On the Searchability of Electronic Ink

Daniel Lopresti *Andrew Tomkins*

Matsushita Information Technology Laboratory
Panasonic Technologies, Inc.
Two Research Way
Princeton, NJ 08540
USA
dpl@mitl.research.panasonic.com
andrewt@mitl.research.panasonic.com

January 9, 1997

Abstract

Pen-based computers and personal digital assistant's (PDA's) are new technologies that are growing in importance. In previous papers, we have espoused a philosophy we call "Computing in the Ink Domain" that treats ink as a first-class datatype.

One of the most important questions that arises under this model concerns the searching of large quantities of previously stored pen-stroke data. In this paper, we examine the ink search problem. We present an algorithm based on a known dynamic programming technique, and examine its performance under a variety of circumstances.

Keywords: pen computing, approximate string matching, edit distance.

1 Introduction

Despite several early, high-profile "flops," pen-based computers and personal digital assistants (PDA's) are important technologies that are now starting to find acceptance. This synthesis of new hardware and software raises many systems-level issues, including the possibility of new paradigms for human-computer interaction. In previous papers, we have espoused a philosophy we have come to call "Computing in the Ink Domain" [3, 2, 4]. Here, the need for traditional handwriting recognition (HWX) is often deferred and sometimes even eliminated. Instead, the day-to-day functionality that users require is realized by treating ink as a first-class datatype.

One of the most important questions that arises under this model concerns the searching of large quantities of previously stored pen-stroke data. While efficient, well-known techniques exist for matching simple text strings exactly (*e.g.*, Knuth-Morris-Pratt, Boyer-Moore), with wild-cards (*e.g.*, Unix `grep`), and approximately (*e.g.*, Levenshtein or edit distance), there is no comparable body of work for the problem of matching ink strings. In this paper, we examine the following question: Given an ink text and an ink pattern, is it possible to search through the text and find all occurrences of the pattern without overwhelming the user with false "hits"?

As we have noted previously (*e.g.*, [4]), ink search serves as the basis for a broad range of user-oriented functionality. The concept is quite general, as illustrated by the following examples:

- Pages of handwritten notes can be searched for keywords.

- Pictographic filenames can be "looked-up" without having to scroll through long browsers.
- Handwritten e-mail addresses can be mapped to previously written and recognized addresses using a translation cache.
- Pen-based commands, including simple gestures, can be matched against a dictionary of possible commands to simplify the user interface.

By not constraining pen-strokes to represent "valid" symbols over a small, fixed alphabet, a much richer input language is made available to the user. This philosophy of *recognition-on-demand* is more distinctly "human-centric" than traditional HWX, which reflects a "computer-centric" orientation.

2 Definitions

Ink is a sequence of time-stamped points in the plane:[1]

$$ink = (x_1, y_1, t_1), (x_2, y_2, t_2), \ldots, (x_k, y_k, t_k) \tag{1}$$

Given two ink sequences T and P (the *text* and the *pattern*), the ink search problem consists of determining all locations in T where P occurs. This differs significantly from the exact string matching problem in that we cannot expect perfect matches between the symbols of P and T. No one writes a word precisely the same way twice. Ambiguity exists at all levels of abstraction: points can be drawn at slightly different locations; pen-strokes can be deleted, added, merged, or split; characters can be written using any of a number of different "allographs," etc. Hence, approximate string matching is the appropriate paradigm for ink search.

A standard model for approximate matching is provided by edit distance, also known as the "k-differences problem" in the literature. In the traditional case [9], the following three operations are permitted:

1. delete a symbol,[2]
2. insert a symbol,
3. substitute one symbol for another.

Each of these is assigned a cost, c_{del}, c_{ins}, and c_{sub}, and the edit distance, $d(P, T)$, is defined as the minimum cost of any sequence of basic operations that transforms P into T. This optimization problem can be solved using a well-known dynamic programming algorithm. Let $P = p_1 p_2 \ldots p_m$, $T = t_1 t_2 \ldots t_n$, and define $d_{i,j}$ to be the distance between the first i symbols of P and the first j symbols of T. Note that $d(P, T) = d_{m,n}$. The initial conditions are:

$$\begin{array}{lcll} d_{0,0} & = & 0 & \\ d_{i,0} & = & d_{i-1,0} + c_{del}(p_i) & 1 \le i \le m \\ d_{0,j} & = & d_{0,j-1} + c_{ins}(t_j) & 1 \le j \le n \end{array} \tag{2}$$

and the main dynamic programming recurrence is:

$$d_{i,j} = \min \left\{ \begin{array}{lcl} d_{i-1,j} & + & c_{del}(p_i) \\ d_{i,j-1} & + & c_{ins}(t_j) \\ d_{i-1,j-1} & + & c_{sub}(p_i, t_j) \end{array} \right. \qquad 1 \le i \le m,\ 1 \le j \le n \tag{3}$$

[1] Pen-tip pressure is also sometimes available – in this paper we do not make use of this.

