On Indexing Handwritten Text

Ibrahim Kamel
Dept. of Electrical and Computer Engineering
University of Sharjah
kamel@sharjah.ac.ae

Abstract

This paper deals with one of the new emerging multimedia data types, namely, handwritten cursive text. The paper presents two indexing methods for searching a collection of cursive handwriting. The first index, called word-level index, treats word as pictogram and uses global features for representing the cursive words and their retrieval. Each word (or stroke) can be described with a set of features and, thus, can be stored as points in the feature space. The Karhunen-Loève transform is then used to minimize the number of features used (data dimensionality) and thus the index size. Feature vectors are stored in an R-tree. The second index, called stroke-level index, treats the word as a set of strokes.

We implemented both indexes and carried many simulation experiments to measure the effectiveness and the cost of the search algorithm. The proposed indexes achieve substantial saving in the search time over the sequential search. Moreover, the proposed indexes improve the matching rate up to 46% over the sequential search. The word-level index is suitable for large collection of cursive text. The stroke-level index is more accurate than the word-level index, but the stroke-level index is more costly than the word-level index in terms of the search time.

1. Introduction

The widespread of tablet computers and Personal Digital Assistants (PDA) like iphone has resulted in an increase interest in manipulating handwritten text and cursive writing. Recent researches focus on the efficient storage and retrieval of handwritten notes and on formulating queries based on handwritten databases. For example, searching a large database which contains one or more handwritten fields (e.g., verifying signature of the bank accounts). One way to handle the handwritten text is to translate it first into ASCII-equivalent characters using pattern recognition techniques and then to store it as ASCII text. Similarly, the search algorithm translates the query string into a sequence of ASCII characters and then performs a traditional search through the database. Thus, the recognition phase is an intermediate step between the input device (pen and tablet) and the storage device. But this is not practical because of the latency delay introduced by the recognition step. Also, research results show that the accuracy of the recognition of cursive writing is low and thus will result in high error rate. It is difficult even to identify letter boundaries in the cursive string. Moreover, by translating the handwritten string into a sequence of predefined symbols (alphabet), we lose much information, such as the particular shape of the letter "allograph", the writing style, etc. Another disadvantage to this method is that the recognition phase renders the system sensitive to the underlying language.

The other alternative is to treat it as a first-class data type. The handwritten string is treated as a pictographic pattern without an attempt to understand it. This is a more natural way to handle the handwritten text. In this approach the query string is compared to database strings using an appropriate distance function. This gives the user more expressive power; he can use non-ASCII symbols, drawings, equations, other languages, etc. Searching

in handwritten cursive text is a challenging problem. A word that is written by two different people cannot look exactly the same. Moreover, a person cannot recreate perfectly even his own previously drawn word. Hence, exact match query will not be appropriate and similarity (or approximate string matching) would be more suitable in this case. The search algorithm should look for all strings which are "similar" to the query string. One additional requirement for pen-based and/or personal digital assistant environment is the need to support online retrieval and fast response time.

In this paper we address the problem of searching for a given cursive string in a database of handwritten text. We proposed two indexes for retrieving handwritten text, one works at the word level by treating the whole word as an image. The other index breaks the handwritten word into strokes. The word-level index, which is the faster one, is suitable for large collection of cursive text. On the other hand, the stroke-level index is more expensive in terms of execution time; however, it is more accurate than the word-level index. Some early results of this project have been presented in [9][10]. The rest of the paper is organized as follow. Section 2 presents background information on the underlying multimedia index that is used. It also presents a sequential search algorithm for cursive handwriting. Section 3 describes the proposed word-level index while section 4 presents the proposed stroke-level index. Prior works related to cursive handwriting is briefly described in Section 5. In Section 6 we show experiments for measuring the response time and the matching rate of the proposed indexes. Section 7 gives our conclusions and future work.

2. Background

This section introduces some background information about the distance function and the underlying index that are used in the proposed technique. Section 2.1 describes briefly a well know multimedia index, R-tree. Section 2.2 describes the VUE technique that is used as a distance function to measure the similarity between two strings. Prior works related to the retrieval of cursive handwriting is outlined in Section 5.

2.1. R-trees

In this section, we provide a brief description of R-trees, which will be the underlying multidimensional index. The R-tree [6] is an extension of the B+-tree for multidimensional objects.

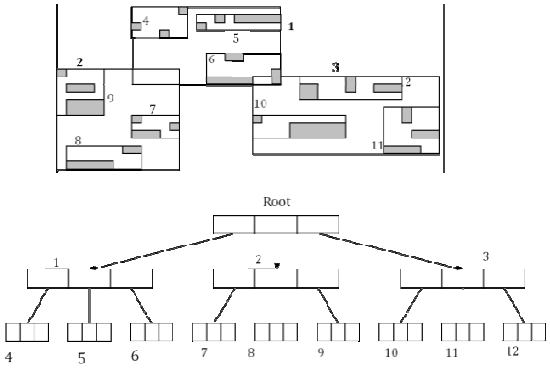

Figure 1: Data rectangles R-tree with fanout value of three

A geometric object is represented by its minimum bounding rectangle (MBR). Non-leaf nodes contain entries of the form (*ptr*,R) where *ptr* is a pointer to a child node in the R-tree; R is the MBR that covers all rectangles in the child node. Leaf nodes contain entries of the form (*obj-id*, R) where *obj-id* is a pointer to the object description, and *R* is the MBR of the object. One of the main features of the R-tree is that it allows nodes to overlap. In this way, the R-tree can guarantee at least 50% space utilization and at the same time remain balanced. Figure 1 illustrates data rectangles (in black) organized in an R-tree with a fanout value of three. On the other hand, excessive overlap between nodes penalizes the search performance. A worst-case scenario would require retrieval of the whole tree, but this rarely happens with practical datasets.

