Abstract—Gesture recognition is an important technology in today’s world dominated by touch devices. This is why in this work we explore different algorithms able to recognize shapes inputed by users. We implemented those algorithms in a provided pre-existing platform and subjected them to a test group. We were able to put forward that some algorithms are more adapted to recognize gestures and others more adapted to recognize shapes.

I. INTRODUCTION

Gesture recognition-based applications are more and more used on small devices such as smartphones, tablets, GPS, etc. It is indeed often difficult to access buttons on these small screens hence gestures can be used to replace these inaccessible buttons. Obviously, this technique is also applicable on desktop software. On desktop modeling applications, gesture recognition can help software architects to work more efficiently, losing less time in menu browsing. The main concern of this kind of applications is the recognition algorithm used. Researches are done in order to improve the efficiency of pattern recognition processes [2]. This document first introduces three new algorithms to be added to François Beuven’s application [1]. Then the integration steps are detailed. The next parts then show the results obtained by a series of tests and the conclusions that can be drawn.

II. THEORETICAL POINTS

Three algorithms were developed in addition to the already implemented ones: Mahalanobis distance, Jaro-Winkler distance and nDollar.

A. Mahalanobis distance

The Mahalanobis distance can be used to measure the separation of two groups of objects. In the literature, it is well known that it is well suited for pattern recognition. In our case, we want to use this method to recognize gestures drawn on a graphical tablet.

This algorithm is based on the correlation between two classes of points representing patterns that can be identified and analyzed. The Mahalanobis distance can have three different forms.

One possibility is to use the Mahalanobis distance between two groups of values with means $\mu_1$ and $\mu_2$:

$$D_M = \sqrt{(\mu_1 - \mu_2)^T \Sigma^{-1} (\mu_1 - \mu_2)}$$  \hspace{1cm} (1)

where $\Sigma^{-1}$ is the inverse of the covariance matrix$^1$.

The second possibility is the Mahalanobis distance between one group of values (with mean $\mu$) and a specific point $(x, y)$:

$$D_M(x) = \sqrt{(x - \mu)^T \Sigma^{-1} (x - \mu)}$$  \hspace{1cm} (2)

And the last possibility is the Mahalanobis distance between two classes of values:

$$d(\vec{x}, \vec{y}) = \sqrt{(\vec{x} - \vec{y})^T \Sigma^{-1} (\vec{x} - \vec{y})}$$  \hspace{1cm} (3)

We can notice that if the covariance matrix is the identity matrix, this distance is reduced to the Euclidean distance. If we consider this latter as the distance between $x$ and the means (black crosses) of each class depicted on figure 1, $x$ will be assigned to $C_2$.

Figure 1: According to euclidean distance, $x$ belongs to the class $C_2$ since it is closest to the center of this distribution. According to the Mahalanobis distance, $x$ belongs to the class $C_1$ since this algorithm takes the sample distribution into account.

Otherwise, if we take the distribution probability into account, it is obvious, regarding figure 1, that $x$ will belong to $C_1$.

The fact that the distribution probability is part of the calculation is interesting for the kind of pattern recognition application we want to develop. Therefore, we chose the third equation (3) in order to measure the distance between each sample gesture recorded in the program and the gesture that the user just drawn.

B. Jaro-Winkler distance

The Jaro-Winkler distance is the Jaro distance with the Winkler modification.

\begin{align*}
\text{Jaro distance} & = \frac{\text{matches}}{\text{characters}} \\
\text{Winkler modification} & = \frac{\text{matches} + \text{characters} \times \text{edit distance}}{2 \times \text{characters}}
\end{align*}

1 Reminder: $\Sigma = \begin{pmatrix}
\text{var}(X) & \text{cov}(X, Y) \\
\text{cov}(Y, X) & \text{var}(Y)
\end{pmatrix}$
The Jaro distance is used to compute the distance between two strings (A & B) and is based on matching and transposition of letters. There are two phases to compute this distance, first the matching and then the transposition.

1) Matching: The matching phase computes a score based on the occurrences of characters in both strings. For each character in String A the algorithm will try to find the same character in String B in a given range around the character’s index in String A considering only String B’s characters that have not yet been matched. This range is half the length of the longest string minus one.

\[
\text{Range} = \frac{\max(\text{length}(A), \text{length}(B))}{2} - 1
\]  

(4)

Each time a match is found a point is awarded. This score is then divided by the longest string’s length to obtain the match score.

