Artificial Intelligence: Handwriting Recognizer

With the widespread adoption of smartphones, tablets, and other hand-held devices, users have deemphasized the traditional role of input methods such as mice and keyboards. With advances in hardware and improved user experiences, stylus pens are becoming a primary input device. Apple, Samsung, and Microsoft are marketing next-generation styluses which can be used for selection, gestures, and hand-written input.

The question is – how do these devices interpret users' writings? Did the user intend to write a letter "O" or a number "0"? Did the user intend to dot an "i" or click a submit button? Engineers who write software to enable these devices use various AI algorithms and techniques to tackle the very difficult problem of determining human intent.

Template Matching

In this assignment, you will implement the technique of hand-sketch recognition called **template matching.** In template matching, the system maintains a set of templates of prototypical examples for each shape it recognizes.

Let's say a system is designed to recognize circles, triangles, and squares. It would keep some number of (assume 6) examples for each of these categories: hand-drawn circles; hand-drawn triangles; and hand-drawn squares. When the user draws a shape, the system compares that shape with all 18 templates to find the templates most *similar* to this new, unknown shape. If the unknown shape is determined to be most similar to a circle, odds are fairly good the unknown shape is a circle.

The basic idea is simple, but how *exactly* does the system compute "similarity"?

Let's start with a more simple question and work our way up to "similarity." We first need to know exactly what a "shape" is. A shape is simply a contiguous series of points collected through hardware sampling.

How The Hardware Collects Points

Every few milliseconds, a computer system will determine whether 3 events just happened and respond accordingly:

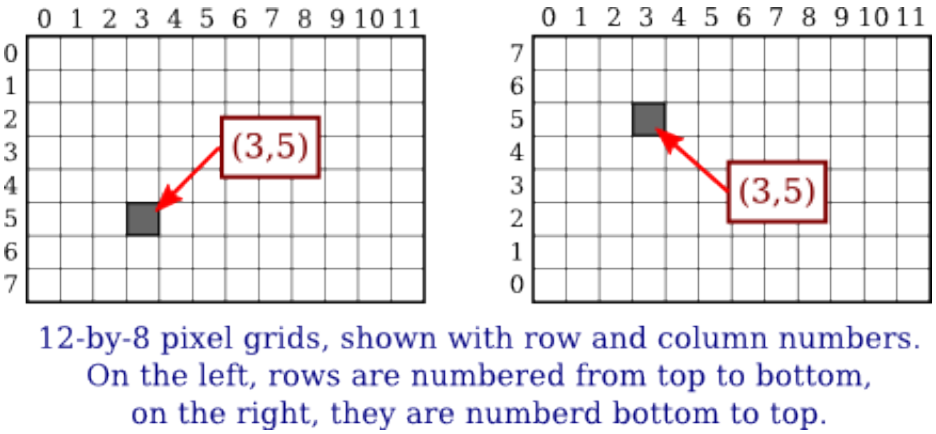
- **Event 1:** Did the pen just touch the screen? If yes, the system says, "Welp, I guess the user wants to draw something, better start recording points." It then creates an empty list to hold points, finds the (x,y) coordinates of the pen's location, and adds the first point to the list.

- **Event 2:** Did the pen (which was already touching the screen) move? If yes and in a new position, it adds the new coordinates to the list of points for the existing shape.
- **Event 3:** Did the pen (which was already touching the screen) leave the screen? If yes, the system knows the user is done inputting a shape (at least for now) and it can start its' recognition magic!

After Event 3, you have a list of a sequence of points on a two-dimensional plane.

Computer Graphics & Coordinate Planes

Note that the computer's coordinate plane is different from the type you've probably seen in mathematics. Consider the image below:



12-by-8 pixel grids, shown with row and column numbers.
On the left, rows are numbered from top to bottom,
on the right, they are numberd bottom to top.

Image Courtesy of Hobart and William Smith Colleges Department of Mathematics and Computer Science (http://math.hws.edu/graphicsbook/c2/s1.html)

The image on the right denotes a coordinate plane used in traditional mathematics and therefore probably the system you are most familiar with. The "origin," or (0,0), is in the bottom-left corner. In computer graphics, however, the "origin" is in the *top-left* corner and we can consider a coordinate in terms of distance in pixels from the origin. For example, the coordinate (3,5) is 3 pixels to the right of the origin and 5 pixels down from the origin.