Subsequent work on R-trees includes: the packed R-tree [17] and Hilbert-packed R-trees [11] for static databases; and the R*-tree 0, and the Hilbert R-tree [12] for dynamic databases. The last of these can increase space utilization to any desired value by employing the concept of deferred splitting (local rotation).

2.2. Sequential Search in Handwritten Database

Although much research has been done in searching handwritten text, not much work has been done in indexing handwritten text. [14] proposed a sequential algorithm for searching a very long cursive string in order to locate all the occurrences of a small cursive string. A set of points that are drawn without raising the pen is called a stroke. The idea is to define a constant alphabet for the strokes. Each stroke can be one of 64 different stroke types (or code books). The problem is then reduced to the traditional ASCII text comparison, but with using a different alphabet (alphabet of strokes).

We can use the edit distance [20] to compare two cursive strings. The "edit distance" aligns the two strings and transforms one string into the other using the following operations: delete a symbol, insert a symbol, and substitute one symbol for another. Each of these operations has a predefined cost associated with it. The "edit distance" technique uses a dynamic programming algorithm to minimize the cost of the transformation. Any substring that is similar to the query string is reported as an answer.

Although reducing each stroke vector to one of 64 alphabet symbols makes the comparison much simpler and saves space, nonetheless, valuable information about the proximity between the different strokes is lost. In addition, it becomes necessary to search the whole database sequentially in order to inspect each substring. We call this method the VUE algorithm.

3. The Proposed Word-level Index

In this section we propose a two-step indexing schema to index a large repository of handwritten cursive text. The proposed index treats each word as a pictogram (or an image) and it consists of two steps, the filtering step and the refinement step. In the first step we use a coarse index that filters out most of the unwanted pictograms and produces a set of pictograms called the candidate set. The second step uses a sequential algorithm that operates on the candidate set to find the best k matches to the query word which are reported as the final answer.

The filtering step uses global features to characterize the different pictograms (words). A set of features $f_1, f_2, \ldots, f_x$ (described in Section 3.2) are calculated for each pictogram in the database as well as for the queries. Thus, each pictogram can be represented as a multidimensional point in x-dimensional space. A good set of features maps two instances

of the same pictogram to points that are close in the multidimensional space. At the same time, two different pictograms should be positioned as far apart as possible.

These points are organized in a multidimensional index. We choose the R-tree because of its ability to prune the search space at early levels of the tree structure and because of the guarantee of good space utilization.

To find the pictograms that are similar to a given query pictogram, we map the query to a point in the *x*-dimensional space, by extracting *x* global features. Then, we perform a similarity search. The length along each axis is a function of the characteristics of the user's handwriting (Section 3.1). The candidate set is formed by the multidimensional points (i.e., pictograms) returned by the query.

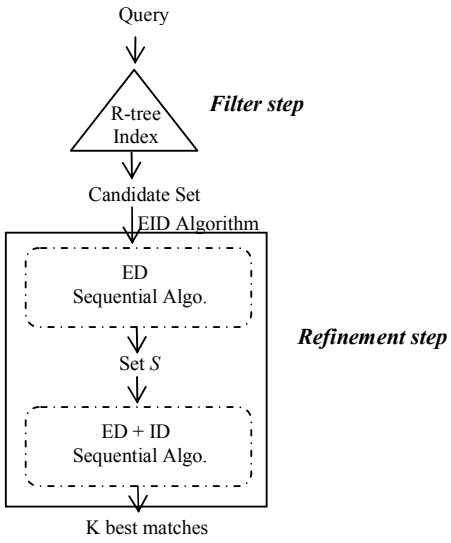

Figure 2: Flow diagram that shows our proposed word-level indexing schema.

Edit distance: is a distance function that quantifies the similarity between two text strings. The edit distance used in [14] aligns the two strings and transforms one string into the other using the following operations:

- Deletion of a symbol.
- Insertion of a symbol.
- Substitution of a symbol by another one.
- Splitting of a symbol into two.
- Merging two symbols into one.

Each of these operations has a predefined cost associated with it. The weighted cost of the transformation is used as a distance between the two strings. We call this metric edit distance ED.

Inflection distance: we define the inflection distance in the same way the edit distance is defined. Ink can also be represented by a sequence of codewords that represent the inflection points of the pictogram. An inflection point marks the change of direction in the pictogram, e.g., going from an upwards direction to a downwards direction, or from a left-

ward direction to a rightward one. We define eight inflection symbols. Using the inflection representation, we can also compute an edit distance between two pictograms. For this, we assume that only insertions and deletions are allowed and that each operation (insertion or deletion) has an associated cost of 1. We call this inflection distance (ID).

