Temporal-Domain Matching of Hand-Drawn Pictorial Queries

Daniel Lopresti   *Andrew Tomkins*
dpl@research.panasonic.com   andrewt@cs.cmu.edu

Matsushita Information Technology Laboratory
Panasonic Technologies, Inc.
Two Research Way
Princeton, NJ 08540

October 27, 1995

Abstract

In this chapter we discuss the problem of searching a database of hand-drawn documents for a particular hand-drawn query. We show how to break the problem into two parts. First, a matching algorithm must extract pieces of the query and pieces of the database that might be placed in correspondence. Second, the algorithm must examine the pieces in detail to determine how similar they are. We present a general framework that generates potential matches to be examined, and we give two candidate algorithms for the underlying comparison. The first underlying match algorithm is result of prior work in script matching; the second is a standard dynamic programming-based elastic match.

We give experimental data based on 125 database pictures and 125 query pictures to show that our algorithms can effectively search handwritten data for pictures, even though stroke orders and overall page structure of queries and their intended hits in the database can differ substantially.

Keywords: *pen computing*, *electronic ink*, *handwriting recognition*, *picture matching.*

1 Introduction

In this chapter, we discuss a new paradigm for pen computing based on the notion of deferring or even eliminating handwriting recognition (HWX). In its place, key functionality is brought closer to the user by implementing it directly in the ink domain. The primary advantage of this approach is increased expressive power, but it also results in a different class of pattern matching problems, some of which may be more tractable and less intrusive than traditional HWX.

For input and interaction, pens have many advantages: they are expressive, lightweight, and familiar. It has been shown, for example, that a pen is better than a mouse or trackball for pointing tasks [11]. But while pen-based computers have met with success in vertical markets, attempts to win mass-market acceptance (*e.g.*, GO's PenPoint, the Apple Newton) have not lived up to early expectations. Indeed, the most recent entry in pen operating systems, General Magic's MagicCap, de-emphasizes HWX and exploits the pen primarily for its navigating capabilities.

There are many possible explanations for this. A lack of "killer" applications, small hard-to-read screens, excessive size and weight (in comparison to paper notepads), and short battery life are undoubtedly contributing factors. Still, the most obvious failing voiced by potential users is the poor quality of handwriting recognition software. To be fair, HWX is still a hard research problem. Some work has focused on techniques to make it easier for the user to correct the errors that inevitably arise during text entry [2]. Another recent approach is to make the HWX problem simpler for the computer by changing the input alphabet [3]. Forcing users to learn a new way of writing, however, is a fairly drastic solution that seems likely to meet with some resistance.

For the most part, today's pen computers operate in a mode which might be described as "eager recognition." Pen-strokes are translated as soon as they are entered, the user corrects the output of the recognizer, and then processing proceeds as if the characters had been typed on a keyboard.

Instead of taking a very expressive medium, ink, and immediately mapping it into a small, pre-defined set of alphanumeric symbols, we suggest that pen computers should support *first-class* computing in the ink domain [8, 9]. While traditional HWX is important for some applications, there are strong arguments for deferring or even eliminating HWX in many cases:

1. Many of a user's day-to-day tasks can be handled entirely in the ink domain using techniques more accurate and less intrusive than HWX.

2. No existing character set captures the full range of graphical representations a human can create using a pen (*e.g.*, pictures, maps, diagrams, equations, doodles). By not constraining pen-strokes to represent "valid" symbols, a much richer input language is made available to the user.

3. If recognition should become necessary at a later time, additional context for performing the translation may be available to improve the speed and accuracy of HWX.

This philosophy of *recognition-on-demand* is more distinctly "human-centric" than HWX, which reflects a "computer-centric" orientation.[1] Figure 1 depicts this state of affairs.

One fundamental problem involves searching handwritten notes (both text and drawings) for close matches to a pictorial "query." This problem appears quite difficult, as there is a great deal of variation in the way people draw the same picture on different occasions. Our approach for dealing with this complexity is to exploit the inherently sequential nature of ink creation in the temporal domain.

The remainder of this chapter is organized as follows. In the next section, we describe a basic algorithm for searching handwritten text. Section 3 considers the more difficult

[1]The concept of "lazy" recognition [13] – delaying HWX so as not to interfere with the creative flow of ideas – is quite similar. However, our proposal is for new functionality at the level of the "raw" ink, making it directly manipulable.

