Fingerprint Recognition

Lưu Nam Đạt

Feb-08-2023

1. Summary

This repository offers 3 components:

1. A fingerprint pre-processing chain based on generic algorithms and tools for filtering image
2. An algorithm for extracting fingerprint's features, which are called "minutiae"
3. A high performance one-to-one fingerprint matching algorithm.

2. System

All the tests are performed on the following system:

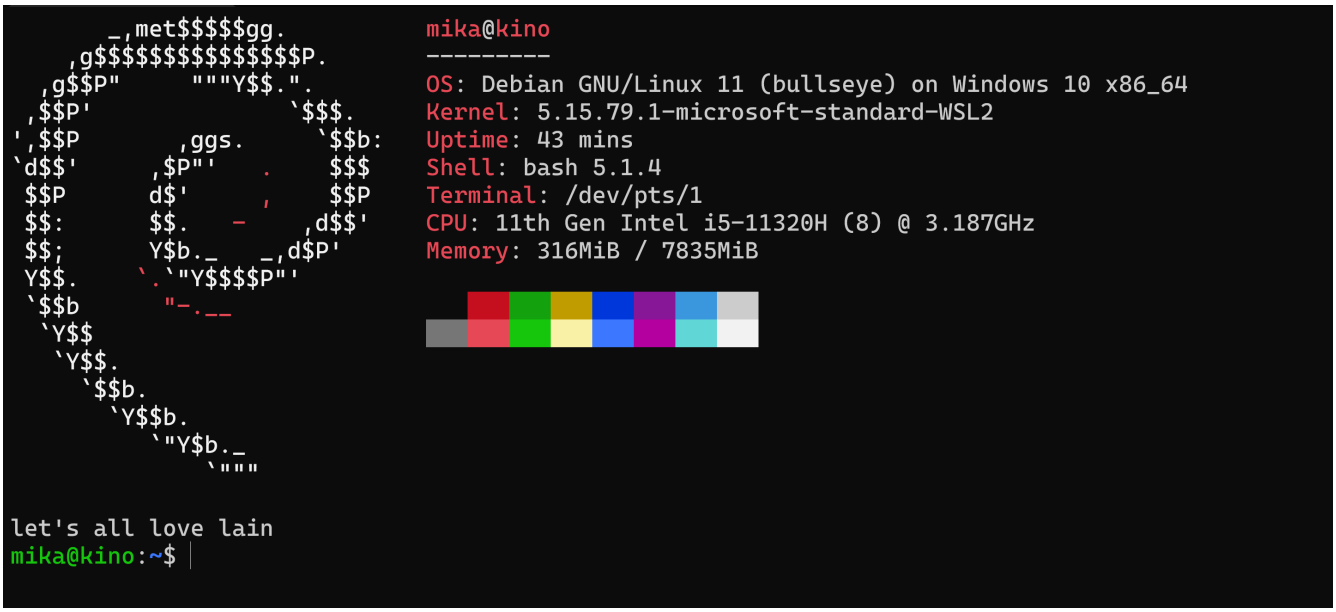


Figure 1: System Information

OS: Debian GNU/Linux 11 (bullseye) on Windows 10 x86_64
CPU: 11th Gen Intel i5-11320H (8) @ 3.187GHz
Memory: 7835MiB (about 320MiB on idle)

3. Pre-processing

The pre-processing pipeline used in this repository is a chain of fingerprint enhancement algorithms. The following images show the steps in order, as well as the result of each step.

