mark.

2. You must use version control from the get-go. This means that there must be a .git folder alongside the code in your project folder. A **10%** penalty will apply should you fail to include a local repository in your submission.

   With regards to git usage, please note the following:

   -10% - usage of git is absent. This refers to both the absence of a git repo and undeniable evidence that the student used git as a last-minute attempt to avoid being penalized.

   -5% - Commit messages are meaningless or lack descriptive clarity. eg: "First", "Second", "Histogram" and "fixed bug" are examples of bad commit messages. A student who is found to have violated this requirement for numerous commits will receive this penalty.

   -5% - frequency of commits. Git practices advocate for frequent commits that are small in scope. Students should ideally be committing their work after a single feature has been added, removed or modified. Tutors will look at the contents of each commit to determine whether this penalty is applicable. A student who commits seemingly unrelated work in large batches on two or more occasions will receive this penalty.

   Please note that all of the git related penalties are cumulative and are capped at -10% (ie: You may not receive more than -10% for git related penalties). The assignment brief has been updated to reflect this new information.

   We cannot provide a definitive number of commits that determine whether or not your git usage is appropriate. It is entirely solution dependent and needs to be accessed on an individual level. All we are looking for is that a student has actually taken the time to think about what actually constitutes a feature in the context of their solution and applied git best practices accordingly.

3. You must provide a README file explaining what each file submitted does and how it fits into the program as a whole and how to compile and run your code. The README file should **not** explain any theory that you have used. The

README be used by the tutors if they encounter any problems.

4. Comment your code appropriately.
5. You may reuse the PGM image class files for Assignment 2 (and extend as needed).
6. Do **not** hand in any binary files. Do **not** add binaries (.o files and your executable) to your local repository.
7. Please ensure that your tarball works and is not corrupt (you can check this by trying to download your submission and extracting the contents of your tarball - make this a habit!). Corrupt or non-working tarballs will not be marked - **no exceptions.**
8. A 10% penalty per day will be incurred for submissions up to 24 hours. No hand-ins will be accepted after 24 hours.
9. **DO NOT COPY. All code submitted must be your own.** *Copying is punishable by 0 and can cause a blotch on your academic record.* **Plagiarism detection software will be used to check that code submitted is unique.**