Andrew Quinn
ECE332 Introduction to Computer Vision, MP#1: Connect Component Labeling

Caution: This was coded in Python 3.8, not MATLAB, and requires two external libraries to run: Pillow, and numpy. If you have Python 3.8 installed you can install these to run the code yourself at the command line with `python -m pip install pillow numpy`.
(The professor said we could use any language we wanted, hence why I went with this option.)

Image results are on the next few pages. I used grayscale to indicate the different connected components.

Algorithm explanation

Initial sweep.

The sequential connected component algorithm itself is pretty straightforward. We first initialize a structure to hold 2-tuples, called the equivalence table. You also need an integer to keep track of the highest potential boundary you have yet come across, which I'll call **hpb**.

Starting at some corner of the bitmap image -- I use the [0, 0] position, but you could do it from any corner really -- you assign a "potential boundary" number to each pixel, iterating from left to right and from up to down, starting from 0, via the following logic:

1. Does the number above me, at the **[row-1, column] position** exist?
    a. If so, does it match the color value I have?
        i. If so, then we take its potential boundary number.
        ii. Either way, continue to #2.
2. Does the number to my left, at the **[row, column-1] position** exist?
    a. If so, does it match the color value I have?
        i. Did I a**lready take the [row-1, column] potential boundary number**?
            1. If so, store the boundary numbers of your two neighbors into the **equivalence table** - they form a reverse "L" with you, and are in fact connected. ***Iteration complete***.
            2. If not, take its potential boundary number. ***Iteration complete.***
        ii. If not, continue to #3.
3. *Neither the pixel above me nor the pixel to the left of me match my color value. This is a new potential boundary.*
    a. Assign to yourself **hpb + 1.**
    b. Update **hpb = hpb + 1**.
    c. Iteration complete.

In my Python code, a single sweep of this is in `check_pxl()`. `set_initial_regions()` applies this to the whole image, to get the first (bad) potential map of connected components.