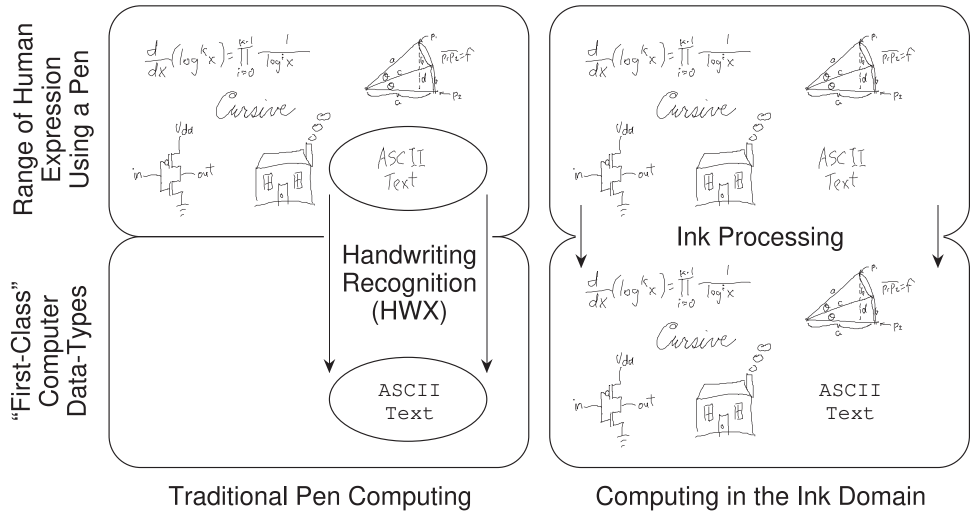

Figure 1: Traditional pen computing versus ink as first-class data.

problem of matching hand-drawn pictures. Here we present a methodology based on a model we have developed for string block editing. In Section 4, we give the results of a simple experiment we performed to test the new approach. Finally, Section 5 offers our conclusions.

2 Searching Handwritten Text

In an earlier paper [9], we describe an approach for searching handwriting text in which pen input from a digitizing tablet is segmented into strokes, a standard set of features is extracted (*e.g.*, stroke length, total angle traversed), and the resulting vectors are clustered into a small number of basic stroke types. Comparisons are then performed between strings over this "ink" alphabet using known approximate string matching techniques.

Ink can be represented at a number of levels of abstraction. 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.

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 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.

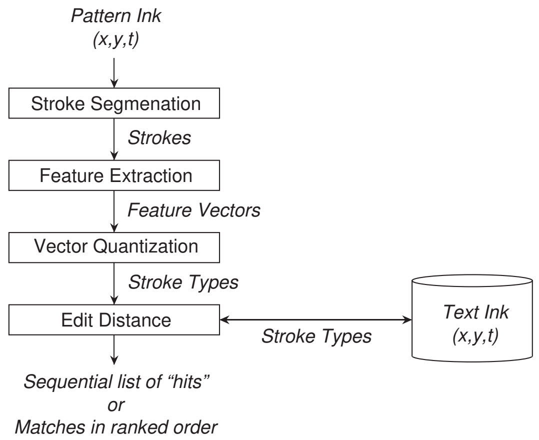

Figure 2: Overview of the ScriptSearch algorithm.

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.

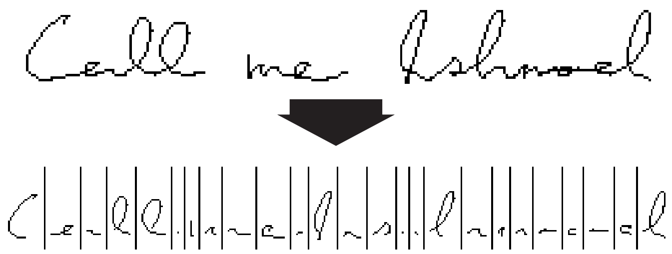

Figure 3: Stroke segmentation example.

Feature Extraction. Rather than propose another new feature set, we have adopted the one used by Rubine in the context of gesture recognition [16]. 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 [6]. 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, μ_i, of the i^{th} feature, and use these averages to compute the covariance matrix Σ as defined by

$$\Sigma_{ij} = E\left[(v_i - \mu_i)(v_j - \mu_j)\right] \tag{1}$$

The main diagonal of Σ, for instance, contains the variances of the features. We employ *Mahalanobis distance* [17] defined on the space of feature vectors as follows:

$$\begin{aligned} \|\vec{v}\|_M^2 &= \vec{v}^T \Sigma^{-1} \vec{v} && (2) \\ d(\vec{v}, \vec{w}) &= \|(\vec{v} - \vec{w})\|_M && (3) \end{aligned}$$

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 sample into 64 groups using a technique from the literature known as the k-means algorithm [12]. 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:

<stroke type 7> <stroke type 42> <stroke type 20> ...