In this example the range will be 3, this means that to find a match for the “B” highlighted in String A the algorithm will search for a “B” in the highlighted zone of string B, note that the first two characters are excluded as they have already been matched to an other character. So if we take String A as reference we have:

![Figure 2: Strings A & B](image)

Note that we would have obtained the same score taking String B as a reference.

![Figure 3: Matching results taking String A as reference](image)

The score of the matching phase is thus 6.

2) Transposing: The transposing phase now evaluates how many characters in String A do not line up with the character at the same index in String B. We then divide this result by two and round down to obtain the transposition score. In this case:

![Figure 5: Exact matching results between the two Strings](image)

The score of the transposing phase is thus 8/2 = 4.

The equation: The Jaro distance equation is

\[
jDistance(A, B) = 1 - jCompare(A, B)
\]

(5)

where \( jCompare(A, B) \) is computed as follows. Note that match and transpose are the scores computed as explained above.

\[
jCompare(A, B) = \frac{1}{3} \frac{\text{match}(A, B)}{\text{length}(A)} + \frac{1}{3} \frac{\text{match}(A, B)}{\text{length}(B)}
\]

\[
+ \frac{1}{3} \frac{\text{match}(A, B) - \text{transpose}(A, B)}{\text{match}(A, B)}
\]

(6)

The compare score for our example would be

\[
\frac{1}{3} \times \left( \frac{6}{8} + \frac{6}{9} + \frac{6 - 4}{6} \right) = 0.58
\]

and thus the distance would be 0.42.

3) The Winkler modification: The Winkler modification acts as a boost on the compare score based on the comparison of the prefixes of the Strings. If the Jaro compare is above a given threshold, 0.7 in Winkler’s papers, the score is boosted accordingly with the number of matches between the prefixes of the Strings. The prefix size is an other variable and is usually set at 4 in Winkler’s papers. The prefix mach score is the number of exact matches in the prefix of the two strings.

\[
jwCompare(A, B, \text{threshold}, \text{prefixSize})
\]

if \( (jCompare(A, B) \leq \text{threshold}) \)

\[jCompare(A, B)\]

else

\[jCompare(A, B) + 0.1 \times \text{prefixMatch}(A, B, \text{prefixSize}) \times (1 - jCompare(A, B))\]

We thus have to change the computed distance accordingly:

\[
jwDistance(A, B) = 1 - jwCompare(A, B)
\]

(7)

This has for effect to further reduce to computed distance if the beginning of the word is similar.

C. nDollar

The third method we have developed in this project is called the \$N Multistroke Recognizer. Since this method is build upon the \$1 Unistroke Recognizer, a short description of the \$1 method is needed.

The principle of the \$1 method is quite simple: given an experimental unistroke E obtained by a graphic tablet or other devices, we have to choose, among a set of stored templates S, the closest template \( S_i \) from our unistroke E. The “Betweenness” or “closeness” between E and one template in S is simply computed using the average euclidean distance between the points forming the two unistrokes. Because both the experimental unistroke and the templates are unistrokes, an experimental unistroke can be immediately added to the set
of templates $S$ in order to allow the method to be taught and “learned” as the experiments are realized.

The main part of the $S1$ recognizer is its way to establish the corresponding points in the recognition process. On both the experimental unistroke $E$ and the templates $S$, this process is divided in four steps which are depicted on Figure 6:

1) **Resample**: The points forming the stroke are resampled such that they are separated by equal distances. This step is necessary since the points are not likely to be uniformly distributed because of the velocity of the move when drawing a stroke.

2) **Rotation**: The stroke is subject to a rotation such that its “indicative angle” is at 0 degree. The indicative angle is determined by the centroid $(\bar{x}, \bar{y})$ to the first point of the stroke. This serves as an approximation for alignment. Later, during the comparison, an other technique is used in order to find the optimal angular alignment minimizing the point-to-point distance.

3) **Scaling**: The stroke is scaled non-uniformly to match a reference square.

4) **Translation**: The stroke is translated such that its centroid is at the origin.

![Drawn Strokes](image)

**Figure 6**: $S1$ dollar steps

The application of these steps ensures that there is a one to one correspondence between points in the resulting stroke and the ones composing the templates. Hence, we can measure the point-to-point distances, given for strokes made of $N$ points by

$$d = \frac{\sum_{j=1}^{N} \sqrt{(E[j]_x - S[j]_x)^2 + (E[j]_y - S[j]_y)^2}}{N}$$

(8)

to find the closest template $S_i$ from our resulting stroke $E$. Once the stored template $S_i$ minimizing the path-distance is found, we convert this distance to a $[1 \ldots 0]$ score by the following formula

$$score = 1 - \frac{d}{\frac{1}{2} \sqrt{size^2 + size^2}}$$

(9)

Given this explanation of the $S1$ dollar algorithm, we can now explain easily the principle of the $S_N$ recognizer.

