Ink as a First-Class Datatype in Multimedia Databases

Walid G. Aref, Daniel Barbará, Daniel P. Lopresti, and Andrew Tomkins

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

1. Introduction

In this chapter, we turn out attention to databases that contain ink. The methods and techniques covered in this chapter can be used to deal effectively with the NOTES database of the Medical Scenario described in the Introduction of the book. With these techniques, doctors would be able to retrieve the handwritten notes about their patients, by using the pen as an input device for their queries.

The pen is a familiar and highly precise input device that is used by two new classes of machines: full-fledged pen computers (*i.e.*, notebook- or desktop-sized units with pen input, and, in some cases, a keyboard), and smaller, more-portable personal digital assistants (PDA's). In certain domains, pen-based computers have significant advantages over traditional keyboard-based machines, including the following:

1. As notepad computers continue to shrink and battery and screen technology improves, the keyboard becomes the limiting factor for miniaturization. Using a pen instead overcomes this difficulty.
2. The pen is language-independent – equally accessible to users of Kanji, Cyrillic, or Latin alphabets.
3. A large fraction of the adult population grew up without learning how to type and have no intentions of learning; this will continue to be the case for many years to come. However, everyone is familiar with the pen.
4. Keyboards are optimized for text entry. Pens naturally support the entry of text, drawings, figures, equations, etc. – in other words, a much richer domain of possible inputs.

In Section 2. of this chapter, we consider a somewhat radical viewpoint: that the immediate recognition of handwritten data is inappropriate in many situations. Computers that maintain ink *as ink* will be able to provide many novel and useful functions. However, they must also provide new features, including the ability to search through large amounts of ink effectively and efficiently. This functionality requires a database whose elements are samples of ink.

In Sections 3. and 4., we describe pattern-matching techniques that can be used to search linearly through a sequence of ink samples. We give data concerning the accuracy and efficiency of these operations. Under certain circumstances, when the size of the database is limited, these solutions are sufficient in themselves. As the size of the database grows, however, faster methods must be used. Section 5. describes database techniques that can be applied to yield sublinear search times.

2. Ink as First-Class Data

For the most part, today's pen computers operate in a mode which might be described as "eager recognition." Using handwriting recognition (HWX) software, pen-strokes are translated into ASCII[1] as soon as they are entered; the user corrects the output of the recognizer; and processing proceeds as if the characters had been typed on a keyboard.

It can be argued, however, that pen computers should not be simply keyboard-based machines with a pen in place of the keyboard. Rather than take a very expressive medium, ink, and immediately map it into a small, pre-defined set of alphanumeric symbols, pen computers could be used to support a concept we call *Computing in the Ink Domain*, as shown in Figure 2.1. Ink is a natural representation for data on pen computers in the same way that ASCII is a natural representation for data on keyboard-based machines. An ink-based system, which defers or eliminates HWX whenever possible, has the following advantages:

1. Many of a user's day-to-day tasks can be handled entirely in the ink domain using techniques more accurate and less intrusive than HWX.
2. No existing character set captures the full range of graphical representations a human can create using a pen (*e.g.*, pictures, maps, diagrams, equations, doodles). By not constraining pen-strokes to represent "valid" symbols, a much richer input language is made available to the user.
3. If recognition should become necessary at a later time, additional context for performing the translation may be available to improve the speed and accuracy of HWX.

The second point – ink is a richer representation language – deserves further discussion. An important advantage of computing in the ink domain is the fact that people often write and draw patterns that have no obvious ASCII representation. With only a fixed character set available, the user is sometimes forced to tedious extremes to convey a point graphically. Figure 2.2 shows an Internet newsgroup posting that demonstrates this awkward mode of communication. Contrast this with Figure 2.1, which illustrates the philosophy of treating all ink patterns as meaningful semantic entities that can be processed as first-class data.

2.1 Expressiveness of Ink

Our intuition tells us that ink is more expressive than ASCII. To test this assertion, we conducted an informal survey of the notebooks of a small number of university students. We asked each to provide examples of their handwritten notes, and broke the pages into three distinct categories:

1. ASCII-representable. This includes straight text, as well as text employing simple "typesetting" conventions such as underlining, etc.
2. Special character set. This includes symbols not found in standard ASCII, but sometimes present in extended character sets, such as mathematical symbols ($\int, \sum, \prod, \ldots$), unusual typographic symbols (Å,§,$\nearrow$,$\natural$,$\overbrace{\qquad\qquad}$), etc.
3. Drawings. This includes all ink not falling into one of the first two categories.

The results of our survey, shown in Figure 2.3, suggest that an electronic "notepad" dependent on converting all input to ASCII would be limiting for these users.

[1] For concreteness, we assume HWX returns ASCII strings, but the reader may substitute whichever fixed character set is appropriate.

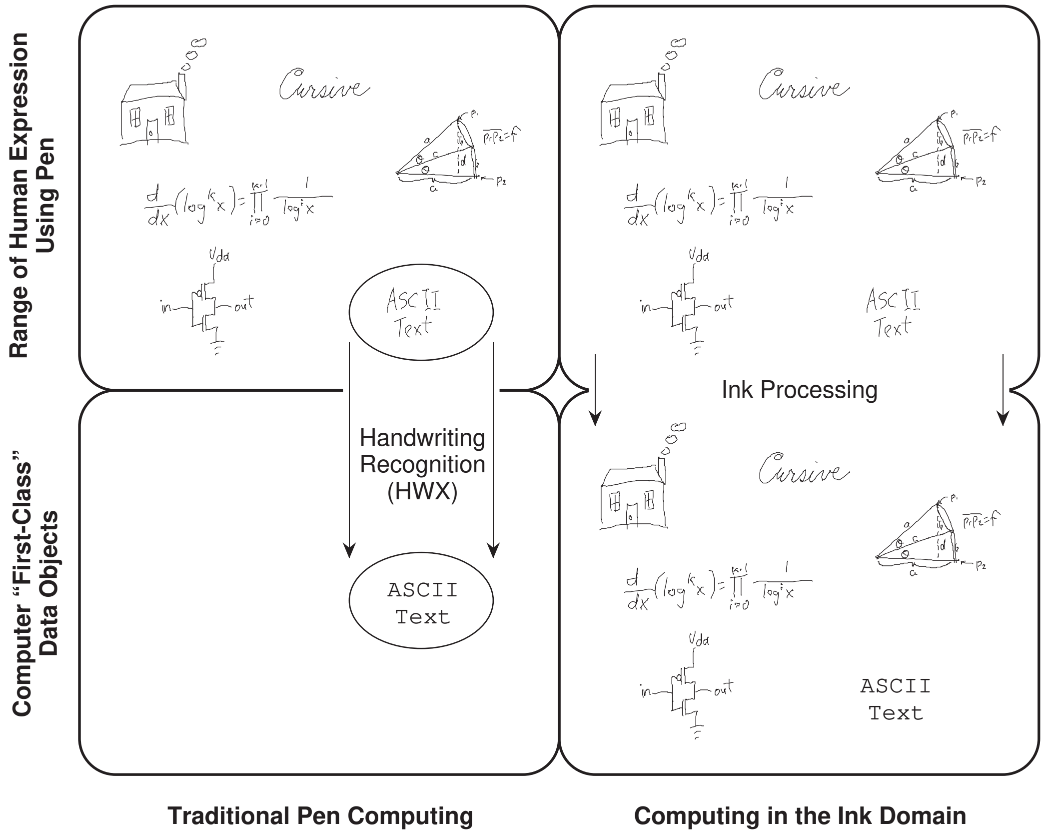

Figure 2.1. Traditional pen computing versus computing in the ink domain.

2.2 Approximate Ink Matching

Ink has the advantage of being a rich, natural representation for humans. However, ASCII text has the advantage of being a natural representation for computers; it can be stored efficiently, searched quickly, etc. If ink is to be made a "first-class datatype" for pen computers, it must be:

- **Transportable.** The ASCII character set made a specific (and somewhat arbitrary) set of 128 characters essentially universal. Standards like JOT [Sla93] are now being developed to make ink data usable across a wide variety of platforms.
- **Editable.** Years of research and development have led to text-oriented word processors that are both powerful and easy-to-use. We need similar editors for ink data. It should be as easy to edit ink (*e.g.* copy, paste, delete, insert) as it is to edit ASCII text.
- **Searchable.** Computers excel at storing and searching textual data – the same must hold for ink. In particular, it should be possible for the user to locate previously saved pen-stroke data by specifying a query and having the computer return the closest matches it can find.

While these three properties are all of fundamental importance, the last, searchability, is a primary topic of this chapter. Since no one writes the same word exactly the same way twice, we cannot depend on exact matches in the case of ink. Instead, search is performed using an *approximate ink matching* (or AIM) procedure. AIM takes two sequences of pen strokes, an *ink pattern* and an *ink database*, and returns a pointer to the location in the ink database that matches the ink pattern as closely as possible.

```
This is from memory, but the schematic below should work...

                            10uF, 63V
    --------o------------||------------------------o-------------- 2
    |       |                           |-----22K-----|
   |0      2K2      ----------------o                                X
    |       |       |                   |-----22K-----|
    -----o--|--o----o----||------------------------o-------------- 3  L
        +|  |  |+           10uF, 63V
         Z  |  C                                                     R
         |  |  |
     ----o--o--o-------------------------------------------------- 1


   Z: Zener diode (I used 15V I think)
   C: 10uF tantalum 25V, bypassed with 10nF plastic film
```

Figure 2.2. An example of ASCII "graphics."

Data Set	Number of Pages ASCII	Special	Drawings	Percent Non-ASCII
A	54	0	58	52%
B	10	9	12	68%
C	0	33	78	100%
D	14	3	18	60%

Figure 2.3. Informal survey of paper notepad users.