In the refinement step, Figure 2, we apply a sequential algorithm to the candidate set to find the best *k* matches to the query word. We use a combination of the two metrics ID and ED. Our experiments show that the cost of the ID comparison is about 5 times less than the cost of ED comparison. Thus, we use the ID as a second stage filter in the retrieval process. The procedure is as follows. First, we perform pair wise comparison (sequential search) of the query against each of the retrieved pictograms by using ID. This results in a ranked set *S* of pictograms. Then, we take the best *m* (in the experiments we set *m = 20%*) elements of *S* and use the metric ED+ID to perform another round of sequential comparison. We call this procedure EID. Increasing *m* improves the retrieval rate but it increases the total response time. The value of *m* can be determined experimentally. After that, the best *k* matches are presented to the user.

3.1. The Size of the Hyper-rectangle

Two instances of the same word will have two different values $v_i, \acute{v}_i$ for feature f_i. To accommodate for this variability, the length l_i of the query hyper-rectangle along feature (axis) f_i should satisfy:

$$l_i \geq |v_i - \acute{v}_i|$$

There is a clear trade-off in the value of l_i. Larger values of l_i will introduce more false alarms and increase the size of the candidate set and consequently, the total response time. On the other hand, small l_i values will be more likely to miss the correct answer and consequently, reduce the retrieval rate. Since the value l_i is a writer dependent, we ask the user to write small sample of pictograms (around 30) twice. We use this sample as a measure for the variance in the values of $f_i, 1 \leq i \leq x$. For each feature i we calculate the differences in the corresponding pictograms in the sample set. The values of $l_i, 1 \leq i \leq x$, is selected so that it covers the differences in the sample set.

3.2. Global Features

Extracting good global features is not an easy task. A good global feature should assign different instances of the same word to close-by values in the feature domain. Features might be correlated and dependent. Adding a new feature could produce no improvement in the performance if the feature is redundant (covered by other features in use.) Using a large number of features not only increases the size of the index but also adversely affects the search performance. This is known as "dimensionality curse" problem.

This section lists the eight global features are used in our indexing scheme. Our model for the cursive string is as follows. A cursive string is read from the tablet as a sequence of points in real time. Each point is represented by the tuple *(x, y)* where *x, y* are the coordinates of the point in two-dimensional space. (Points are taken at equal time intervals, so the time information is also available.) These points are grouped into strokes. Strokes are defined by local minima in the *x*-*y* coordinates. A new stroke starts at each local minimum. (This method is known as local minima segmentation.) The minimum rectangle that encloses the stroke is called Minimum Bounding Rectangle (MBR).

Some of the features we collected use aggregate functions over the set of strokes that constitute the string. Some of the resulting strokes are very small and do not contribute to the final image of the string. We also noticed that these small strokes can be produced simply by pressing or raising the pen. In the evaluation of the following features, we filtered

out such strokes from both the database and the query strings and only included strokes whose MBR area is larger than 15 points (where the point is the unit distance in the tablet device).

- **Number of strokes:** that constitutes the string.
- **Number of points:** in the string.
- **Number of vertical inversions:** an inversion occurs when either the *y*-coordinate of the point decreases after it was increasing or the *y*-coordinate of the point increases after it was decreasing.
- **Total-change-MBR-height:** the accumulation of the absolute differences between the height of a stroke's MBR and that of its predecessor, i.e., :

$$\sum_{i=2}^{s} |Height(i) - Height(i-1)|$$

where *s* is the total number of strokes in the string.

- ***Avg-weighted-MBR-area***: the average weighted area of stroke's MBRs, i.e., :

$$\frac{\sum_{i=2}^{s} Area(i) \times i}{s}$$

where *s* is the total number of strokes in the string.

- ***Number of thin strokes***: thin strokes are defined as those stroke with height >1.5 × width.
- ***X-centroid:*** the position of the centroid of the stroke areas, calculated as follows:

$$\frac{\sum_{i=1}^{s} i \times Area(i)}{\sum_{i=1}^{s} Area(i)}$$

where *s* is the total number of strokes in the string.

- **Y-centroid***:* the position of the centroid of the stroke areas in the *y*-direction, calculated as:

$$\frac{\sum_{i=1}^{s} (y_i - y_0) \times Area(i)}{\sum_{i=1}^{s} Area(i)}$$

Where y_i is the highest *y*-value for the i^{th} stroke, y_0 is the *y*-value for the first point of the string, and *s* is the number of strokes.

This word-level index is suitable for searching large number of cursive handwriting. In Section 6, we perform simulation experiments to measure the effectiveness of the proposed index in filtering unwanted pictograms and identifying the most similar word.

4. The proposed stroke-level index

In this section, we propose an index that is suitable for small database of cursive handwriting. Unlike the word-level index, the stroke-level index compares text at the stroke level rather than word level. Thus, this index is more costly than the word-level index but it is more accurate in identifying the query work. The stroke-level index allows fast retrieval of similar strings and can handle insertion, deletion, *m-n* substitution errors and substring matching. This index is dynamic in the sense that insertion and deletion operations can be

intermixed in real time with the search operations. Given a search query string, the answer would be a set of the strings or substrings that look like the query string.

4.1. The basic idea

We model the cursive string as a sequence of strokes. Each stroke is described by a set of features and thus can be represented by a point in the multidimensional feature space. We propose to store these points in a multi-dimensional index, and more specifically, the R-tree because of its ability to prune the search space at early levels of the tree structure and because of the guarantee of good space utilization. A cursive string is read from the tablet as a sequence of points in real time. Each point is represented by the tuple *(x, y, t)* where *x, y* are the coordinates of the point in two-dimensional space and *t* is the time at which the point is printed. These points are grouped into strokes. A stroke ends and a new one starts at each local minimum in the *x-y* coordinates. This method is known as local minima segmentation.