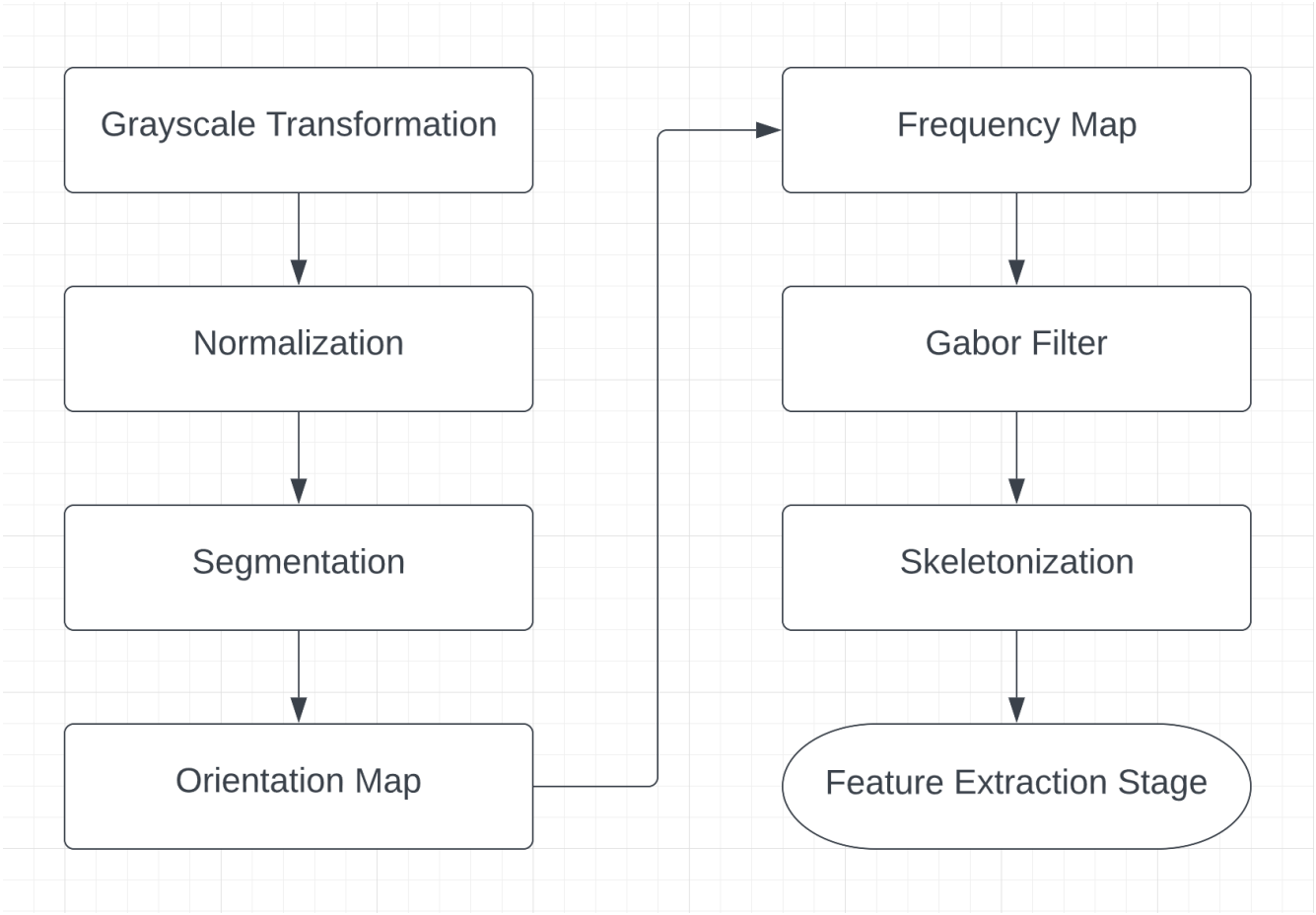


Figure 2: Pre-processing Pipeline

Figure 3: Pre-processing Pipeline - Visualized

3.1. Pipeline

This section summarizes the usage of each step. The detailed implementation explaination shall be addressed in another document.

- **Normalization**: Normalization is used to standardize the intensity values in an image by adjusting the range of gray level values so that they extend in a desired range of values and improve the contrast of the image. The main goal of normalization is to reduce the variance of the gray level value along the ridges to facilitate subsequent processing steps.
- **Segmentation**: In order to eliminate the edges of the image and areas that are too noisy, segmentation is necessary.
- **Orientation Map**: Tells the ridges' orientation in any region. This step provides necessary data for the *Gabor Filter*.
- **Frequency Map**: Tells the ridges' frequency in any region. This step also provides necessary data for the *Gabor Filter*.
- **Gabor Filter**: A linear filter used for edge detection. Gabor filter can be viewed as a sinusoidal plane of particular frequency and orientation, modulated by a Gaussian envelope (see Figure 4 - Gabor Filter)
- **Skeletonization**: Thins the gabor image (which is now a binary image) into its skeleton, using Zhang-Suen thinning technique.

3.2. Complexity Analysis

Let:
- N be the resolution of the input image
- R be the number of Gabor Filter's kernels variance (see Figure 4 - Gabor Filter)
- T be the maximum thickness of a fingerprint's ridge, measured in pixels

The complexity of the pipeline is then:

Step	Time Complexity	Space Complexity
Normalization	$O(N)$	$O(N)$
Segmentation	$O(N)$	$O(N)$
Orientation Map	$O(N)$	$O(N)$
Frequency Map	$O(N)$	$O(N)$
Gabor Filter	$O(RNlogN)$	$O(RN)$
Skeletonization	$O(TN)$	$O(N)$