Such a procedure is a surprisingly general tool for ink-based computing. We now give several examples to show how AIM can be used to provide a wide range of functionality to the user:

- Andrew writes a short note to Bill. Using AIM, Bill's address is located in a database of past addresses to which Andrew has sent mail. The message itself is compressed and sent to Bill to be read as ink – full HWX of the message body is never performed. Indeed, Bill will do a far better job of reading the message than current HWX algorithms, especially if it contains cursive script, diagrams, or other non-ASCII symbols. Figure 2.4 illustrates this. Figure 2.2, on the other hand, is an example of a message that would have been more simply and effectively communicated via digital ink.
- Martha runs an application on her pen computer and names all of her documents using pen strokes. In many cases, she finds her documents by browsing through the names – HWX is not necessary. In other cases, she enters a query for which the system searches using AIM. This particular AIM problem is made simpler by the fact that the query must only be matched against the current database of filenames instead of a larger, more general database.
- Joe has an on-line discussion with Martha about a mathematical idea they have been considering. Later, he wishes to retrieve the document. He enters one of the equations he recalls from the conversation. The system searches through his pen-stroke data and finds a similar-looking sequence of strokes, returning the page in question.

Thus, AIM is central to computing in the ink domain. Of course, we can think of approximate ink matching as exactly the problem of searching a database in which the keys are pen-strokes. Thus, we expect ink to become an important new form of multimedia data.

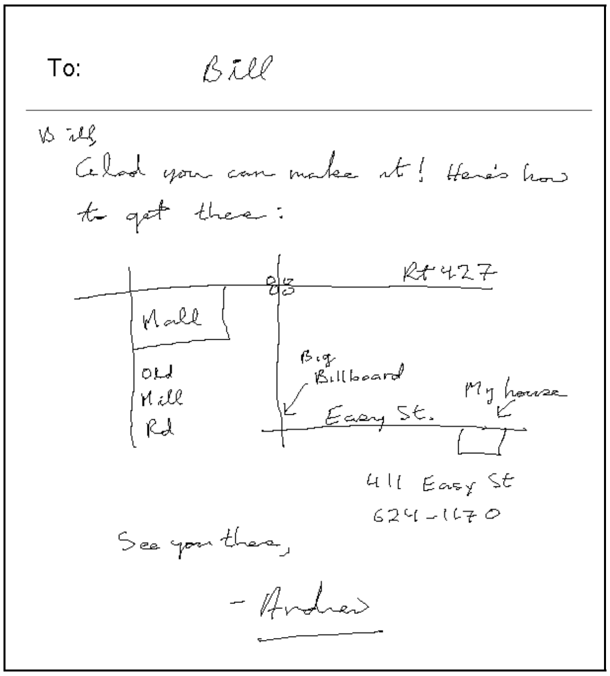

Figure 2.4. A sample ink e-mail message.

3. Pictographic Naming

We now consider an application in which AIM can be applied to provide necessary functionality, allowing a traditionally text-based operation to be performed entirely using ink. The domain is that of file names. Traditionally, a name is a short string of alphanumeric characters with the property that it can be easily stored, recognized, and remembered. However, the current approach to specifying names using a pen has received widespread criticism: the user writes the name letter-by-letter into a comb or grid and the computer performs HWX on each character. Error rates are high enough that the user must often pause to redraw an incorrectly recognized letter. Other options seem even less appealing: the user could follow a path through a menu system to specify a letter uniquely, or tap a pen on a simulated keyboard provided by the system. None of these methods feels like a natural way to specify a name, though.

Consider instead extending the space of acceptable names to include arbitrary hand-drawn pictures of a certain size, which we call *pictographic names*, or simply *pictograms* [LT93c, LT93b]. The precise semantics of the pictograms are left entirely to the user. Intuitively, the major advantages of the pictogram approach are ease of specification and a much larger name-space. A disadvantage is that people cannot be expected to re-create perfectly a previously drawn pictogram; hence, looking up a document by name requires AIM. In this section we study techniques for performing ink search in this limited domain, and present some experimental results.

3.1 Motivation

Broadly speaking, a computer user can specify the name of an existing file or document in either of two ways:

1. *Direct manipulation.* Selecting the desired name from a list of possible options in a scrollable graphical browser.

2. *Reproduction.* Duplicating the original name by retyping or redrawing it, and then letting the computer search for a match.

The motivation behind pictographic naming is the following: since users of graphical interfaces often specify files through direct manipulation rather than reproduction, HWX of a name may never be necessary and should be deferred whenever possible. A natural way to defer HWX is to leave the name as ink, in the form of a pictogram, which the user can browse later. This leaves the user free to choose complex and varied pictographic names, but forces the operating system to search for reproduced names approximately rather than exactly, a more difficult problem.

As a concrete example, suppose the user has produced the note shown in Figure 3.1. Perhaps it is part of a paper he/she is working on, a slide for a presentation, or something drawn to communicate an idea to a friend. The user wants to store the document, and to be able to access it later on. A natural idea is to write a small pictogram describing its contents – for example, Figure 3.2. When the user wants to retrieve the set of equations, he/she can easily browse through the list of pictograms to find the appropriate one, without resorting to the use of a keyboard, and without a complicated and inaccurate translation to a computer representation of characters.

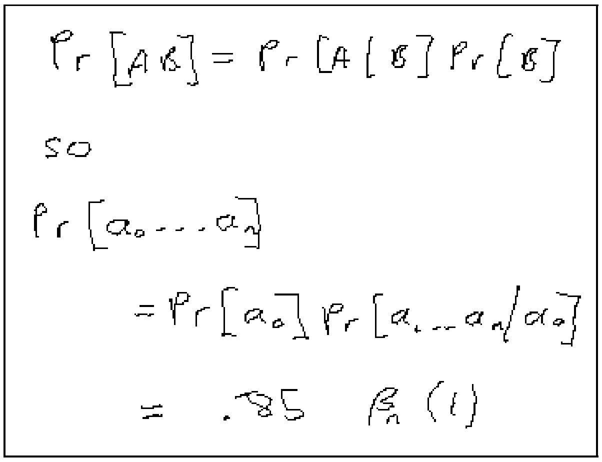

Figure 3.1: An example document.

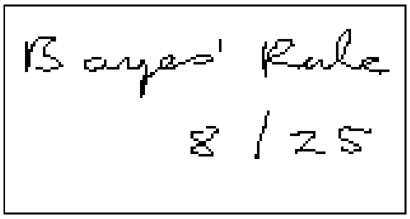

Figure 3.2: Its pictographic name.

3.2 A Pictographic Browser

To implement a file naming paradigm such as this, consider providing the user with a document browser, much like the browsers used in traditional mouse-based graphical user interfaces. However, rather than select a text string, the user selects an appropriate pictographic name. Such a browser is shown in Figure 3.3.

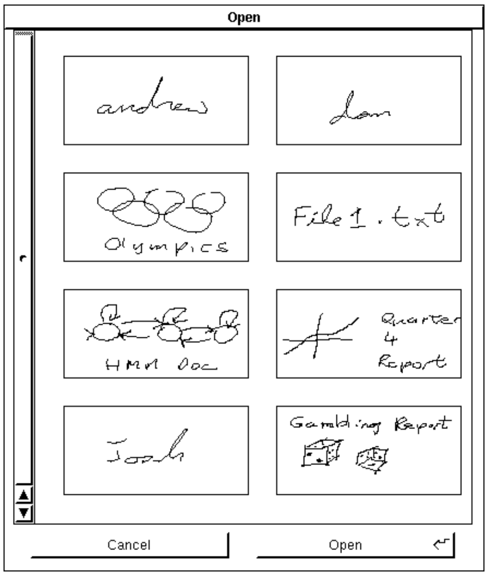

Figure 3.3: Pictographic browser.

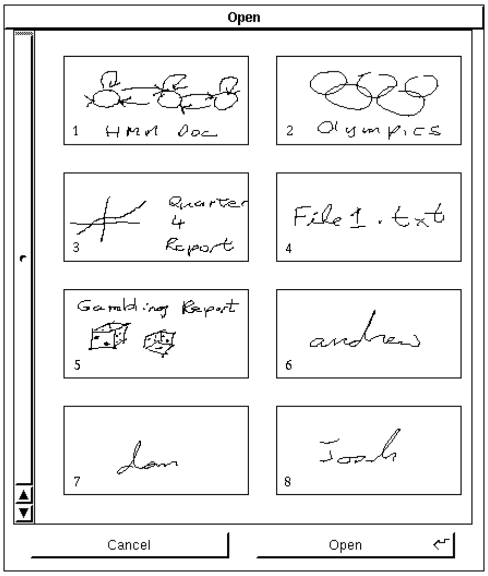

Figure 3.4: Ranked browser.

Note that pictographic names are very simple, and provide the user with far more flexibility than character strings. When new users are first introduced to standard file systems, they sometimes have difficulty adapting to the rigid conventions of traditional document storage and retrieval. Pictographic names allow the user to specify names rapidly and easily, while making available a much larger name space than traditional ASCII strings. Written words, sketches, non-ASCII characters, cursive script, symbols, Greek or Cyrillic letters, Kanji or other Eastern characters, or any combination of these are all valid names, as long as the user can recognize what he/she drew at a later time.

When the number of names becomes too large to browse manually, automatic search methods must be employed. We now consider an algorithm for solving this pictogram matching problem. While simple, this approach can produce results better than those obtained using nominally more powerful techniques such as Hidden Markov Models and Neural Nets. This seems appropriate for domains of moderate complexity (*e.g.*, the browser of Figure 3.3). As the complexity increases, however, more advanced tools may be necessary; in later sections, we examine a number of these more difficult tasks.