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:

$$\Sigma^{-1} = A^T A \tag{4}$$

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

$$\vec{v}^T \Sigma^{-1} \vec{v} = \vec{v}^T (A^T A) \vec{v} = (\vec{v}^T A^T) \cdot (A\vec{v}) = \vec{w}^T \vec{w} \tag{5}$$

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. We use a well-known dynamic programming algorithm to determine the edit distance between the sequences [18]. 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 new editing operations to the three standard ones: split one stroke into two, and merge two strokes 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 α merged with an average stroke of type β results in a stroke of type γ." The cost of splitting stroke δ into a pair of strokes $\alpha\beta$ is a function of the distance between δ and $merge(\alpha, \beta) = \gamma$. We compute edit distance using these costs and operations to find matches for the pattern in the text ink.

Let $d_{i,j}$ represent the cost of the best match between the first i strokes of P and a substring of T ending at stroke t_j. The recurrence, modified to account for these new operations, is

$$d_{i,j} = \min \begin{cases} d_{i-1,j} & + \quad c_{del}(p_i) \\ d_{i,j-1} & + \quad c_{ins}(t_j) \\ d_{i-1,j-1} & + \quad c_{sub}(p_i, t_j) \\ d_{i-1,j-2} & + \quad c_{split}(p_i, t_{j-1}t_j) \\ d_{i-2,j-1} & + \quad c_{merge}(p_{i-1}p_i, t_j) \end{cases} \qquad 1 \le i \le m,\ 1 \le j \le n \tag{6}$$

This 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 [7].

For handwritten text (English and Japanese, cursive and printed), empirical studies using this approach, as well as other, similar ones, have demonstrated good performance [8, 9, 1, 5, 14]. In one experiment we conducted, two subjects each wrote a reasonably large amount of English text drawn from Herman Melville's famous novel *Moby-Dick*. They then wrote 30 short words and 30 longer phrases (2-3 words), taken from the same passages. These served as our search strings, which we also refer to as "patterns" or "queries."

We used the following two standard criteria for judging the success of ScriptSearch:

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.

Table 1 presents the results of this experiment. Note that ScriptSearch returns mostly the desired "hits," with relatively little superfluous "noise."

	Subject A Patterns			*Subject 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 1: Average precision as a function of recall for Subjects A and B.

3 Matching Pictorial Queries

While the approach just described works well for text, when processing more complicated pictorial data, certain sub-structures within a larger image will correspond stroke-for-stroke, but these basic "blocks" may be drawn by the user in an otherwise arbitrary order. Figure 4 demonstrates this; the two trees in Picture A are drawn last, while the very similar tree in Picture B is drawn first. Moreover, if the goal is to search a database, the best match may be "partial" in the sense that certain elements are omitted or repeated. This phenomenon is illustrated in Figure 4 as well. Existing string matching algorithms are not flexible enough to capture these kinds of *block motion*.

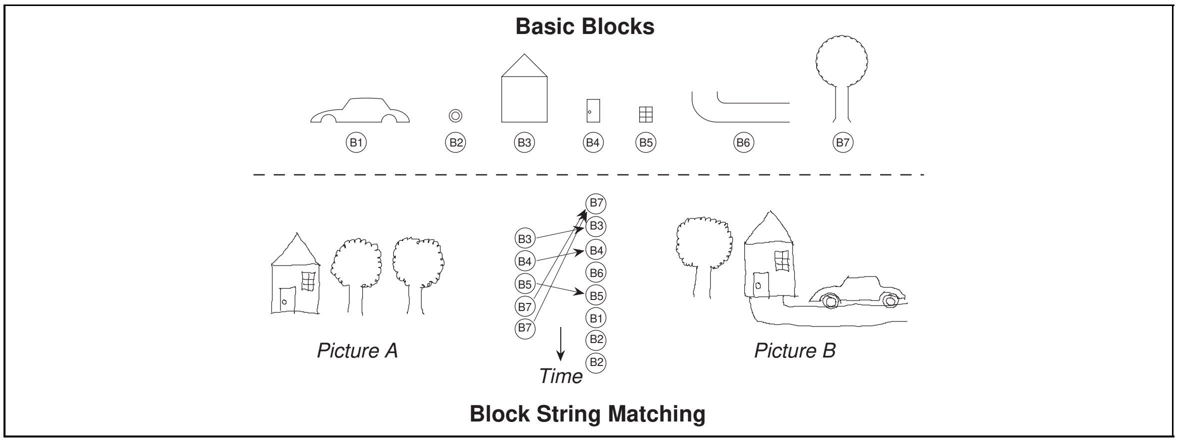

Figure 4: Approximate string matching applied to hand-drawn pictorial queries.

For this problem, we have developed new block edit algorithms for string matching [10]. Briefly, our approach takes a database $d_1 \ldots d_n$ and a query string $q_1 \ldots q_m$ as input, partitions the query string into blocks optimally, and matches each block to a database block such that the sum of the distances between corresponding blocks is minimized. Thus, the procedure allows high-level block motion by placing arbitrary blocks in correspondence, and also incorporates low-level stroke edits (deletion, insertion, substitution, merge, split) in an underlying distance function *dist*.

