The Handwritten Trie: Indexing Electronic Ink

Walid Aref Daniel Barbará Padmavathi Vallabhaneni
Matsushita Information Technology Laboratory
2 Research Way, 3rd Floor
Princeton, N.J. 08540
{aref,daniel,vpadma}@mitl.research.panasonic.com

Abstract

The emergence of the pen as the main interface device for personal digital assistants and pen-computers has made handwritten text, and more generally *ink*, a first-class object. As for any other type of data, the need of retrieval is a prevailing one. Retrieval of handwritten text is more difficult than that of conventional data since it is necessary to identify a handwritten word given slightly different variations in its shape. The current way of addressing this is by using handwriting recognition, which is prone to errors and limits the expressiveness of ink. Alternatively, one can retrieve from the database handwritten words that are similar to a query handwritten word using techniques borrowed from pattern and speech recognition. In particular, Hidden Markov Models (HMM) can be used as representatives of the handwritten words in the database. However, using HMM techniques to match the input against every item in the database (sequential searching) is unacceptably slow and does not scale up for large ink databases. In this paper, an indexing technique based on HMMs is proposed. The new index is a variation of the trie data structure that uses HMMs and a new search algorithm to provide approximate matching. Each node in the tree contains handwritten letters, where each letter is represented by an HMM. Branching in the trie is based on the ranking of matches given by the HMMs. The new search algorithm is parametrized so that it provides means for controlling the matching quality of the search process via a time-based budget. The index dramatically improves the search time in a database of handwritten words. Due to the variety of platforms for which this work is aimed, ranging from personal digital assistants to desktop computers, we implemented both main-memory and disk-based systems. The implementations are reported in this paper, along with performance results that show the practicality of the technique under a variety of conditions.

1 Introduction

In the last few years a number of pen computers have been released or announced (such as the Apple Newton, EO and NCR). These devices range from *personal digital assistants* (PDAs), intended to keep personal data such as schedules, notes, address books and so on, to full-scale computers with pens instead of (or in addition to) keyboards. They come equipped with large pen-based tablets or small writeable screens and provide the ability to perform functions such as note taking, browsing and mark-up of electronic documents. For PDAs, given the size limitations imposed by the need for portability, pens are the most natural (and some times the only) way to input data. Market research studies estimate there are 48 Million potential users of PDAs as a mobile office tool and 80 Million potential users of PDAs as a personal communications and data organization tool [11].

As in any other type of computer, users need to search for data that was previously input. When the data has been handwritten, the problem becomes more difficult than in conventional situations. The current way of addressing this is by using *handwriting recognition*, i.e., a procedure for converting pen strokes into strings of ASCII characters (or any other fixed character set). Once converted into ASCII, strings can be manipulated and searched in conventional ways. The problem is that handwriting recognition is prone to errors. Many critics point out that handwriting recognition does not meet the need of users. In fact, it is widely believed that this shortcoming is one of the main causes why the sales of PDAs have fallen short of the expected figures.

Many systems (such as the PenPoint operating system [6]) force the user to write each character in a separate fixed-size box. This is clearly a very unnatural interface to a computer. Some systems (like the Newton [20]) add word recognition on top of character recognition. Unfortunately, the software is not highly accurate and must deal with the fact that many words that people write are not in the system's dictionary. Notice also that these systems are highly dependent on the user's language and alphabet.

Moreover, even if handwriting recognition became highly accurate, it is clear that it provides a mapping from a highly expressive medium such as ink to a constrained medium, such as ASCII strings. There are

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
SIGMOD ' 95, San Jose , CA USA
© 1995 ACM 0-89791-731-6/95/0005..$3.50

no ASCII parallels to diagrams, drawings and many special symbols that the user can write.

In [13, 14], the authors introduce the notions of *pictograms* and *approximate ink matching* (AIM) as an alternative to handwriting recognition. Pictograms are simply handwritten pictures, while AIM is the technique that evaluates how well two pictograms match. By using pattern recognition techniques, AIM can take an input pictogram and evaluate how well it matches each one of the previously stored pictograms. Obviously, AIM eliminates the problems of expressiveness, language and alphabet dependency and inaccuracy of handwriting recognition. The procedure simply focuses in finding a pictogram that resembles the input, without trying to "understand" or translate its meaning. AIM algorithms with high matching accuracy have been developed.

The problem with the AIM technique is that it could be computationally expensive. Without any other tool, we are forced to do sequential searching on the entire pictogram repository. As the size of the pictogram repository grows, this process becomes painfully slow and impractical.

For instance, in [14] Lopresti and Tomkins report an experiment that uses Hidden Markov Models (HMM) [16] to represent pictograms. Following the AIM approach, each pictogram in the database is modeled by an HMM. The HMM is constructed so it accepts the specific pictogram with high probability (relative to other pictograms in the database). Given a pictogram, an HMM can be executed against it, rendering an output probability (an estimation of how likely it is that this HMM "models" the pictogram). In order to match a give input pictogram, each HMM in the database is executed against it. The system selects the one that generates the output with the highest probability. They report that this approach is extremely slow. A database of 100 pictograms takes 16 seconds to be searched on a 68040-based NeXT.

The key to make these ideas scale is to build indices that allow the system to prune out some of the pictograms and select a subset of them that are likely candidates for a good matching with the input pictogram. In this way, we can make AIM scale to repositories that contain large number of pictograms. In this paper, we describe the implementation of one such index, which we term the *handwritten trie*. This data structure is based on the trie structure ([8]), but incorporates HMMs actively in each level of the tree. The handwritten trie provides approximate matching by using the HMMs and a new search algorithm to decide how the branching is done.

We envision two ways in which indexing techniques can benefit PDAs. First, even though data repositories in PDAs are likely to remain small due to the space restrictions dictated by the PDA size, sequential searches can be prohibitively expensive in terms of energy utilization. Indices will speed up the process thereby requiring less energy consumption. Secondly, as the need to access larger repositories arises, PDAs will be used as input/output devices that send queries to servers and receive the answers. Again, the natural way to express the queries will be handwriting. The query will consist of a set of pen strokes (a pictogram) that will be sent to the server to be matched against a large pictogram repository. Of course, many other pen-based applications that do not involve PDAs can benefit from our techniques. With these applications in mind, we study the performance of a main-memory handwritten trie (for data repositories in PDAs) and that of a disk-based handwritten trie (for large server repositories).