3.3 The Window Algorithm

If we knew that the same pictogram drawn twice by the same individual would tend to line up point-for-point, we could measure similarity between pictograms by summing the distances between corresponding points. Unfortunately, for real-world samples the points are not likely to correspond so closely. A two-step approach allows us to overcome this difficulty. First we compress the curves down to a small number of points. Then we allow the two curves to "slide" along one another. Given two pen-stroke sequences p and q, each re-sampled to contain N points, and an integer Δ representing the maximum "slide" we are willing to allow, define the distance D to be

$$D(p,q) \equiv \sum_{i=1}^{N} \left(\sum_{\delta=-\Delta}^{\Delta} w_\delta \, d(p_i, q_{i+\delta}) \right) \tag{3.1}$$

We assume that the point-wise distance function d returns 0 for boundary conditions where $i + \delta \notin [1..N]$. The values for w_δ are a parameter – we typically use $w_\delta = 1/(|\delta| + 1)$.

This procedure is similar to the dynamic programming "template matching" algorithms used in character recognition. However, it is computationally more efficient, and it allows us to make use of the fact that given two similar sequences p and q, we expect some similarity between p_i and all of $\{q_{i-1}, q_i, q_{i+1}\}$.

Experimental results for the Window algorithm are given in Figure 3.5. Each of four subjects created a database of 60 names. The first was in Japanese, the remainder in English. Each subject then re-drew each of the 60 names three times to create a 180-word test set. For each element of the test set, we used the Window algorithm to select and rank the eight most similar-looking words in the database. On average, this operation took 1/3 of a second to complete for each element of the test set, running on a 40MHz NeXT workstation. The table shows how often the correct element of the database was ranked first ("Ranked First"), and how often it was ranked in the top eight choices ("Ranked In Top 8").

	Data Set				
Success Criterion	A	B	C	D	Ave.
Ranked First	97%	83%	94%	88%	90%
Ranked In Top 8	100%	95%	99%	98%	98%

Figure 3.5. Experimental evaluation of the Window algorithm.

3.4 Hidden Markov Models

In this section, we present an overview of Hidden Markov Models in the context of handwritten pictogram matching. The reader is referred to [Rab89] for a tutorial. We assume that each of the pictograms is modeled by a Hidden Markov Model (HMM) as done in [LT92b, LT93a]. The HMM of a pictogram is stored along with the document to allow subsequent matching with the input.

Formally, an HMM is a doubly stochastic process that contains a non-observable underlying stochastic process (hidden) that can be uncovered by a set of stochastic processes that produce the sequence of observed symbols. Mathematically, an HMM is a tuple $< \Sigma, Q, a, b >$, where

- Σ is a (finite) alphabet of output symbols.
- Q is a set of states, $Q = \{0, ..., N-1\}$ for an N-state model.
- a is a probability distribution that governs the transitions between states. The probability of going from state i to j is denoted by a_{ij}. The transition probabilities a_{ij} are real numbers between 0 and 1, such that
  for all $i \in Q$: $\sum_{j=0}^{N-1} a_{ij} = 1$
  The distribution includes the initial distribution of states, that is the probability a_i of the first state being i.
- b is an output probability distribution $b_i(s)$ that govern the distribution of output symbols for each state. That is, $b_i(s)$ is the probability of producing the symbol $s \in \Sigma$ while being in state i. These probabilities follow the rules:
  for all $i \in Q$ and $s \in \Sigma : 0 \leq b_i(s) \leq 1$
  for all $i \in Q$, $\sum_{s \in \Sigma} b_i(s) = 1$

A variety of HMMs have been used to model handwriting. Also, a variety of features have been selected to describe the output symbols. In [LT93a], the authors divide the hand-drawn figure in points and extract four features per point: direction, velocity, change of direction and change of velocity. Each feature is drawn from a set of four possible values, hence the feature vector for a point is represented using eight bits. Each vector value is one of the output symbols in Σ.

Usually the transition probabilities (a) and the state set (Q) are computed by best-fitting the model to a series of samples. This is known as training the model. Algorithms for training models using samples of handwriting are described in [AVB94]. These algorithms are fast and require no intervention from the writer. Each sample used for the training consists of a sequence of output symbols (points), with which the parameters of the model can be adjusted. However, in applications like the one we are describing, the model has to be described using a single sample (a sequence of output symbols for the document that is to be filed). Quite commonly, then, the structure of the model is fixed to accommodate for the lack of samples with which to train it. A choice used in [LT92a] is that of a left-to-right HMM, i.e. a model in which it is only possible to remain in the current state or to jump to the next one in sequence. These models are sufficiently powerful to capture pictograms, as we have found out in practice. The rest of the adjustable parameters (branching probabilities, number of states, and output probabilities) provide a broad spectrum of choices to accomodate for pictogram differences. An example of such model is given in Figure 1. This model contains 5 states numbered from 0 to 4, and the probability to jump from state i to $i+1$ is 0.5, while the probability of staying in the same state is 0.5. For the last state, the probability of staying in it is 1.0.

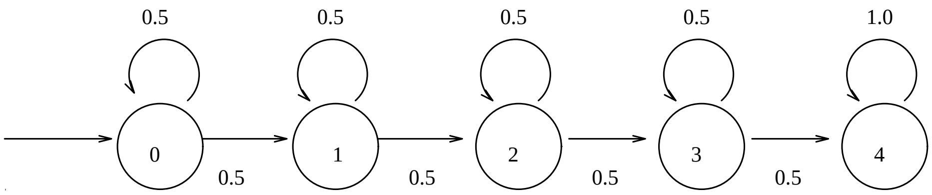

Figure 3.6. A left-to-right HMM.

Several authors have used HMMs to model handwriting and hand-written documents (e.g., [LT92b, LT93a, BK92, TSW90]).

We assume that each pictogram in the database is modeled by a left-to -right HMM, i.e., a model in which it is only possible to remain in the current state or to jump to the next one in sequence. An example of such model is given in Figure 3.7. The HMM of a pictogram is stored along with the pictogram to allow subsequent matching with the input.

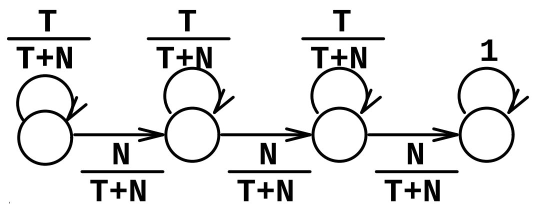

Figure 3.7. A left-to-right HMM.

This model contains 4 states numbered from 0 to 3, and the probability to jump from state i to $i+1$ is $\frac{N}{N+T}$, while the probability of staying in the same state is $\frac{T}{N+T}$. For the last state, the probability of staying in it is 1.0. Notice that we adjust the transition probabilities so that the HMM is *encouraged* to remain in the same state until it consumes the symbols in the input pattern that correspond to this state. More concretely, consider an HMM with N states and an input pattern with

T symbols. Assume that each of the N states is responsible for consuming $\frac{T}{N}$ symbols. Therefore, we can adjust the transition probability matrix a in the following way:

$$a_{ii} = \frac{\frac{T}{N}}{\frac{T}{N}+1} = \frac{T}{N+T} \text{ for } i = 0, ..., N-1 \tag{3.2}$$

$$a_{i,i+1} = \frac{1}{\frac{T}{N}+1} = \frac{N}{N+T} \text{ for } i = 0, ..., N-1 \tag{3.3}$$

This way, it is expected that the HMM consumes the symbols intended for a given state before moving to the next state. The model is then trained using multiple sample inputs (see [AVB94] for a detailed discussion about training using multiple patterns). Smoothing is performed upon completing the training stage by assigning an epsilon value to the output probability value for all the output symbols which did not appear in the training patterns.

Table 3.1 shows the results obtained with the various training methods presented in [AVB94].

Row	Training Method	Rank					Total Rank
		1st	2nd	3rd	5th	10th	
0	No Training	42.25%	55.00%	63.75%	74.75%	87.50%	1685
1	Levinson's [LRS83]	85.25%	91.25%	94.25%	96.25%	98.50%	252
2	Plain average	85.50%	90.75%	93.75%	97.00%	98.25%	240
3	Biased average (normalized)	80.25%	90.00%	92.75%	96.25%	98.25%	285
4	Biased average (unnormalized)	85.00%	91.00%	94.25%	97.00%	98.75%	220
5	Binary merge (normalized)	81.00%	90.00%	93.00%	96.25%	98.25%	283
6	Binary merge (unnormalized)	81.75%	90.25%	94.00%	96.50%	98.75%	239

Table 3.1. A comparison of various training methods.

4. The ScriptSearch Algorithm

We now turn our attention to a more difficult problem, that of searching through a continuous ink text. In the domain of pictographic naming, we are essentially solving a dictionary look-up problem: given a dictionary of words and a search key, we wish to locate the key in the dictionary. We know that the key will never span multiple entries, and that it will always match the intended entry from beginning to end.

Now, however, imagine a pen computer on which a user has written many pages of notes. If the user wishes to re-enter and search for a particular phrase, the system must be able to locate the phrase even though it crosses an unknown number of word boundaries. The problem is made all the more difficult when one considers that word segmentation algorithms sometimes make mistakes, breaking one word into two or merging two into one. We next describe an algorithm that requires no *a priori* segmentation of the database – it is searched as though it were a continuous stream of text [LT94]. We begin with some definitions.