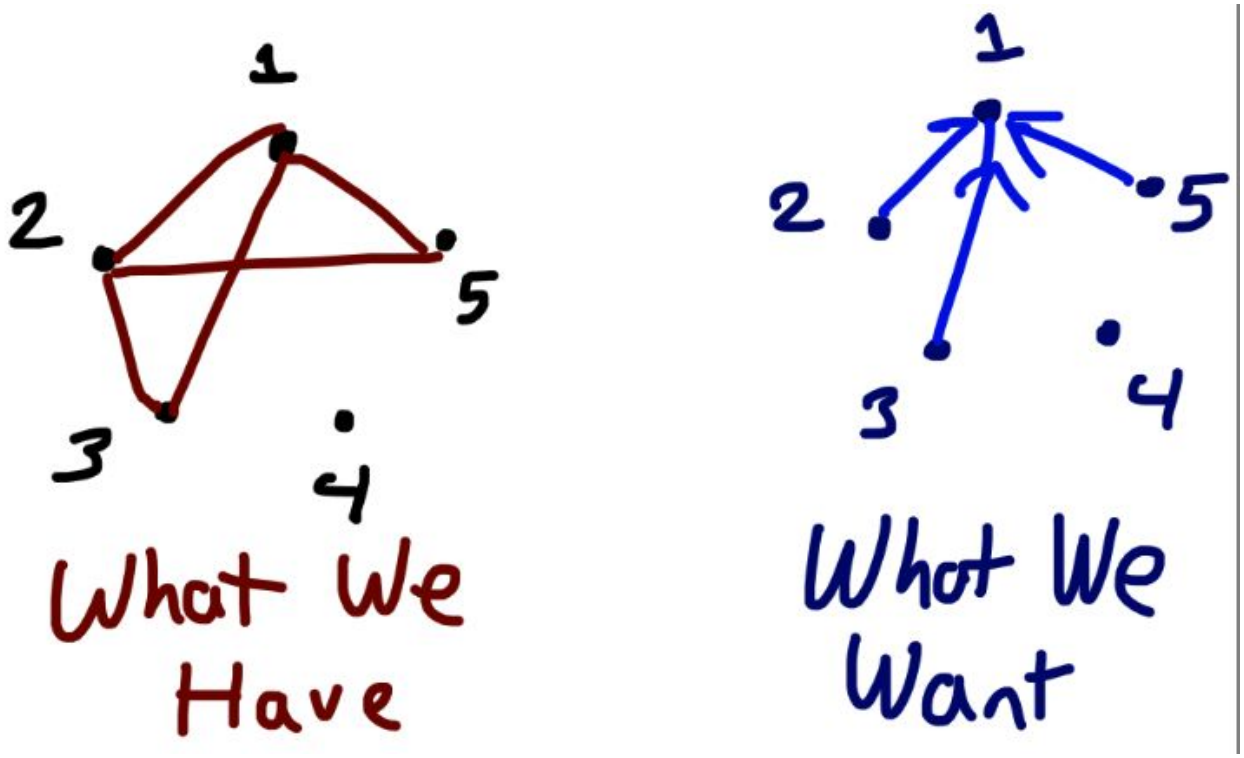
Reducing the equivalence table

Once the initial sweep is done you will have an equivalence table that, for me, looked something like

```
[ (1, 2), (1, 3), (1, 5), (2, 3), (2, 5) ]
```

The way to read this is that `(1, 2)` means potential boundaries 1 and 2 actually share a pixel in common, and so they are the same boundary - you can replace every 2 with a 1, or vice versa.

One way to interpret this as forming the edge set of an undirected graph (the arrows on the right are just for illustration). We would like to use this equivalence class to build a straightforward table that we can then quickly consult in a secondary sweep through the image to replace all of the potential boundaries with actual, unique boundaries.



Now I'm sure if I were a computer science student, and not an electrical engineering student, I would see this and immediately go "That's a minimum spanning tree!" and pinpoint a highly performant algorithm to do this off the top of my head. But I'm not, and I didn't. My solution is a naive recursive one.

eq_reduce(list_of_2_tuples)

1. Sort the list.
2. (Base case) If there are zero elements in the list, we're done. **Return the list as-is.**
3. Otherwise, let **head = (x, y)** be the first element of the list, and let **tail** contain the rest of the elements.
4. For each **(t, s) in tail:**
    a. If **(x, y) = (t, s),** then you have a duplicate. Just delete it and move on.
    b. Otherwise, if **t = y,** then **replace (t, s) with (x, s).** *It's worth noting here that from the initial sweep algorithm, it is always true that t < s and x < y.*
    c. Otherwise, if **s = y,** then **replace (t, s) with (x, t).** *This is why it was important to sort.*

5. Return a list concatenation of **eq_reduce(tail).append(head).** (If I remember right, appends to the ends of lists are the most performant way to work with them - but it's not necessary. You still need to go through and manually remove duplicates at the end anyway.)

Now I'll walk you through what this does for each element in the example list.

```
sort!
Head: (1, 2)   Tail: (1, 3), (1, 5), (2, 3), (2, 5)
Head: (1, 2)   Tail: (1, 3), (1, 5), (2, 3), (2, 5)
Head: (1, 2)   Tail: (1, 3), (1, 5), (2, 3), (2, 5) <- s = y
Head: (1, 2)   Tail: (1, 3), (1, 5), (1, 2), (2, 5)
Head: (1, 2)   Tail: (1, 3), (1, 5), (1, 2), (2, 5) <- s = y
Head: (1, 2)   Tail: (1, 3), (1, 5), (1, 2), (1, 2)
                          sort!
                          Head: (1, 2)   Tail: (1, 2), (1, 3), (1, 5) <- (x, y) = (t, s)
                          Head: (1, 2)   Tail:         (1, 3), (1, 5)
                          Head: (1, 2)   Tail:         (1, 3), (1, 5)
                    sort!
                                                Head: (1, 3)   Tail: (1, 5)
                              Sort!
                              Head: (1, 5)   Tail: Empty! <- base case
                                                                                  End val: [(1,2), (1,2), (1,3), (1,5)]
                                                                                  Still have duplicates, but now everything
                                                                                  Goes to the lowest place!
```

Dealing with duplicates in Python in this situation is very straightfoward: Turn the list into a set, and then back into a list again. `list(set([(1,2), (1,2), (1,3), (1,5)]))`

Final (necessary) sweep

In Python we can now take these reduced equivalence tables and form key-value pairs in an object known as a dictionary. Dictionaries have O(1) lookup time and O(N) creation time, so they are blazing fast in this language. So you just iterate through the array of initial possible boundary values, and if a boundary has a dictionary entry, you just replace it with the value that key points to. In my code this is `regions_finalizer()`.