[2] The term "symbol" is often taken to mean a text character. Here we use it much more generally – a symbol could also be a pen-stroke, for example.

When Equation 3 is used as the inner-loop step in an implementation, the time required is $O(mn)$ where m and n are the lengths of the two strings.

This formulation requires the two strings to be aligned in their entirety. The variation we use for ink search is modified so that a short pattern can be matched against a longer text. We make the initial edit distance 0 along the entire length of the text (allowing a match to start anywhere), and search the final row of the edit distance table for the smallest value (allowing a match to end anywhere). The initial conditions become:

$$\begin{array}{rcll} d_{0,0} & = & 0 & \\ d_{i,0} & = & d_{i-1,0} + c_{del}(p_i) & 1 \leq i \leq m \\ d_{0,j} & = & 0 & \end{array} \tag{4}$$

The inner-loop recurrence (*i.e.*, Equation 3) remains the same.

Finally, we must define our evaluation criteria. It seems inevitable that any ink search algorithm will miss true occurrences of P in T, and report false "hits" at locations where P does not really occur. Quantifying the success of an algorithm under these circumstances is not simple. The field of information retrieval concerns itself with a similar problem in a different domain, however, and seems to have converged on the following two measures [7]:

Recall The percentage of the time P is found.
Precision The percentage of reported matches that are in fact true.

Obviously it is desirable to have both of these measures as close to 1 as possible. There is, however, a fundamental trade-off between the two. By insisting on an exact match, the precision can be made 1, but the recall will undoubtedly suffer. On the other hand, if we allow arbitrary edits between the pattern and the matched portion of the text, the recall will approach 1, but the precision will fall to 0. For ink to be searchable, there must exist a point on this trade-off curve where both the recall and the precision are sufficiently high.

3 Approaches to Searching Ink

Ink can be represented at a number of levels of abstraction, as indicated in Figure 1. At the lowest level, ink is a sequence of points. At the highest, ink is ASCII text.[3] It is natural to assume that ink search could take place at any given level, with its attendant advantages and disadvantages.

As can be seen from the figure, at each step ink is represented as a collection of higher-level objects. Some of the earlier information is lost, and a new representation is created that (hopefully) captures the relevant information from the previous level in a more concise form. So, for instance, it may be impossible to know from the final word which allographs were used, or to know from the feature vectors exactly what the ink looked like, etc. Each stage in the process can be viewed as a recognition task (*e.g.*, strokes from points, words from allographs), and introduces the possibility of new errors.

An ink search algorithm could perform approximate matching at any level of representation. At one end of the spectrum, the algorithm could attempt to match individual points in the pattern to points in the text. At the other extreme, it could perform full HWX on both the pattern and the text, and then apply "fuzzy" matching on the resulting ASCII strings (to account for recognition errors).

[3] For concreteness, we assume HWX returns ASCII strings, but the reader may substitute any fixed character set as appropriate.

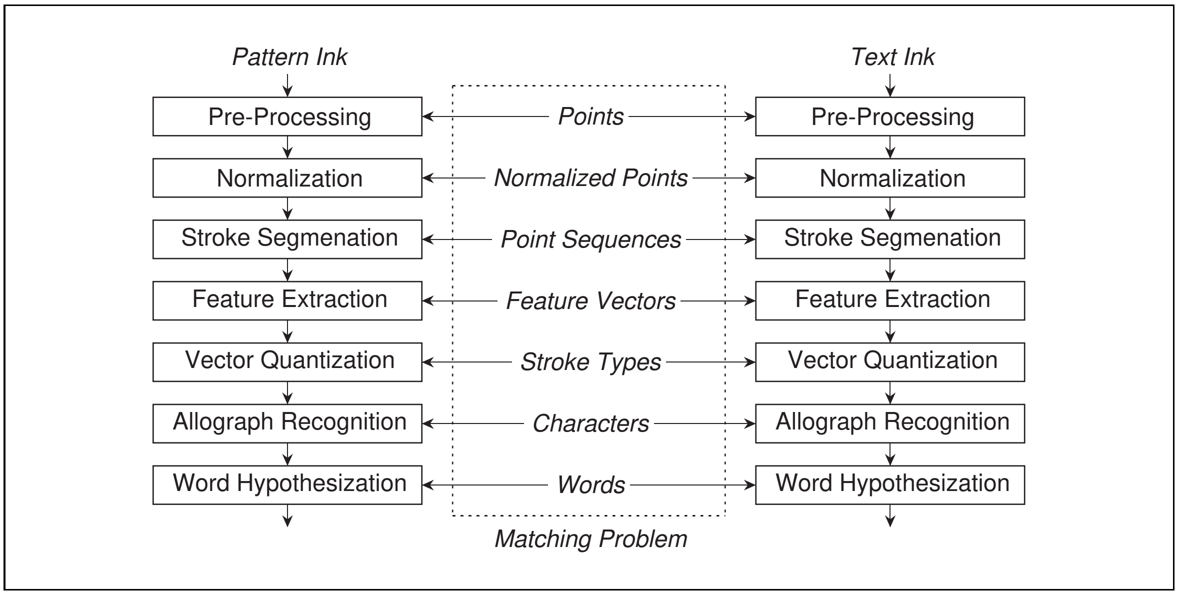

Figure 1: Handwriting recognition stages and potential matching problems.

