# ON THE SEARCHABILITY OF ELECTRONIC INK[1]

Daniel Lopresti Andrew Tomkins

Matsushita Information Technology Laboratory
Panasonic Technologies, Inc.
Two Research Way
Princeton, NJ 08540-6628 USA
`[dpl,andrewt]@mitl.research.panasonic.com`

**Abstract**

Pen-based computers and personal digital assistant's (PDA's) are growing in popularity. An important issue that arises when ink is treated as a first-class datatype concerns the searching of large quantities of pen-stroke data. In this paper, we examine the problem of finding all occurrences of an ink pattern in a longer ink text. We present an algorithm based on a dynamic programming technique, and evaluate its performance under a variety of circumstances. Our results show that ink can be searched effectively. They also raise new questions in pen computing and the manipulation of ink data.

## 1 Introduction

Pen-based computers and personal digital assistants (PDA's) are a synthesis of new hardware and software technologies. As such, they raise many systems-level issues, including the possibility of new paradigms for human-computer interaction. In two earlier papers, we presented a philosophy we have come to call "Computing in the Ink Domain" [4, 5]. Here, the need for traditional handwriting recognition (HWX) is often deferred and sometimes even eliminated. Instead, much of the functionality that users require is realized by treating ink as a first-class datatype.

One of the most important issues that arises under this model concerns the searching of large quantities of pen-stroke data. While efficient, well-known techniques exist for matching text strings exactly (*e.g.*, Knuth-Morris-Pratt, Boyer-Moore), approximately (*e.g.*, Levenshtein or edit distance), and with wild-cards (*e.g.*, Unix `grep`), there is no comparable body of work for the problem of matching ink "strings." In this paper, we examine a particular instance of this problem which can be stated as follows: given an ink pattern and a longer ink text, search through the text and find as many occurrences of the pattern as possible without overwhelming the user with false hits.

[1]Presented at the *Fourth International Workshop on Frontiers of Handwriting Recognition*, Taipei, Taiwan, December 1994.

## 2 Definitions

We view *ink* as a sequence of time-stamped points in the plane:

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

For two ink sequences $P$ (the *pattern*) and $T$ (the *text*), 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 $T$. 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 [11], the following three operations are permitted: (1) delete a symbol, (2) insert a symbol, and (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

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

and the main dynamic programming recurrence is

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

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

As it stands, this formulation aligns the two strings in their entirety. The variation we use for ink search is modified so that a short pattern can be matched against a longer text. The initial edit distance is made 0 along the entire length of the text (allowing a match to start anywhere), and the final row of the edit distance table is searched for its smallest value (allowing a match to end anywhere). The initial conditions become

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

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

Lastly, we must specify evaluation criteria. It seems inevitable that an ink search algorithm will miss some true occurrences of $P$ in $T$, and report false "hits" at locations where $P$ does not really occur. How can we quantify the success of an algorithm under such conditions? The field of Information Retrieval concerns itself with a similar problem in a different domain, and has settled on the following two measures [9]:

| | |
|---|---|
| **Recall** | The percentage of true matches that are reported. |
| **Precision** | The percentage of reported matches that are in fact true. |

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, it is text in a predetermined character set (*e.g.*, ASCII). Proceeding upwards in the hierarchy, some information is lost, while a new representation is created that (hopefully) captures all that is relevant from the lower level in a more concise form. So, for instance, it may be impossible to deduce from the final word which allographs were used, or from the feature vectors exactly what the ink looked like.

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). Each level has its attendant advantages and disadvantages.

The next section presents the primary contribution of this paper: an algorithm we call *ScriptSearch* that performs matching at the level of pen-strokes. This approach has the advantage of allowing us to do well against a broad range of handwriting, including some so poor that it is effectively illegible. ScriptSearch also makes possible the matching of strings with no obvious ASCII representation, including equations, drawings, doodles, etc.