Optional sweep

There is no guarantee that the final sweep returns boundaries *in order*, however. You may end up with a 2D boundary array that has four unique boundaries labelled with the values 0, 1, 3, 4327. This is an easy fix with an integer *next* and another dictionary:

1. Initialize a dictionary with {0:0}, the 'default' boundary value of the first pixel. Set the integer *next* to 1.
2. For every pixel, register shift by 1 bit[1], *and then* look up its 2x'd boundary value in the dictionary. This is another O(1) average case action.
    a. If it's in there already, assign the value that's in the dictionary. **Iterate**.
    b. If not, take the 2x'd boundary value 2*b* of the pixel, and add 2*b: next* to the dictionary as a key-value pair. Set the 2x'd boundary value of the pixel itself to *next*. Set **next = next + 1,** and **iterate**.

Now your boundary values move up in a nice 0, 1, 2, 3, 4, 5, …. Progression. In my code this is the function `file_off_the_serials()`.

Size filter

[1] This is a quick fix that isn't strictly needed. In my original code I ran into an issue where I didn't write this algorithm 100% clean, and I got an issue when it turned out next was *exactly equal* to the boundary value of a pixel, when I needed a strict inequality. The shift means this will never be the case for you; every time next grows by 1, the actual next new boundary it will meet will grow by a minimum of 2.

For single-pixel size filters like the one we needed to use on gun.bmp, it suffices to make an algorithm that merely checks to see whether any of a pixel's immediate neighbors match its regional value. If it doesn't, it gets "deleted" (in this context it just flips from black to white, or vice versa).

For an arbitrary *n*-volume size filter, we go through with a dictionary after and count the number of pixels each unique region post-`regions_finalizer()` has. We then make a second sweep through; any region that had fewer than *n* pixels has its pixels' regional boundary values set to 0. (There are more complicated ways to do this, I'm sure, but these are just black and white images for now.)

Code

Again, make sure you have Python 3.8, numpy, and Pillow installed if you want to run this yourself. You can install both of those packages from the command line via

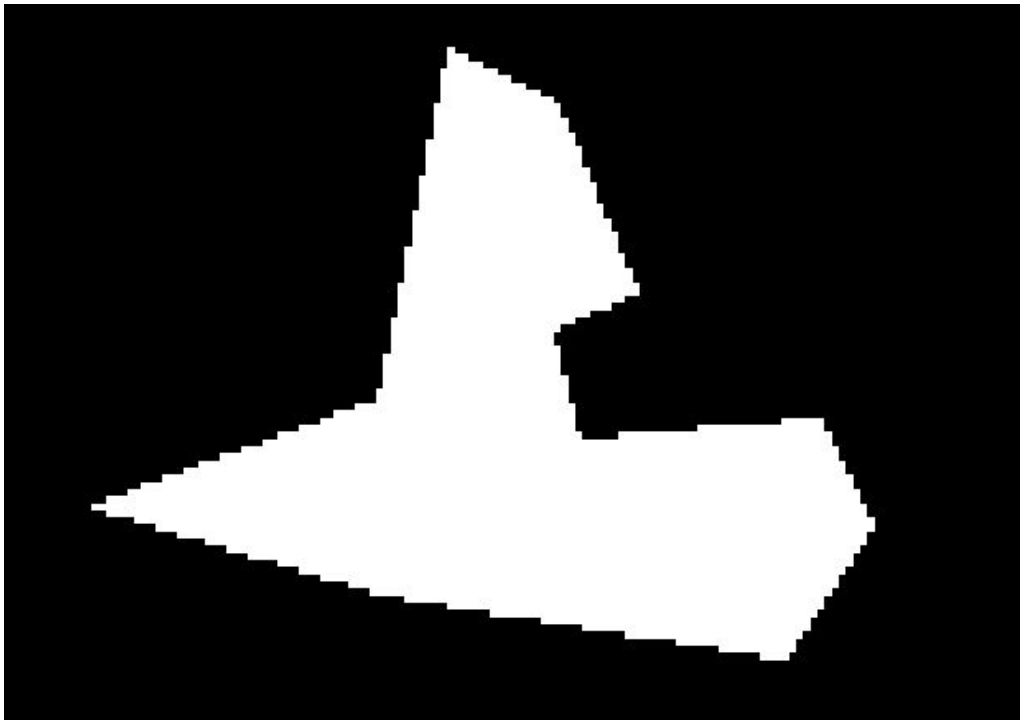
```
python -m pip install pillow numpy
```