In Section 4, we consider the latter option by examining how randomly introduced "noise" affects recall and precision for text searching. The point here is to gain some intuition about the performance of ink search algorithms built on top of traditional handwriting recognition.

Section 5 presents the primary contribution of this paper: an in-depth examination of an algorithm we call *ScriptSearch* that performs matching at the level of pen-strokes. This approach has the advantage of allowing us to do quite well against a broad range of handwriting, including some so bad that a human might find it illegible. ScriptSearch also allows the possibility of matching between strings with no obvious ASCII representation, such as equations, drawings, doodles, etc.

4 Searching for Patterns in Noisy Text

In this section we assume that the text and pattern are both ASCII strings, but that characters have been deleted, inserted, and substituted uniformly at random. This "simulation" has two purposes. First, it allows us to apply the recall/precision formulation in a familiar domain to develop intuition about acceptable values. Second, this model corresponds to the problem of matching ink that has been translated into ASCII by HWX with no manual intervention to correct recognition errors. Of course, these values are only an approximation since HWX processes in general do not exhibit uniform error behavior across all characters.

To illustrate the effects of noise on pattern matching, consider what happens when we search for a number of keywords in Herman Melville's novel *Moby-Dick*. Figure 2 tabulates average recall and precision under a variety of scenarios. Here *garble rate* represents a uniformly random artificial noise source that deletes, inserts, and substitutes characters in the pattern and the text. Note that when there is some "fuzziness," the precision can drop off rapidly if we require perfect recall. At some point, the text is no longer searchable as too many false hits are returned to the user. This is what we mean when we ask the question: Is ink searchable?

Another view of the data is to consider the precision realizable for a given recall rate. This is shown in Figure 3. An intuitive interpretation of this figure is that no threshold is necessary if a

Edit Distance Threshold	Garble Rate 0% Recall	Precision	10% Recall	Precision	20% Recall	Precision
0	1.000	1.000	0.274	0.995	0.003	0.996
1	1.000	0.875	0.643	0.901	0.280	0.944
2	1.000	0.610	0.910	0.581	0.664	0.700
3	1.000	0.329	0.986	0.326	0.886	0.424
4	1.000	0.121	1.000	0.097	0.981	0.154
5	1.000	0.021	1.000	0.015	0.999	0.048
6	1.000	0.010	1.000	0.010	1.000	0.013

Figure 2: Searching for keywords in *Moby-Dick* (as a function of threshold).

Recall	Garble Rate 0% Precision	10% Precision	20% Precision
0.1	1.000	0.950	0.901
0.2	1.000	0.950	0.901
0.3	1.000	0.950	0.896
0.4	1.000	0.950	0.771
0.5	1.000	0.928	0.678
0.6	1.000	0.909	0.616
0.7	1.000	0.814	0.564
0.8	1.000	0.744	0.408
0.9	1.000	0.604	0.289
1.0	1.000	0.102	0.018

Figure 3: Searching for keywords in *Moby-Dick* (as a function of recall rate).

ranked list of matches is returned to the user. In this case, for example, at a 10% garble rate, the user will experience a precision of 0.928 in viewing 50% of the true hits for the pattern.

Of course, real text (without noise) is searchable using routines like Unix `grep`, etc. However, handwriting is inherently "noisy" – it is not possible to say *a priori* that a given handwriting sample is just as searchable as its textual counterpart. That is the purpose of this study.

5 The ScriptSearch Algorithm

As we noted above, representations for ink exist at various different levels of abstraction. In this section we examine an algorithm for writer-dependent ink search at the stroke level. The algorithm applies dynamic programming with a recurrence similar to that used for string edit distance, but with a different set of operations and costs. The top-level organization of the ScriptSearch algorithm is shown in Figure 4.

As can be seen from the figure, there are four phases to the algorithm. First, the incoming pen points are broken into strokes. Next, the strokes are converted into vectors of descriptive features.