Let $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$ be strings over the alphabet, $a_i, b_j \in \Sigma$. We say that a *t-block substring family of A*, $A|_t$, is a multiset containing t substrings of A, some of which may be identical. In the following, we will write $A|_t = \{A^{(1)}, \ldots, A^{(t)}\}$, with the understanding that the $A^{(i)}$'s need not be distinct. A corresponding t-block substring family of B, $B|_t$, is a multiset of t substrings of B.

If the substrings in $A|_t$ do not overlap, we say the family is *disjoint*. If each character of A is contained in some substring, we say the family represents a *cover* of A. Thus, Figure 4 shows a mapping between substring families such that $A|_5$, on the left, is a disjoint cover, and $B|_5$, on the right, is neither disjoint nor a cover. In general, we may require that either or both of the substring families be disjoint and/or a cover. Each possible combination of constraints represents a particular block edit model. For succinctness, we introduce the following notation:

C must be a cover,
$\overline{\mathrm{C}}$ need not be a cover,
D must be disjoint,
$\overline{\mathrm{D}}$ need not be disjoint.

To refer to the model in which the first substring family must be a disjoint cover, and the second substring family is unconstrained, we write CD-$\overline{\mathrm{CD}}$. (Note: from a computational standpoint, by symmetry $\overline{\mathrm{CD}}$-CD is exactly the same problem.)

3.1 Block Edit Distance

Before defining block edit distance, we require an underlying function *dist* that returns the cost of corresponding a substring of A with a substring of B:

$$dist : \{i, j \mid 1 \le i \le j \le |A|\} \times \{k, l \mid 1 \le k \le l \le |B|\} \to \mathcal{R}$$

In some domains it is natural to assume that *dist* is traditional string edit distance, but any cost function could be used, and later we shall describe another distance function that has been developed for matching ink trajectories.

The *block edit distance* $\mathcal{B}$ between two strings A and B is determined by finding the best way to choose substring families of A and B and correspond each member of $A|_t$ with some member of $B|_t$. For each pairing, a cost is assessed based on the distance between the two substrings. The correspondence between blocks is given by a permutation $\sigma \in S_t$ from the symmetric group on t elements. We impose the additional restriction that if $i \neq j$ and $A^{(i)} = A^{(j)}$, then $B^{(\sigma(i))} \neq B^{(\sigma(j))}$. That is, a particular pair of blocks cannot be placed into correspondence more than once. This allows us to keep the measure from diverging if a negative-cost pairing exists and the substring families do not have to be disjoint. More formally,

$$\mathcal{B}(A, B) \equiv \min_{t} \min_{A|_t, B|_t} \min_{\sigma \in S(t)} \left\{ \sum_{i=1}^{t} dist\left(A^{(i)}, B^{(\sigma(i))}\right) \right\} \tag{7}$$

	$\overline{\mathrm{CD}}$	$\overline{\mathrm{C}}\mathrm{D}$	$\mathrm{C}\overline{\mathrm{D}}$	CD
$\overline{\mathrm{CD}}$	$O(m^2n)$			
$\overline{\mathrm{C}}\mathrm{D}$	$O(m^2n)$	NP-complete		
$\mathrm{C}\overline{\mathrm{D}}$	$O(m^2n)$	NP-complete	NP-complete	
CD	$O(m^2n)$	NP-complete	NP-complete	NP-complete

Table 2: Complexities of block matching problems.

Equation 7 does not specify whether the particular substring families must be covers, disjoint, or both. In an earlier paper [10], we examined the various cases, showed which are hard, and presented algorithms for those that are solvable in polynomial time. Table 2 summarizes these results.

3.2 An Algorithm for Block Edit Distance

We now present a polynomial-time algorithm for one of the variants of block edit distance that can be used for matching pictorial queries, CD-$\overline{\mathrm{CD}}$.

Say that B is the string whose substring family need not be disjoint or a cover (*e.g.*, B corresponds to the database entry). For the discussion that follows, it will be convenient to assume we have an array W^1 defined as below for $1 \leq i \leq j \leq m$:

$$W^1(i,j) \equiv \min_{k \leq l} \{dist(a_i \ldots a_j, b_k \ldots b_l)\} \tag{8}$$

That is, $W^1(i,j)$ gives the value of the best possible match between $a_i \ldots a_j$ and any substring of B. Since portions of B can be re-used, and it need not be covered, the information in W^1 is sufficient to perform the needed calculations for the CD-$\overline{\mathrm{CD}}$ problem. (Similar matrices W can be defined for the other solvable versions of the problem.) We write $T(W)$ to mean the time required to compute a matrix W, and shall discuss later how W can be computed more efficiently than the naive implementation when *dist* is standard edit distance.