Regarding the Gabor Filter,

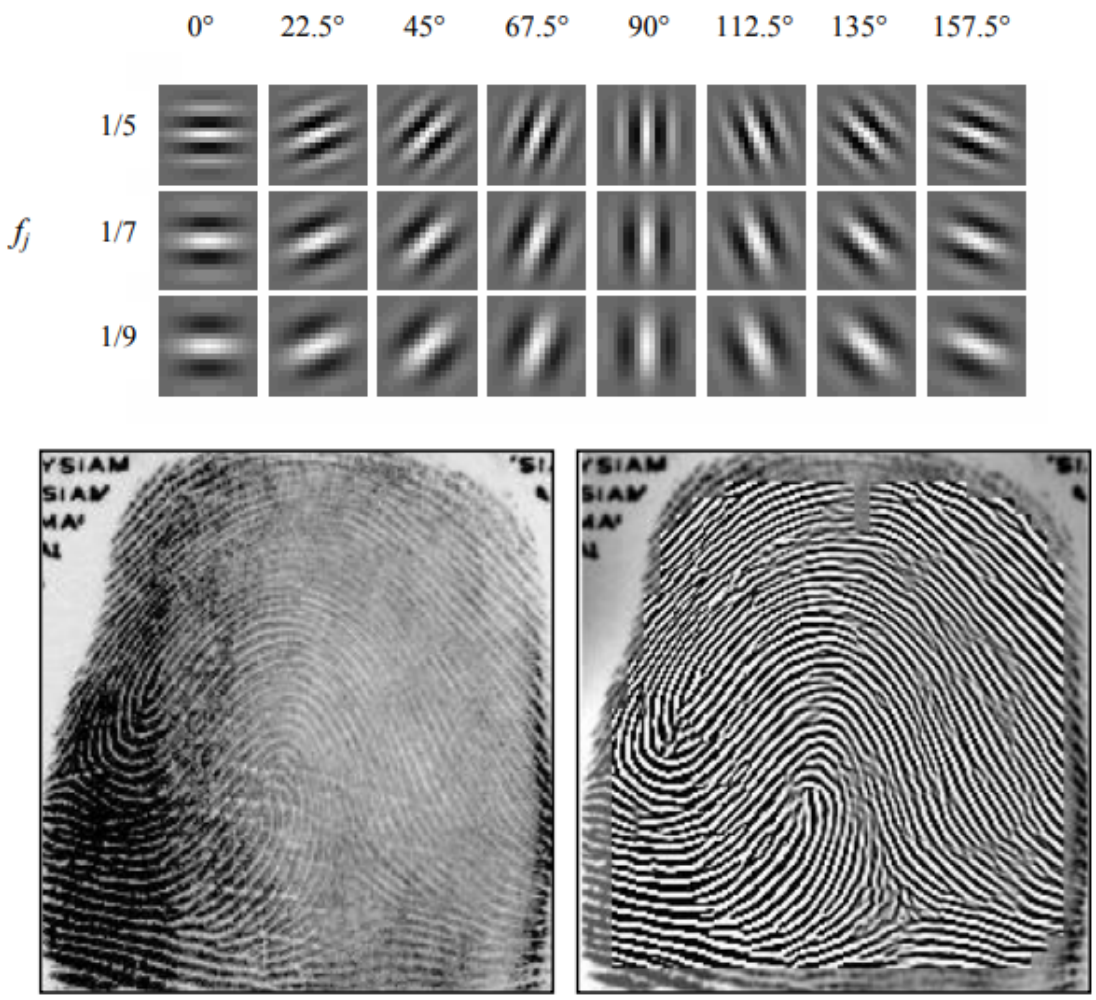


Figure 4: Gabor Filter

As portrayed in the figure above, Gabor Filtering is performed by using various kernels in their corresponding fingerprint blocks. The R variable in complexity calculation is determined by how many kernels are in use.

3.3. Benchmark

This benchmark shows the performance of this repository's preprocessing pipeline. It also add the former project into comparison.

The data set in used consists of 500 fingerprint images, each has the resolution of 320×480 pixels (Width $\times$ Height), white ridges & black background. You can see the data set here.

In order to make the benchmark result accurate and reliable, the program is single-thread. The fingerprints will be processed one after another. There is no paralleled process.

Processing 500 Images - Runtime of Individual Steps

Step	Old Version	This Version
Normalization	00:09.671	00:00.739
Segmentation	02:52.806	00:03.047
Post-Segmentation	00:09.447	–
Orientation Map	00:56.922	00:01.282
Frequency Map	00:14.788	00:01.573
Gabor Filter	17:02.519	00:45.201
Skeletonization	00:37.697	00:04.757