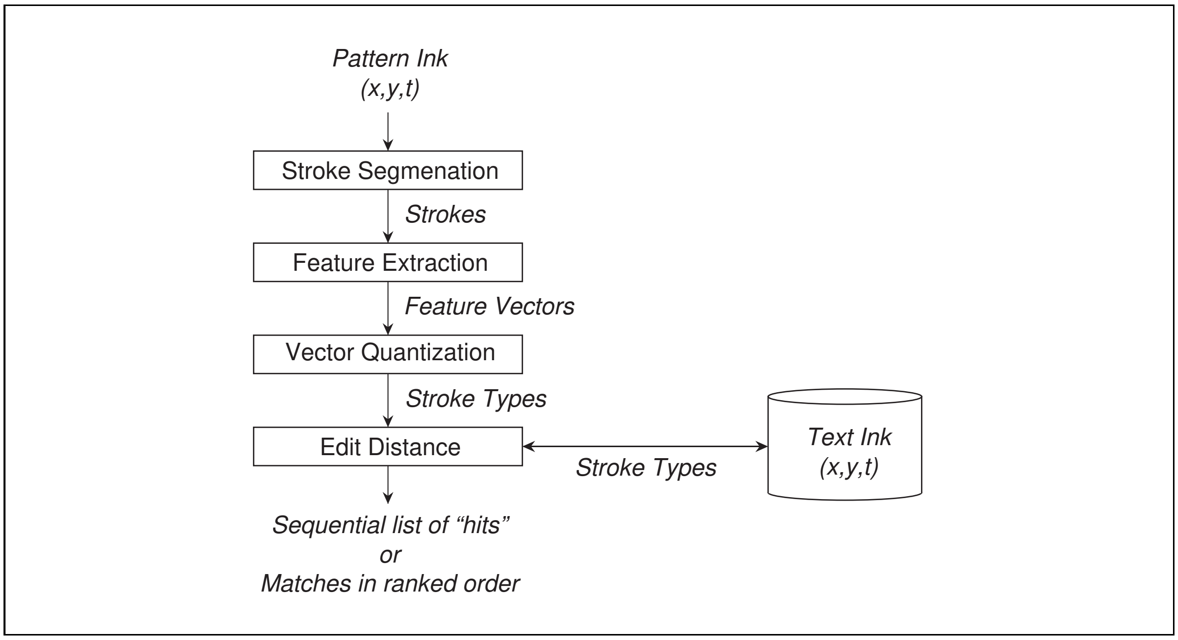

Figure 4: Overview of the ScriptSearch algorithm.

Third, the feature vectors are classified according to writer-specific information. Finally, the resulting sequence of classified strokes is matched against the text using approximate string matching over an alphabet of “stroke classes.” We now describe the four phases in more detail.

Stroke Segmentation. We have investigated several common stroke segmentation algorithms from handwriting recognition. Currently we break strokes at local minima of the y values. Figure 5 shows a sample line of stroke-segmented text. The bounding boxes of all strokes are shown.

Feature Extraction. Rather than propose another new feature set, we have taken a set created by Dean Rubine in the context of gesture recognition [6]. This particular feature set, which converts each stroke into a real-valued 13-dimensional vector, seems to do well at discriminating single strokes, and is efficient to update as new points arrive. For intuition, the feature set includes the length of the stroke, total angle traversed, angle and length of bounding box diagonal, and so on.

Vector Quantization. In the vector quantization stage the complex 13-dimensional feature space

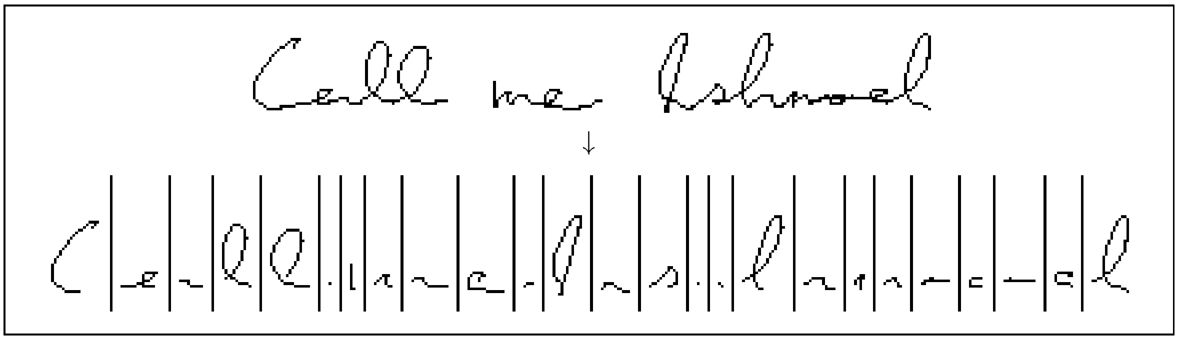

Figure 5: Example of pen-stroke segmentation.

is segmented or "quantized" into 64 clusters. From then on, instead of representing a feature vector by the 13 real values of the features, we represent it instead by the index of the cluster to which it belongs. Thus, instead of maintaining 13 real numbers, we maintain 6 bits. This technique is common in speech recognition and many other pattern recognition domains. ([1]). The quantization makes the remaining processing much more efficient, and seeks to choose clusters so that useful semantic information about the strokes is retained by the 6 bits of the index. We now describe how to build and use the clusters. First, we must describe how distances are calculated in feature space.