Consider the diagram shown in Figure 5. Each of the intervals $a_i \ldots a_j$ in the figure represents a substring of A, and is labelled with $W^1(i,j)$. Note that $W^1(i,i)$ represents the best match between the single character a_i and any interval of B. As before, a substring family of A is a multiset of substrings (*i.e.*, intervals). If the intervals do not overlap, the family is disjoint; if the union of the intervals is the entire line, the family is a cover. Enforcing or relaxing these constraints (all relative to string A) results in different versions of the block edit distance problem.

It is clear from Figure 5 that W^1 induces a complete interval graph, a well-studied class for which most known problems have efficient solutions [15]. We define $\mathcal{M}(i)$ to be the best block match between $a_1 \ldots a_i$ and B. We can then extract blocks of the query string optimally, with corresponding blocks of the database, using the following recurrence:

$$\mathcal{M}(i) = \min_{j<i} \{\mathcal{M}(j) + W(j+1, i)\} \tag{9}$$

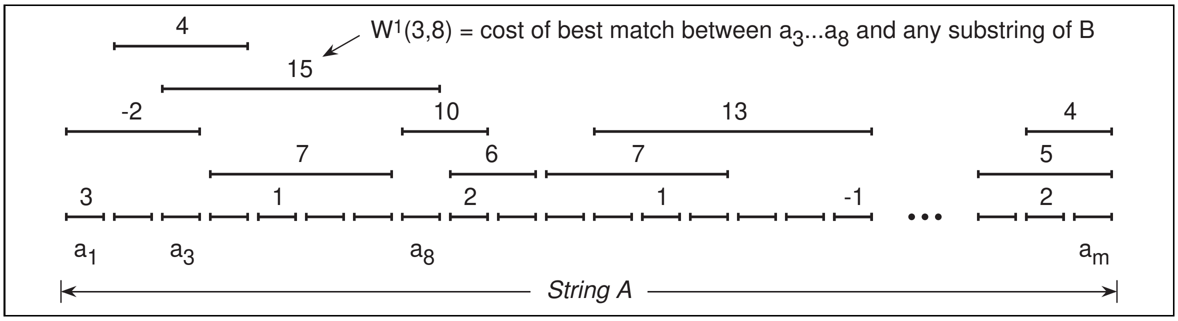

Figure 5: Possible string matches viewed as intervals.

In the recurrence, $\mathcal{M}(i)$ allows the best match in the database corresponding to $q_{j+1} \ldots q_i$ to be added to the optimal solution for $q_1 \ldots q_j$ for all possible "cuts" in the query, j. Once we have computed $\mathcal{M}$ for $i = 1, 2, \ldots, m$ (recall that $|A| = m$), our final answer is $\mathcal{B}(A, B) = \mathcal{M}(m)$.

3.3 Time Complexity

The recurrence above requires $O(m^2)$ time, where $m = |A|$. However, it depends on having a matrix W^1, so the full time bound is $O(m^2) + T(W^1)$.

If we build W^1 according to its definition (*i.e.*, Equation 8), we must fill in $O(m^2)$ entries by comparing $O(n^2)$ values, each of which can take time $O(mn)$ to compute when *dist* is standard string edit distance. Thus, naively, $T(W^1) = O(m^3 n^3)$. There is, however, a well-known modification of the basic dynamic programming algorithm that allows the best match in B for a fixed substring in A to be found in time $O(mn)$. This saves a factor of $O(n^2)$ over the naive approach. Furthermore, a property of this computation is that the table generated for matching $a_i \ldots a_n$ to B contains information about the best substring matches for $a_i \ldots a_k$ as well, for $i \leq k \leq n$. Hence, only $O(m)$ such tables need be built to compute W^1, saving another $O(m)$. Thus, $T(W^1) = O(m^2 n)$.

3.4 Elastic Match Distance

The block edit algorithm just described can be used with ScriptSearch as its underlying distance function. However, we have also begun to explore another technique, based on elastic matching, that appears to offer even better results.

Elastic matching is simply edit distance over an alphabet of points in 2-space, and is commonly used in on-line stroke processing. Temporal information is used in the sequence of points, but not elsewhere. Given a subsequence of the pattern $P = p_1, p_2, \ldots, p_r$, where each p_i is a point (x, y), and a subsequence of the text $T = t_1, t_2, \ldots, t_s$, where again each t_i is a point, we define the elastic distance E as follows:

$$\begin{aligned} e_{i,0} &= e_{i-1,0} + del(p_i) \\ e_{0,j} &= e_{0,j-1} + ins(t_j) \end{aligned}$$

$$e_{i,j} = \min \begin{cases} e_{i-1,j} + del(p_i) \\ e_{i,j-1} + ins(t_j) \\ e_{i-1,j-1} + sub(p_i, t_j) \end{cases}$$

where $del(p_i) = \delta_1$, $ins(t_j) = \delta_2$, and $sub(p_i, t_j) = \sqrt{(p_i.x - t_j.x)^2 + (p_i.y - t_j.y)^2}$. We then define $E(P,T) = e_{r,s}$.