The $S_N$ Multistroke Recognizer, as hinted by his name, has the capability to recognize a multistroke gesture where the component strokes are realized in any order and in either direction. To permit this flexibility, given a multistroke defined by a designer, the $S_N$ recognizer computes and stores each multistroke’s “unistroke permutations”. Each permutation represents one possible combination of stroke order and direction which is then made into a unistroke by simply connecting the endpoints of component strokes. This is, in effect, treating a multistroke as if it were fundamentally a unistroke, but where “junction” strokes are made away from connecting the endpoints of component strokes. This technique of course gives us a huge number of unistroses from a multistroke made of $N$ strokes. First, we have to compute all the possible orders for our strokes which is simply given by the number of possible permutations of our $N$ strokes:

$$N! \times 2^N$$

(11)

This technique, we are able to treat any multistroke gesture made by a user as a unistroke gesture where the $S1$ dollar principle is applied on it to find the most similar multistroke using its unistroke permutations.

### III. Implementation

This section explains how we have implemented the algorithms described in the previous section.

We have used the platform developed by François Beuven and Thierry Dullier for their master thesis [1]. Basically, this platform allows two things:

1) **Record templates.** A template is a gesture drawn by a user. The user is asked to draw a specific pattern. The platform will then record this gesture for a future use.

2) **Recognize gestures.** A gesture is drawn by a user on a tablet. The platform will then analyze this gesture and compare it to the set of templates it knows from the first step. The platform will then output the result to the user.

Thanks to the fact that this platform is open-source, we have modified it in order to add the three algorithms previously discussed. Note that the platform allows to store the templates, so they can be used afterwards. Except for $S_n$, we have used the recorded templates provided to the platforms in order to train our algorithms.

#### A. Mahalanobis distance

The Mahalanobis distance algorithm has been completely integrated into the existing implementation of the $S1$ algorithm. Initially, the $S1$ implementation used a euclidian distance. We have improved this algorithm in such a way that the user have now the choice of the distance algorithm the $S1$ recognizer should use: euclidian distance or Mahalanobis distance.

$$d = \frac{\sum_{j=1}^{N} (E[j]_x - S[j]_x)^2 + (E[j]_y - S[j]_y)^2}{N}$$

(10)
B. Jaro-Winkler distance

In this section, we will describe how we have implemented the Jaro-Winkler algorithm. This implementation is very close to the implementation of the Levensthein algorithm. Indeed, since the Jaro-Winkler is an algorithm that compares two strings, we begin by re-using the utils.java class from the Levenshtein algorithm that provides some functionalities such as a method that transforms a set of points into a string of orientation values. These orientation values are calculated by using the compass defined at figure 7.

![Figure 7: Compass for orientation value.](image)

Let’s take the point \((x_1, y_1)\) and the point \((x_2, y_2)\) where \(y_2 > y_1\) then the direction between the first and the second point will be 0. Note that the direction between the second point and the first one will be 4.

In fact with this example we can see two major limitations of the implementation that prevents it to correctly use the Levenshtein algorithm and also the Jaro-Winkler algorithm. The first one is that no threshold was used when the application calculates the orientation. Indeed, if someone wants to draw a vertical down arrow, so from the top to the bottom of the tablet, it will be impossible for this person to draw a perfectly vertical line. The string of orientation values will probably be something like \([1,7,7,1,1,1,7,1,0,7,1,7,1,7,1,1,1]\) instead of \([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]\). This result comes from the fact that the application returns a direction equals to 0 only if \(x_2 = x_1\) and \(y_2 > y_1\). A good idea would have been to use a kind of threshold. For example \(T = 1\) and the comparison would be something like

```java
if (abs(x1-x2) < T && y2 > y1)
    return 0;
```

This modification would avoid the algorithm to return Zigzag when the user draws a simple line. In addition it would have been an interesting parameter to calibrate in order to check performances. Note that, in practice, this imprecision is invisible because the comparison of the drawing of a user is made with other drawings done by the same user during the training.