This paper is organized as follows. In Section 2 we study the representation of ink and the matching algorithms. In Section 3 we present the description of the handwritten trie and the search algorithm, and describe briefly the main-memory and disk-based implementations of the trie. In Section 4 we present experimental results for both implementations of the trie. Finally Section 5 summarizes the work and shows future avenues of research.

2 Ink representation

Given a pictogram, whether it is an item to be inserted into the database, or a query that is used for retrieval from the database, we need to transform it to a more robust form before passing it to the database system for further processing. (The original representation of the pictogram, which comes in terms of a series of x, y coordinates is too vulnerable to slight changes. This makes it innapropiate for matching.)

Traditional databases use an alphanumeric representation of the data. This representation is ideally unique, precise and stable. Ink data lacks these qualities, making its matching a difficult problem. The data is often corrupted with noise. Even ideal ink does not provide an adequate basis for sequence identification,because we want to be able to identify a pictogram given slightly different variations in its shape. You cannot find two people that write the same word in the same way. Even for the same person, it is very difficult to generate *exactly* the same pictogram twice (it will almost always be the case that the stroke information varies each time the person writes the same word).

The first issue that needs to be settled is the selection of the granularity that is to be represented in the database. In Figure 1, we show a pictogram and its segmentation into pen strokes and handwritten symbols. We can choose to represent pictograms by any of the three granules presented in the figure, i.e., as one

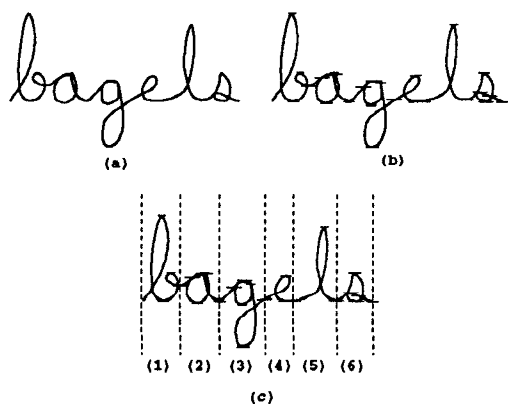

Figure 1: Example illustrating the segmentation of the pictogram in (a) into (b) strokes, (c) alphabet symbols.

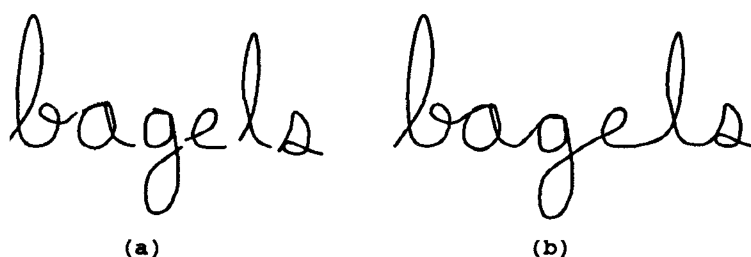

Figure 2: An example of (a) a handprinted word, (b) a cursive handwritten word.

entity containing the entire pictogram (Figure 1a), as a sequence of pen strokes (Figure 1b), or as a sequence of alphabet symbols (Figure 1c). Of course, in order to select, for instance, the symbols as granules, we have to have a segmentation algorithm that properly separates the symbols. For instance, for strokes, a simple segmentation algorithm picks local minimum (or maximum) points and uses them to segment the curve. Segmentation could be a difficult task for some types of pictograms, such as cursive handwritten words (see Figure 2b), or a simpler task as in handprinted words (see Figure 2a). Some languages, like Japanese, lend themselves easily to symbol segmentation. In Japanese, Kanji symbols are already separated by blank spaces. The choice of granularity has an impact on the type of indices we build.

The second issue is that, for matching purposes, it is better to talk about pictogram (symbols, strokes) classes instead of individual pictograms. A pictogram class is the set of pictograms that have the same semantics, according to the user. Figure 3 shows a (non-exhaustive) list of pictograms that represent the concept "gold." Of course, it would be impractical to store a pictogram class by storing the list of pictograms that belong to it. Alternatively, we have to choose a *representative* of the class and a *distance metric.*

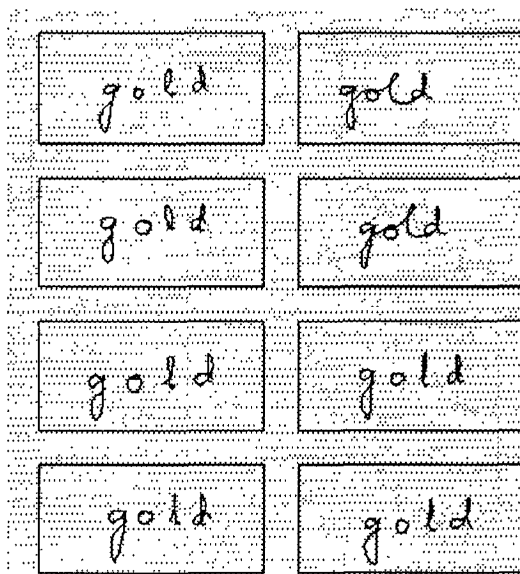

Figure 3: A list of pictograms, all representing the concept "gold".

Inputs are matched against the representative (after the necessary preprocessing) and the distance metric is used to rank the matches. The representative is not a pictogram, but rather a model that captures the essential qualities of the class.

To build the representative model one needs to use *features* of the pictograms that belong to the class. Features can be of two types.

- Local Features: One way of representing ink is to pick some sample points from the pictogram (or symbol, or stroke) and compute some local features. By local, we mean that the computed features depend on a sample point and possibly the one (or two) surrounding points from each side. It is important to mention that, depending on the application, some of these features may be more relevant than others, and hence not all of them need be computed at a given point in the sequence.

  Common features are: direction, velocity, change in direction, change in velocity, accumulative angle (with respect to the initial point), accumulative length and angle of bounding box diagonal (also with respect to the initial point), and accumulative sequence length.

  The input format of our pictogram is an array s of P time-stamped sample points:

  $$s_p = (x_p, y_p, t_p),\ 0 \le p < P \qquad (1)$$

  More feature definitions as well as ways of computing them can be found in [18].