## 4 The ScriptSearch Algorithm

Figure 2 shows an overview of ScriptSearch, which consists of four phases. First, the incoming points are grouped into strokes. Next, the strokes are converted into vectors

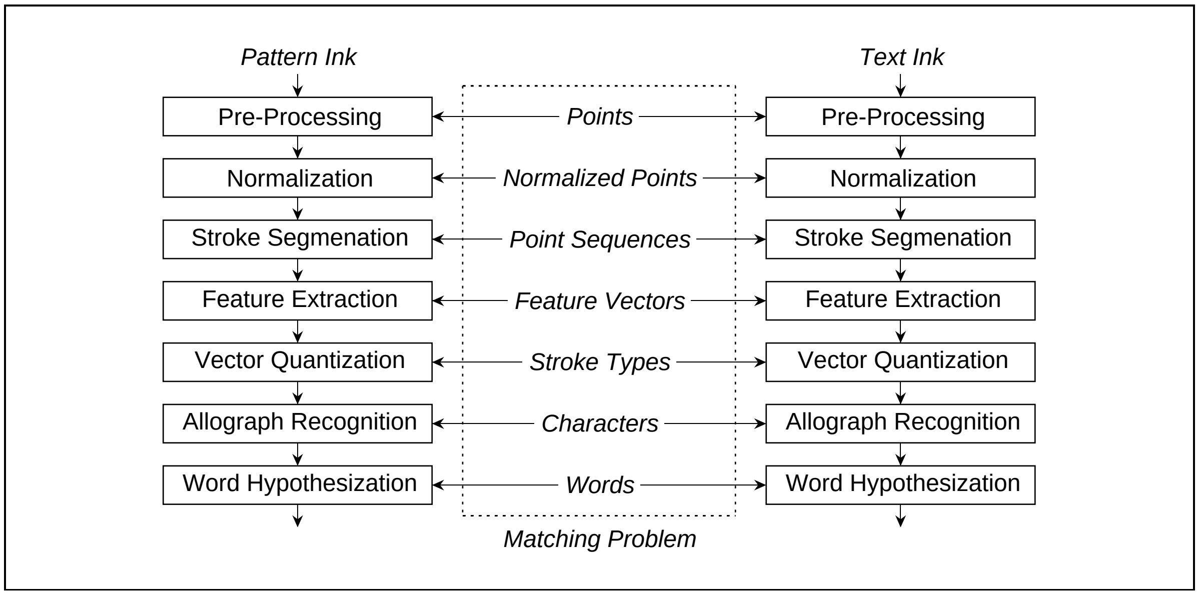

Figure 1: Handwriting recognition stages and potential matching problems.

of descriptive features. Third, the feature vectors are classified according to writer-specific information. Finally, the resulting sequence is matched against the text using approximate string matching over an alphabet of "stroke types." We now describe each of the phases in more detail.

**Stroke Segmentation.** We have investigated several common stroke segmentation algorithms used in handwriting recognition. Currently we break strokes at local $y$-minima. Figure 3 shows a handwritten text sample, with the stroke segmentation depicted in the lower half of the figure.

**Feature Extraction.** Rather than propose another new feature set, we have adopted the one used by Rubine in the context of gesture recognition [8]. This particular feature set, which converts each stroke into a real-valued 13-dimensional vector, seems to do well at discriminating single strokes and can be updated efficiently as new points arrive. Some of the features it incorporates include the length of the stroke, the total angle traversed, and the angle and length of the bounding box diagonal.

**Vector Quantization.** In the VQ stage, the complex 13-dimensional feature space is "quantized" into 64 clusters. From then on, we represent a feature by the index of the cluster to which it belongs. Thus, instead of having to maintain 13 real numbers, we need only keep six bits. This technique is common in speech recognition and other related domains [2]. 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 index.