The second issue appears during the training phase. Let us take a circle. There is an infinite number of ways to draw a circle, beginning by the top, by the left and so on. These cases are correctly checked by the algorithm due to the fact that user is able to allow rotation for the Levenshtein and Jaro-Winkler algorithm. The problem is the answer to the following question:

*What happens when the application compares two circles drew in the two different ways described at figure 8?*

![Figure 8: The two ways of drawing a circle.](image)

In fact, we have seen that the application correctly identifies the first circle but completely failed to recognize the second circle and returns something like W. This problem is due to the fact that the first circle is represented by a string equals to \([6,6,5,4,4,3,3,2,2,1,1,1,0,7,7,7,6,6]\) and the second one will be represented by a string equals to \([2,2,2,3,3,3,4,5,5,6,6,7,7,7,0,1,1,2,2,2]\). The template registered only corresponds to the first way to draw a circle. This problem can be avoided by implementing a function that would be able to reverse the way of drawing by addition and subtraction of some points to each value in the string.

This improvement might interfere with shapes that rely on specific directions, such as arrows. It should thus be optional or there should be a way to specify that the gesture should not be reversed.

C. nDollar

The $n algorithm provides a unique feature that none of the others algorithms provides. Indeed, the $n takes into account the number of strokes the user does when she draws her gesture. Unfortunately, the platform does not support that feature, since the gesture is recorded as one unique unistroke. Thus, the $n implementation does not use the training facilities provided by the platform. Instead, we have hard-coded a set of gestures that our implementation is able to recognize. Besides, it would not make sense to test the unistroke gesture against our implementation, since $n is an improvement from $1, and the sole purpose of this improvement is to support multi-strokes gestures.

When the platform tries to recognize a gesture with our implementation, it does not take into account the number of strokes. That is really not a big deal, since the fist step of
the recognition part of \( n \) is to transform the multi-stroke into one resampled unistroke. However, due to this limitation of the platform, we have not implemented an optimization proposed by [5]. This optimization consists of considering only the templates that have the same number of stroke that the gesture drawn by the user.

IV. Tests

This section details how the tests were led and the results that came from those tests.

A. Procedure

The same test was proposed to 16 people of different age, different gender, different computer knowledges and different educational background. The tests were led as following:

Three roles were defined:
1) The user: the person who passes the test.
2) The supervisor: the person who launches the recognition process with different algorithms.
3) The scribe: the person who shows to the user the gesture he must perform and writes down the results.

Before starting the test phase, the supervisor is responsible for training the algorithms. In this context, training an algorithm means giving it some examples of gestures to permit it to recognize future gestures. The reference gestures that were used have been taken from François Beuvens’ master thesis. They contain five sets of all gestures made ten times.

The rules that the user must respect during the test are simple:
- A gesture can be done only once. The user cannot ask to restart the test or to redraw an already drawn gesture.
- The user must write as naturally as possible. He must use the pen on the tablet as a normal pencil on a sheet of paper.

We had an issue when we had to define the set of gesture our tests will have to recognize. Indeed, the \( n \) algorithm is optimized in order to recognize multisroke gestures. If the input of the \( n \) algorithm is a unistroke, the result will be very likely the same than the \( 1 \) algorithm. In the meanwhile, the two others algorithms can only recognize unistroke gestures. Thus, we have defined three sets of gestures to be recognized. The first set contains gestures that can be unistroke or multistroke. We have not given any informations of the user if she had to drawn unistroke or multistroke gesture. We will then analyze which algorithm performs best, depending of the user preference. The second and third sets contains respectively unistroke and multisroke gestures. Again, we have not informed the user if she had to draw unistroke or multistroke gesture.

The first set of gestures that the users were asked to perform (Figure 9) is composed of five geometrical shapes and five action commands. Note that the small arrows on the action commands only indicate the direction of the gesture and must not be drawn. Each of those ten gestures was analysed by the three newly implemented algorithms.

The second set of gestures is composed of five lower-case letters of the alphabet (Figure 10). The difference with the first set is that these five gestures were only analysed by Jaro-Winkler and Mahalanobis algorithms.

The last set of gestures is composed of five upper-case letters of the alphabet (Figure 11). As with the second set, the third one was not analysed by all algorithms but only with nDollar.

Every time a gesture is performed by the user, the supervisor launches the corresponding algorithms and the scribe records the result returned by the algorithm.

At the end of this process, fifteen gestures have been analysed by each algorithm.

B. Results

In this section, we analyze the performance of the three different algorithms described previously. To have a relevant and detailed comparison between these algorithms, we have divided the results according to the type a gesture, i.e the geometrical forms, the arrows, the lower case letters and the upper case letters.