After computing the local features, the pictogram can be represented by a sequence of feature vectors $(v_1, t_1), (v_2, t_2), \ldots$ The dimensionality of the v_i corresponds to the number of local features at each point.

- Global Features: Global features are characteristics of the entire pictogram. Among them are: the bounding box coordinates, the total angle traversed by the pictogram (measured by the angle from the beginning point to the endpoint), and length of the bounding box diagonal.

  After computing global features, the pictogram can be represented by a vector of global feature values.

Having collected a set of features for the pictograms (symbols or strokes), we can proceed to model the class. We show here how to build two representative models. The choice of model also has an impact on the index technique that we use.

2.1 Hidden Markov Models

HMMs are already used in the field of speech and handwritten recognition as a powerful tool for speech and handwritten document matching. (e.g., see [14, 13, 9, 15, 17, 19]). Each pictogram in the database can be modeled by an HMM. The HMM is constructed so that it accepts the specific pictogram with high probability (relative to the other pictograms in the database). In order to recognize a given input pictogram, we execute each HMM in the database and select the one that generates the input sequence with the highest probability. Since each HMM in the underlying sequence database has to be tested, this results in a linear process where the speed of execution is the primary difficulty.

An HMM is a doubly stochastic process, where there is a probability distribution that governs the transitions between states and an output probability distribution that identifies the distribution of output symbols for each state. An excellent coverage of HMMs can be found in [17].

We use one type of HMM structures, termed *left-to-right HMM* [4] (e.g., see Figure 4b). The left-to-right type of HMMs is useful for modeling temporal signals as in sound (e.g., see [17]) and cursive handwritten text (e.g., see [14]) because the underlying state sequence associated with the model has the property that, as time increases, the state index increases (or stays the same) — that is, the system states proceed from left to right.

A left-to-right HMM can be constructed to model a handwritten word or an alphabet symbol. The HMM is constructed so that it accepts the word (or the symbol) with high probability (relative to the other words in the database). [1] The probability given by the HMM is the distance metric used for ranking purposes.

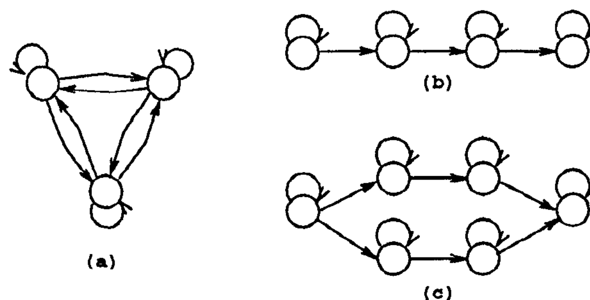

Figure 4: Several examples of HMMs: (a) the ergodic model with three states, (b) a left-to-right model with four states, and (c) a parallel path left-to-right model with six states.

Given an HMM that models a word (or symbol), we can run an input symbol against it and obtain as an output a matching probability. Given a set of stored words (or symbols), one can match an input word (or symbol) by running each one of the corresponding HMMs against the input and choosing those with the best matching probability. In fact, we can keep the size of the answer set as a parameter and choose the k best matches.

2.2 Indexing Ink

As we shall see shortly, sequential searching does not scale well. The process becomes unnaceptably slow as the number of items stored grows. Therefore, indexing techniques that help pruning the choices are extremely helpful. However, indexing techniques for handwritten pictograms must exhibit two characteristics that makes them different from traditional indexing techniques:

- The structures must incorporate the underlying model that is chosen to represent ink. The model must play an active part of the index.

- Due to the high variability of the ink data, the indices must provide approximate matching.

This paper describes one such technique, termed by us the handwritten trie, which exhibit both of the characteristics stated above.

As we mentioned before, the choice of granularity and representative model dictates the type of index that we build. For instance, we can choose to model

[1] We may be able to apply training techniques in the case where multiple instances of the word are available (e.g., a word can be handwritten multiple times, resulting in more than one sample). These samples can be used to train the HMM so that it can match similar words with higher probabilities. The interested reader is referred to [3, 12] for a more detailed discussion.

entire pictograms with HMMs and build indices that use the HMM characteristics to guide the search (we call such an index the HMM-tree [2]). Alternatively, we can choose to deal with alphabet symbols for granularity and represent the symbol classes by using HMMs. The handwritten trie, which will be described in the next section, uses the second approach.

3 The handwritten trie

The trie structure [8] is an M-ary tree, whose nodes have M entries each, and each entry corresponds to a digit or a character of the alphabet. Searching for a word in the trie is simple. We start at the root and look up the first letter in the word, and we follow the pointer next to the letter and look up the second letter in the word in the same way (see [10] for a detailed description).

The handwritten trie is an extension of the trie data structure. It differs from it, however in two important ways:

- Each letter in the trie is handwritten and modeled by an HMM.
- The search algorithm (which we will explain in detail shortly) starts at the root and descends the tree using the ranking provided by the HMMs found at every level of the tree.

These two characteristics follow the guidelines stated in Section 2 for ink indices.

More concretely, we are given as input to the index, a handwritten cursive word w that is composed of a sequence of letters $a_1 a_2 \cdots a_n$, where n is the number of characters in w. In order to search for a handwritten word in the trie we need to match the letters of w with the letters in the trie. We start at the root and descend the tree so that the path that we follow depends on the ranking of matches between the letter a_i of w and the letters at level i of the trie. Each handwritten letter in our alphabet is modeled by an HMM. The HMM is constructed so that it accepts the specific letter with high probability (relative to the other letters in the alphabet). In order to match a given input letter, we can execute each of our alphabet HMMs and rank them according to their output probabilities. (As we shall see shortly, it is not necessary to run all the alphabet HMMs at every level.) Due to the difficulty in matching perfectly the individual letters in w with the letters in the trie (since it is nearly impossible to handwrite a word twice in exactly the same way), it would be erroneous to just follow the path indicated by the best match. A more elaborate method to traverse the trie is needed (and explained shortly). An example handwritten trie is given in Figure 5.

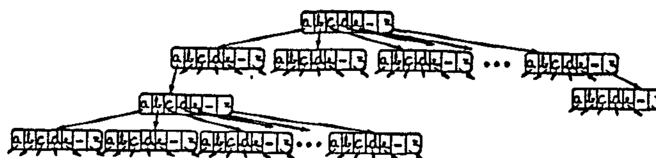

Figure 5: An example of a handwritten trie.