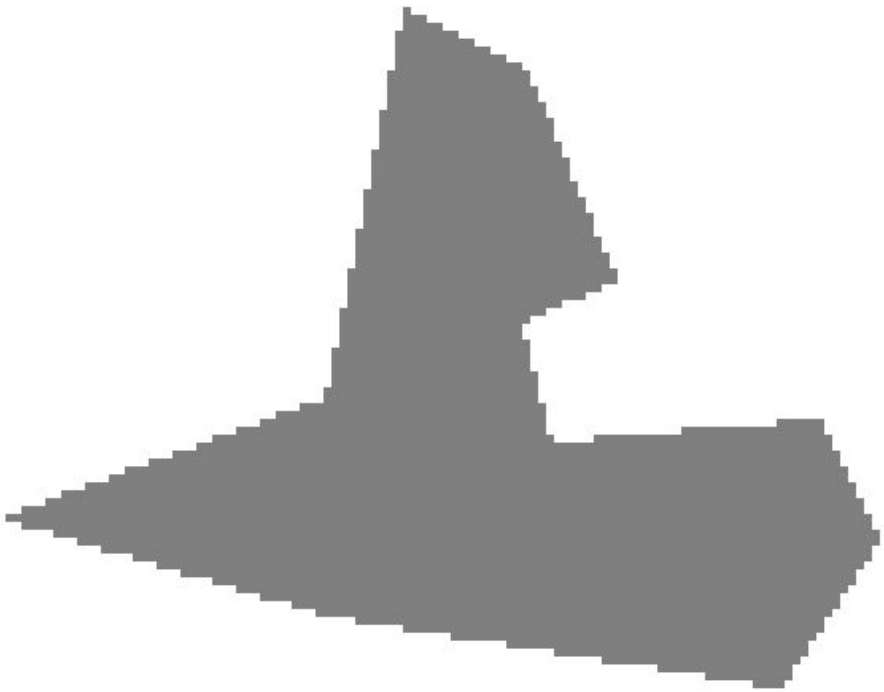
To run:

1. **Unzip the .zip file attached.** (All of the images are in there; you can't run the Python file as a standalone thing.)
2. **Run mp1.py by going to the folder location and typing "python mp1.py".** You will see a lot of text, but the important thing is all of the .bmp files that will be created. You can redirect the text to a file by doing "python mp1.py > output.txt" if you like.
3. **Look at the grayscale "gs.bmp" files created.** Those are the same ones shown on the next several pages.