We begin by collecting a small sample of handwriting from the user in advance. This is segmented into strokes, each of which is converted into a feature vector $\vec{v} = < v_1, v_2, \ldots, v_{13} >^T$. We use this sample to calculate the average value, $\mu_i$, of the $i^{th}$

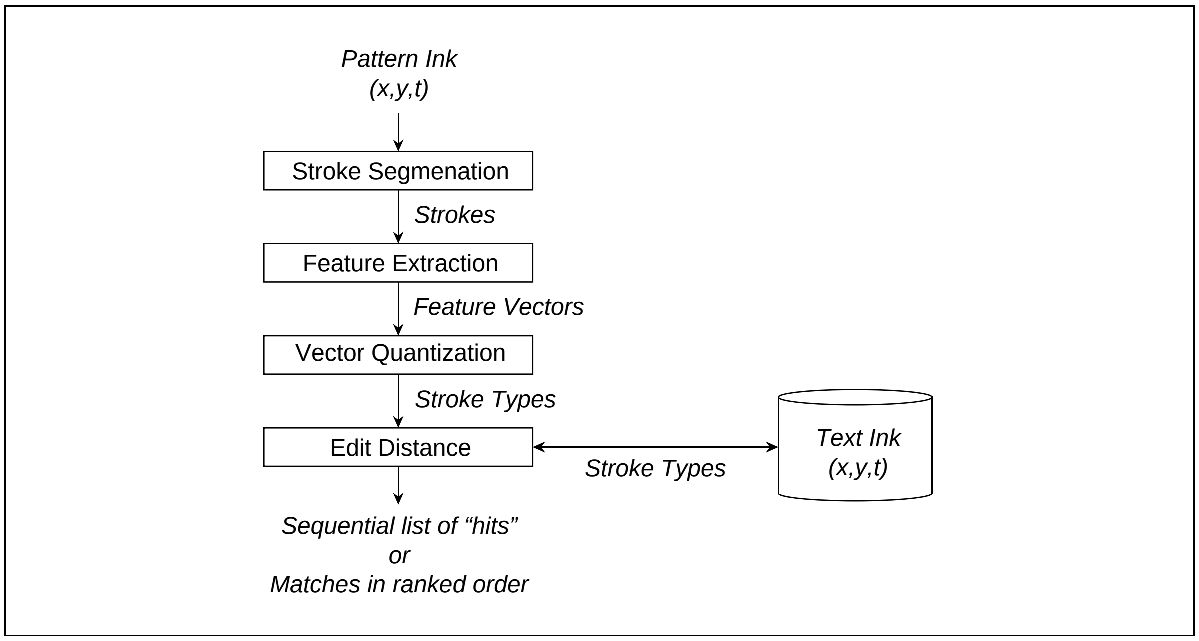

Figure 2: Overview of the ScriptSearch algorithm.

feature, and use these averages to compute the covariance matrix $\Sigma$ as defined by

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

The main diagonal of $\Sigma$, for instance, contains the variances of the features. We employ *Mahalanobis distance* [10] defined on the space of feature vectors as follows:

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

With a suitable distance measure for the feature space, we can now proceed to describe our vector quantization scheme. We cluster the feature vectors of the ink

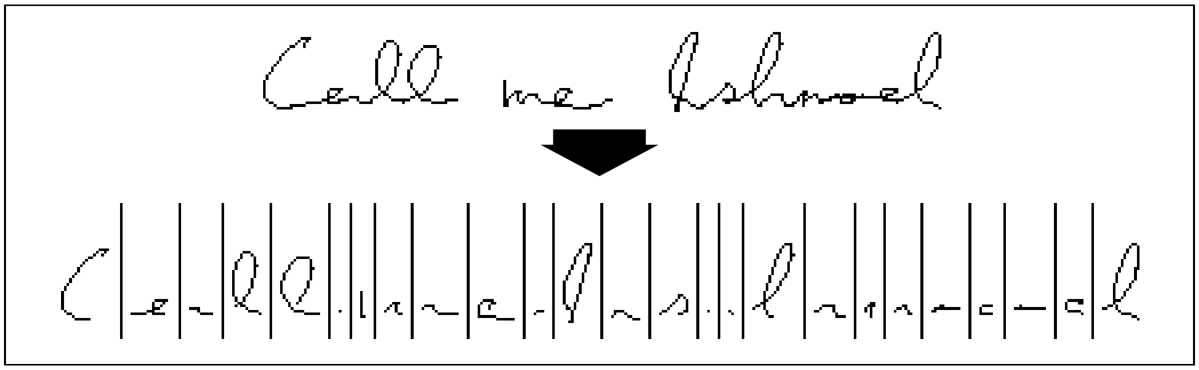

Figure 3: Stroke segmentation example.