In order to reduce the memory space of the trie structure, we do not store all the M symbols in each node of the trie. We only store the letter combinations that correspond to words that exist in our database. This technique was also described in [7] based on the observation that the full trie tends to be sparse. This amounts to replacing the trie by a forest of trees.

Several advantages emerge from using a trie.

1. Using the trie serves as a way of pruning the search space since the search is limited only to those branches that exist in the trie.
2. Using the trie also helps add some semantic knowledge of the possible words to look for.

In order for the handwritten trie to function properly, the segmentation issue has to be addressed. If the input handwritten word is cursive (see Figure 2b), characters in the word has to be segmented so that each character can be used to match the corresponding character in one of the trie nodes. The extra strokes that are used to connect the letters in cursive writing have to be treated in such a way that they do not interfere with the matching process. If the input is handprinted (see Figure 2a), segmenting the pictogram is a simpler task. For instance, we can measure the x coordinate of the first point of each stroke. Then, by computing the difference between coordinates of consecutive strokes and comparing it to a threshold, we can determine that a new character has begun.

In this paper, we will not address these issues any further and will assume that the words are segmented into alphabet symbols.

3.1 The search algorithm

In principle, to traverse the trie using an input word, we should use each symbol in the word to decide what is the next node we need to visit. However, since at each level the decision involves running HMMs that represent symbols and analyzing the probabilities that they output, the search algorithm is far from straightforward. If the algorithm just takes the best probability at each level and follows the corresponding path, it is possible to eliminate words that may be very close to the input (and in fact, they might be globally,

the best matches). On the other extreme, traversing *all* the paths would certainly guarantee that all words are considered and that the best global match is selected. However, this would be as bad as sequential searching. In this section, we present an algorithm for searching the trie, termed *the limited-budget search algorithm.* The algorithm *approximates* the answer while having good performance. This approximation proves to be extremely good in practice, as our results will show in the next section.

Assume that we are given an input handwritten word w that is segmented into a sequence of alphabet symbols $a_1, a_2, \cdots, a_n$ (e.g., see Figure 2 where the word "bagels" is segmented into 6 alphabet symbols, i.e., $n = 6$). Assume further that we have an HMM for each symbol in the alphabet set A. Let the number of symbols in A be N_A. Therefore, we have N_A HMMs $H_1, H_2, \cdots, H_{N_A}$. We are also given a running-time budget which is expressed in terms of the maximum number of HMMs that we can execute during the search.

Instead of executing all the HMMs $H_1, H_2, \cdots, H_{N_A}$ for each letter of the input word, we limit ourselves to running the HMMs that correspond to letters that exist in the trie. The algorithm maintains the following data structures: the array *HMM_tried*$[1 \cdots n,\ 1 \cdots N_A]$, the integer variable *HMM_Budget*, and the list *best_matches.*

For each letter a_i in the input word,

$HMM_tried[i, 1 \cdots N_A]$

contains the HMMs that have been tried or executed (zero means not *executed*, and a non-zero value means *executed*) with a_i as input, and if executed, a non-zero value is stored which indicates the probability of accepting a_i by the corresponding HMM. For example, $HMM_tried[1, \text{“}c''] = 0.3$ means that the HMM corresponding to the letter "c" accepts the first letter of the input word (i.e., a_1) with probability 0.3, and $HMM_tried[1, \text{“}d''] = 0$ means that the HMM corresponding to the letter "d" was not executed with a_1 as input.

Each time the entire input word is matched by a word in the trie, the matched word as well as its matching probability are stored in the sorted list *best_matches.* The size or number of words that can be kept in *best_matches* at a given time can be limited by a constant, say k. By the end of the searching process, the k words in *best_matches* can be reported as the best k matches that are found so far.

The integer variable *HMM_Budget* is initialized by the maximum number of HMM executions that are permitted during the search. Each time a node of the trie is visited and some HMMs are executed, *HMM_Budget* gets decreased by the number of HMMs that are executed. The algorithm is forced to a halt the searching process once *HMM_Budget* reaches zero. In this case, the words that are stored in *best_matches* are reported as the final answer to the search.

The search starts at the root r of the trie, where all the HMMs that correspond to the letters in r are executed with the letter a_1 as input. The resulting probabilities of accepting a_1 are stored in array $HMM_tried[1, 1 \cdots N_A]$, and are sorted based on the probability with which they accept a_1. We store the best l HMMs (i.e., the ones that accept a_1 with highest probabilities) in a stack, and visit the best l children of r in the order of their acceptance probability. Notice that when there are less than l entries in r, we store all of the children in the stack. For example, the child of r that corresponds to the highest matching probability is visited first. The above procedure is repeated by matching the second letter of the input word, i.e., a_2 with the HMMs that correspond to the alphabet symbols that are stored in the visited child node. *HMM_Budget* is decremented each time an HMM is executed. If a leaf node is reached without having all the symbols in the input word being processed, this implies that the input word has more symbols than the word stored at this leaf node and hence we exclude this word.

Analogously, if all the symbols of the input word are processed and we have not reached a leaf node, then this implies that the input word has fewer symbols than the words stored along this path of the trie, and hence we exclude the path as well. When a leaf node of the trie is reached and at the same time all the symbols of the input word are processed, the word stored at the leaf node is further tested in order to decide if it is among the best k matches found so far. This is achieved by comparing the word's matching probability with the matching probabilities of the words currently stored in *best_matches.* If the new word has a matching probability that is greater than any of the words already existing in *best_matches* (or if the number of words in *best_matches* is less than k), then the word is added into *best_matches* along with its matching probability. This may result in excluding the word in *best_matches* that have the lowest matching probability. This happens when the number of words in *best_matches* is equal to k words. The matching probability of a word is computed as the product of all the probability values of the HMMs that lie in the path from the root of the trie to the leaf node. As long as *HMM_Budget* still permits for more searches to take place, alternative paths are visited by considering the second, through lth best HMM matches at each node in the trie path and then traversing alternative routes in the trie. In this case, the array *HMM_tried* is consulted before executing any HMMs since there is a chance that a particular HMM has been executed before given the same input symbol. In this regard, *HMM_tried* helps avoid the

redundant execution of the HMMs. A more detailed listing of the algorithm is given below.