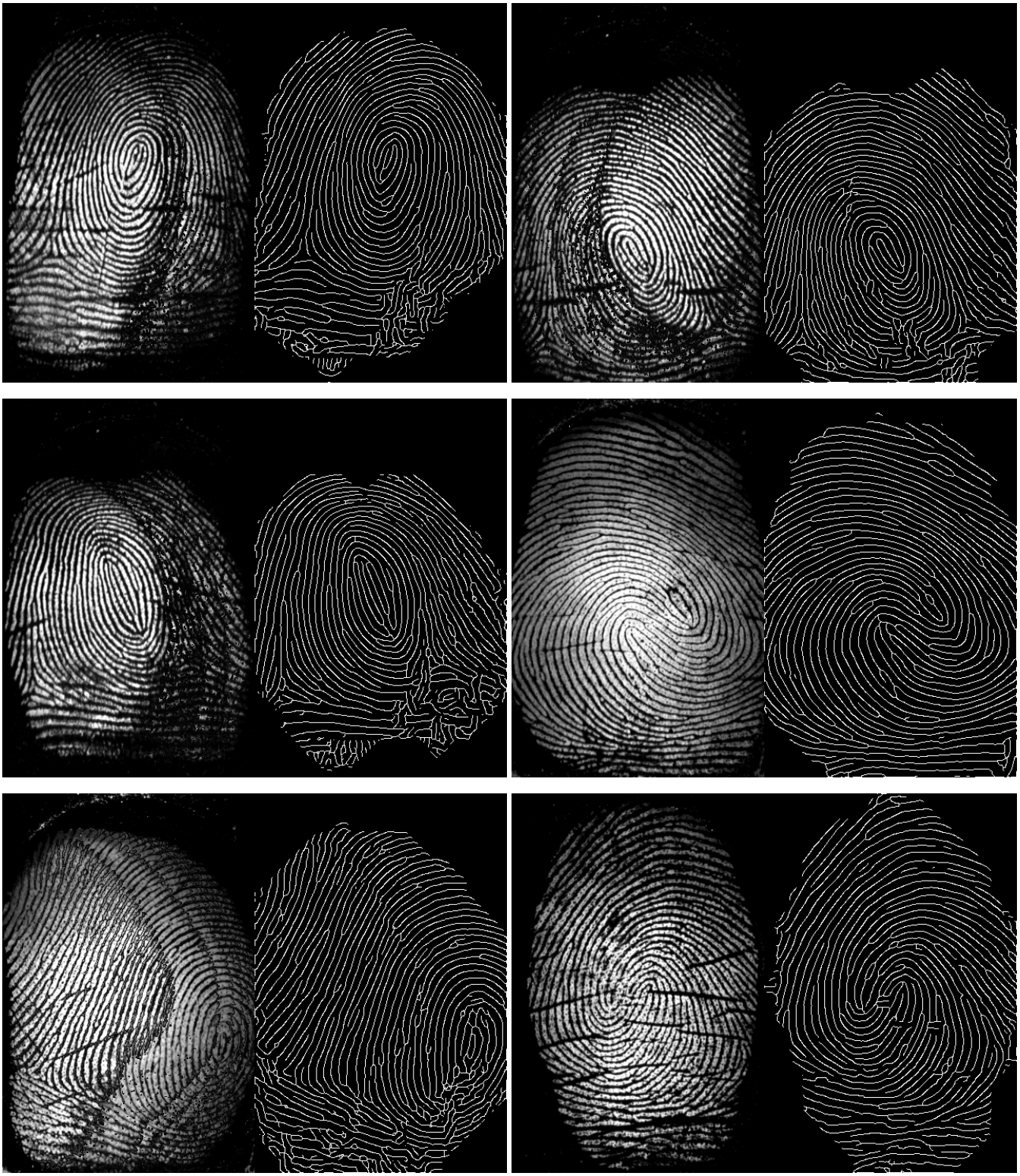
Processing 500 Images - Cummulative Runtime

Step	Old Version	This Version
Normalization	00:09.671	0:00.739
Segmentation	03:02.477	0:03.786
Post-Segmentation	03:11.924	–
Orientation Map	04:08.846	0:05.068
Frequency Map	04:23.634	0:06.641
Gabor Filter	21:26.153	0:51.842
Skeletonization	22:03.850	0:56.599

Average runtime for each fingerprint:
- Old version: 2.6477s
- This version: 0.1132s

Processing speed:
- Old version: 0.38 img/s
- This version: 8.83 img/s

3.4. Result Showcase



4. Feature Extraction

This module proposes an algorithm that extracts features from a fingerprint's skeleton image.

The features to be extracted are *terminations* and *bifurcations*, which look like this in a fingerprint:

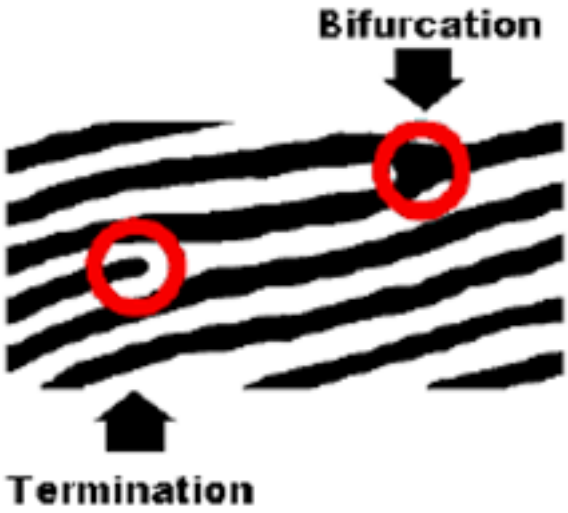


Figure 5: Termination and Bifurcation

4.1. Definitions

For each cell P_0 in the skeleton image, let its adjacent cells be called:

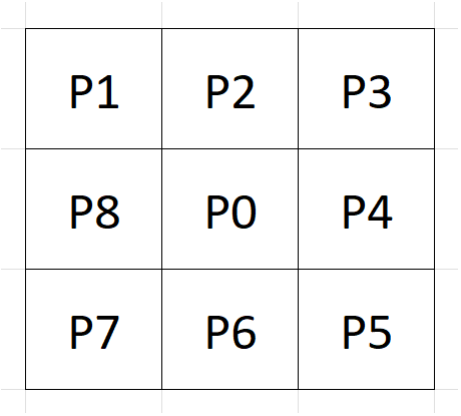


Figure 6: Relative position of each cell

Let c be the transitions count. Traverse through $P_1 \to P_2 \to P_3 \to P_4 \to P_5 \to P_6 \to P_7 \to P_8 \to P_1$, for each time a cell turns from white to black (in other words, `false` to `true`), increase c by one.