Thus, two sequences of points in space can be placed into correspondence by deleting points when necessary, and aligning similar points that are close, paying a penalty proportional to their distance.

Given two trajectories that we wish to match, we perform some pre-processing before running the elastic match algorithm as specified. First, we process the trajectories so that each has n points, as follows: treat each sequence of points as a piecewise-linear path through the plane (by "connecting-the-dots"). Calculate the distance of the total path, and break it into $n-1$ pieces of equal length. The new points are the breaks between the $n-1$ pieces. Each trajectory is rescaled so that its bounding rectangle has origin at the origin and size 100×100. We then compute the elastic matching. (For performance reasons, we replace the Euclidean distance with "Manhattan" distance given by $d(p,q) = |p.x - q.x| + |p.y - q.y|$.)

Elastic matching performs well with the re-sampling policy defined above. Unfortunately, this policy makes it impossible to use the same "trick" we can employ in the case of ScriptSearch to reduce the overall complexity of block matching from $O(m^3n^3)$ to $O(m^2n)$. Thus, at heart, block distance with elastic matching is an $O(m^3n^3)$ algorithm. An actual implementation of block matching, however, need not compare every possible subsequence of the pattern to every subsequence of the text. We can perform significant "pruning" of the block comparisons based on the following heuristics: (1) number of pen-ups in path, (2) number of sampled points in path, (3) bounding box of path, (4) total amount of ink in path, and (5) reachability of path.

In our experiments, we prune using all of the above techniques. The first four are straightforward, but the last requires some discussion. Block distance has several possible variants, some of which are harder than others. The polynomial-time versions offer different properties that can be matched to the requirements of the search task, but for most tasks, it is reasonable to assume that the query should be matched in its entirety and each part of the query should be used only once. Reachability is the following condition: if we know that no good matches can be found that end at a certain point in the query, then we need not consider matches that begin at the next point, because such matches will never be used in the final partition.

4 Experimental Results

We have tested the block matching algorithm with both VQ distance (ScriptSearch) and elastic matching. In Section 4.1, we describe sample databases and query sets generated from a number of subjects. In Section 4.2, we give the results of our algorithms on these datasets.

4.1 Design of the Experiment

We designed our experiment so that purely local similarity measures would not be effective, but more powerful global approaches should perform well. Each of five subjects was asked to create an ink database consisting of 25 sketches, and a set of 25 ink queries to the database. An example instruction for creating a database page might be the following:

"Draw a chair, a desk, a computer, and a lamp."

Instructions for creating queries always consisted of either 1 or 2 elements, so an example query meant to match the database element just presented might be the following:

"Draw a lamp and a chair."

Note that the same query might match other database elements as well. Of the 25 queries, three matched three elements of the database, nine matched two elements, and 13 matched only one element.

Writers were asked to draw the elements in the order in which they appeared. Thus, the database element above contains ink for a chair, followed by some other ink, followed by ink for a lamp. The query contains ink for a lamp followed by ink for a chair. The block matching algorithm must determine an appropriate segmentation of the query into, say, two pieces representing the chair and the lamp. It must then match each of those blocks to the appropriate part of the database element.

Three writers drew the queries a day later than the database (Subjects 1, 2 and 4), while the other two drew both sets in the same session. The mean database element contained 1,502 sampled points; the mean pattern contained 713 points, with a total of 279,210 points collected in the experiment. The average database element was draw with 32 pen-ups, and the average pattern with 16.

4.2 Evaluation

Tables 3-4 and 5-6 give the raw data for rankings obtained using block editing with VQ and elastic match distance, respectively. It is clear from the tables that elastic matching yields substantially better results than ScriptSearch. As we mentioned earlier, ScriptSearch is optimized for searching through handwritten script, and uses a representation based upon a space of "important" features. In the picture matching domain, an algorithm like elastic matching that operates more firmly in the spatial domain seems better-suited to the problem.

More concretely, out of 125 matching tasks (25 queries for each of 5 writers), elastic match performed perfectly, ranking all correct sketches above all other sketches, 75% of the time. In the other 25% of the cases, the correct sketches were almost always ranked close to the top. VQ distance found the perfect set of matches only 16% of the time, although its accuracy for textual searches is substantially higher.