Image results

Test.bmp

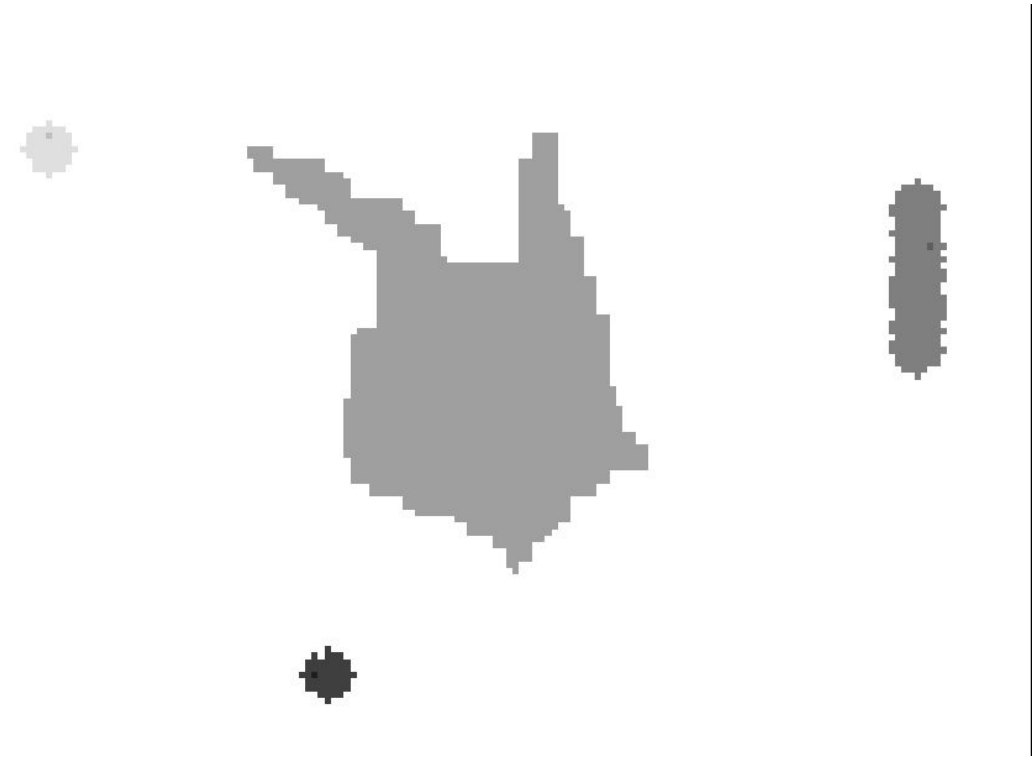


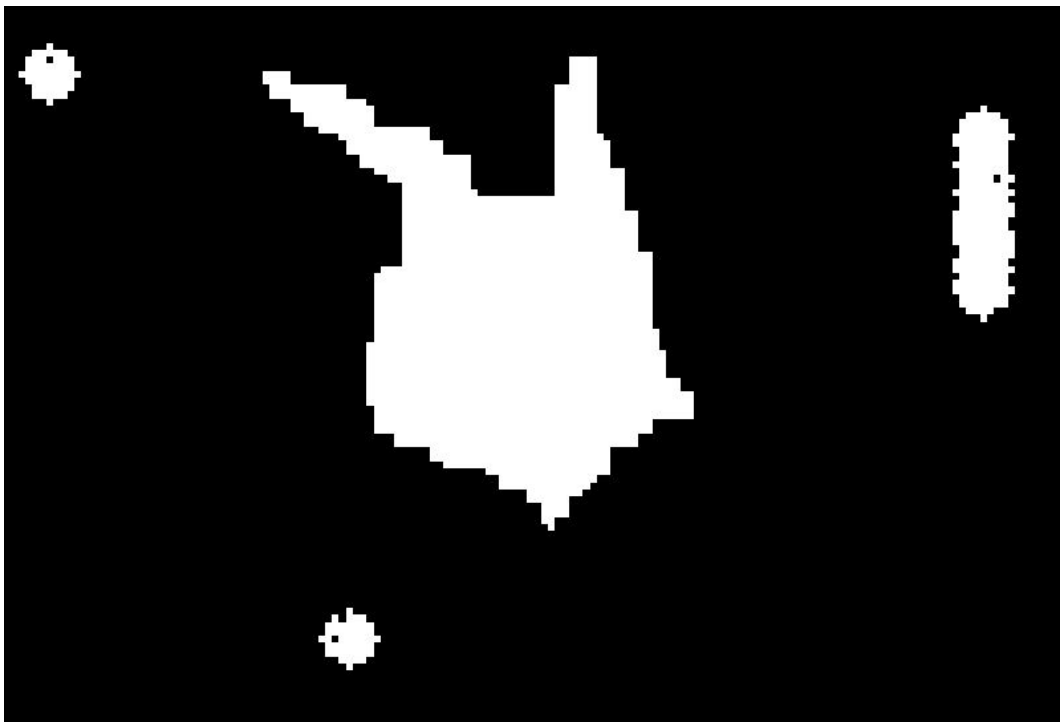


face.bmp