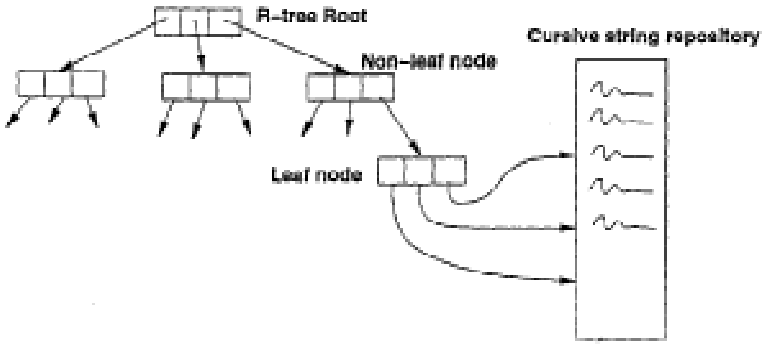

Figure 3: Outline of the word-level index

We describe each stroke with a set of 11 features. The features, which have been described in [18], describe the geometric properties of the stroke, e.g., the length of the stroke, the total angle traversed, and the angle and length of the bounding box diagonal. The features are selected so that strokes that look alike tend to have similar vector values according to some distance functions. Due to the variability in handwriting, the feature vectors that correspond to different instances of one stroke tend to vary slightly. Vectors that represent different instances of the same stroke form a cluster in the feature space. Thus, strokes that look similar will have their representative clusters close to each other or even overlapping in the multi-dimensional space.

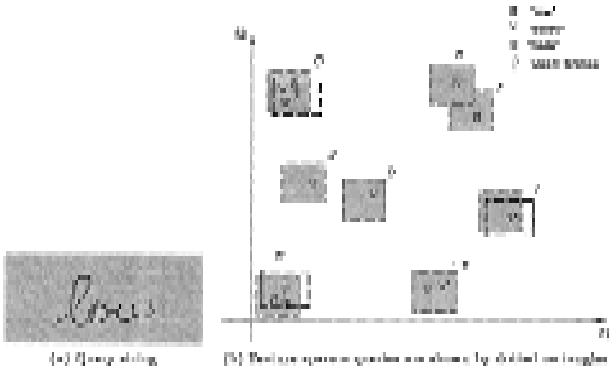

Figure 4: (a) shows a pictogram of the cursive word "low" (b) the representation of the cursive word "low" in the feature space.

Given a string *S*, the stroke segmentation program decomposes *S* into a sequence of *t* stokes. Each stroke S_i , $1 < i < t$, is represented as a point in an 11-dimensional space formed by the features $f_1, f_2, \ldots, f_{11}$. The string *S* is represented by *t* points in the space. These multi-dimensional points (strokes) are stored in an R-tree index. Each R-tree node occupies one disk page. Non-leaf nodes, which are small in number, are kept in main memory, while leaf nodes are stored on the disk. Figure 3 shows an example of the proposed index assuming two features only (in two dimensional space). A set of points that are close to each other will be stored in the same leaf node (level 0 in the tree). Each entry in the leaf node is in the form of (*word-id*, *P*) contains the coordinates of a point *P* (stroke) and a *word-id* for the pictographic description of the string that contains (owns) this stroke. Non-leaf nodes in level *i*, where $i > 0$, have entries of the form (*ptr, R*) where *ptr* points to a child node and *R* is the Minimum Bounding Rectangle (MBR) that encloses all the entries in the child node.

Figure 4 shows an example of three cursive words "node, row, below." For the sake of this illustration and to simplify the presentation and the drawing of the example, let us assume that each stroke is described with only two features, namely f_1 and f_2. For simplicity of the presentation, let us assume that each stroke represents one letter in the string. Note that in practice, each letter is represented by several strokes but we make this assumption to simplify the explanation. Figure 4 shows the representation of the three strings in the multi-dimensional space. Each string is represented by several points (equal to the number of strokes/characters in the string) in the two-dimensional space. Strokes that represent the same letter tend to form a cluster. The shaded area in Figure 4 shows the cluster for each letter (stroke). Letters that are written in a similar way (e.g. *r, n*) might have clusters close to each other or even overlapping.

4.2. Similarity search queries

To search for a string *Q*, we treat the query string in a manner similar to that described in the previous section. The set of strokes $q_1, q_2, \ldots, q_x$: are extracted from *Q*. Since it is impossible to write the same word twice identically, we need similarity queries. For each stroke q_i, a range query in the form of a hyper-rectangle is formed in the 11-dimensional space. The center of the hyper-rectangle is the query point, and the length along each axis is a ratio ($2 \times p$) of the standard deviation of the data along that axis.

The output of each query would be a set of *word-ids* for those words which contain a stroke similar to the query stroke. We call this set the candidate set *C*. We then apply a simple voting algorithm as follows. Each *word-id* takes a score that indicates how many times it has appeared as an answer for the queries q_i, $1 < i < x$. The set of *word-ids* that have the highest scores are reported as the answer. Note that we did not use any expensive operations, nor we accessed any of the pictographic representation of the strings from the database.