1. let input word w_{in} be $a_1, \cdots, a_n$, and maximum allowable budget be *Max_Budget*.

2. start at the root r of the trie

3. store r into a STACK.

4. initialize budget:
   $budget \leftarrow Max_Budget$

5. Loop until budget is exhausted or no more nodes in STACK
   while $budget \geq 0$ and STACK not empty do

   (a) node $t \leftarrow$ top element of STACK

   (b) if t has no children (i.e., is a leaf node), then

      i. $w_t \leftarrow$ word associated with leaf node t

      ii. if input word still has more unmatched characters (i.e., $n > depth(t)$ or even $length(w_t) \neq length(w_{in})$) then discard w_t

      iii. otherwise, compute $P(w_t)$, the entire probability of matching w_t. This is the product of all the probabilities of the HMM letter matchings in the path from r to t.

      iv. if number of words in *best_matches* $< k$ or if $P(w_t)$ is higher than any of the words in *best_matches*, insert w_t into *best_matches*. This may result in excluding the word in *best_matches* that have the lowest matching probability.

   (c) expand node t (t is a non-leaf node):
   if all symbols of w_{in} are consumed, then skip t, else

      i. $a_i \leftarrow$ next symbol in w_{in}

      ii. avoid executing the same HMMs again for a_i: execute all HMMs that correspond to the alphabet symbols of children in t that are not in $HMM_tried[i, 1 \cdots N_A]$

      iii. decrease budget:
      $budget \leftarrow budget - number\ of\ executed\ HMMs$

      iv. store probabilities of accepting a_i in array $HMM_tried[i, 1 \cdots N_A]$

      v. store the l-best HMMs in STACK:
      sort the probabilities with which the HMMs accept a_i and store the best l HMMs (i.e., the ones that accept a_i with highest probabilities) in *STACK*.

3.2 Implementation issues

The hardware of our system is composed of the following components: a NeXT 68040 with 32 Megs of main-memory and a pen-based tablet. The software is composed of a graphical user interface to handle both the pen-based tablet and graphical displays, a home-coded package for handling hidden Markov Models (constructing, training, and matching software), buffering software, and two implementations of the trie: the main-memory and the disk-based versions.

We omit the details of the implementations here for lack of space. They can be found in [1].

The storage requirements for the main memory implementation are $13N$ where N is the number of nodes in the trie. Each node is structured as having four fields: the alphabet symbol that corresponds to the node, a pointer to its parent, a pointer to its lefmost child, and a pointer to its sibling.

The node structure for the disk-based implementation is a bit more elaborated. Since we pack nodes in pages, we have to worry about a node shifting inside a page and invalidating the pointers from the parents to the shifted nodes. To avoid this, we partition the node structure into two parts. A fixed-size portion contains the alphabet symbol of the node, a pointer to its parent, the number of children, and an offset to the location of the variable-size portion. The variable-size portion contains the alphabet symbol of each child, and a pointer to each of the children. Both parts of the node are stored in the same page so as to avoid extra I/O. However, the fixed-size portion of all the nodes in a given page are stored as a heap at the beginning of the page and grow forward, while the variable-size portions of all the nodes in the same page are stored as a heap at the end of the page and grow backwards. The total size of the trie can be computed as being $17N$.

4 Experiments and Performance Results

We have implemented both main memory and disk-based versions of our trie. In this section, we report the performance and space measurements for these implementations. These measurements are taken in a 40MHz NeXT Workstation.

4.1 Main memory implementation

We built a main memory version of the trie for an alphabet size of 26 symbols. Figure 6 shows how the trie grows with the number of words in the database.

Table 1 shows the total space requirements for the first database size (17,903 items) in Figure 6. (The number of items was selected to fit in exactly one megabyte of memory.)

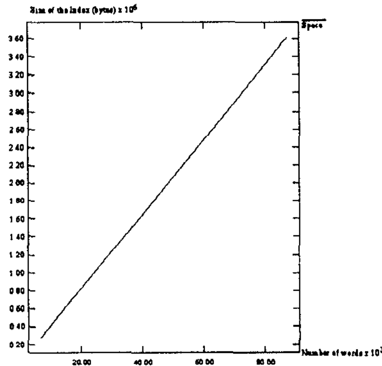

Figure 6: The space requirements of the main-memory implementation of the handwritten trie with the increase in the number of words in the database.

code size	150,000
number of words	17,903
size of a symbol pictogram	100
index size	375,963
size of alphabet pictograms	2,600
size of one HMM	20,000
total size of HMMs	520,000
total space requirements	1,048,563

Table 1: Space requirements for the main-memory implementation (using 1 megabyte as a base).

The size of a symbol pictogram row refers to the space taken to store a pictogram that represents a symbol of the alphabet. We choose to store a representative of each symbol to avoid storing handwritten words altogether. In that way, after the search, we can show the user the word(s) found by the search by putting together the pictograms of the symbols encountered in traversing the trie. Doing this saves a significant amount of space. (This is an additional benefit of the trie structure). The size of HMM refers to the number of bytes needed to store the HMM model for each of the alphabet symbols. The total space requirements for almost 18,000 words (including the executable modules of the code) is about 1 Mbyte. Thus, our technique is extremely practical from a space utilization standpoint.

In order to measure the matching rate of our search algorithm we conducted the following experiment. First, we chose 80 words at random from the dictionary of words that are stored in the trie. Then, we handwrited the words, and considered them as our query words to drive the search algorithm. For each handwritten query word, we found the best matching word in the trie using the search algorithm given in Section 3.1. Finally, we count the number of properly matched query words over