1) Geometrical forms: In order to evaluate the performance of our three algorithms on geometrical forms (see Figure 9), we compute the percentage of correct recognition for each algorithm. For each users test, we count the number of geometrical forms that were correctly recognized divided by the number of forms, 6. Hence, for each test, we obtain a percentage of recognition. The average percentage of recognition on all tests for each algorithm is depicted on Figure 12.

As we can see, the Mahalanobis algorithm has the highest rate of recognition (72%). Statistical tests confirm that the Mahalanobis technique has better results than the \( n \) technique \((t(30) = 2,6, p > 0,01)\) and also the Jaro technique of course \((t(30) = 4,30, p > 0,0005)\).
Also, the results reveals the poor efficiency of the Jaro technique on geometrical forms (42,5%) which is worse than the $sn$ method ($t(30) = 2,19, p > 0,025$).

This is due to the fact that the Jaro Wrinkler algorithm has been designed to recognise words and searches for letter inversions. In this case the string given as input is not a word but rather a set of direction values. Trying to search for inversions inside those strings is thus not very relevant as this would mean the user has inverted two directions while drawing the shape. Furthermore the Wrinkler modification which boosts the score when detecting a similar prefix has no real reason to be used in this context as it would only try to recognise the beginning of the gesture, which is often irrelevant due to the stylus making unwanted movements when first touching the tablet. Mahalanobis has a good recognition rate for geometrical forms because it uses an improved euclidian distance algorithm which has a good efficiency for common shapes recognition.

2) Arrows forms: The same principle is applied in this section to evaluate the efficiency of the algorithms but on the arrow forms (see Figure 9). The average recognition rate for the algorithm is represented in Figure 13.

Figure 13 reveals very different performance for our algorithms. Indeed, the Mahalanobis is significantly worse than the Jaro techniques ($t(30) = 5,19, p > 0,0005$) unlike results computed on the geometrical forms. Also, the Jaro method has the best recognition rate (70,2%) and is significantly better than the $sn$ method ($t(30) = 4,16, p > 0,0005$).

The Jaro-Wrinkler algorithm achieves the best performances on this test. This is probably due to the fact that the algorithm takes directions and not points as input and is thus more able to recognise the arrow directions. Mahalanobis has a bad recognition rate for arrows because it does not take rotation and direction of the strokes into account. Indeed, some of the stored arrow templates are simple rotation of each others. Since the direction of the stroke is not taken into account, it is not possible to distinguish some of them and the recognition is ambiguous.

3) Lower case letters: Again, the same principle is applied in this section to evaluate the efficiency of the algorithms but on the lower case letters (see Figure 10). The average recognition rate for the algorithm is represented in Figure 14. Since the test on lower case letter is only applied on the Jaro-Wrinkler method and the Mahalanobis method as explain previously, there is no possible comparison with the $sn$ method.

Figure 14 reveals a poor recognition rate for the Jaro-Winkler method while the Mahalanobis method has a much better rate of 70% as compared to the Jaro-Wrinkler method ($t(30) = 4,7, p > 0,0005$).

As we have seen during our tests on the Geometrical forms, the Jaro-Wrinkler algorithm is not an algorithm adapted to shape recognition which explains its low score. Mahalanobis has a good recognition rate for lower case letters for the same reason as for geometrical forms.

4) Upper case letters: The last evaluation concerns the $sn$ method on the upper case letters (see Figure 11). The average recognition rate for the algorithm is represented in Figure 15.

The recognition rate of the $sn$ method is not as good as we expect it (66%) but is still encouraging since several optimization in our implementation can be made in order...
to improve its performance as explain in the implementation section.

V. CONCLUSION

The goal of this work was to evaluate the ability of three different algorithms to recognize shapes and gestures. During the first stage we implemented them into the existing platform developed by François Beuvens and Thierry Dullier. In the second phase we asked our test group to use those algorithms in order to recognize what they wrote. From those tests we can conclude that some algorithms are not adapted to recognize handwriting and others are not adapted to recognize directions, or at least not in the way their inputs are currently modeled. For instance the Wrinkler modification in the Jaro-Wrinkler algorithm only makes sense to compare words but not to compare strings containing directions values representing gestures. On the other hand algorithms such as Mahalanobis are not really adapted to recognize movements. We thus think it would be best to use different algorithms depending if we want to recognize shapes or gestures. As future works we think it could be interesting to develop a mechanism able to recognize if the user drew a shape or a gesture in order to further analyze it with an adequate algorithm.

REFERENCES

[2] François Beuvens and Jean Vanderdonckt. UsiGesture: an Environment for Integrating Pen-based Interaction in User Interfaces. 2010, Pisa, Italy