We collect a small sample of handwriting from each writer in advance. This is segmented into strokes and each strokes is converted into a feature vector $\vec{v} =< v_1, v_2, \ldots, v_{13} >^T$. We use the sample to calculate the average μ_i of the i^{th} feature, and we use these averages to compute the covariance matrix Σ defined by:

$$\Sigma_{ij} = E\left[(v_i - \mu_i)(v_j - \mu_j)\right] \tag{5}$$

Hence, for instance, the diagonal of Σ contains the variances of the features. Instead of using standard Euclidean distance we now define the *Mahalanobis distance* [8] that we will use upon the space of feature vectors as follows:

$$\begin{aligned} \|\vec{v}\|_M^2 &= \vec{v}^T \Sigma^{-1} \vec{v} && (6) \\ d(\vec{v}, \vec{w}) &= \|(\vec{v} - \vec{w})\|_M && (7) \end{aligned}$$

We now have a suitable distance measure for feature space, and can proceed to describe our vector quantization scheme. We cluster the feature vectors of the ink sample into 64 groups using a well-known clustering algorithm from the literature known as the k-means algorithm [5]. The feature vectors of the sample are processed sequentially. Each vector in turn is placed into a cluster, which is then updated to reflect the new member. Each cluster is represented by its centroid, the element-wise average of all vectors in the cluster.

The rule for classifying new feature vectors uses the centroids that define each cluster: a new vector belongs to the cluster with the nearest centroid, under Mahalanobis distance. The 64 final clusters can be thought of as "stroke types," and the feature extraction and VQ phases can be thought of as classifying strokes into stroke types.

After these phases of processing have been performed the text and pattern are represented as sequences of quantized stroke types:

$$< \textit{stroke type 7} >< \textit{stroke type 42} >< \textit{stroke type 20} > \ldots \tag{8}$$

Recall that $P = p_1 p_2 \ldots p_m$ and $T = t_1 t_2 \ldots t_n$. From now on, we shall assume that the p_i's and t_j's are vector-quantized stroke types.

The operations described above can be computed without significant overhead from the Mahalanobis distance metric. First, note that the inverse covariance matrix is positive definite (in fact, any matrix defining a valid distance must be positive definite). So we perform a Cholesky decomposition to write:

$$\Sigma^{-1} = A^T A \tag{9}$$

This being the case, we note that the new distance represents simply a coordinate transformation of the space:

$$\vec{v}^T \Sigma^{-1} \vec{v} = \vec{v}^T (A^T A) \vec{v} = (\vec{v}^T A^T) \cdot (A \vec{v}) = \vec{w}^T \vec{w} \tag{10}$$

where $\vec{w} = A\vec{v}$. Thus, once all points have been transformed, we can perform all future calculations in standard Euclidean space.

Text	Strokes	Characters	Words	Lines	Style
Writer A	34,560	23,262	4,045	625	Cursive
Writer B	19,324	12,269	2,194	363	Printed

Figure 6: The ink texts.

Edit Distance. Finally, we compute the similarity between the sequence of stroke types associated with the pattern ink, and the pre-computed sequence for the document ink. We use dynamic programming to determine the edit distance between the sequences. The cost of an insertion or deletion is a function of the "size" of the ink being inserted or deleted, where size is defined to be the length of the stroke type representing the ink, again using the Mahalanobis distance. The cost of a substitution is the distance between the stroke types. We also add two additional operations: two-to-one *merges* and one-to-two *splits*. This accounts for imperfections in the stroke segmentation algorithm. We build a merge/split table that contains information of the form "an average stroke of type 1 merged with an average stroke of type 4 results in a stroke of type 11." The cost of a merge or split from stroke a to strokes bc is a function of the distance from a to $merge(b, c)$. We compute dynamic programming edit distance using these costs and operations to find the best match in the document ink.

Again, recall that $d_{i,j}$ represents the cost of the best match of the first i symbols of P and a substring of T ending at symbol j. The recurrence, modified to account for our new types of substitutions (1:2 and 2:1), is as follows:

$$d_{i,j} = \min \left\{ \begin{array}{lcl} d_{i-1,j} & + & c_{del}(p_i) \\ d_{i,j-1} & + & c_{ins}(t_j) \\ d_{i-1,j-1} & + & c_{sub1:1}(p_i, t_j) \\ d_{i-1,j-2} & + & c_{sub1:2}(p_i, t_{j-1}t_j) \\ d_{i-2,j-1} & + & c_{sub2:1}(p_{i-1}p_i, t_j) \end{array} \right. \qquad 1 \le i \le m,\ 1 \le j \le n \qquad (11)$$

6 Experimental Data

In this section we describe our procedure for evaluating the algorithm. We asked two individuals to hand-write a reasonably large amount of data. Figure 6 gives some statistics about the sizes of the texts used. Both individuals wrote based on the beginning of the novel *Moby-Dick*. We refer to the two copies as Writer A and Writer B.