4.1 Definitions

Ink is a sequence of time-stamped points in the plane:[2]

[2] Pen-tip pressure is another parameter that is sometimes available, but we do not make use of it in this chapter.

$$ink = (x_1, y_1, t_1), (x_2, y_2, t_2), \ldots, (x_k, y_k, t_k) \tag{4.1}$$

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

A standard model for approximate string matching is provided by *edit distance*, also known as the "k-differences problem" in the literature. In the traditional case [WF74], the following three operations are permitted:

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

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

$$\begin{array}{rcll} d_{0,0} & = & 0 & \\ d_{i,0} & = & d_{i-1,0} + c_{del}(p_i) & 1 \le i \le m \\ d_{0,j} & = & d_{0,j-1} + c_{ins}(t_j) & 1 \le j \le n \end{array} \tag{4.2}$$

and the main dynamic programming recurrence is

$$d_{i,j} = \min \left\{ \begin{array}{lcl} d_{i-1,j} & + & c_{del}(p_i) \\ d_{i,j-1} & + & c_{ins}(t_j) \\ d_{i-1,j-1} & + & c_{sub}(p_i, t_j) \end{array} \right. \qquad 1 \le i \le m,\ 1 \le j \le n \tag{4.3}$$

When Equation 4.3 is used as the inner-loop step in an implementation, the time required is $O(mn)$ where m and n are the lengths of the two strings.

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

$$\begin{array}{rcll} d_{0,0} & = & 0 & \\ d_{i,0} & = & d_{i-1,0} + c_{del}(p_i) & 1 \le i \le m \\ d_{0,j} & = & 0 & \end{array} \tag{4.4}$$

The inner-loop recurrence (*i.e.*, Equation 4.3) remains the same. Finally, we must define our evaluation criteria. It seems inevitable that any ink search algorithm will miss true occurrences of P in T, and report false "hits" at locations where P does not really occur. Quantifying the success of an algorithm under these circumstances is not straightforward. The field of information retrieval concerns itself with a similar problem in a different domain, however, and has converged on the following two measures [SM83]:

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

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

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

4.2 Approaches to Searching Ink

Ink can be represented at a number of levels of abstraction, as indicated in Figure 4.1. At the lowest level, ink is a sequence of points; at the highest, ink is ASCII text. It is natural to assume that ink search could take place at any given level, with attendant advantages and disadvantages.

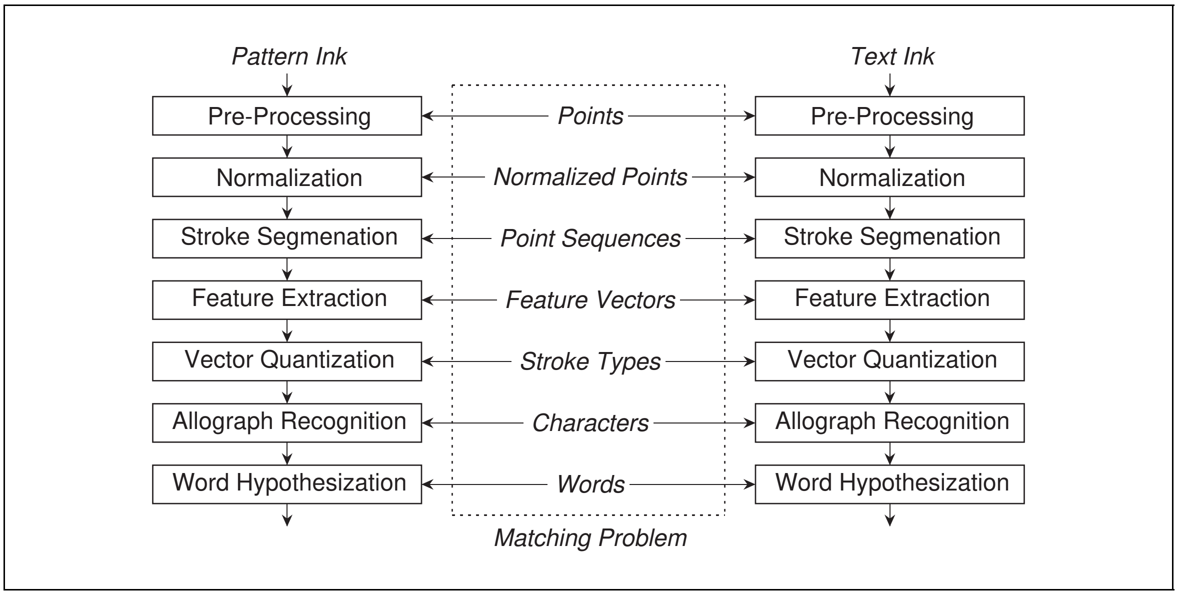

Figure 4.1. Handwriting recognition stages and potential matching problems.

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

An ink search algorithm could perform approximate matching at any level of representation. At one end of the spectrum, an algorithm like the Window algorithm of Section 3. could be used to match individual points in the pattern to points in the text. At the other extreme, we could perform full HWX on both the pattern and the text, and then apply "fuzzy" matching on the resulting ASCII strings (to account for recognition errors).

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

Section 4.4 presents an in-depth examination of an algorithm we call *ScriptSearch* that performs matching at the level of pen-strokes. This approach has the advantage of allowing us to do quite well against a broad range of handwriting, including some so bad that a human might find it illegible. ScriptSearch also allows the possibility of matching strings with no obvious ASCII representation, such as equations, drawings, doodles, etc.

4.3 Searching for Patterns in Noisy Text

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

To illustrate the effects of noise on pattern matching, consider what happens when we search for a number of keywords in Herman Melville's famous novel, *Moby-Dick*. Figure 4.2 tabulates average recall and precision under a variety of scenarios. Here *garble rate* represents a uniformly random artificial noise source that deletes, inserts, and substitutes characters in the pattern and the text. Note that when there is some "fuzziness," the precision can drop off rapidly if we require perfect recall. At some point, the text is no longer searchable as too many false hits are returned to the user. This is what we mean when we ask the question: Is ink searchable?

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

Figure 4.2. Searching for keywords in *Moby-Dick* (as a function of threshold).

Another view of the data is to consider the precision realizable for a given recall rate. This is shown in Figure 4.3. An intuitive interpretation of this figure is that setting a threshold is unnecessary if a ranked list of matches is returned to the user. In this case, for example, at a 10% garble rate, the user will experience a precision of 0.928 in viewing 50% of the true hits for the pattern.

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

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

Figure 4.3. Searching for keywords in *Moby-Dick* (as a function of recall rate).

4.4 The ScriptSearch Algorithm

As we noted above, representations for ink exist at various different levels of abstraction. In this subsection we examine an algorithm for writer-dependent ink search at the pen-stroke level. The algorithm applies dynamic programming with a recurrence similar to that used for string edit distance, but with a different set of operations and costs. The top-level organization of the ScriptSearch algorithm is shown in Figure 4.4.

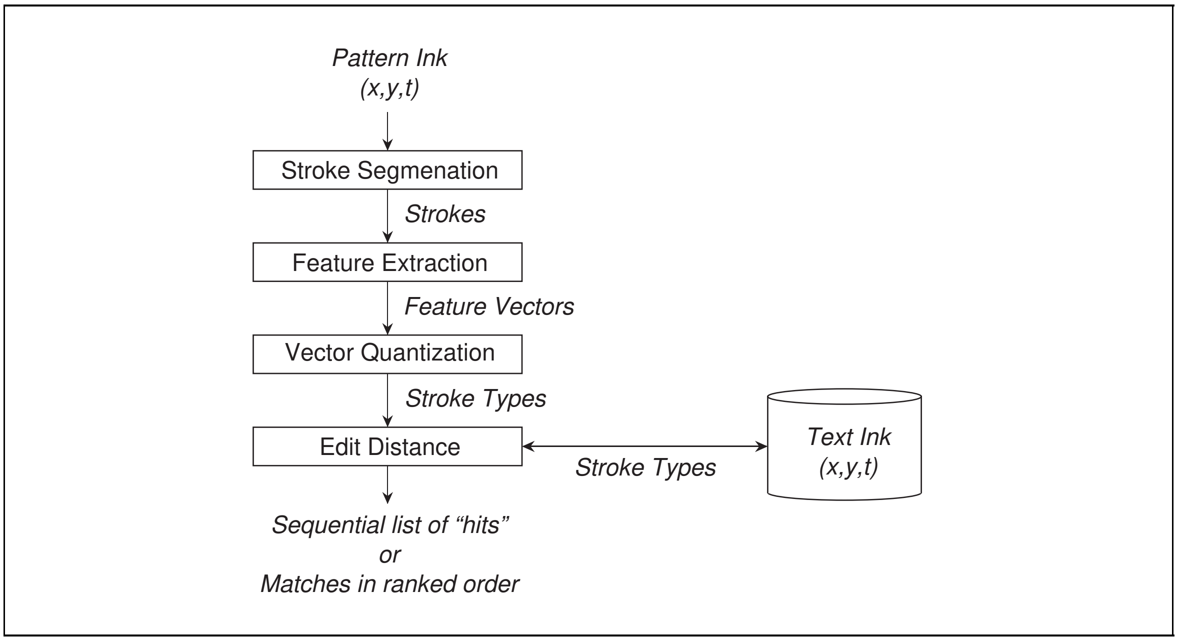

Figure 4.4. Overview of the ScriptSearch algorithm.

As can be seen from the figure, there are four phases to the algorithm. First, the incoming ink points are grouped into strokes. Next, the strokes are converted into vectors of descriptive features. Third, the feature vectors are classified according to writer-specific information. Finally, the resulting sequence of classified strokes is matched against the text using approximate string matching over an alphabet of "stroke classes." We now describe the four phases in more detail.

4.4.1 Stroke Segmentation. There are several common stroke segmentation algorithms used in handwriting recognition. For our experiments, we break strokes at local minima of the y values. Figure 4.5 shows a sample line of stroke-segmented text.