sample into 64 groups using a technique from the literature known as the $k$-means algorithm [7]. The feature vectors of the sample are processed sequentially. Each vector in turn is placed in the appropriate cluster, which is represented by its centroid, the element-wise average of all the vectors it contains. A new vector is placed in the cluster with the nearest centroid, as determined using Mahalanobis distance. The resulting 64 clusters can be thought of as an alphabet of stroke types, and the feature extraction and VQ phases can be viewed as a stroke classification process.

After these steps are completed, the text and pattern can be represented as sequences of quantized stroke types:

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

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 just described can be computed without incurring significant overhead from the Mahalanobis distance metric. First, note that the inverse covariance matrix is positive definite. We perform a Cholesky decomposition to write:

$$\Sigma^{-1} = A^T A \tag{8}$$

The new distance simply represents a coordinate transformation of the space:

$$\vec{v}^T \Sigma^{-1} \vec{v} = \vec{v}^T (A^T A) \vec{v} = (\vec{v}^T A^T) \cdot (A\vec{v}) = \vec{w}^T \vec{w} \tag{9}$$

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

**Edit Distance.** In the last phase, we compute the similarity between the stroke sequence associated with the pattern ink and the pre-computed sequence for the text ink. As noted previously, we use dynamic programming to determine the edit distance between the sequences. The cost of a deletion or insertion is a function of the "size" of the ink involved; this is defined as the length of the stroke type representing the ink, again using Mahalanobis distance. The cost of a substitution is the distance between the two stroke types in question.

We also add two operations to the three listed earlier: (4) split one symbol into two, and (5) merge two symbols into one. These account for imperfections in the stroke segmentation. We build a split/merge table that contains information of the form "an average stroke of type $\alpha$ merged with an average stroke of type $\beta$ results in a stroke of type $\gamma$." The cost of splitting stroke $\delta$ into a pair of strokes $\alpha\beta$ is a function of the distance between $\delta$ and $\mathit{merge}(\alpha, \beta) = \gamma$. We compute edit distance using these costs and operations to find matches for the pattern in the text ink.

Again, recall that $d_{i,j}$ represents the cost of the best match between the first $i$ symbols of $P$ and a substring of $T$ ending at symbol $t_j$. The recurrence, modified to account for these new operations, is

$$d_{i,j} = \min \begin{cases} d_{i-1,j} & + \quad c_{del}(p_i) \\ d_{i,j-1} & + \quad c_{ins}(t_j) \\ d_{i-1,j-1} & + \quad c_{sub}(p_i, t_j) \\ d_{i-1,j-2} & + \quad c_{split}(p_i, t_{j-1}t_j) \\ d_{i-2,j-1} & + \quad c_{merge}(p_{i-1}p_i, t_j) \end{cases} \qquad 1 \le i \le m,\ 1 \le j \le n \qquad (10)$$

As before, the computation takes time $O(mn)$. If the pattern and text are long, this can be appreciable. There exist, however, asymptotically faster algorithms for computing edit distance, as well as parallel versions that are orders of magnitude faster than the obvious sequential implementation [3].

## 5 Experimental Procedure

In this section, we describe our procedure for evaluating the ScriptSearch algorithm. The two authors each wrote a reasonably large amount of English text drawn from Herman Melville's famous novel *Moby-Dick.* We refer to these two datasets as "Writer A" and "Writer B." Table 1 presents some basic statistics.

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

Table 1: Summary of the ink texts used to evaluate ScriptSearch.