We then asked each writer to write a sequence of 30 short words and 30 phrases of two or three words, taken from the same passages of *Moby-Dick*. These are the "search strings," which we refer to as patterns, or queries. The short patterns ranged in length from 5 to 11 characters, with average length 8. The long patterns ranged from 12 to 24 characters, with average length 16. We are primarily interested in the results of searching the text drawn by a particular writer for the patterns drawn by the same writer since the ScriptSearch algorithm is meant to be writer-dependent.

The task of the algorithm is to find all lines of the text that contain the pattern. For each writer (A and B) we have an ASCII representation of the ink, augmented by hand with the locations of all line breaks, which we refer to as the ASCII text. We also have an ASCII version of the patterns.

Thus, the ink text corresponds line-to-line with the ASCII text. Using exact matching techniques, we find all occurrences of the ASCII patterns in the ASCII text, and note the lines on which all the matches occur. The ink patterns must occur on the same lines of the ink text.

We then segment both ink texts into lines using simple pattern recognition techniques, and associate each stroke of the ink text with a line number. Figure 7 shows an example of a page of

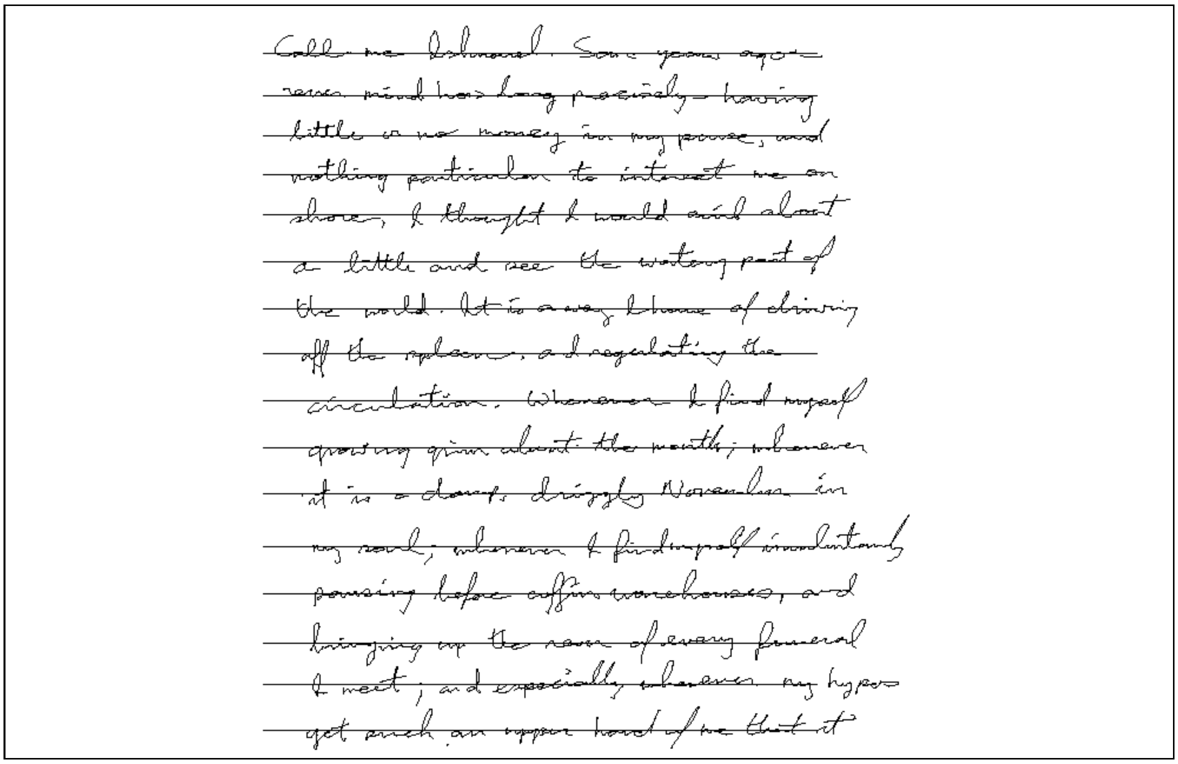

Figure 7: Estimation of line center-points.

ink with the center-points of the lines drawn in by the algorithm, and also gives an example of the handwriting quality of the text.

Using ink search we find all matches of the ink pattern within the ink text, and from the line segmentation information, we determine the lines of the ink text upon which matches have occurred. Since the ASCII text corresponds line-to-line with the ink text, we now check to see which matches are valid. From this information we compute the recall and precision of the ink search procedure.