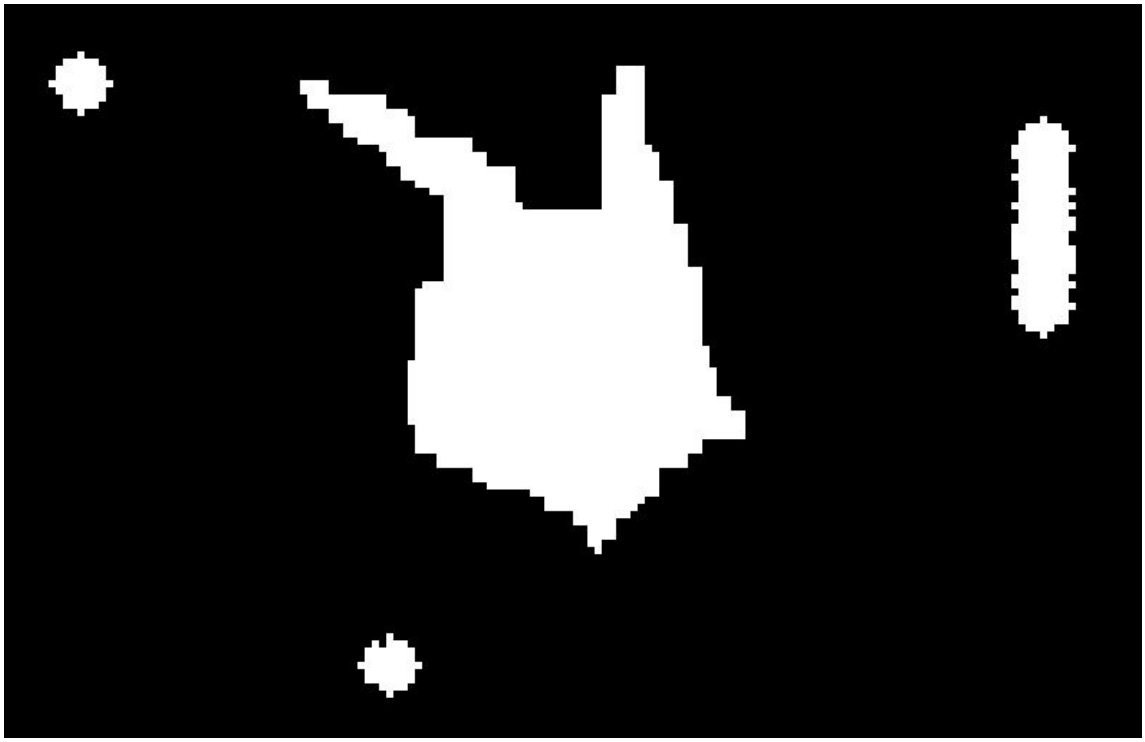
A few randomly-generated, 10 by 10 pixel white noise blocks I generated to help debug