Determining Similarity Between Shapes

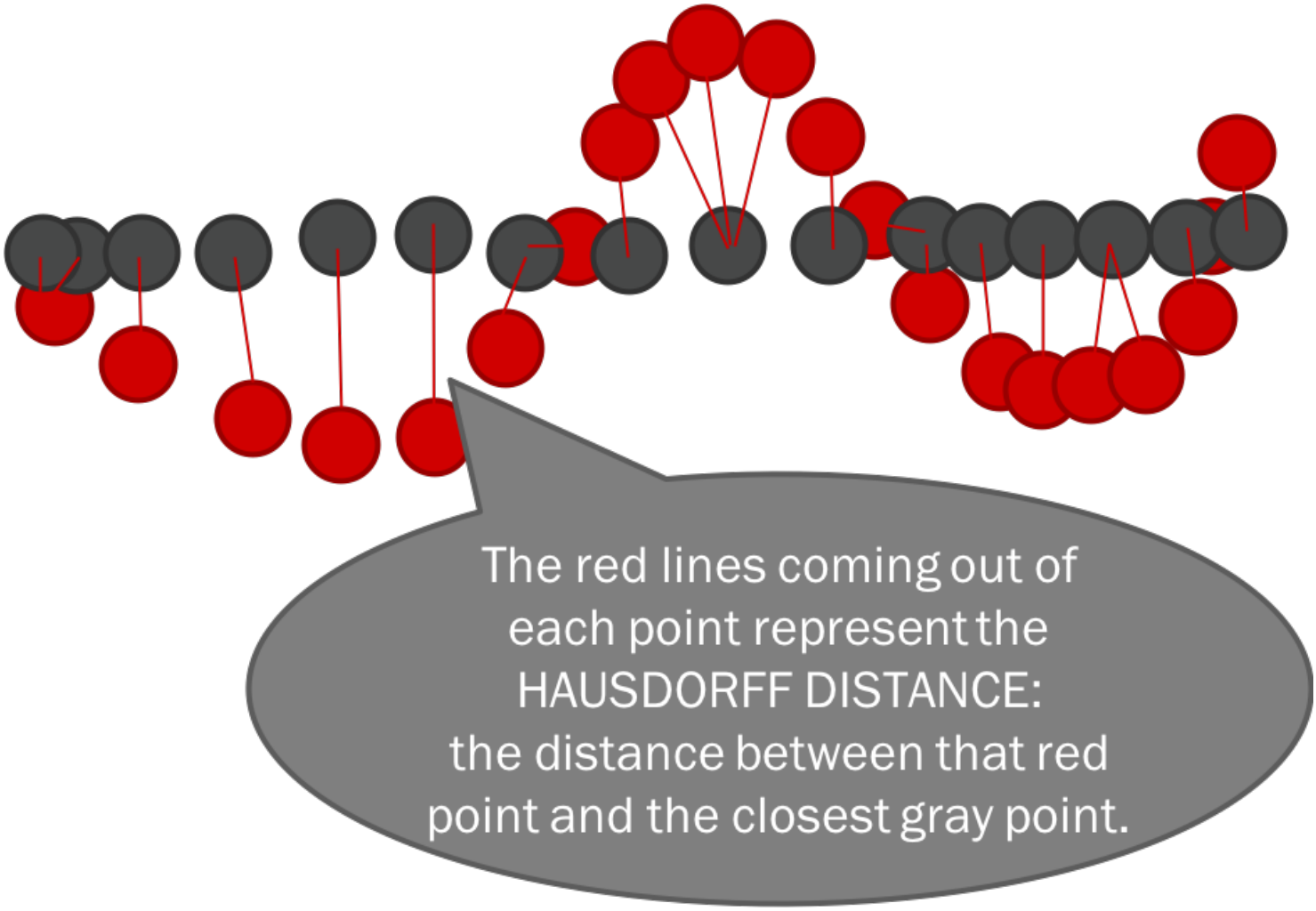
Researchers in computer science and mathematics have researched the idea of shape similarity extensively and have developed multiple metrics for determining shape similarity. In this assignment, we are going to be working specifically with the "Tanimoto Coefficient." Our calculations will require some preprocessing, so let's start there.

Step 1: Preprocessing. First, you need to make sure both shapes (the new shape and the template shape) are the same size (*i.e.*, they are as large as they can be and centered within a 300x300 bounding box). Additionally, it is a good idea to resample (interpolate) your shapes, ensuring your user's shape and all your templates have the same (or close to the same) number of points. If the shapes you're trying to recognize are highly detailed, you probably want a larger bounding box and a larger quantity of points. Our shapes in this assignment (lowercase letters from the English alphabet) are pretty simple, so we're using a 300x300 pixel bounding box and about 90 points. This preprocessing work has already been done for you in this assignment. You are welcome! ;-)

Step 2: Calculate Hausdorff Distances Defined simply, a Hausdorff distance is the shortest distance from a point to a shape. The number of feet between you and the nearest wall is a Hausdorff distance. The number of miles between your chair and the nearest international border is a Hausdorff distance. Because we are trying to determine the similarity of two sketched shapes (a template shape whose categorization is known and a user-drawn shape), we will calculate the distance from a point to a shape.