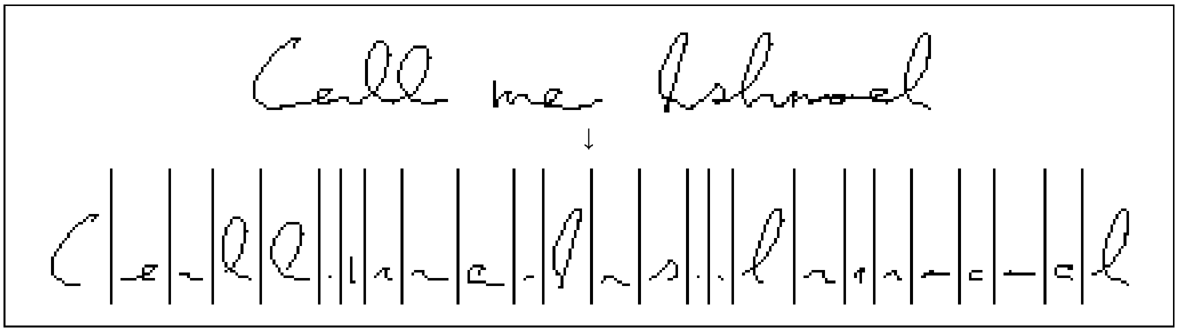

Figure 4.5. Example of pen-stroke segmentation.

4.4.2 Feature Extraction. As with segmentation algorithms, there are many different feature sets employed by handwriting researchers today. We have taken a set created by Dean Rubine in the context of gesture recognition [Rub91]. This particular feature set, which converts each stroke into a real-valued 13-dimensional vector, seems to do well at discriminating single strokes, and can be efficiently updated as new points arrive. The feature set includes the length of the stroke, total angle traversed, angle and length of the bounding box diagonal, etc.

4.4.3 Vector Quantization. In the vector quantization stage, the complex 13-dimensional feature space is segmented or "quantized" into 64 clusters. From then on, instead of representing a feature vector by 13 real values, we represent it by the index of the cluster to which it belongs. Thus, rather than maintain 13 real numbers, we maintain 6 bits. This technique is common in speech recognition and many other pattern recognition domains [LBG80]. The quantization makes the remaining processing much more efficient, and seeks to choose clusters so that useful semantic information about the strokes is retained by the 6 bits of the index. We now describe how to build and use the clusters. First, we must describe how distances are calculated in the feature space.

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

$$\Sigma_{ij} \equiv E\left[(v_i - \mu_i)(v_j - \mu_j)\right] \tag{4.5}$$

Hence, for instance, the diagonal of Σ contains the variances of the features.

Instead of standard Euclidean distance, we employ *Mahalanobis distance* [Sch92]. This is defined on the space of feature vectors as follows:

$$\|\mathbf{v}\|_M^2 \equiv \mathbf{v}^T \Sigma^{-1} \mathbf{v} \tag{4.6}$$

$$d(\mathbf{v}, \mathbf{w}) \equiv \|(\mathbf{v} - \mathbf{w})\|_M \tag{4.7}$$

With a suitable distance measure for our feature space, we can now proceed to describe a vector quantization scheme. We cluster the feature vectors of the ink sample into 64 groups using a clustering algorithm from the literature known as the k-means algorithm [Mac67]. The feature vectors of the sample are processed sequentially. Each vector in turn is placed into a cluster, which is then updated to reflect the new member. Each cluster is represented by its centroid, the element-wise average of all vectors in the cluster.

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

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

$$< \textit{stroke-type 7} >< \textit{stroke-type 42} >< \textit{stroke-type 20} > \ldots \tag{4.8}$$

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

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

$$\Sigma^{-1} = A^T A \tag{4.9}$$

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

$$\mathbf{v}^T \Sigma^{-1} \mathbf{v} = \mathbf{v}^T (A^T A) \mathbf{v} = (\mathbf{v}^T A^T) \cdot (A\mathbf{v}) = \mathbf{w}^T \mathbf{w} \tag{4.10}$$

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

4.4.4 Edit Distance. Finally, we compute the similarity between the sequence of stroke-types associated with the pattern ink, and the pre-computed sequence for the text ink. We use dynamic programming to determine the edit distance between the sequences. The cost of a deletion or an insertion is a function of the "size" of the ink being deleted or inserted, where size is defined to be the length of the stroke-type representing the ink, again using Mahalanobis distance. The cost of a substitution is the distance between the stroke-types. We also allow two additional operations: two-to-one *merges* and one-to-two *splits*. These account for imperfections in the stroke segmentation algorithm. We build a merge/split table that contains information of the form "an average stroke of type 1 merged with an average stroke of type 4 results in a stroke of type 11." The cost of a particular merge involving strokes α and β and resulting in stroke γ is, for instance, a function of the distance between $merge(\alpha, \beta)$ and γ. We compute the edit distance using these operations and their associated costs to find the best match in the text ink.

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

$$d_{i,j} = \min \begin{cases} d_{i-1,j} & + \quad c_{del}(p_i) \\ d_{i,j-1} & + \quad c_{ins}(t_j) \\ d_{i-1,j-1} & + \quad c_{sub1:1}(p_i, t_j) \\ d_{i-1,j-2} & + \quad c_{sub1:2}(p_i, t_{j-1} t_j) \\ d_{i-2,j-1} & + \quad c_{sub2:1}(p_{i-1} p_i, t_j) \end{cases} \qquad 1 \le i \le m,\ 1 \le j \le n \tag{4.11}$$

4.5 Evaluation of ScriptSearch

In this section, we describe the procedure we used when evaluating the ScriptSearch algorithm. We asked two individuals to hand-write a reasonably large amount of text taken from the beginning of *Moby-Dick*. Throughout the remainder of this discussion, we shall refer to these two primary datasets as "Writer A" and "Writer B." Figure 4.6 summarizes some basic statistics concerning the test data.

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

Figure 4.6. Statistics for the test data used to evaluate ScriptSearch.

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

As indicated earlier, the task of the algorithm is to find all the lines of the text that contain the pattern. For each writer (A and B), we augmented by hand the ASCII source text with the locations of the line breaks. Thus, the ASCII text corresponded line-for-line to the ink text. Using exact matching techniques, we found all occurrences of the ASCII patterns in the ASCII text, and noted the lines on which they occurred. For an ink search to be successful, the ink patterns must be found on the corresponding lines of the ink text.

We then segmented the ink texts into lines using simple pattern recognition techniques, and associated each stroke of the ink text with a line number. Figure 4.7 shows an example of a page of ink with the center-points of the lines determined by the algorithm, and also serves to illustrate the quality of the handwriting in the test data.

Using ScriptSearch, we found all matches for the ink pattern in the ink text. When combined with the line segmentation information, this determined the lines of the ink text upon which matches occurred. Since the ASCII text had been placed in line-for-line correspondence to the ink text, we could quickly determine which matches were valid, which were "false hits," and which were missed by the algorithm. From this information, we computed the recall and precision of the ScriptSearch procedure.

4.6 Experimental Results

As mentioned previously, there are two ways of viewing the output of a pattern matching algorithm like ScriptSearch. If hits are returned in a ranked order, precision can be calculated by considering the number of spurious elements in the ranking above a certain recall value. If all hits exceeding a fixed threshold are returned, recall and precision can be calculated by determining the total number of hits returned and the number of valid hits returned for a particular threshold.

There is a common thread relating these two points of view. If it were possible to choose an optimal threshold for each search, then a system that returns all hits above that threshold will have the same recall (*i.e.*, 1) and precision as a ranked system. Thus, a ranked system represents, in some sense, an upper-bound on the performance that can be obtained with a thresholded system. In contrast, a thresholded system has the advantage that ink can be processed sequentially – hits are

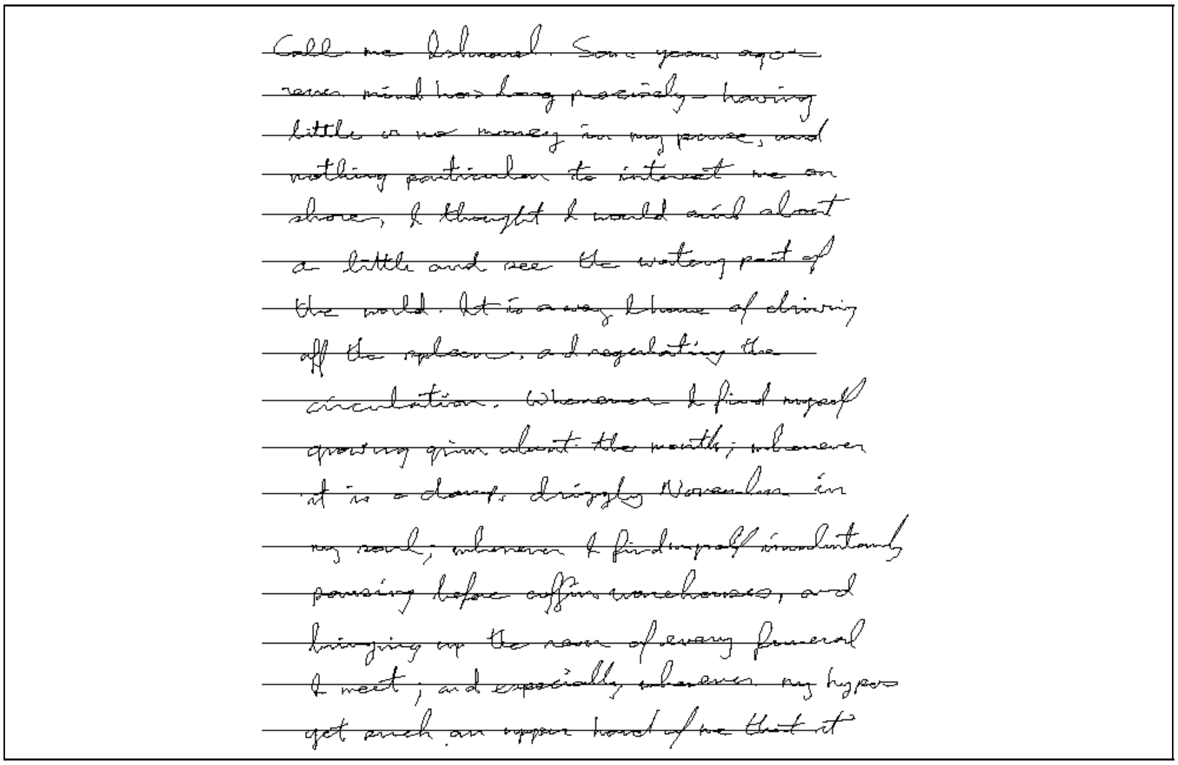

Figure 4.7. Estimation of line center-points (ScriptSearch line segmentation).

returned as soon as they are found, without waiting for the entire search to complete. If ScriptSearch is used as an intermediate stage in a "pipe," thresholding might be required in certain applications. Hence, as before, we present experimental results that reflect both viewpoints.