Algorithm Search (**node** Root, **string** *Q*):

S1. *Preprocessing*:
Use *Q* to build the set of strokes $q_1, q_2, \ldots, q_x$.
Extract the set of features for each q_i, $1 < i < x$.
S2. *Search the index*:
For each stroke q_i, perform a range query.
Form the candidate sets.
S3. *Voting algorithm*:
Words with the highest score are the answer.

4.3. Feature space dimensionality

Our goal here is to reduce the number of features needed to describe the stroke by transforming the data points into another space with smaller dimensions. This problem is known as dimensionality reduction. Until now, we used 11 highly correlated features to describe each stroke. We use the *Karhunen-Loève* transform [5], also known as *Hotelling* transform or *Principal Component Analysis*) to reduce the dimensionality of the feature space. The transform maps a set of vectors to a new space with an orthogonal uncorrelated axis. The *Karhunen-Loève* transform consigns most of the discrimination power to the first few axes. Hopefully, using only *k* axes, $k < 11$, we lose little information while reducing the index size significantly.

The axes of the new feature space are the Eigen vectors of the auto correlation (covariance) matrix for the set of data points. The *Karhunen-Loève* transform sorts the eigenvectors in decreasing order according to the eigen values and approximates each data vector with its projections on the first *k* eigenvectors, $k < 11$.

We collect a small sample from the writer in advance and apply the *Karhunen-Loève* transform to calculate the vector transformation matrix. All strokes (vectors) are mapped to the new space and then inserted in the index.

4.4. Reducing the candidate set size

Two strings are similar if they have similar strokes in the same order. The output of the search query gives a set of strings which has strokes similar to the query stroke but they do not necessarily occur in the same location. The candidate set is thus large because it contains many false candidates. Moreover, the voting algorithm does not take into consideration the location of the stroke.

To make use of the stroke location and to reduce the size of the candidate set, we store the location of the stroke inside the string as one more dimension in the feature space. Each stroke is then represented by *k* features $f_1, f_2, \ldots, f_k$ and by its location stk_{loc} inside the string in $(k + 1)$ dimensional space.

Two instances of the same string will not, in general, have equal number of strokes. The difference, however, is expected to be small. Thus, the answer to the range query that corresponds to stroke q_i should include strings that have similar strokes not only in the

position i but also in a window of length w around i. We found experimentally that $w = 3$ gives the best results (thus covering stroke numbers i - 1, i, and i + 1).

In substring matching, however, we want to allow the query string to start at any position inside the database string. In this case, a partial match query rather than a range is used. In a partial match query, the extent of the query rectangle is specified for all axes $f_1, f_2, \ldots, f_k$ as before. For the stroke location stk_{loc} axis, the extent is left open $(-\infty, +\infty)$ to allow the query string to start at any position inside the database string. Otherwise, the algorithm is similar to that for similarity query.

5. Prior work

There are many researches on modeling, retrieval, and annotation of cursive handwriting. However, there are not many works on indexing cursive handwritten text. Research included handling different languages like Arabic [1], Chinese [15], and Indian languages [4] and on-line handwriting [16].

[4] proposed a technique that is based on an additive fusion resulted after a novel combination of two different modes of word image normalization and robust hybrid feature extraction. They employ two types of features in a hybrid fashion. The first one, divides the word image into a set of zones and calculates the density of the character pixels in each zone. In the second type of features, they calculate the area that is formed from the projections of the upper and lower profile of the word.

[22] proposed a method for generating a large database of cursive handwriting. Synthesized data are used to enlarge the training set. He proposed method learns the shape deformation characteristics of handwriting from real samples; then used for handwriting synthesis.

[18] also proposed a method to synthesize cursive handwriting of the user's personal handwriting style, by combining shape and physical models together. In the training process, some sample paragraphs written by the user are collected and these cursive handwriting samples are segmented into individual characters by using a two-level writer-independent segmentation algorithm. Samples for each letter are then aligned and trained using shape models.

[7] word-spotting system operates on a database containing a number of handwritten pages. The method used for word matching is based on a string matching technique, called Dynamic Time Warping (DTW). The following three features are computed at each sample point in the word, resulting in a sequence of feature vectors: The height (y) of the sample point: This is the distance of the sample point from the base of the word; the stroke direction; and the curvature of the stroke at point p. The word to be compared is first scaled so that it is of the same size (height) as the keyword, and translated so that both words have the same centroid. The DTW technique then aligns the feature vector sequence from a database word to that of the keyword using a dynamic programming-based algorithm. The algorithm computes a distance score for matching points by finding the Euclidean distance between corresponding feature vectors and penalizes missing or spurious points in the word being tested.

6. Experimental results

This section presents experimental results that show the effectiveness of our proposed indexes. The two proposed methods are implemented in *C*. Our database consists of 8,000 handwritten cursive words produced by one writer. The same writer then recreated 100 words to be used as queries. Since our data is static (no insertion nor deletion) we used the Hilbert packed R-tree [11] because of its high space utilization. For dynamic data, other R-

tree variants that allow insertions and deletions can be used (such as R*-tree [0], and the Hilbert R-tree [12]).