We will calculate "how close" a point (let's call it `p`) is to a shape (let's call it `S`) by finding the minimum Euclidean distance between `p` and a point `s<sub>min` in `S.` Let's unpack that a bit first by talking about Euclidean distances. Do you remember the Pythagorean theorem: `A<sup>2</sup> + B<sup>2</sup> = C<sup>2</sup>`? Euclidean distances are calculated using that same idea, except it's `(difference in x coordinates)<sup>2</sup> + (difference in y coordinates)^2^ = (distance between points)<sup>2</sup>.` Next, let's talk about "the minimum." You'll calculate Euclidean distances between `p` and every point in `S,` choose the minimum distance, and then call it the Haudorff distance.

After calculating those distances, we will ask: how close is each point in our user's shape to the template shape? We also want to know the reverse: how close is each point in the template shape to the user's shape? How many of these Housdoff distances do we need to calculate? Each time you compare the user's shape to a template shape you will calculate and record `n+m` Hausdorff distances, where `n` is the number of points in the unknown shape and `m` is the number of points in the template shape.



Hausdorff Distances

Step 3: Calculate the Tanimoto Coefficient Defined simply, the Tanimoto Coefficient tells us how many of the `n+m` points "overlap" the other shape. More specifically, the Tanimoto Coefficient is the percentage of the `n+m` Hausdorff distances that are close enough to zero. We define "close enough to zero" to be a distance that's less than 10% of the bounding box, so within 30 pixels in our example. (Remember not to hard-code *magic numbers* like 10 and 30, though!).

Once computed, you can think of the Tanimoto Coefficient as a similarity confidence value (*e.g.*, "I am 96.524% confident that these two shapes are similar.")

Step 4: Repeat
You will compute the Hausdorff distances and Tanimoto coefficient between your unknown shape and every template known to your system. It's important to keep track of every result because we will need to find the `k` (in our case `k=8`) highest confidence values. How you find those top `k` matches is up to you.

For this assignment, you will stop at this point and display the 8 most similar templates and marvel at your brilliance! :-) However, in a deployed sketch recognition system, one would find the label (*e.g.*, “circle,” “square,” or “triangle”) of the shape that shows up the most in your top-k matches, and that will be the label that you assign to your previously unknown shape.

Okay! Now you understand the theory, but what is it you actually need to do to complete the assignment???

Your Task

Your task has two parts. You can do them in any order.

Part 1: Calculating the Matches

In this part, you will do the work necessary to calculate Hausdorff Distances, Tanimoto Coefficients, and to find the top-k matches between the user-drawn shape and the template shapes. Basically, you’re going to implement all the theory you just learned.