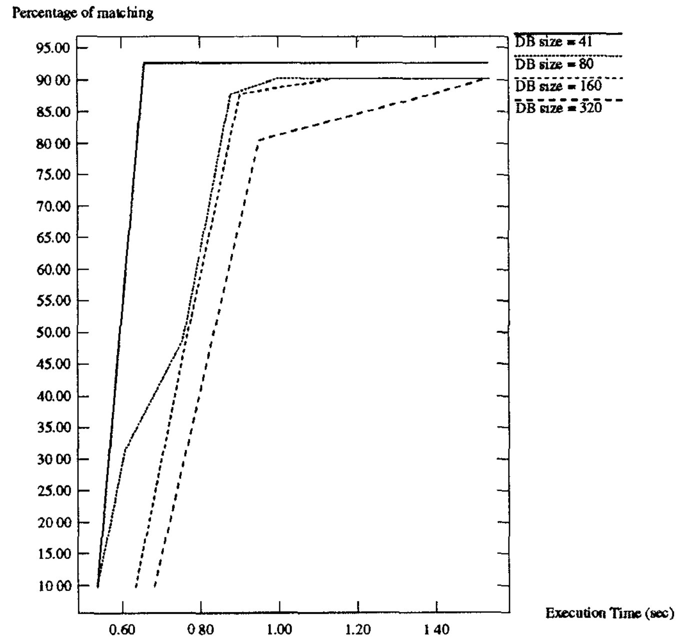

Figure 7: The matching rate for various handwritten database sizes. The x-axis corresponds to a tuning parameter of the search algorithm (main-memory implementation).

the set of 80 query words. The matching rate was measured as the percentage of queries for which the query word is matched with the correct word in the database, i.e., when the search procedure ranks the correct word that matches the input word at the top of the best-matches word list (i.e., the 1-best selection). Notice that, since in this database there is only **one** "correct" answer to each query, this definition of matching corresponds to the more traditional definition of *recall*. We repeated this experiment for various sizes of the underlying handwritten database. Figure 7 gives the result of our experiment. The figure gives the matching rate (y-axis) for different sizes of the underlying handwritten database. The x-axis corresponds to the time spent in the search. The Figure shows that the overall matching rate is around 90%. This value can always be reached for the various sizes of handwritten database by increasing the budget (and therefore the execution time), as shown in the figure.

Figure 8 compares the matching time when using our indexing technique versus using a sequential matching algorithm. As expected, the search time of the sequential matching algorithm grows linearly with the size of the database. On the other hand, the search time of our indexing technique tends to grow logarithmically (slow growth) in the size of the database. The search

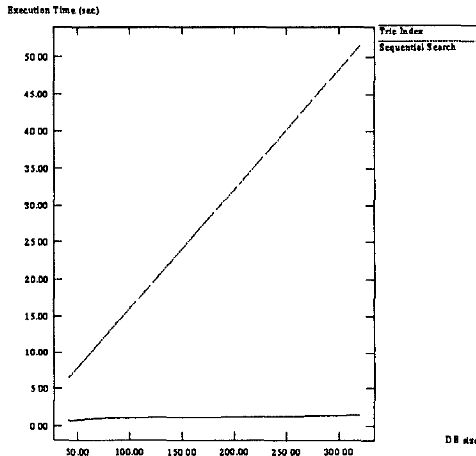

Figure 8: A comparison between the matching time using our indexing technique (main-memory implementation) versus using a sequential algorithm. The x-axis corresponds to various database sizes.

time using the index is around one second for 150 items.

4.2 Disk-based implementation

For the disk-based implementation, we built our database of words from an online dictionary that contains over 150,000 words. In order to avoid handwriting all the words, we insert the ASCII version of the words into the trie. This also guarantees that words are inserted properly, i.e., without any errors at insertion time. Notice, however, that we are not matching handwritten text with ASCII, but with the corresponding HMMs for each character. We still entered handwritten pictograms for each alphabet symbol, trained the corresponding HMMs and made the connection between ASCII characters and the HMMs. In other words, we use as symbol names the 26 ASCII characters that represent lower case roman characters.

The input query words are all handwritten. We categorize the input words into the following four groups, based on the length of the handwritten words: long (more than seven characters), medium (between four and seven characters), short (less than four characters), and mixed (a mix of long, medium, and short words).

We implemented the disk-based handwritten trie and the search algorithm as described in sections ?? and 3 respectively. We also implemented a page buffer management system. It uses a least-recently used (LRU) replacement policy. Unless mentioned otherwise, in the following experiments we pre-allocate a buffer pool of 1000 pages, where each page is of size 1K bytes, i.e., a total of 1 megabyte of buffer space and use no clustering (nodes are allocated by order of insertion).

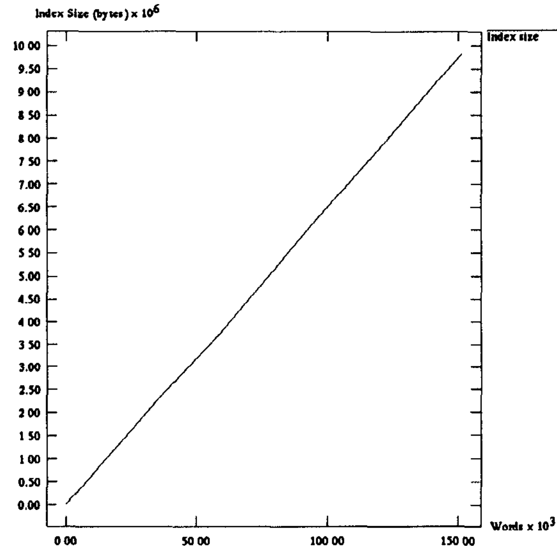

Figure 9: The space requirements of the handwritten trie with the increase in the number of words in the database (disk-based implementation).

In addition, we vary the page size as well as the number of pages in the buffer and study their effect on the search time performance.

Figure 9 illustrates how the disk-based trie grows with the increase in the number of words in the database. The size of the trie is expressed in terms of the total number of bytes.

In order to measure the matching rate of our search algorithm we perform the same experiment described in Section 4.1. We maintain a buffer pool of size 1 megabytes, as explained earlier in this section. Figure 10 shows the matching rate for a query set of mixed lengths when the budget is expressed in terms of the total search time allowed for the limited-budget search algorithm. The x-axis reflects the search time (in seconds). The y-axis shows the matching rate.

From the figure, we observe that as the number of words in the database increase, the matching rate decreases. This is because it is harder to rank the correct word from among the database words to appear at the top of the best-matches list as the database size increases.