6.1. Evaluation of the global features

In this section we evaluate how good the set of global features, listed in Section 3.2, are in pruning the search space and retrieving the most similar set of words. We store the leaf nodes (which account for the large portion of the R-tree) on the disk and keep the non-leaf nodes in main memory (non-leaf nodes occupy about 50 K-bytes for 8000 words). Figure 5 shows the percentage of the database that is filtered out by the R-tree as a function of the database size. Notice that the pruning capability is increasing with increasing the database size. The reason for this is that the query size is constant regardless of the database size (recall that the query size is defined by the characteristics of the user handwriting.)

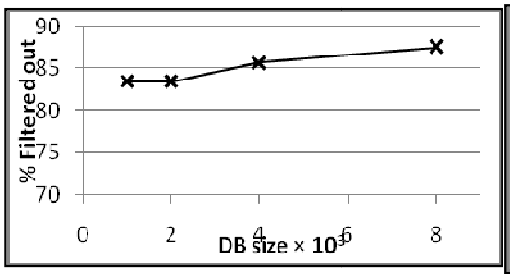

Figure 5. The percentage of the database filtered out by the R-tree

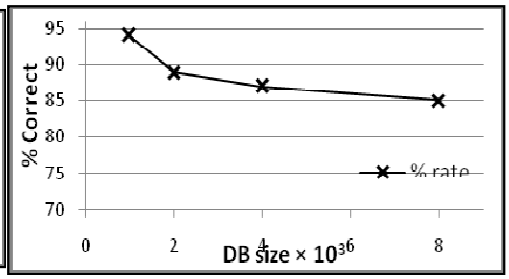

Figure 6. The retrieval rate of the R-tree only

To see how good the global features are in describing the cursive handwriting, we show in Figure 6 the percentage of cases in which the correct answer to queries are in the candidate sets (R-tree retrieval rate). The graph show high retrieval rate 86% - 95%. As expected, the retrieval rate decreases with increasing the size of the database.

6.2. Comparison between the proposed Index and the Sequential Scan

In this section we compare the proposed schema (R-tree + EID) with ED and EID sequential searches.

Figure 7 shows the total search time per query for various database sizes. The above R-tree package stored the tree in main-memory; thus, we had to simulate each disk access with a 15 *msec* delay. For our method (marked as "R-tree + EID"), it shows the time per query after searching the R-tree and screening the resulting subset of pictograms with EID. We compared our method with the ED sequential algorithm. The figure also shows the time it takes to perform sequential search over the entire database using our EID sequential algorithm. Our method "R-tree+ EID" outperforms ED sequential search in the entire range of database sizes. For 8,000 pictograms the ratio of search times is 12:1. We included in the graph the performance of our sequential EID (no R-tree) which also outperform ED. Note that, for small databases 1000 or less, EID is little faster than "R-tree + EID". This is because of the overhead of the R-tree (the relative cost of node access increases with decreasing the database size). As expected the sequential search times grow linearly with the size of the database, while for our method "R-tree + EID" the search times grow sub-linearly in the entire range.

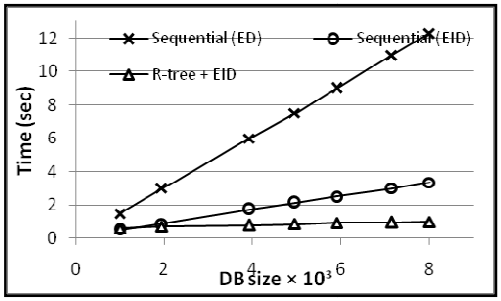

Figure 7. Total search time

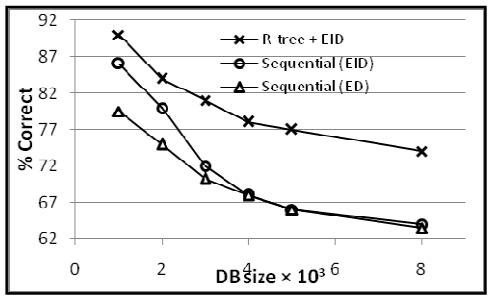

Figure 8. Matching rate (top 5)

Figure 8 and Figure 9 plot the matching rates obtained when showing the best $k = 3$ and $k = 5$ pictograms respectively. As we can see, the matching rates for the index outperform those of sequential search.

Figure 10 and Figure 11 explain the sub-linear behavior of our method. Figure 10 shows the percentage of pictograms returned by querying the R-tree. As we can see, the relative size of the subset obtained by searching the tree decreases with increasing the size of the database.

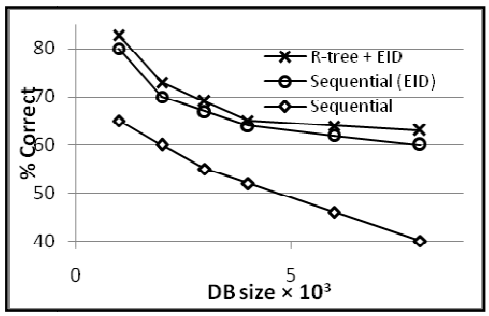

Figure 9. Matching rate (top 3)

Figure 11 shows the percentage of blocks retrieved by the tree search as a function of the database size. Again, since the relative size of the retrieved subset decreases, so does the percentage of blocks brought to memory.