If a cell P is *black* has the value c equals to 1, consider it a *termination candidate*. If a cell P is *black* has the value c equals to 3, consider it a *bifurcation candidate*. For example, this is how a *termination* and a *bifurcation candidate* looks like:

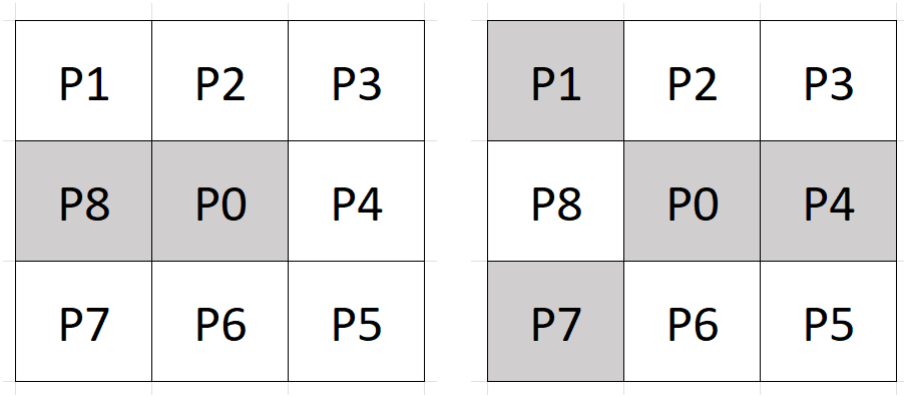


Figure 7: Termination Candidate and Bifurcation Candidate

The above procedure counts the transitions of cells that are one pixel away from P_0. Let c_i be the transitions count of cells that are i pixels away from P_0. If P_0 is a *termination candidate*, it becomes a *termination* if and only if:

$\forall i \in 2..r,\ c_i = c_1$

The same applies to *bifurcation candidates*.

5. Matching

5.1. Naive Approach

I'll explain this later on.

5.2. Improved Approach

This repository implements a matching algorithm proposed in *_doc/references/Improving Fingerprint Verification Using Minutiae Triplets.pdf*

In short, imagine a comparison between a fingerprint called *Probe* and another called *Candidate*. *Probe* has the minutiae set P and *Candidate* has Q. For each minutia p in P, found c nearest minutiae and build triplets (p_1, p_2, p_3) such as this:

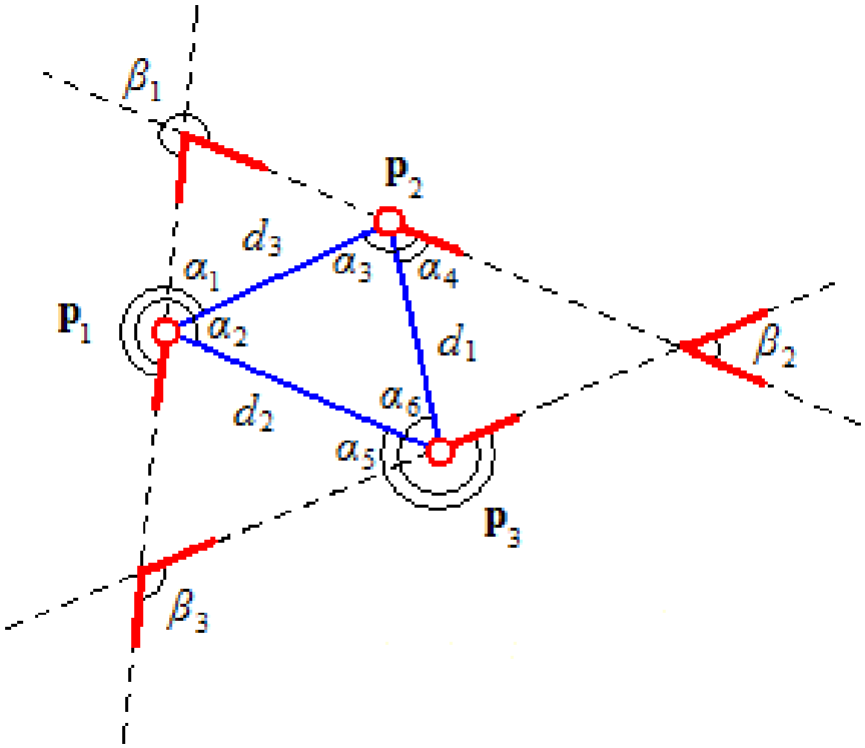


Figure 8: A Triplet

For each minutia q in Q, build triplets (q_1, q_2, q_3) just like so. Then for each triplets in *Probe*, find similar triplets in *Candidate*, add the pairs $(p_1, q_1), (p_2, q_2), (p_3, q_3)$ to collection A.

Then, bruteforce A in $O(N^2)$ with N is $|A|$. For each pair (p_i, q_i) in A: Assume these two are the same minutia in a fingerprint, find out how many pairs (p_j, q_j) that make (p_i, p_j) similar to (q_i, q_j). The number of pairs found is also the number of matching keypoints in the result.