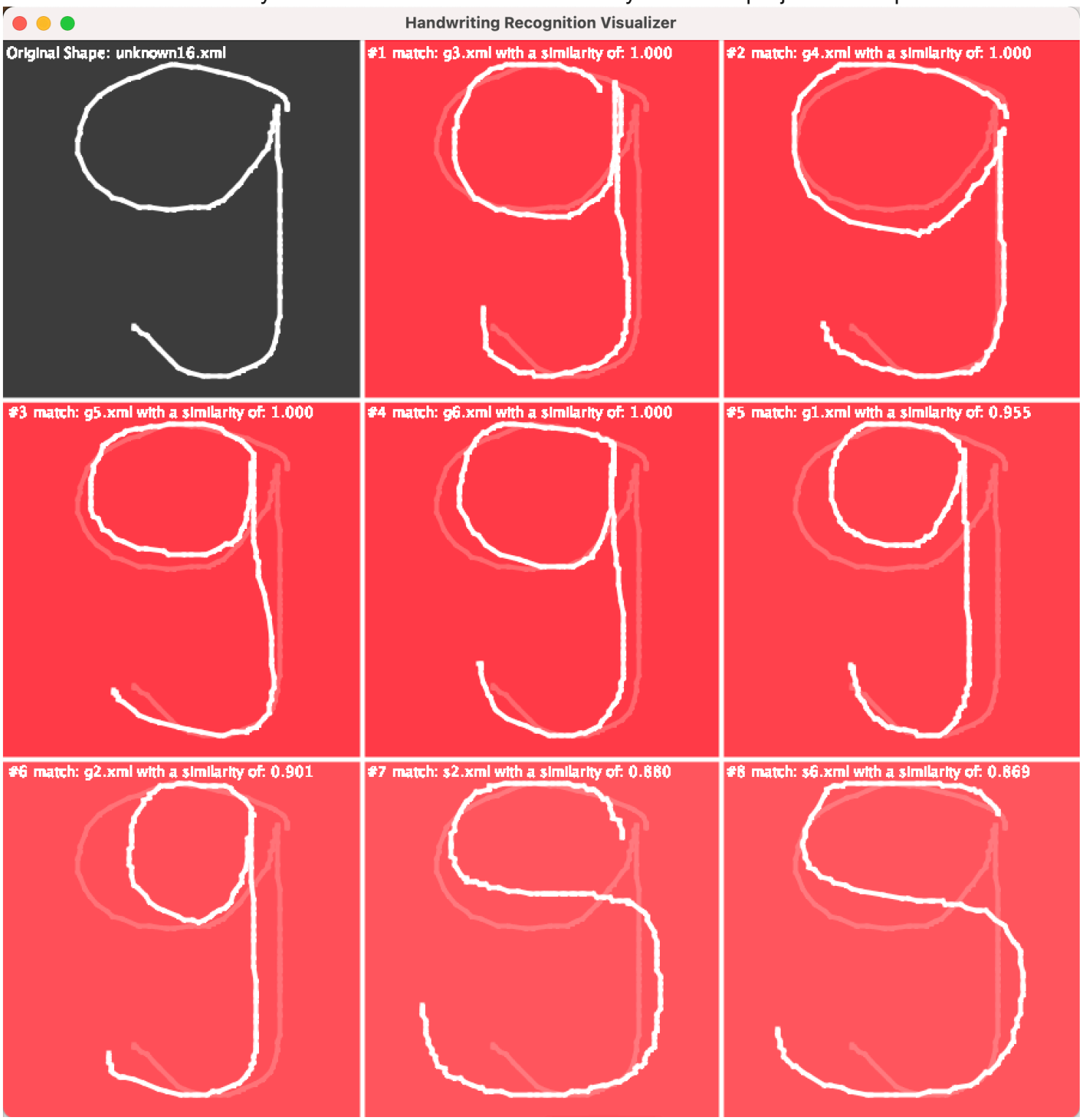
Here’s what to do:

1. You will need to fill in the `calculateSimilarity()` function in `SketchComparison.java`. This is the public function that triggers the computation of the Tanimoto Coefficient, but it shouldn’t do all the work itself. This algorithm doesn’t really lend itself to a monolithic function. Think about which useful tasks you can pull out into helper functions (*e.g.*, calculating the distance between two points, finding a Hausdorff distance, *etc.*). Design your suite of helper functions, write them, and then link them all up in `calculateSimilarity()`.
2. You will need to complete the `calculateAllMatches()` function in `SketchRecognizer.java` according to the provided JavaDoc. Basically, this function executes the idea of comparing an unknown shape to all templates in the system.
3. Likewise, you will need to complete the `calculateTopKMatches()` function in `SketchRecognizer.java` according to the provided JavaDoc. This function searches through all templates to find the `k` with the highest Tanimoto Coefficients. Your `calculateTopKMatches()` function must sort the `ShapeComparison` objects according to their natural ordering (higher similarity score is better; ties should be broken with lexicographical ordering of filenames). Remember, comparing according to natural ordering means `ShapeComparison` needs to implement the `Comparable` interface and you need to write a `compareTo()` function. You should use that `compareTo()` function in your searching algorithm. Note that you may NOT use any built-in sorting functions anywhere in this assignment.
4. You will need to fill in the `getTopKMatches()` function in `SketchComparison.java`, which should use the functions written in the previous two instructions.
5. You should be able to tell whether your algorithms are working properly through console print

statements.

Part 2: Displaying The Things

In this part, you will use the functions provided for you in `SketchCanvas.java` to draw things onto the canvas. This is what your canvas should look like when your whole project is completed:



Note: the “headquarters” for your GUI drawing (the big overarching calls, like a call to draw the contents of an entire cell or calls to draw the borders) should exist in the provided `updateGUI` function in `SketchRecognizer.java`. All other, more fine-grained functions for drawing should be written in `SketchCanvas.java`.