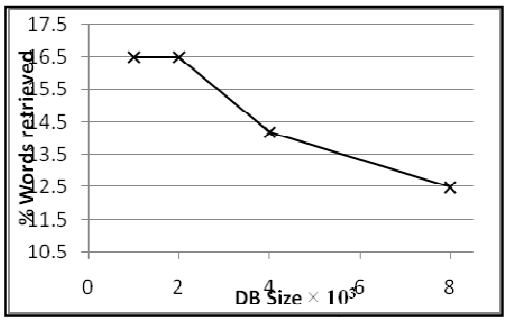

Figure 10. Percentage of pictograms retrieved

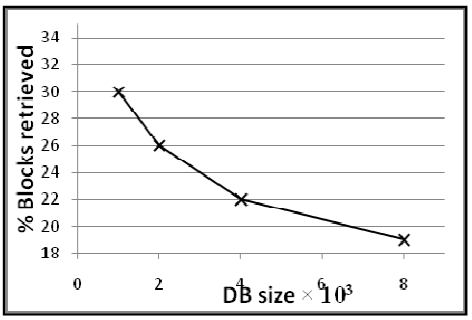

Figure 11. Percentage of blocks retrieved

6.3. Evaluation of the Stroke-based Index

We implemented the proposed stroke-based index and the VUE algorithm [14]. We carried several experiments to evaluate the performance of our proposed index and compare it with the VUE algorithm. Due to the space limitation, we do not show all the results. We asked one writer to produce 200 handwritten cursive words. The same writer then recreated 74 words to be used as search strings. In all the experiments, the stroke location was stored as additional feature as explained in Section 4.4. For the experiments shown here, the value *w* was set at 3 and the value of *s* was set at 1 (each stroke was stored as a separate point.) Since the data used in the experiments were static, we used the Hilbert-packed R-trees [11] as an underlying multi-dimensional index. For data that has dynamic nature (where data can be inserted or deleted at any time), the R-tree [6], Hilbert R-tree [12] or the R*-tree [3] can be used. Node size was fixed at one KByte.

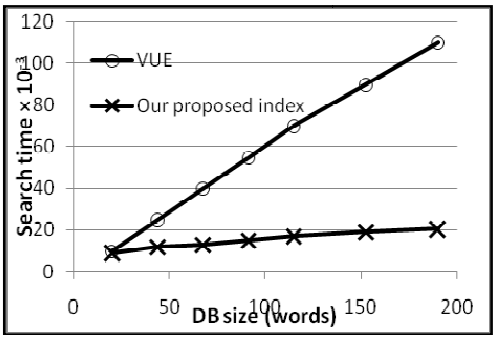

Figure 12. Response time of our proposed index versus the VUE algorithm

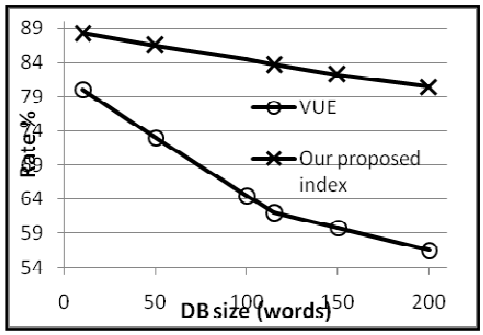

Figure 13. Matching rate of our proposed index versus VUE algorithm

The segmentation algorithm cuts the stroke once it encounters a local minimum. During our experiments, we noticed that some of the resulting strokes are tiny and do not contribute to the final image of the string, and thus considered noise. These tiny strokes can be produced simply by pressing or raising the pen. These strokes not only increase the size of the database but might also adversely affect the retrieval performance. We filtered out such strokes from both the database and the query strings. We only included strokes whose MBR diagonal is larger than 15 points (where the point is the unit distance in the tablet device.)

Table 1. Matching rates for index that uses all 11 features vs. index that uses 6 features only

Voting algo, Rank	Matching rate	
	11 features	6 features
first	80	73
top 2	85	82.5
top 3	89	84

Figure 12 compares the search time of our proposed index with the search time of the VUE algorithm for different database sizes. As expected the VUE algorithm time increases linearly with the database size. Our proposed index achieves substantial saving in response time over the VUE. Note that the VUE algorithm is faster than the index for small database

(less than 15 words) because of the constant overhead of the R-tree. The saving in time, when using the index, increases with the database size.

We also compared the matching rate of the proposed index and the VUE algorithm. Figure 13 shows the number of times the correct answer (matching rate) is ranked among top two for different database sizes. We also carried experiments that show the matching rate when the answer is ranked the first (received highest score) and among the top 5 for different database sizes (not shown for space limitation). The common observation is that the matching rate of our proposed index is consistently higher than that of the VUE algorithm. The improvement in the matching rate is up to 46%.

To evaluate the index when it uses the reduced feature space (as discussed in Section 4.4), we carried out two sets of experiments, one using the full set of features (= 11). In the second set of experiments we applied the *Karhunen-Loève* transform to a sample of 30 words to calculate the transformation matrix, and then all words in the database were mapped to the new six-dimensional space. The queries were also mapped using the same transformation matrix before searching the tree. Our experiments measured matching rate. We count the number of search words that were ranked first (received the highest score), among top two, and among top three by the voting algorithm. As we see in Table 1 the matching rate is about 84% when reporting strings with the highest three scores. As expected, the matching rate decreased as we used a smaller number of dimensions. The good news is that, although we cut the space required to store a stroke to nearly half, we nevertheless achieved about 93% of the matching power of the index that used all 11 features.

7. Conclusions

This paper introduced two indexing schemes for cursive handwriting. The first index works at the word-level and suitable for large database of cursive handwritten text. While the second index, which works at the stroke level is more accurate but it is also more costly.