5.3. Benchmark

This benchmark uses the same data set as the one in the **Pre-processing** section. The program is then challenged:

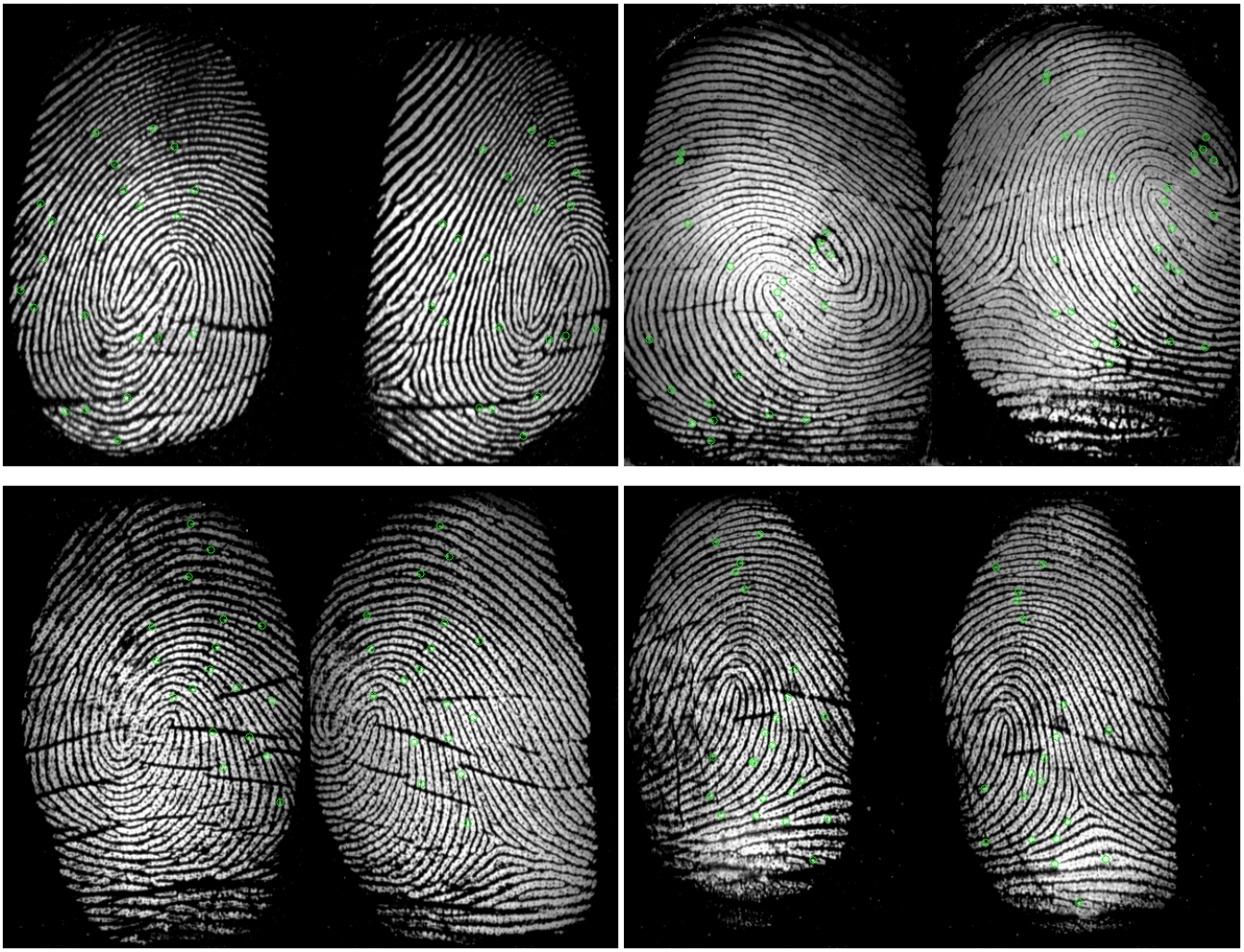
- Given N fingerprints. For each fingerprint, find another one that is the most similar.
- Because the project aims at fingerprint recognition, not fingerprint grouping, each comparison must be one-to-one. Thus, for this data set, there are 500×499 comparisons expected.

Again, the program is single-thread, which ensures the average time for each comparison is accurate and reliable. Each comparison is performed one after another. There is no paralleled process.