During our experiments, we observed that the number of letters of the query word (i.e., its length) has an effect on the matching rate of our search algorithm. In order to test this hypothesis, we constructed four sets of handwritten query words: short, medium, long, and mixed, where each set contains a group of handwritten words of a given range of word lengths (*short* means words with length ≤ 3, *medium* means words with $3 <$

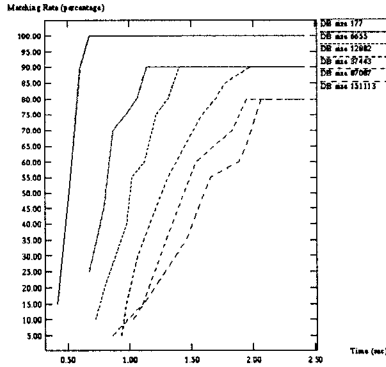

Figure 10: The matching rate vs. search time using the limited-budget search algorithm

length ≤ 7, *long* means words with length > 7. The size of each data set is 80 words. Table 2 shows the effect of word size in the matching rate. The entries on the table are pairs (b, r), where b is the HMM budget value for which the matching rate peaks at a value r (i.e., for budgets bigger than b, the matching rate remains at r). As we can see, long words reach higher points (100%) than short or medium words. (In general, the matching rate of long words is higher than that of the shorter words.) This is because for longer words we have more information that enables us to narrow the search significantly. Also, since the trie is much denser at the levels close to the root, long words tend to use the characters towards the end of the word to produce a better matching. On the other hand, short words are more likely to be mismatched because of the high density of the top levels of the trie. Nevertheless, on the average, we are able to achieve around 80% matching rate for the mixed set of query words in the case of the 150,000 words database, which is very promising. For each database size, the matching rate peaks and does not improve any further, even with the availability of more budget. This is because for the given query word, the subtrees in the trie that are likely to contain the word are already explored, and hence additional budget will not help.

Figure 11 shows the results when we change our ranking method to report the k-best items instead of just reporting the best match. This is practical since more than a relatively small number of answers (in this case k answers) can easily be reported to the user in a small display device. (Or if this is not possible, the k items can be matched against the input sequentially to select which one of them is the best match. We could use for this comparison HMMs or other matching techniques.) From the figure, we can see that by considering the k-best matches, the matching rate enhances significantly.

Database Size	Short Words		Medium Words		Long Words	
	b	r	b	r	b	r
6,655	25	100	70	100	60	100
12,882	50	95	90	95	80	100
37,443	70	75	110	80	125	100
87,087	50	65	110	80	150	100
151,113	50	65	110	75	150	100

Table 2: The matching rate saturation points for different word and database sizes.

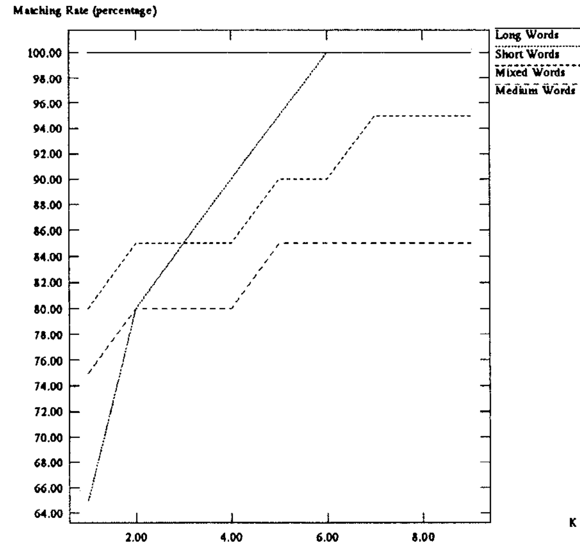

Figure 11: The matching rate of k-best matching with varying k for different input word lengths.

Figure 12 shows the search time as the database size increases for query sets with various word size combinations. For this experiment, we pre-allocate a buffer pool of 1000 pages, where each page is of size 1K bytes, i.e., a total of 1 megabyte of buffer space (the effect of changing the buffer pool size is studied below). In each case, the budget was adjusted (using information from previous experiments) so that the maximum possible matching rate is achieved (these matching rates are listed in the map legend of Figure 12). Notice that the response time is very reasonable for an interactive online query environment. Notice that for longer words, the search algorithm takes more time to perform the matching because there are more letters to match in this case. Notice also that the search time increases almost logarithmically (and not linearly) as the size of database increases. This demonstrates the usefulness of the index.

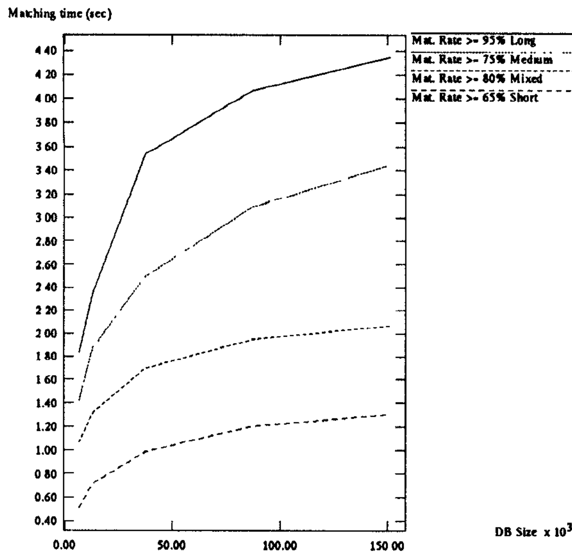

Figure 12: The search time vs. database size using the limited budget search algorithm for various query sets.

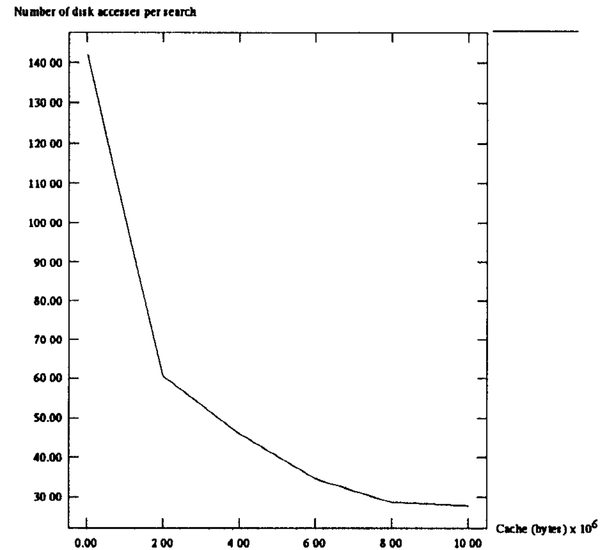

Figure 13: The average number of disk accesses required during the trie search while varying the number of pages in the buffer pool.