The word-level index uses a set of global "word" features that provides an effective way of reducing searching cost. The experimental results showed that the proposed index, which is using R-trees followed by EID clearly outperforms the ED and EID sequential searches. The space overhead incurred by the R-tree is low. The sequential algorithm EID outperforms ED. Another important contribution is the identification of a small set of global features (eight features) that can be used to characterize cursive handwriting.

In the second index, each string is divided into a set of strokes; each stroke is described with a feature vector. Subsequently, the feature vectors can be stored in any multi-dimensional access method, such as the R-tree. A similarity search can be performed by executing a few range queries and by then applying a simple voting algorithm to the output to select the most similar strings. The stroke-level index is resilient to the errors resulting from segmentation errors, such as insertion, stroke deletion, or *m-n* substitution. Our experiments showed that the extra effort we spent in mapping the data to lower dimensionality space pays off. The stroke-level index achieves substantial saving in search time over the VUE algorithm and improves the matching rate up to 46% over the VUE algorithm. With a sacrifice of less than 10% of the matching accuracy we saved almost half of the space required to represent a stroke.

Our results showed that the word-level index is less accurate than the stroke level index, however it is much faster. The stroke-level index, on the other hand, is more accurate than the word-level index, but it is more costly than the word level index in terms of the search time. Thus the word-level index can be used as a filter to reduce the size of the candidate set. Then the stroke-level index can be used on a smaller data set to produce the final results.

References

[1] Al Aghbari, Z., Brook, S., HAH manuscripts: A holistic paradigm for classifying and retrieving historical Arabic handwritten documents. In: Journal of Expert Systems with Applications (2009)
[2] Aref, W., Kamel, I., Lopresti, D., "On Handling Handwritten Electronic Ink", The international Journal of ACM Computing Survey, Symposium on Multimedia Systems, December 1995, Vol 27, No 4, Pages 564 – 567.
[3] Beckmann, N. *etal*: The R*-tree: an efficient and robust access method for points and rectangles. In: Proc. of ACM SIGMOD (1990)
[4] Gatos, B., Pratikakis, I., Perantonis, S.J.: Hybrid Off-Line Cursive Handwriting Word Recognition, Pattern Recognition, vol. 2, pp. 998—1002 (2006)
[5] Gersha, A., Gray R.: Vector Quantization and Signal Compression. In: Kluwer Academic (1992)
[6] Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proc. of ACM SIGMOD (1984)
[7] Jain, A., Namboodiri, A.: Indexing and Retrieval of On-line Handwritten Documents. In: Proc. of the 7th International Conference on Document Analysis and Recognition, p. 655 (2003)
[8] Jawahar, C. V., Balasubramanian, A., Meshesha, M., Namboodiri, A.: Retrieval of online handwriting by synthesis and matching. In: Pattern Recognition, vol. 42, Issue 7 (July 2009)
[9] Kamel, I., Fast Retrieval of Cursive Handwriting, 5th international Conference on Information and Knowledge Management CIKM, 1996.
[10] Kamel, I., and Barbara, D., Retrieving Electronic Ink by Content, 1996 International Workshop on Multimedia Database Management Systems, 1996.
[11] Kamel, I., Faloutsos, C.: On packing Rtrees. In: Proc. of CIKM (1993)
[12] Kamel, I., Faloutsos, C.: Hilbert R-tree: an improved r-tree using fractals. In: VLDB (1994)
[13] Lopresti, D., Tomkins, A.: Pictographic naming. In: Tech. Rep. MITL- TR-21-92, Matsushita Information Technology Lab, (1992).
[14] Lopresti, D., Tomkins, A.: On the searchability of electronic ink. In: Tech. Rep. MITL- TR-114-94, Matsushita Information Technology Lab, (1994).
[15] Ma, Y., Zhang, C.: Retrieval of cursive Chinese handwritten annotations based on radical model United States Patent 6681044 (2004)
[16] Oda, H., Akihito Kitadai, Motoki Onuma, Masaki Nakagawa: A Search Method for On-Line Handwritten Text Employing Writing-Box-Free Handwriting Recognition. In: Proc. of the 9th International Workshop on Frontiers in Handwriting Recognition, pp. 545—550 (2004)
[17] Roussopoulos, N., Leifker, D.: Direct spatial search on pictorial databases using packed r-trees. In: ACM SIGMOD, pp. 17—31 (1985)
[18] Rubine, D.: The automatic recognition of gestures. In: PhD thesis, Carnegie Mellon University (1991)
[19] Varga, T., Bunke, H.: Generation of Synthetic Training Data for an HMM-based Handwriting Recognition System. In: Proc. of the 7th International Conference on Document Analysis and Recognition, p.618 (2003)
[20] Wagner, R., Fisher, M.: The string-to-string correction problem. In: Journal of ACM, vol. 21, pp. 168—173 (1974)
[21] Wang, J., Wu, C., Xu, Y., Harry Shum: Combining shape and physical models for on-line cursive handwriting synthesis. In: International Journal of Document Analysis and Recognition, vol. 7, No. 4, pp. 219—227 (2005)
[22] Zheng, Y., Doermann, D.: Handwriting Matching and Its Application to Handwriting Synthesis. In: Proceedings of the 8th International Conference on Document Analysis and Recognition, pp. 861—865 (2005)