What you need to draw:

- **Draw 9 rectangles**, one for each cell.
- The size of each cell in the grid is 300x300. (See the `SHAPE_HEIGHT_WIDTH` variable in SketchedShape.)
- The cell in the first row/first column uses the background color `ORIGINAL_BG_COLOR` from the SketchCanvas class.
- The remaining cells are bleached versions of `MATCH_BG_COLOR`. The amount of bleaching should be equivalent to `1 - similarity`. So if the similarity value was `.97`, the bleach amount should be `.03`. The function to bleach colors has been provided for you (see `SketchCanvas.java`).
- **Draw 4 lines in a tic-tac-toe pattern** to create the white borders you see in the example. They are drawn in `NORMAL_STROKE_COLOR`.
- **Draw the unknown (original) shape in the top-left corner.** The sketched shapes are made up of points (which are drawn, though they can be difficult to see) and lines between adjacent points. The original shape is drawn in `NORMAL_STROKE_COLOR`. You should add a function in the SketchCanvas class that draws shapes (utilizing the `drawPoint()` and `drawLine()` functions that already exist). It would be a pain to write that code more than once!
- **Draw the other 8 unknown shapes.** These shapes are drawn in the same process listed above, but use a faded, pinkish-red color. The faded, pinkish-red color is a bleached version of the background color for that cell (which may have been bleached already, but you will bleach it again). The amount of bleaching for the stroke color can be found in the `SketchCanvas.FADED_STROKE_BLEACH_AMOUNT` variable.
- **Draw the template shapes.** Like the unknown shape in the top left corner, the template shapes are drawn in `NORMAL_STROKE_COLOR`.
- **Add the text.** The text is written in `NORMAL_STROKE_COLOR` and displays the similarity score to three decimal points.

If you do this part first, you’ll have to use some dummy data. That’s okay, just instantiate a shape or two, make up some numbers, and get everything displayed properly. This will allow you to visually check your work in Part 2.

Extra Credit

- If you’re looking for more of a challenge and a few points of extra credit, consider implementing a few other (at least two) metrics for calculating shape similarity:
    - **Average Hausdorff Distance**: calculate the average of all `n+m` Hausdorff Distances. This metric represents the “average” distance between the two shapes. Note that this metric is a *distance* and not a *confidence value*. If you want to use it as a confidence value (*e.g.*, “I am 95% confident that these shapes are similar.”), you will need to normalize it in some way. I

have had luck with dividing the average Hausdorff Distance by the greatest distance possible between any two points, and subtracting that value from 1.
In our application, the greatest distance between two points is the distance between opposite corners of our 300x300 pixel bounding box (about 424.3). The glory in this kind of work, though, is you get to be creative. Try different ways of normalizing and see how they affect the final similarity scores!
    - **Maximum Hausdorff Distance:** find the largest of the Hausdorff Distances. This metric is a good indicator of the greatest amount of dissimilarity between the two shapes. Note, again, this is a distance and not a confidence value. You will need to normalize this as well.
    - **Your Own Metric:** What kind of metric can *you* think up? What information is relevant for the comparisons?

Once you've implemented 2 or more additional metrics, you will need to combine these metrics (plus the original Tanimoto Coefficent) in some way. Will you average them? Will you use a weighted average? Will you pick the best? The worst? Be creative!

If you choose to do this extra credit and thus change the way similarity values are calculated, you will need to add a message (a paragraph or two) explaining: exactly which metrics you are using to calculate the similarity score; how you are normalizing them (if you are doing so); and how you are combining those metrics into a single confidence value. This message should be printed to the console.

General Notes

- You will need to create many new functions to do the work that you need. Don't be afraid to create more functions to do repetitive work.
- Remember that each function should be responsible for doing one thing. It is okay to have a function that is only called once because it prevents monolithic functions that do many things. Monolithic functions can be very difficult to debug and maintain!
- No function should be longer than 100 lines long (including internal comments).
- Pay close attention to which class you add a new function to. Make sure the placement of your functions makes sense. For example, you won't want to add any canvas-drawing code in the SketchComparison class. That class deals with comparing shapes and should not need to know about canvases.
- As a fun fact, the metrics you're dealing with in this assignment are components of a technique in sketch recognition and AI that I worked on as an undergraduate researcher and young graduate student. My specific concoction of metrics (including the Tanimoto Coefficient, among others) has been called the "Valentine Recognizer" by some members of the research community. So, you know, I'm kinda famous. NBD. :-)

Rubric

Points	Description
20	code/functionality for drawing results to the screen
20	code/functionality for calculating Hausdorff Distances
15	code/functionality for calculating Tanimoto Coefficient
10	code/functionality for comparing the selected shape with all templates
10	code/functionality for searching, sorting, and returning the top-k matches
10	structure: code appropriately modular
8	sufficient unit tests for calculation functions
2	unit tests run & pass
5	good software engineering principles (formatting, commenting, variable names, *etc.*)
5	**Extra Credit:** adding 2 or more other metrics for determining shape similarity
100+5	**TOTAL**