Figure 13 shows the effect on the number of disk reads when varying the number of buffer pages from 1 to 10 megabytes for a database size of 150,000 items. Notice that the total size of the trie index is 9.8 megabytes (Figure 9). In order to conduct these experiments, we warm-up the buffer pool by issuing many queries to reduce the transient effect. Notice that the number of disk reads reduces as the number of buffer pages increases since more pages that are common among successive queries can now be kept in the additional buffer space. Also, notice that the number of disk reads tends to level as the size of the buffer pool exceeds 7 megabytes since the portion of the trie that corresponds to the words of the input queries is loaded entirely into the buffer pool and hence any additional buffer pages that are allocated are not utilized and hence they do not enhance the performance. The behavior of the average search time as the number of pages in the buffer increases was found to be similar to that of the disk reads [1].

5 Concluding Remarks

We have studied in this paper the implementation of the handwritten trie, a tool to index handwritten words. The results show conclusively that the usage of the index improves the search times considerably. In both, the main-memory and disk-based implementation of the trie, the search times are very reasonable for an interactive online query environment. The matching rates obtained are also very good and can be tuned up by showing the k best matches instead of only the best match. Space utilization is specially important for main-memory databases: main-memory implementations will be the likely choice for PDAs, and these machines do not usually have large main memories (main memory in current PDAs range from half megabyte to two megabytes). Our technique allows the implementation of an index for almost 18,000 words in 1 megabyte of main memory. The space utilization of the disk-based implementation is also very reasonable (around a 65 bytes of index per stored word). We need a total of 9.8 megabytes of index space for a database of over 150,000 items. Although this amount of space could even be found in the main memory of some contemporary workstations, it certainly would have to be allocated to disk in labtops and notebooks, where pen-based technology is most likely to find its place. We will experiment in the future with larger databases.

We can further reduce the storage overhead per node by packing the subtrees that are linear chain of nodes containing just one word at the leaf. The packing is performed by clustering together all the alphabet symbols that appear in the nodes of the chain. This idea was also used by Knuth [10]. Experiments show that we achieve a reduction of space by a factor of 4.5. The index requirements for the disk-based implementation for a database of 150,000 items decrease from 9.8 megabytes to 2.1 megabytes. This packing would also reduce the number of disk operations, thereby reducing the search time. In the case of the main-memory, the database index for 17,903 items would occupy around 80,000

bytes instead of 375,000. (That means we can place an index for many more items in the same amount of memory.)

In [1], we also report the effect that different clustering techniques have on the performance of the trie. We have found that with breadth-first mapping (i.e., performing a breadth-first traversal of the trie, and clustering the nodes into disk pages as they are traversed), we can achieve improvements of around 40% in the number of I/O's performed during the search.

In addition to the handwritten trie, we are currently investigating other ways of indexing handwritten text [5, 2]. Future research includes performance and comparative studies among the various types of handwriting indexes.

References

[1] W. Aref, D. Barbará, and P. Vallabhaneni. The Handwritten Trie: Indexing Electronic Ink. Technical report, M.I.T.L, October 1994.

[2] Walid Aref and Daniel Barbará. The Hidden Markov Model Tree Index: A Practical Approach to Fast Recognition of Handwritten Documents in Large Databases. Technical Report MITL-TR-84-93, MITL, January 1994.

[3] Walid G. Aref, Padmavathi Vallabhaneni, and Daniel Barbará. Towards a realization of handwritten databases: Training and recognition. Technical Report MITL-TR-98-94, Matsushita Information Technology Laboratory, Princeton, NJ, March 1994.

[4] R. Bakis. Continuous speech word recognition via centisecond acoustic states. In *Proc. ASA Meeting*, Washington, DC, April 1976.

[5] Daniel Barbará. Method to index electronic handwritten documents. Technical Report MITL-TR-77-93, Matsushita Information Technology Laboratory, Princeton, NJ, November 1993.

[6] R. Carr and D. Shafer. *The Power of PenPoint.* Addison-Wesley, 1991.

[7] Rene de la Briandais. File searching using variable length keys. In *Proceedings of the Western Joint Computer Conference*, pages 295–298, 1959.

[8] E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–500, 1960.

[9] A. Kaltenmeier, T. Caesar, J. M. Gloger, and E. Mandler. Sophisticated topology of hidden markov models for cursive script recognition. In *Proceedings of the 2nd. International Conference on Document Analysis and Recognition*, pages 139–142, Tsukuba Science City, October 1993.

[10] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, MA, 1978.

[11] K. Landau, S. Major, and C. Wiederhold. The Role of PDA in the Office. Notes of the Seminar at PC-Expo, June 1994.

[12] S. E. Levinson, L. R. Rabiner, and M. M. Sondhi. An introduction to the application of the theory of probabilistic functions of a markov proces to automatic speech recognition. *Bell System Technical Journal*, 62(4):1035–1074, April 1983.

[13] D. P. Lopresti and A. Tomkins. Approximate Matching of Hand-Drawn Pictograms. In *Proceedings of the Third International Workshop on Frontiers in Handwriting Recognition*, May 1993.

[14] D. P. Lopresti and A. Tomkins. Pictographic Naming. In *Proceedings of INTERCHI93*, 1993.

[15] Hee-Seen Park and Seong-Whan Lee. Large-set handwritten character recognition with multiple stochastic models. In *Proceedings of the 2nd. International Conference on Document Analysis and Recognition*, pages 143–146, Tsukuba Science City, October 1993.

[16] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceeding of the IEEE*, 77(2):257–285, February 1989.

[17] Lawrence Rabiner and Biing-Hwang Juang. *Fundamentals of Speech Recognition.* Prentice Hall, Englewood Cliffs, N.J., 1993.

[18] Dean H. Rubine. *The Automatic Recognition of Gestures.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, December 1991. Also technical report number CMU-CS-91-202.

[19] Bong Kee Sin and Jim Hyuung Kim. A statistical approach with HMMs for on-line cursive Hangul (Korean script) recognition. In *Proceedings of the 2nd. International Conference on Document Analysis and Recognition*, pages 147–150, Tsukuba Science City, October 1993.

[20] W. Smith. The Newton Application Architecture. In *Proceedings of the IEEE Computer Conference, San Francisco*, 1994.