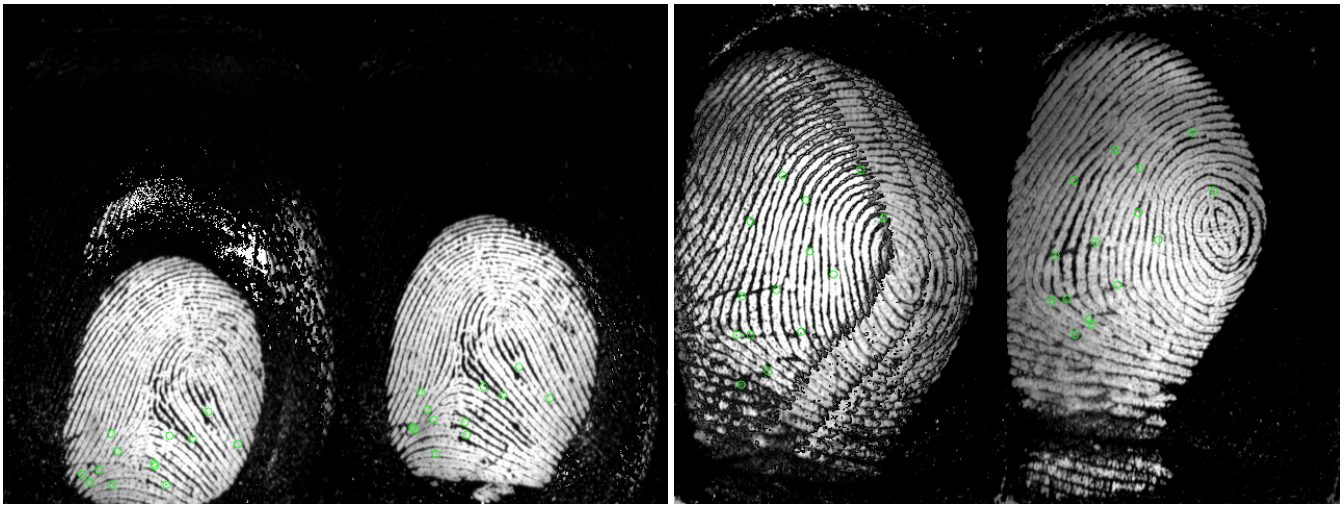
5.3.1. Test Result - Accuracy

481 out of 500 given fingerprints successfully found an accurate match. That makes the accuracy of **96.2%**, or **3.8%** mismatch rate.

In the used dataset, it's common that there's only one front version of a fingerprint and multiple versions of that same fingerprint in various angles. Regardless, the algorithm is proven to accurately match these pairs of fingerprints:



The algorithm also shows a decent resistance to noises, either by eliminating them, or by reconstructing them:



I decided to make things more challenging. I took an image out, erased more than half of the foreground, then replaced the old image in the data set with the new, edited one. Still, the algorithm performed another successful match.