Figure 4.8 shows the performance of the algorithm when returning ranked hits. These results demonstrate that pattern length has a large impact on performance. For example, at 100% recall, there is a 47% difference in the average precision for long and short patterns for Writer A, and a 50% difference for Writer B.

Figures 4.9 and 4.10 present recall and precision as a function of edit distance threshold for Writers A and B, respectively. From these results, we can conclude that thresholds should be chosen dynamically based on properties of the pattern such as length. As before, we see that long patterns are more "searchable" than short ones.

In order to explore our intuition that this form of stroke-based matching is not appropriate for multiple authors, we asked three more writers (C, D, and E) to write the entire set of 60 search patterns. We then matched these patterns against the text of Writer A. The results for this test are shown in Figure 4.11 for the ranked case. As expected, the performance of the algorithm degrades dramatically. This implies that ink search at the stroke level should probably be restricted to patterns and text written by the same author, unless a more complex notion of stroke distance can be developed.

4.7 Discussion

In this section, we have discussed techniques for searching through an ink text for all occurrences of a pattern. We presented data that suggests using HWX and then performing fuzzy matching at the

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

Figure 4.8. Ranked precision values for Writers A and B.

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

Figure 4.9. Recall and precision as a function of edit distance threshold for Writer A.

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

Figure 4.10. Recall and precision as a function of edit distance threshold for Writer B.

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

Figure 4.11. Cross-writer precision (text by Writer A).

character level is one viable option. We also described ScriptSearch, a pen-stroke matching algorithm that performs quite well for same-author searching, both in thresholded and ranked systems. The latter approach has a paradigmatic advantage as it treats ink as a first-class datatype.

In the future, it would be interesting to evaluate approaches that represent ink at different levels of abstraction (recall Figure 4.1), for example as allographs, perhaps performing dynamic programming on the associated adjacency graph to locate matches. Another intriguing extension of the work we have just described concerns searching non-textual ink, and languages other than English. We observe that if the VQ classes are trained using a more general set of strokes, it should be possible to run ScriptSearch as-is on drawings, figures, equations, other alphabets, etc. It would be instructive to examine its effectiveness in these domains, especially since traditional HWX-based methods do not apply.

It is also clearly important to address the issue of writer-independence with regard to ink matching. We now briefly sketch an approach that appears to have some potential. Recall that since the VQ codebooks for two authors may be different, there is no natural stroke-to-stroke mapping. Let us assume that by some means it is possible to put text from two authors A and B into a rough correspondence, and then to determine for each of A's strokes a distribution of similarities to B's strokes. We can represent these distributions as a *Stroke Similarity Matrix*, S. The i^{th} row of such a matrix describes how A's i^{th} stroke corresponds to all of B's strokes. Assume that the $(i,j)^{th}$ entry of matrix $D_{B\rightarrow B}$ gives the Mahalanobis distance from B's stroke i to stroke j. We wish to compute $D_{A\rightarrow B}$, the matrix giving distances from each of A's strokes to each of B's strokes. We can do so as follows:

$$D_{A\rightarrow B} = S \cdot D_{B\rightarrow B} \tag{4.12}$$

That is, to compute the distance between the i^{th} stroke of A and the j^{th} stroke of B, we view the i^{th} stroke of A as corresponding to various strokes of B with the weights given in the i^{th} row of S. We extract the distance from each of these strokes to B's j^{th} stroke, and take the weighted sum of these values. This is the inner product of the i^{th} row of S with the j^{th} column of $D_{B\rightarrow B}$, as indicated in Equation 4.12. This approach should yield a reasonable "cross-writer" distance measure that we can substitute for Mahalanobis distance. The ScriptSearch algorithm could then be used without further changes.

Finally, since the amount of ink to be searched will undoubtedly grow as pen computers proliferate, it is important to consider sub-linear techniques that employ more complex pre-processing of the ink text. Some of these are treated in the next section.

5. Searching Large Databases

Now that we have discussed some of the issues regarding ink as a first class datatype, we consider the issues of large ink databases. Using sequential searching techniques (like the ones explained previously), the running time grows linearly with the database size. This is clearly unacceptable for large databases. Thus, more sofisticated methods should be used to do the searching. In this section we show some techniques to index pictograms and speed up the searches for large databases.

As pointed out in Section 4.2, ink can be represented at a number of levels of abstraction. Different types of indices can be built for each one of these *granules* of representation. For instance, we can choose to model entire pictograms with HMMs and build indices that use the HMM characteristics to guide the search. We call such an index the *HMM-tree* [AB94]. Alternatively, we can choose to deal with alphabet symbols (or strokes) for granularity and represent the symbol classes by using HMMs. We call the resulting index the Handwritten Trie.

In the next subsections, we describe each one of these two approaches.

5.1 The HMM-Tree

Assume that we have M pictograms in our database and that each document has been modeled by an HMM (and hence we have M HMMs in the database). Each one of the HMMs has the same transition distribution (a), number of states (N), output alphabet (Σ), and a fixed-length sequence of output symbols (points) (T) (i.e., that each input pattern is sampled by T sample points, each of which can assume one of the possible symbols of the output alphabet). Let the size of the output alphabet be n (i.e., $|\Sigma| = n$). The output distribution is particular to each HMM (and hence to each document). For each document D_m in the database, with $0 \leq m \leq M$, we call H_m the HMM associated with the document.

As suggested in [LT92b], we use the following two measures of "goodness" of a matching method:

- a method is good if it selects the right picture first for reasonable size databases, because this way the user can simply confirm the highlighted selection.
- a method is good if it often ranks the right picture on the first k items (so that those can fit in the first page of a browser [LT92b]) for reasonable size databases, because this way the user can easily select the picture.

In order to recognize a given input pattern I, we execute each HMM in the database and find which k models generate I with the highest probabilities. This approach is extremely slow in practice, as shown in Figure 5.1.

One way to avoid this problem is to move the execution of the HMMs to the preprocessing phase in the following way (which we term the *naive* approach). At the preprocessing phase we enumerate all the possible output sequences of length T. Since each output symbol can assume one of n values, we have n^T possible output sequences. For each sequence, we execute all the HMMs in the database and select the top k HMMs that generate the sequence with highest probability. We repeat this process for all the sequences. The output is a table of size kn^T where for each possible sequence the identifiers of the best k HMMs for this sequence are stored. At run time, for a given input pattern, we access this table at the appropriate entry and retrieve the identifiers of the corresponding HMMs. In order to insert a new document D_m (modeled by the HMM H_m) into the database, we need to execute H_m for every possible sequence (out of the n^T sequences) and for each output sequence S compare the probability, say p_m, that results from executing $H_m(S)$, with the other k probabilities associated with S. If $H_m(S)$ is higher than any of the other k probabilities, the list of identifiers associated with S is updated to include m. If the list of probabilities is kept sorted, then $\log k$ operations are needed to insert i and p_i in the proper location.

The complexity of the naive approach can be summarized as follows:

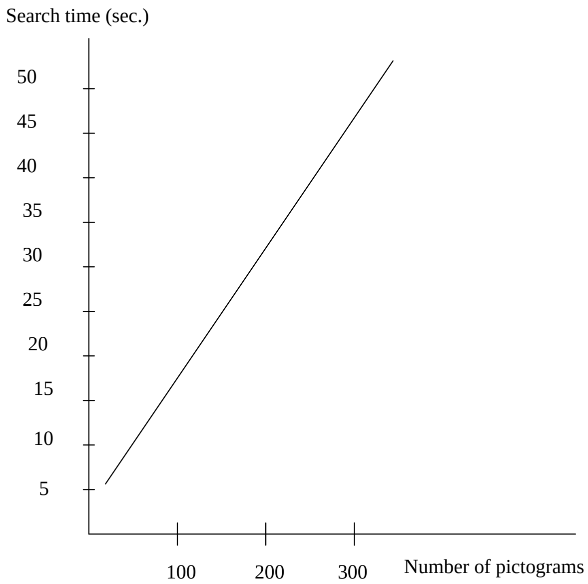

Figure 5.1. Matching time using a sequential algorithm.

- **Preprocessing time**: $Mn^T C_H(\log k + \lceil \frac{k}{2} \rceil)$, where C_H is the average time to execute an HMM, given an input sequence, and $\log k + \lceil \frac{k}{2} \rceil$ is the time to maintain the indexes for the k HMMs with best probability values.
- **Space**: $(T+2k)n^T$, i.e., is exponential in the number of sample points in the input pattern T. The factor (T+2k) is the size of each entry in the table; T is the number of symbols per sequence, and $2k$ is due to storing the HMM identifiers along with the probability that each of them generates the pattern. The latter is used when inserting a new document to test whether the new model generates the corresponding pattern with better probability than any of the given k HMMs.
- **Searching** (at runtime): $\log_2 n^T = T \log n$.
- **Insertion**: $n^T C_H(\log k + \lceil \frac{k}{2} \rceil)$.