7 Experimental Results and Discussion

As we mentioned previously, the system can be used in two different ways which yield two different sets of recall/precision values. First, the hits can be returned in ranked order. Recall and precision can be calculated by considering the number of spurious elements in the ranked list above a certain recall point. Second, all hits that exceed a fixed threshold can be returned. Recall and precision can then be calculated for a particular threshold by determining the total number of hits returned and the number of valid hits returned.

There is a connection between the two modes of use. If a perfect threshold can be chosen for each search then a system that returns all values above that threshold will have the same recall and precision as a ranked system. If the threshold cannot be chosen perfectly then the performance will decline. Thus, a ranked system represents an upper bound on the performance possible with a thresholded system. In contrast, a thresholded system has the advantage that the ink can be processed sequentially, returning hits as they become available, without waiting for the entire search to complete. Also, certain automatic operations require thresholding. Thus, we give results for both

	Writer A			Writer B		
	Short	Long	All	Short	Long	All
Recall	Patterns	Patterns	Patterns	Patterns	Patterns	Patterns
0.1	0.506	1.000	0.753	0.522	0.826	0.674
0.2	0.494	0.983	0.738	0.493	0.826	0.659
0.3	0.452	0.983	0.718	0.452	0.814	0.634
0.4	0.431	0.973	0.702	0.440	0.814	0.627
0.5	0.403	0.968	0.686	0.416	0.814	0.615
0.6	0.349	0.917	0.633	0.272	0.721	0.496
0.7	0.271	0.873	0.572	0.226	0.678	0.452
0.8	0.268	0.873	0.571	0.217	0.681	0.449
0.9	0.227	0.687	0.457	0.179	0.681	0.430
1.0	0.215	0.684	0.450	0.179	0.681	0.430

Figure 8: Ranked precision values for Writers A and B.

	Writer A					
Threshold	Short Patterns		Long Patterns		All Patterns	
	Rec	Prec	Rec	Prec	Rec	Prec
10	0.023	0.916	0.000	1.000	0.011	0.958
20	0.357	0.652	0.000	1.000	0.178	0.826
30	0.632	0.299	0.011	1.000	0.321	0.649
40	0.955	0.071	0.119	0.988	0.537	0.529
50	1.000	0.010	0.322	0.910	0.661	0.460
60	1.000	0.010	0.572	0.643	0.786	0.326
70	1.000	0.010	0.783	0.431	0.891	0.220
80	1.000	0.010	0.909	0.268	0.954	0.139
90	1.000	0.010	0.961	0.115	0.980	0.062
100	1.000	0.010	0.991	0.075	0.995	0.042
110	1.000	0.010	1.000	0.024	1.000	0.017
120	1.000	0.010	1.000	0.011	1.000	0.010

Figure 9: Recall and precision as a function of edit distance threshold for Writer A.

models.

Figure 8 shows the performance of the algorithm when returning ranked data. The figure shows that the difference in pattern length is extremely important to the performance. For example, at 100% recall there is a 47% difference in average precision values for long and short patterns in Writer A, and 50% difference in Writer B.

Figure 9 gives results for a system that does not give ranked output, for a wide variety of thresholds on the final edit distance between the pattern and a subsequence of the text. The figure shows recall and precision values for each threshold. Figure 10 shows the same information for Writer B. These figures suggest that it might be possible to choose thresholds dynamically based on properties of the pattern such as length.

In order to explore our intuition that stroke-based matching is not likely to be effective for multiple writers, we asked three additional writers (C, D, and E) to write the entire set of 60 patterns. We then matched these patterns against Writer A. The results are shown in Figure 11 for ranked recall and precision. As expected, the performance of the system is drastically worse than the results given above. This implies that performing ink search at the stroke level will probably

Threshold	Writer B					
	Short Patterns		Long Patterns		All Patterns	
	Rec	Prec	Rec	Prec	Rec	Prec
10	0.041	0.973	0.000	1.000	0.020	0.986
20	0.215	0.677	0.000	1.000	0.107	0.834
30	0.539	0.383	0.017	1.000	0.278	0.691
40	0.757	0.094	0.075	1.000	0.416	0.547
50	0.946	0.041	0.195	0.948	0.570	0.494
60	1.000	0.010	0.500	0.679	0.750	0.344
70	1.000	0.010	0.626	0.398	0.813	0.204
80	1.000	0.010	0.914	0.304	0.957	0.157
90	1.000	0.010	0.931	0.103	0.965	0.062
100	1.000	0.010	1.000	0.039	1.000	0.024
110	1.000	0.010	1.000	0.006	1.000	0.008
120	1.000	0.010	1.000	0.005	1.000	0.007

Figure 10: Recall and precision as a function of edit distance threshold for Writer B.