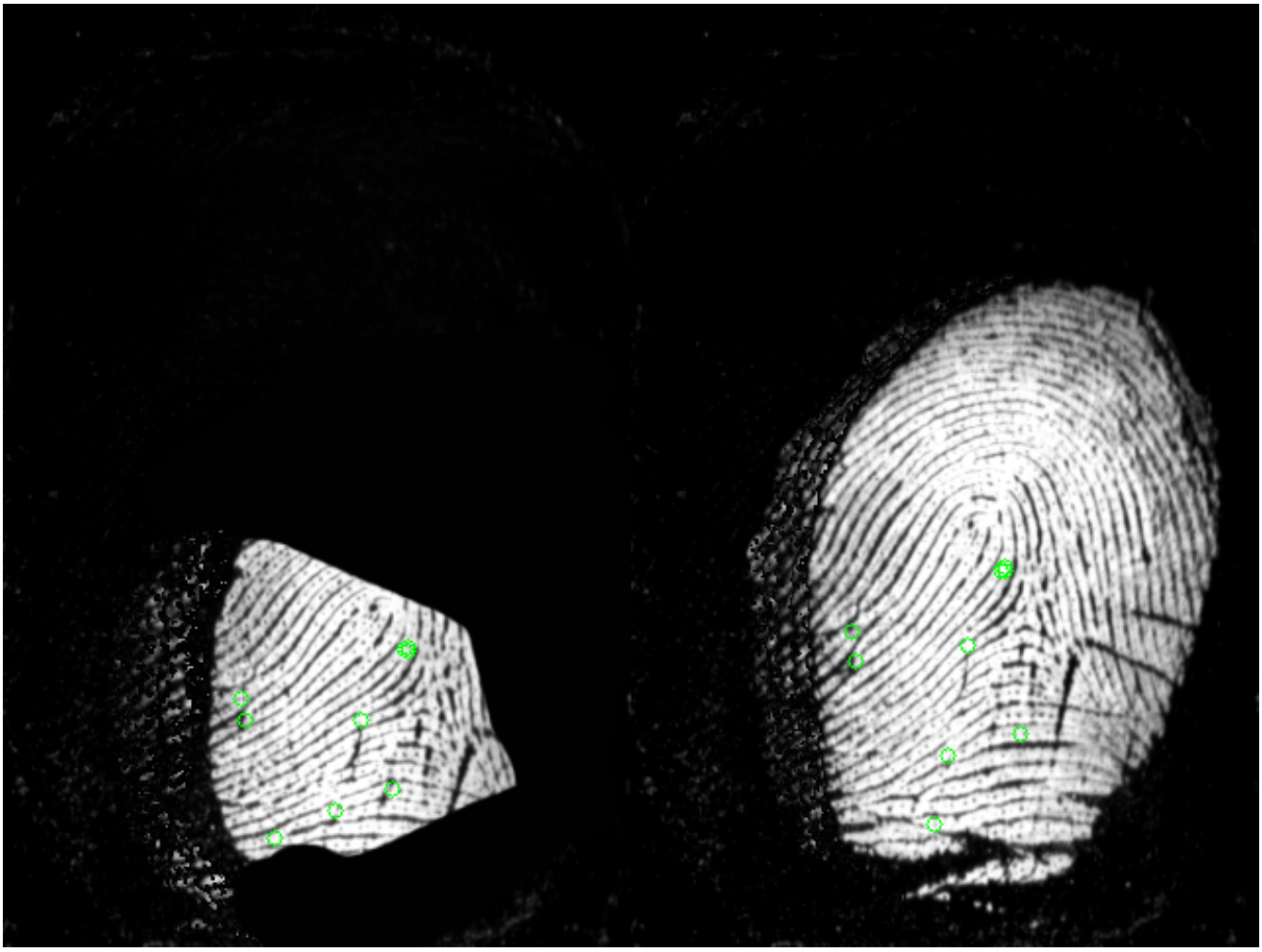


Figure 9: A little challenge

The algorithm can even accurately match fingerprints that share so little information to each other:

Figure 10: An amazing match

The match above seems like a false match, but it is not. They are really of a same finger, but are scanned from very different angles, which proves how amazing this matching algorithm is.

Here is the finger in question:

5.3.2. Test Result - Inaccuracy

This matching algorithm is imperfect, however. For this instance, the algorithm only took the minutiae in calculation, while ignoring the fingerprints' singularities:

Figure 11: The best match for the left fingerprint, according to the algorithm

Only once had I manually removed all the fingerprints that have different singularity type than the left one, the algorithm produced an accurate match:

Figure 12: The true closest match for the left fingerprint in the data set

Nonetheless, the fact that there are no other front versions of this fingerprint also plays a major role in this mismatch. The other versions of this fingerprint, however, were successfully paired.

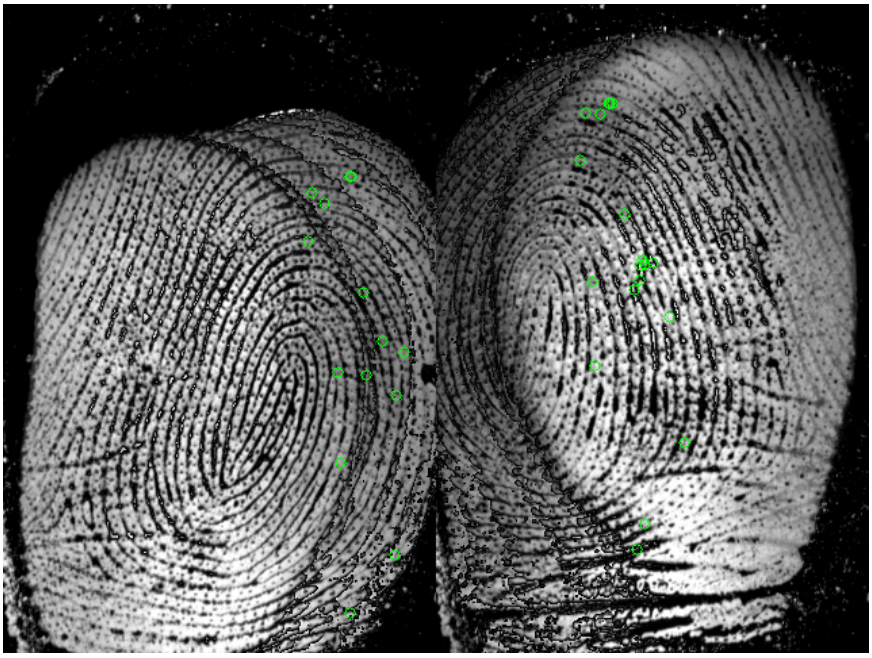


Figure 13: Other variations are paired accurately

Another instance of imperfectness was this case, in which the number of ridges between the minutiae was not put into consideration:

Figure 14: The best match for the left fingerprint, according to the algorithm

While the true match looks like this:

Figure 15: The true closest match for the left fingerprint

These three images have the same singularity type, have similar relative position between keypoints. Nevertheless, if the distances measured in ridges are taken into calculation, we can conclude that the algorithm made a false match.

5.3.3. Test Result - Performance

The comparison algorithm performs as follow:

- Executed $500 \times 499 = 249500$ comparisons: 37.738 s
- Average 0.1512×10^{-3} s, or 0.1512 ms per comparison
- Speed: 6611.37 comparisons/s

This repository offers a decent performance in single-thread, though it can be improved by 5x-10x if:

- All calculations use *int_8t* and *int_16t* instead of floating points
- Bit manipulation is used to speed up logical operations, addition, multiplication, division,…
- Memory manipulation and caching

In fact, those improvements are already implemented in experimental state.

Additionally, here is the benchmark of the former project compared to the current one:

Version	Old	New
Performs 249500 comparisons	4161.086s	37.738s
Average runtime per comparison	16.68ms	0.1512ms
Comparisons per second	59.96	6611.37
Accurate pairs	301	481
Accuracy	60.2%	96.2%