In order to organize the above table, we use a tree structure. One possible tree (which we term the *HMM1-tree*) is a balanced tree of depth T and fanout n. Each internal node has a fixed capacity of n elements where an element corresponds to one of the symbols of the alphabet. Figure 5.2 shows an example of the HMM1-tree. The HMM1-tree is a variation of the *Pyramid* data structure [TP75], where in the case of the HMM1-tree, the fanout is not restricted to a power of 2 as in the case of a pyramid. In addition, the pyramid is used to index space while the HMM1-tree is used to index HMMs. However, the structure of both the HMM1-tree and the pyramid is similar. In the example of Figure 5.2, the alphabet has two symbols (and hence the nodes have two entries each), and the length of the sequence is 3 (3 output symbols must be entered to search documents). We see how nodes in the last level of the tree point to linked lists of documents. The dotted path in the tree shows the path taken by the traverse algorithm when the input contains the symbols $0, 1, 0$. This particular search retrieves documents D_3 and D_4.
More formally, the HMM1-tree is constructed as follows.

- The HMM1-tree has $T + 1$ levels (the number of steps or length of the output sequence in the HMMs associated with the documents in the repository). The root of the tree is the node at level 0 and is denoted by r.

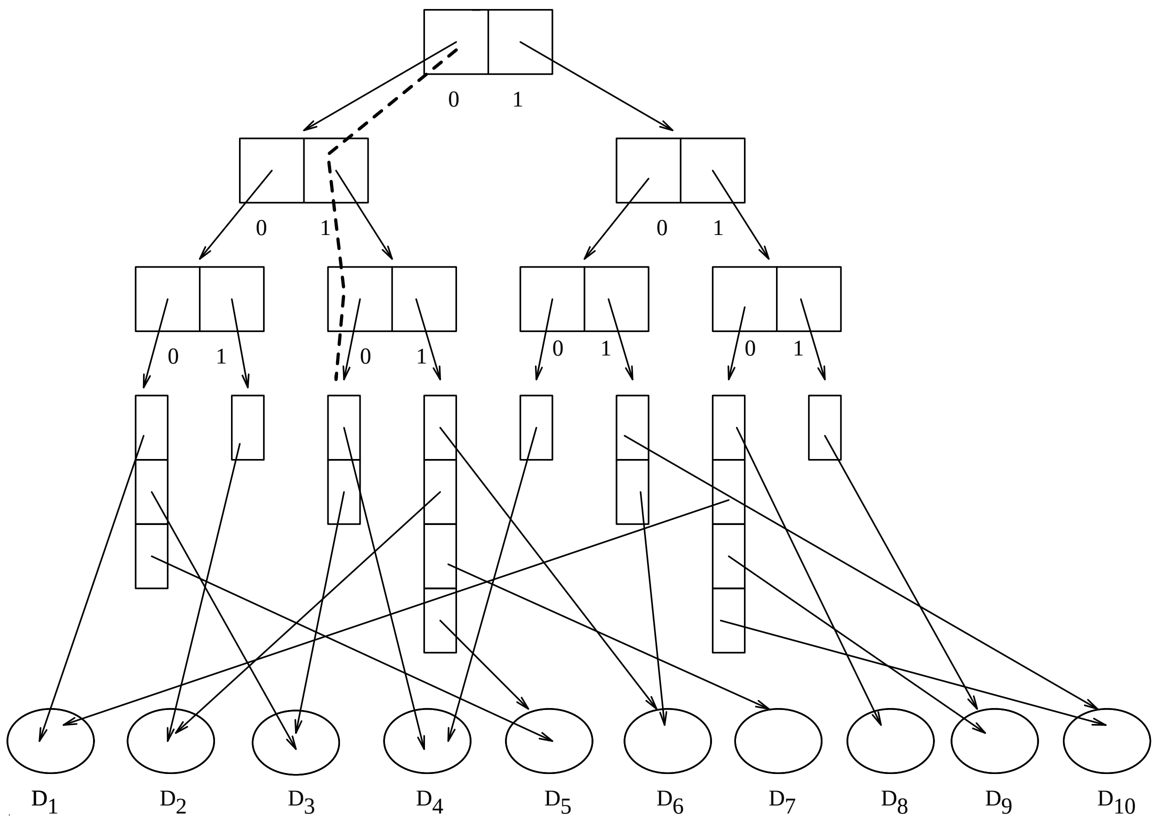

Figure 5.2. An example of an HMM1-tree.

- Each internal node (including the root) in the tree is an n-tuple, where each entry in the n-tuple corresponds to a symbol of the output alphabet Σ and has a pointer to a subtree[4]. We denote by $v[k]$ the kth entry on the node v.
- Each internal node in the Tth level points to a leaf node that contains a linked list. The linked lists store pointers to the files that contain the documents in the repository.

The preprocessing time for the HMM1-tree is still $Mn^T C_H$, since we need to traverse each node at the leaf level and for each node find the best HMMs (by executing all M of them and selecting the ones with highest probabilities) that generate the output sequence that corresponds to this node.

To insert a document, we traverse all the nodes at the leaf level without having to descend the tree starting from the root. For each leaf node, we follow the same approach as the table approach described above and hence the complexity of insertion is the same, i.e., $\log kn^T C_H$.

To select a set of documents that are similar to an input D, we extract a set of T output symbols $O = \{O[i], 0 \leq i \leq T \; and \; 0 \leq O[i] \leq n-1\}$ from D and run the following algorithm.

Procedure traverse(O)
begin
    $v = r$
    for $(0 \leq level \leq T)$

[4] A pyramid can be implemented as a heap array where the address of any internal or leaf node can be computed and directly accessed if the symbols that lead from the root to that node are known [AS90, Tuc84]. As a result, we can avoid storing explicit pointers and compute the address of each node instead.

    $v = v[O[l]]$
  return every element in the list pointed by v
end

An alternative approach to traversing the tree which avoids storing pointers is based on the observation that since the HMM1-tree is a complete tree, i.e., none of the internal nodes is missing, then the addresses of the nodes can be easily computed and there is no need to store pointers to subtrees explicitly in the tree (this is similar to the technique used in the pyramid data structure [AS90, Tuc84]).

The storage complexity of the HMM1-tree can be computed as follows. The number of non-leaf (internal) nodes is $\frac{n^T-1}{n-1}$ where each node is of size n (notice that since we assume that the addresses of the nodes can be easily computed we do not store pointers to subtrees explicitly as described above), while the number of leaf nodes is n^T where each node is of size $2k$ (to store the k HMM identifiers along with their corresponding probabilities). Therefore, the total space complexity is:

$$n\frac{n^T-1}{n-1} + 2kn^T$$

which is still exponential in the number of sample points in the input pattern T, although is less than the storage complexity of the naive approach (since $T > \frac{n}{n-1}$). The saving is due to the fact that for any two sequences that share the same prefix, this prefix is stored in the tree approach only once while is being repeatedly stored with each sequence in the naive approach.

The complexity of the HMM1-tree approach is summarized as follows.

- **Preprocessing time**: $Mn^T C_H(\log k + \lceil \frac{k}{2} \rceil)$, since at the leaf level we still have to store the k HMMs.
- **Space**: $n\frac{n^T-1}{n-1} + 2kn^T$ (still exponential).
- **Insertion**: $n^T C_H(\log k + \lceil \frac{k}{2} \rceil)$.
- **Searching** (at runtime): $O(T)$ since computing the address of the node depends on the path length to reach that node (or the length of the sequence that leads to the node)

5.1.1 Reducing the Preprocessing and Insertion Times. In this section, we show how to reduce the times for preprocessing and insertion.

The HMM2-tree

We show how to reduce the preprocessing and insertion times of the HMM1-tree. This results in what we term the HMM2-tree. Recall that in the case of the HMM1-tree, both the preprocessing and insertion times are exponential in the number of symbols per sequence. The HMM2-tree has the following additional properties.

- Each level l $(0 \leq l \leq T)$ in the HMM2-tree is associated with a threshold value ϵ_l $(0 \leq \epsilon_l \leq 1)$.
- For each node q in the HMM2-tree, at level l, and each symbol o in the output alphabet, let $O_q = O[i_1]O[i_2]\cdots O[i_l]$ denotes the sequence of symbols in the path from the root of the HMM2-tree to the node q. Then, there is an associated pruning function $f^m(l, q, O_q, o)$ that is computable for every model in the database. The use of the pruning function is demonstrated below.

To insert a document D_m (modeled by the HMM H_m) into the HMM2-tree, we perform the following algorithm.

Procedure HMM2-Insert(D_m)
begin
  Let r be the root of the tree

```
        level = 0
        call search(r, level)
end

Procedure search(v, l)
begin
        for 0 ≤ k ≤ n − 1
            if (f^m(l, v, O_v, k) ≥ ε_l)
              if (l ≤ T − 1)
                call search(v[k], l + 1)
              else
                include a pointer to D_m in the list pointed by v[k]
end
```

In other words, during the insertion procedure, when processing node v at level l and output symbol k, if the condition $(f^m(l, v, O_v, k) \geq \epsilon_l)$ is true the subtree $v[k]$ is investigated. Otherwise, the entire subtree is skipped by the insertion algorithm. This helps reduce the time to insert each document into the database.

The preprocessing stage is reduced to inserting each of the documents into the database by following the above insertion algorithm for each document. Therefore, the reduction in insertion time is also reflected into the preprocessing time.

To select a set of documents that are similar to an input D, we extract a set of T output symbols $O = \{O[i], 0 \leq i \leq T \textit{ and } 0 \leq O[i] \leq n-1\}$ from the input and we run procedure *traverse*, the one used for the HMM1-tree. Similar to the HMM1-tree, we can also compute the address of the leaf node from O and directly access the k HMMs associated with it.