Recall	Writer C			Writer D			Writer E		
	Short	Long	All	Short	Long	All	Short	Long	All
0.1	0.024	0.027	0.025	0.033	0.070	0.052	0.048	0.099	0.073
0.2	0.022	0.014	0.018	0.032	0.041	0.037	0.032	0.028	0.030
0.3	0.013	0.014	0.013	0.031	0.042	0.036	0.032	0.024	0.028
0.4	0.013	0.015	0.014	0.029	0.023	0.026	0.033	0.021	0.027
0.5	0.013	0.015	0.014	0.030	0.022	0.026	0.034	0.021	0.028
0.6	0.010	0.013	0.011	0.018	0.016	0.017	0.018	0.018	0.018
0.7	0.010	0.013	0.011	0.017	0.015	0.016	0.018	0.018	0.018
0.8	0.010	0.013	0.011	0.017	0.014	0.016	0.016	0.017	0.017
0.9	0.010	0.012	0.011	0.017	0.013	0.015	0.015	0.017	0.016
1.0	0.010	0.012	0.011	0.017	0.013	0.015	0.015	0.016	0.016

Figure 11: Cross-writer query precision (text by Writer A).

restrict the pattern and text to be written by the same author, unless a more complex notion of distance between strokes can be developed.

8 Conclusions and Future Research

In this paper we have given some approaches to the problem of searching through ink in an attempt to determine the tractability of the problem. We have given some data to suggest that matching at the character level after performing HWX might be a tractable option. We have also presented a stroke-level matching algorithm that performs well for writer-dependent matching, in both ranked and thresholded systems. We also showed results suggesting that the unmodified algorithm does not perform well in writer-independent domains.

In the future it would be interesting to evaluate approaches to the problem that represent ink at different levels of abstraction, such as at the allograph level, perhaps performing dynamic programming on the adjacency graph to find the best match.

Further, since the amount of ink being searched is likely to become very large, especially for multi-author techniques, it would be very interesting to investigate sub-linear techniques that use more complex pre-processing of the ink text.

Also, we have not evaluated searches through parts of the ink domain other than English text. If the VQ classes were trained with a more general set of strokes then the ScriptSearch algorithm could run unchanged on drawings, figures, equations, other alphabets, and so on. It would be interesting to evaluate its effectiveness in these domains, especially since HWX-based methods do not apply.

Finally, it would be interesting to examine extensions to writer independence at the stroke level of matching. We sketch briefly an approach that we believe has potential for this problem:

Recall that, since the VQ codebooks of two authors may be different, this is no natural stroke-to-stroke correspondence. Let us assume that by some approximate means it is possible put text from two authors A and B into a rough correspondence, and then to determine for each of A's strokes a distribution of similarity with B's strokes. We can describe these distributions for each of A's strokes in a *Stroke Similarity Matrix* S. The i^{th} row of such a matrix describes how A's i^{th} stroke corresponds to all of B's strokes. Assume that the $(i,j)^{th}$ entry of matrix $D_{B\rightarrow B}$ gives the Mahalanobis distance from writer B's stroke i to stroke j. We wish to compute $D_{A\rightarrow B}$, the matrix giving distances from each of writer A's strokes to each of writer B's strokes. We do so as follows:

$$D_{A\rightarrow B} = S \cdot D_{B\rightarrow B} \tag{12}$$

That is, to compute the distance between the i^{th} stroke of A and the j^{th} stroke of B we do the following. Think of the i^{th} stroke of A as corresponding to various strokes of B with the weights given in the i^{th} row of S. Now extract the distance from each of these strokes to B's j^{th} stroke, and take the weighted sum of these distances. This is the inner product of the i^{th} row of S with the j^{th} column of $D_{B\rightarrow B}$, as shown in Equation 12.

This approach gives a reasonable "cross-author" distance measure that we can substitute for the Mahalanobis distance used above. The ScriptSearch algorithm can then be used without further changes.

References

[1] Yoseph Linde, Andres Buzo, and Robert M. Gray. An algorithm for vector quantizer design. *IEEE Transactions on Communications*, COM-28, No 1:84–95, 1980.

[2] Daniel Lopresti and Andrew Tomkins. Approximate matching of hand-drawn pictograms. In *Proceedings of the Third International Workshop on Frontiers in Handwriting Recognition*, pages 102–111, May 1993.

[3] Daniel Lopresti and Andrew Tomkins. Pictographic naming. In *Adjunct Proceedings of the 1993 Conference on Human Factors in Computing Systems (INTERCHI'93)*, pages 77–78, April 1993.

[4] Daniel Lopresti and Andrew Tomkins. Computing in the ink domain. Submitted for publication, March 1994.

[5] J. MacQueen. Some methods for classification and analysis of multivariate observations. *Proceedings of the Fifth Berkeley Symposium on Mathematics, Statistics and Probability*, 1:281–296, 1967.

[6] Dean Rubine. *The Automatic Recognition of Gestures.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1991.

[7] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, Inc., 1983.

[8] Robert Schalkoff. *Pattern Recognition. Statistical, Structural and Neural Approaches.* John Wiley & Sons, Inc, 1992.

[9] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the Association for Computing Machinery*, 21(1):168–173, 1974.