If we assume that any sketch matching the query would be an acceptable response from the search algorithm, then elastic matching gives an acceptable response as its first choice in 78% of the cases. Likewise, the average number of incorrect sketches ranked ahead of a

	Query Pattern												
	1	2	3	4	5	6	7	8	9	10	11	12	13
Subject 1	1,11	16	7	2	1,4,6	2,3	8,11,14	8,19	4,7	15	2,5	2	4,6
Subject 2	1,2	1	2	4	3,10,16	1,4	2,3,9	2,10	2,4	15	3,12	2	9,19
Subject 3	5,9	11	4	1	1,7,9	4,6	2,9,17	3,19	13,19	6	3,6	3	4,10
Subject 4	1,8	19	16	20	1,5,14	3,8	5,12,16	8,9	8,20	10	1,8	2	2,8
Subject 5	1,3	12	1	9	1,8,11	1,8	1,3,11	7,16	6,22	9	1,2	3	9,12

Table 3: Rankings for block editing with underlying VQ distance.

	Query Pattern											
	14	15	16	17	18	19	20	21	22	23	24	25
Subject 1	8	2,3,13	7	1,6	4,9	3	1	15	2	4	18	1,3
Subject 2	1	1,2,17	8	1,3	2,3	2	1	13	4	1	1	1,8
Subject 3	1	1,3,7	4	1,2	1,6	5	4	18	2	4	4	8,20
Subject 4	4	4,10,11	4	2,3	3,6	1	1	3	4	3	8	1,7
Subject 5	4	1,2,12	2	2,3	2,7	22	4	21	3	5	1	4,5

Table 4: Rankings for block editing with underlying VQ distance (cont.).

true hit is 0.52, which drops to 0.32 if we remove four outlying points of the 125 queries. More precisely, 92% of all queries had one or fewer incorrect sketches ranked ahead of them, 95.5% had two or fewer incorrect matches ahead of them, 97.5% had four or fewer incorrect matches ahead of them. We do not have data at this point to indicate how these numbers would scale.

Figure 6 shows the effectiveness of the ScriptSearch (VQ) and elastic match distances respectively on a per-hit basis. Recall that the 25 queries contained 13 queries with one hit, nine with two hits, and three with three hits, for a total of 40 hits. With five writers, this means we have a total of $40 \cdot 5 = 200$ hits. We will say that the performance of a particular hit is the number of spurious drawings ranked ahead of the hit. As an example, assume that a 3-hit query has rankings "1,3,7." This means that one hit was ranked first, which means that no drawings were ranked ahead of it. The next hit was ranked third, but we will not penalize it for the correct hit at position 1; there was one spurious drawing ahead of it (ranked second by the algorithm). Finally, the last hit was ranked number 7, with four spurious drawings ahead of it (ranked 2, 4, 5 and 6). The figures are histograms showing how many hits had no spurious drawings ranked ahead of them, how many had only one

	Query Pattern												
	1	2	3	4	5	6	7	8	9	10	11	12	13
Subject 1	1,2	1	1	2	1,2,3	1,2	1,2,3	1,2	1,2	1	1,2	1	1,2
Subject 2	1,4	1	1	1	1,2,3	1,2	1,2,3	1,2	2,4	1	1,2	1	1,2
Subject 3	2,5	1	2	2	1,3,10	1,2	1,2,3	1,2	1,2	1	1,2	1	1,2
Subject 4	1,2	4	1	1	1,2,3	1,2	1,2,3	1,2	1,2	1	1,3	1	1,2
Subject 5	1,2	1	2	3	1,2,3	1,2	1,2,3	1,2	2,4	1	1,2	1	1,5

Table 5: Rankings for block editing with underlying elastic match distance.

	Query Pattern											
	14	15	16	17	18	19	20	21	22	23	24	25
Subject 1	2	1,3,4	1	1,20	6,15	1	1	2	1	1	2	1,4
Subject 2	1	1,2,10	1	1,2	1,2	2	1	2	1	1	1	1,2
Subject 3	1	2,3,4	1	1,2	1,2	2	2	1	1	1	1	1,2
Subject 4	1	2,3,5	1	1,2	1,2	5	1	1	2	1	1	1,2
Subject 5	1	1,3,4	1	1,4	1,2	2	1	1	1	1	1	1,2

Table 6: Rankings for block editing with underlying elastic match distance (cont.).

spurious drawing ranked ahead of them, etc.