The authors then each wrote 30 short words and 30 longer phrases (2-3 words), taken from the same passages of *Moby-Dick.* These served as our search strings, which we also refer to as "patterns" or "queries." Table 2 provides details concerning this data. Since ScriptSearch is writer-dependent (*i.e.,* no attempt is made to account for the natural variation in stroke-order from person-to-person), we were primarily interested in matching patterns and text from the same writer.

| *Patterns* | | *Characters* | | | *Pen Strokes* | | | *Style* |
|---|---|---|---|---|---|---|---|---|
| | | *Min.* | *Max.* | *Ave.* | *Min.* | *Max.* | *Ave.* | |
| Writer A | Short | 5 | 11 | 8 | 8 | 23 | 13 | Cursive |
| | Long | 12 | 24 | 16 | 23 | 45 | 30 | |
| Writer B | Short | 5 | 11 | 8 | 9 | 26 | 15 | Printed |
| | Long | 12 | 24 | 16 | 23 | 41 | 29 | |

Table 2: Summary of the ink patterns used to evaluate ScriptSearch.

The goal of the algorithm is to identify all the lines of the text that contain the pattern. To establish our "ground-truth," we started with an ASCII representation of the text. We then annotated this with the locations of the line breaks that arose when the text was handwritten. Using exact string matching, we found all occurrences of the patterns in the ASCII text, and recorded which lines contained matches. We then segmented our ink texts into lines using a straightforward approach, and associated each stroke with a line number.

Using the ScriptSearch algorithm, we matched the ink patterns to the ink text and, from the line segmentation information, determined the lines on which matches occurred. Since the ground-truth ASCII text corresponded line-for-line to the ink text, we could check to see which matches were missed, and which of the reported matches were true. We used this data to compute the recall and precision of our ink search procedure.

## 6 Results and Discussion

The results of an ink search can be formulated in two different ways. First, all hits exceeding a fixed threshold can be returned. Recall and precision are then calculated by determining the number of true matches that are found, and the number of reported matches that are valid. Second, hits can be returned in ranked order. In this case, precision is calculated as a function of recall by proceeding down the ranking and determining the number of false matches above a certain recall point.

There is a relationship between these two formulations. If a "perfect" threshold could be chosen in advance for each search, then a system that returns all matches above the threshold would have the same precision as a ranked system. Thus, in some sense, a ranked system represents an upper bound on performance. In contrast, a thresholded system has the advantage that hits are returned as soon as they are found, in sequential order, without having to wait for the entire search to complete. If ink search is the first step in a multi-stage process, then thresholding may be unavoidable. Hence, our analysis is in terms of both formulations.

Table 3 gives results for Writer A's searches for a range of edit distance thresholds. The table shows recall and precision values for each threshold, further broken down by the length of the pattern. Table 4 presents the same information for Writer B. Note that the choice of threshold is critical and, not surprisingly, a function of the pattern length. This data suggests that the threshold should be determined dynamically based on the pattern, and that length is a key criterion. If less than perfect recall is acceptable, then ink appears to be quite searchable using a technique such as this. If it is not, the user can expect to see up to 50 false hits for every true match, a scenario that would undoubtedly lead to great frustration.

The performance of ScriptSearch when returning ranked data is shown in Table 5. Note that ink appears to be far more searchable under this formulation. This is because the process of generating a ranking automatically determines an optimal threshold for each pattern (as opposed to using the same, predetermined threshold across multiple patterns). This presentation also makes clear the fundamental difference between short and long patterns; the latter can be searched with approximately two to four times

| *Edit Distance Threshold* | *Writer A* | | | | | |
|---|---|---|---|---|---|---|
| | *Short Patterns* | | *Long Patterns* | | *All Patterns* | |
| | *Recall* | *Precision* | *Recall* | *Precision* | *Recall* | *Precision* |
| 10 | 0.023 | 0.916 | 0.000 | n/a | 0.011 | 0.958 |
| 30 | 0.632 | 0.299 | 0.011 | 1.000 | 0.321 | 0.649 |
| 50 | 1.000 | 0.010 | 0.322 | 0.910 | 0.661 | 0.460 |
| 70 | 1.000 | 0.010 | 0.783 | 0.431 | 0.891 | 0.220 |
| 90 | 1.000 | 0.010 | 0.961 | 0.115 | 0.980 | 0.062 |
| 110 | 1.000 | 0.010 | 1.000 | 0.024 | 1.000 | 0.017 |