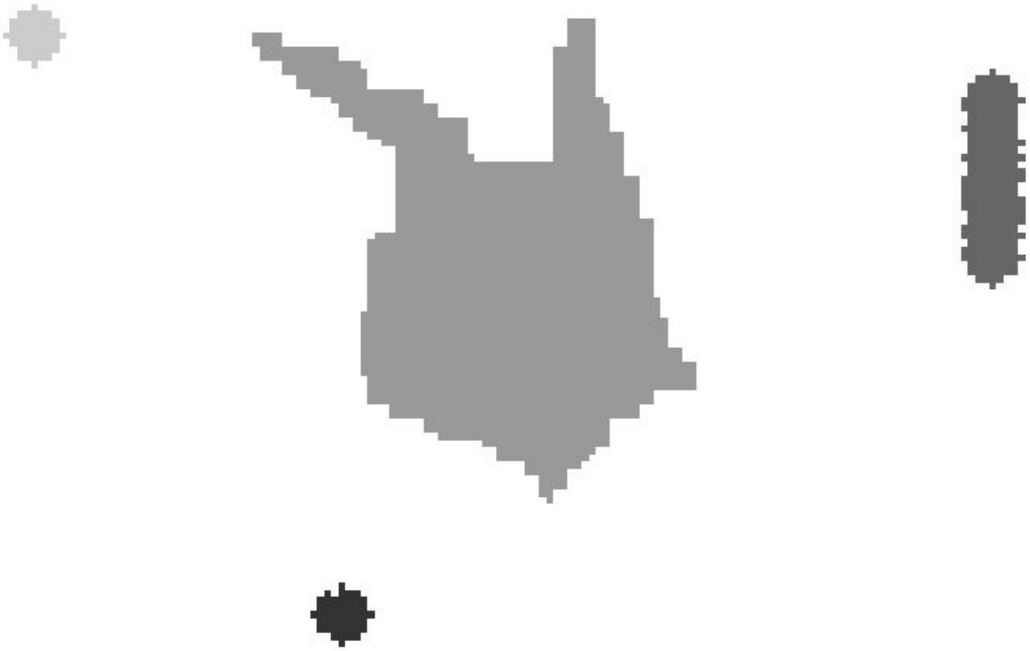
Gun.bmp, **without** size filter



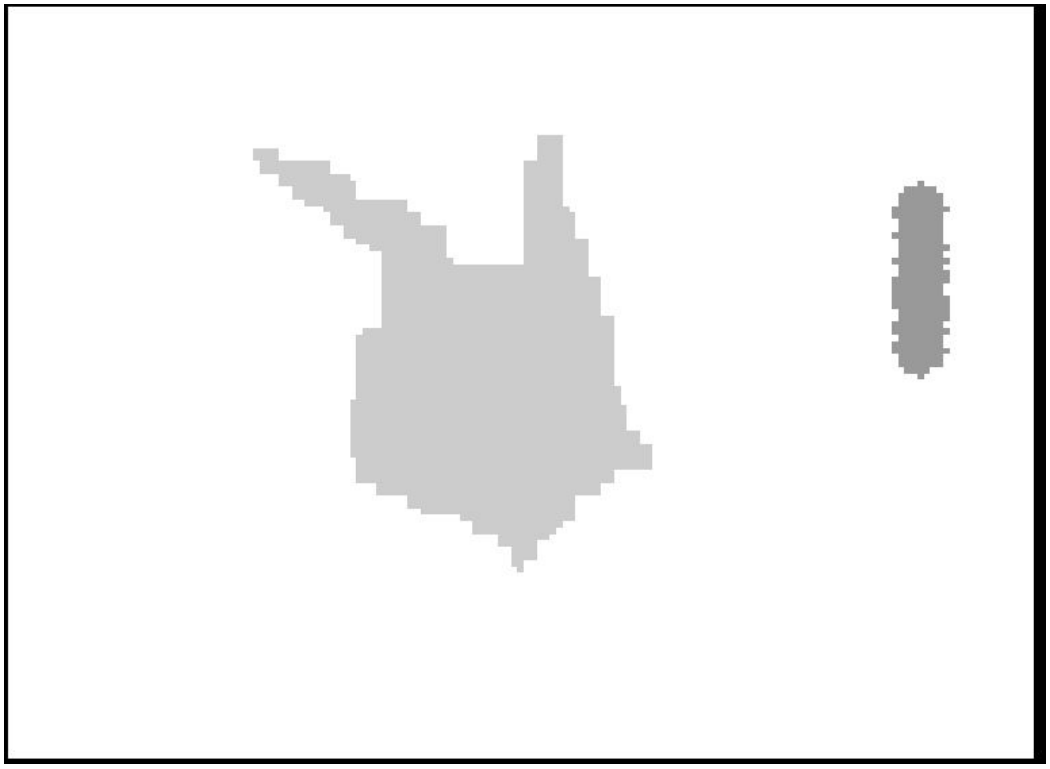


Gun.bmp, **with size filter *n* = *1*** (single pixels removed before component processing)





Gun.bmp, with **aggressive size filter *n = 50*** (big dots wiped out!)



Gun.bmp, with a **very aggressive size filter *n = 250***