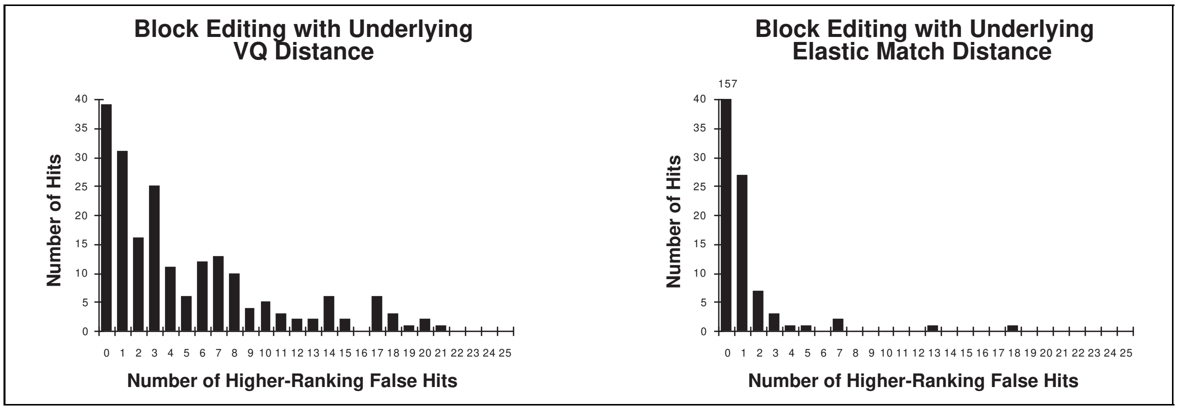

Figure 6: Histograms showing ranking effectiveness of VQ and elastic match distances.

5 Conclusions

Our primary conclusion is that successful picture matching requires solutions to two difficult problems. First, reasonable choices for possible corresponding blocks have to be found, and second a good spatially-based matching algorithm must be used to compare these blocks.

Our algorithm performed well in most cases, but occasionally a writer will draw a representation of the same object differently on two occasions. The difference could be invisible to an off-line algorithm (*i.e.*, the same points could have been drawn in a different order. Or the difference could be visible, if for instance the figure was drawn in a different orientation, substantially different scale, rotated due to the position of the writer, drawn carefully versus scribbled, and so on.

One can imagine algorithms to address the first problem, in which the strokes are drawn differently but the figure looks the same, by finding a canonical representation for different trajectories that look the same. The second class of problems seems more difficult, but it is likely that many instances of these problems could be handled by simple heuristics such as matching both a block of ink, and the same block rotated across a vertical axis — thus, a car facing left would match a car facing right.

There are also several algorithmic issues raised by this work. Most importantly, is it possible to implement elastic matching such that the complexity of the overall block distance problem can be reduced to cubic? But in addition, are there other approaches to pruning that could reduce the number of times an underlying distance measure needs to be called. And of course, researchers have often studied the problem of finding linear-time variations of edit distance style algorithms such as elastic matching (see, for example, [8]) – are there approaches to this problem that are particularly suitable for ink matching?

References

[1] W. Aref, D. Barbará, D. Lopresti, and A. Tomkins. Ink as a first-class datatype in multimedia databases. In S. Jajodia and V. S. Subrahmanian, editors, *Multimedia Databases*. Springer-Verlag, 1995.

[2] D. Goldberg and A. Goodisman. Stylus user interfaces for manipulating text. In *ACM Symposium on User Interface Software and Technology*, pages 127–135, Hilton Head, SC, November 1991.

[3] D. Goldberg and C. Richardson. Touch-typing with a stylus. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 80–87, Amsterdam, The Netherlands, April 1993.

[4] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.

[5] R. Hull, D. Reynolds, and D. Gupta. Scribble matching. In *Proceedings of the Fourth International Workshop on Frontiers in Handwriting Recognition*, pages 285–294, Taipei, Taiwan, December 1994.

[6] 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.

[7] 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.

[8] D. Lopresti and A. Tomkins. Pictographic naming. In *Adjunct Proceedings of the Conference on Human Factors in Computing Systems*, pages 77–78, Amsterdam, The Netherlands, April 1993.

[9] D. Lopresti and A. Tomkins. On the searchability of electronic ink. In *Proceedings of the Fourth International Workshop on Frontiers in Handwriting Recognition*, pages 156–165, Taipei, Taiwan, December 1994.

[10] D. Lopresti and A. Tomkins. Block edit models for approximate string matching. In R. Baeza-Yates and U. Manber, editors, *Proceedings of the Second Annual South American Workshop on String Processing*, pages 11–26, Valparaíso, Chile, April 1995.

[11] I. S. MacKenzie, A. Sellen, and W. Buxton. A comparison of input devices in elemental pointing and dragging tasks. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 161–166, New Orleans, LA, April 1991.

[12] 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.

[13] M. Nakagawa, K. Machii, N. Kato, and T. Souya. Lazy recognition as a principle of pen interfaces. In *Adjunct Proceedings of the Conference on Human Factors in Computing Systems*, pages 89–90, Amsterdam, The Netherlands, April 1993.

[14] A. Poon, K. Weber, and T. Cass. Scribbler: A tool for searching digital ink. In *Human Factors in Computing Systems Conference Companion*, pages 252–253, Denver, CO, May 1995.

[15] F. S. Roberts. *Graph Theory and Its Applications to Problems of Society.* SIAM, Philadelphia, PA, 1978.

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

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

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