Table 3: Average recall and precision as a function of threshold for Writer A.

| *Edit Distance Threshold* | *Writer B* | | | | | |
|---|---|---|---|---|---|---|
| | *Short Patterns* | | *Long Patterns* | | *All Patterns* | |
| | *Recall* | *Precision* | *Recall* | *Precision* | *Recall* | *Precision* |
| 10 | 0.041 | 0.973 | 0.000 | n/a | 0.020 | 0.986 |
| 30 | 0.539 | 0.383 | 0.017 | 1.000 | 0.278 | 0.691 |
| 50 | 0.946 | 0.041 | 0.195 | 0.948 | 0.570 | 0.494 |
| 70 | 1.000 | 0.010 | 0.626 | 0.398 | 0.813 | 0.204 |
| 90 | 1.000 | 0.010 | 0.931 | 0.103 | 0.965 | 0.062 |
| 110 | 1.000 | 0.010 | 1.000 | 0.006 | 1.000 | 0.008 |

Table 4: Average recall and precision as a function of threshold for Writer B.

better precision at a given recall rate. For example, at 100% recall there is a 318% difference between long and short patterns for Writer A, and a 380% difference for Writer B. This confirms our intuition that more context (data) should result in a better (tighter) search.

| | *Writer A Patterns* | | | *Writer B Patterns* | | |
|---|---|---|---|---|---|---|
| *Recall* | *Short* | *Long* | *All* | *Short* | *Long* | *All* |
| 0.2 | 0.494 | 0.983 | 0.738 | 0.493 | 0.826 | 0.659 |
| 0.4 | 0.431 | 0.973 | 0.702 | 0.440 | 0.814 | 0.627 |
| 0.6 | 0.349 | 0.917 | 0.633 | 0.272 | 0.721 | 0.496 |
| 0.8 | 0.268 | 0.873 | 0.571 | 0.217 | 0.681 | 0.449 |
| 1.0 | 0.215 | 0.684 | 0.450 | 0.179 | 0.681 | 0.430 |

Table 5: Average precision as a function of recall for Writers A and B.

We have also investigated the degree to which ScriptSearch is writer-dependent by performing cross-writer searches. As expected, its performance is markedly worse in this case; see [6] for further details.

## 7 Conclusions and Future Research

In this paper, we have presented a technique for searching through an ink text for all occurrences of an ink pattern. Using recall and precision as figures of merit, our pen-stroke matching algorithm performs quite well, especially when hits can be output in ranked order. In the future, it would be interesting to investigate approaches that treat ink at other levels of abstraction (recall Figure 1). Moreover, since the amount of ink to be searched will undoubtedly grow as pen computers proliferate, it may be important to consider sub-linear techniques that employ more complex pre-processing of the ink text. Some of these are described in [1].

An intriguing extension of this work concerns searching languages other than English as well as non-textual ink. If the VQ classes are trained on a more general set of strokes, it may be possible to run ScriptSearch on different alphabets, drawings, figures, equations, etc. Finally, the issue of writer-independent ink matching remains an important open problem.

## References

[1] W. G. Aref and D. Barbará. The Hidden Markov Model tree index: A practical approach to fast recognition of pictures and handwritten documents in large databases. Technical Report 84-93, MITL, January 1994.

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

[3] R. J. Lipton and D. P. Lopresti. A systolic array for rapid string comparison. In H. Fuchs, editor, *Proceedings of the 1985 Chapel Hill Conference on Very Large Scale Integration*, pages 363–376. Computer Science Press, 1985.

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

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

[6] D. Lopresti and A. Tomkins. On the searchability of electronic ink. Technical Report 114-94, MITL, June 1994.

[7] 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.

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

[9] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, Inc., 1983.

[10] R. Schalkoff. *Pattern Recognition: Statistical, Structural and Neural Approaches.* John Wiley & Sons, Inc, 1992.

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