At this point it is worth mentioning that the index described above will work provided that we supply the pruning function $f^m(l, q, O_q, o)$. The performance of the index will be affected by how effective the pruning function is. In the following Section, we describe several methods to compute such a function provided that some conditions are met by the underlying database of documents.

5.1.2 Pruning Functions. In this section, we present several methods for computing pruning functions.

In order to compute $f^m(l, q, O_q, o)$, we assume that the following conditions are met by the underlying database of documents.

- All the documents in the database are modeled by left-to-right HMMs with N states.
- The transition probabilities of these HMMs are the following:

$$a_{ij} = 0.5 \text{ for } i = 0, ..., N-2 \text{ and } j = i \text{ or } j = i+1 \tag{5.1}$$

$$a_{N-1N-1} = 1.0 \tag{5.2}$$

$$a_0 = 1, a_i = 0 \text{ for } i = 1, ..., N-1 \tag{5.3}$$

- For all the documents in the database, a sequence of output symbols of length T has been extracted. All inputs for which the index is going to be used have to be presented in the form of a T sequence of output symbols, taken from the alphabet (Σ) of the HMMs.

The Unconditional Method

Define $\phi^m_{i,j}$ to be the probability that the HMM H_m is in state j at step i of its execution $(0 \leq i \leq T-1$ and $0 \leq j \leq N-1)$. Notice that $\phi^m_{i,j}$ is independent of the output sequence. Now, define $\Phi_i^m(o)$ to be the probability that the HMM H_m outputs the symbol o at step i of execution. We can compute $\Phi_i^m(o)$ using $\phi_{i,j}^m$ as follows:

$$\Phi_i^m(o) = \sum_{j=0}^{N-1} \phi_{i,j}^m b_j(o) \tag{5.4}$$

$\Phi_i^m(o)$ is used as the pruning function f_1, i.e.,

$$f_1^m(i, q, O_q, o) = \Phi_i^m(o) \tag{5.5}$$

It remains to show how we compute ϕ_{ij}^m. Based on the HMM structure of Figure 3.6, $\phi_{i,j}^m$ can be expressed recursively as follows:

$$\phi_{i,0}^m = 0.5^i, \text{ for } i = 0, ..., T-1 \tag{5.6}$$

$$\phi_{0,j}^m = 0, \text{ for } j = 1, ..., N-1 \tag{5.7}$$

and

$$\phi_{i,j}^m = 0.5(\phi_{i-1,j-1}^m + \phi_{i-1,j}^m) \text{ for } i = 1, ..., T-1 \text{ and } j = 1, ..., N-1 \tag{5.8}$$

Notice that $\phi_{0,0}^m = 1$ and $\phi_{i,i}^m = 0.5^i$ for $1 \le i \le N-1$. An additional optimization that is based on the structure of the HMM of Figure 3.6 is that at step i, H_m cannot be past state $j > i$ since at best, H_m advances to a new state at each step. In other words,

$$\phi_{i,j}^m = 0 \text{ for } 0 \le i < j \le N-1 \tag{5.9}$$

Therefore, the recurrence for $\phi_{i,j}^m$ reduces to

$$\phi_{i,j}^m = 0.5(\phi_{i-1,j-1}^m + \phi_{i-1,j}^m) \text{ for } 1 \le j \le i \le N-1 \text{ and } i = 1, ..., T-1 \tag{5.10}$$

Figure 5.3 illustrates the recursion process for computing $\phi_{i,j}^m$.

The process of computing $\phi_{i,j}^m$ and $\Phi_i^m(o)$ is independent of which branch of the HMM2-tree we are processing. It is dependent only on the HMM model (H_m). As a result, when inserting an HMM model H_m into the HMM2-tree, we build a two-dimensional matrix Φ^m of size $T \times N$ such that $\Phi^m[i][j]$ corresponds to the probability that the jth output symbol appears at the ith step of executing the HMM H_m (i.e., $\Phi^m[i][j] = \Phi_i^m(o_j)$). This matrix is accessed while inserting the model H_m into the HMM2-tree to prune the number of paths descended by the algorithm (see procedure HMM2-Insert, given at the beginning of Section 5.1.1).

The Conditional Method

An alternative approach to computing pruning functions is to make use of the dependencies between the output symbols. Instead of computing the probability that an output symbol appears at step i of the execution of an HMM, we compute the probability that the sequence $O[0]O[1]\cdots O[i]$ appears after executing the first i steps of the HMM. This leads to the following new pruning function which depends on the path in the HMM2-tree where we are to insert a new HMM model into.

Our objective is to insert the index m of an HMM H_m into the linked list belonging to a leaf node q, when the probability that the sequence $O_q = O[0]O[1]\cdots O[T-1]$ (denoting the sequence of symbols in the path from the root of the HMM2-tree to the node q) is produced by H_m is high (or above a given threshold). This corresponds to the probability: $Prob[O[0]O[1]\cdots O[T-1]|H_m]$. In order to save on insertion and preprocessing times, we need to avoid computing this probability for every possible pattern (of length T) in the tree. As a result, we use the following pruning function which we apply as we descend the tree, and hence can prune entire subtrees.

Define $\alpha_{i,j}^m$ to be the probability that the sequence $O[0]O[1]\cdots O[i]$ is produced by the HMM after executing i steps and ending at state j. In other words,

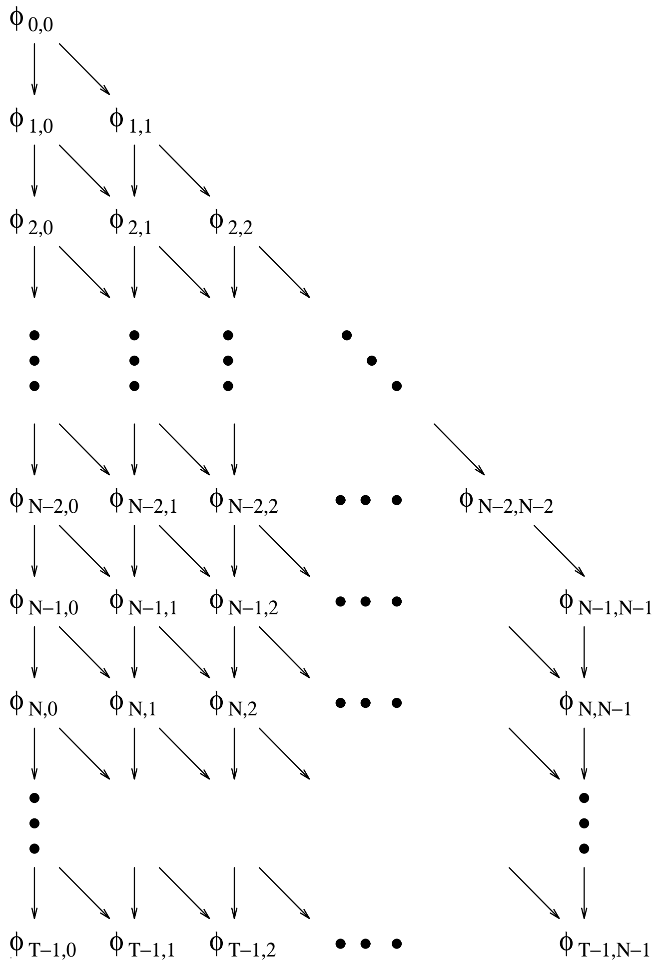

Figure 5.3. An illustration of how $\phi^m_{i,j}$ is computed recursively.

$$\alpha^m_{i,j} = Prob[O[0]O[1]\cdots O[i] | \text{ the state at step } i \text{ is equal to } j] \tag{5.11}$$

At the time an HMM model H_m is inserted into the HMM2-tree, $\alpha^m_{i,j}$ is computed dynamically as we descend the tree while constructing the sequence $O[0]O[1]\cdots O[i]$ on the fly. Assume that we descend the tree in a depth-first order, and we are in level j of the tree at node q. The sequence $O_q = O[0]O[1]\cdots O[i]$ corresponds to the symbols encountered while descending from the root to q. In this case, $\alpha^m_{i,j}$ can be computed as follows:

$$\alpha^m_{0,0} = b_0(O[0]) \tag{5.12}$$

$$\alpha^m_{i,0} = \alpha^m_{i-1,0} b_0(O[i]) \tag{5.13}$$

$$\alpha^m_{0,j} = 0 \text{ for } j = 1, ..., N-1 \tag{5.14}$$

In general,

$$\alpha^m_{i,j} = 0 \text{ for } 0 \le i < j \le N-1 \text{ and } i = 1, ..., T-1 \tag{5.15}$$

and

$$\alpha^m_{i,j} = 0.5(\alpha^m_{i-1,j} + \alpha^m_{i-1,j-1}) b_j(O[i]) \text{ for } 1 \le j \le i \le N-1 \text{ and } i = 1, ..., T-1 \tag{5.16}$$

The difference between this method and the unconditional method is that $\alpha^m_{i,j}$ depends on the output sequence produced up to step i of the computation, while $\phi^m_{i,j}$ does not. In addition, Φ^m_i depends only on one output symbol and not the sequence of symbols as does $\alpha^m_{i,j}$. The recursion process for computing $\alpha^m_{i,j}$ is the same as the one of Figure 5.3 except that we replace the computations for the ϕ's with the ones for the α's.

One way to save on the time for computing α for all the paths, is that we maintain a stack of the intermediate results of the recursive steps so that when we finish traversing a subtree we pop the stack up to that level and restart the recursion from there, instead of starting the computations from the $\alpha^m_{0,0}$. As we descend the HMM2-tree in order to insert a model H_m, when we visit a node q, we start from the α's in q's parent node, and incrementally apply one step of the recursive process for computing α for each of the symbols in q. We save the resulting n computations in the stack (we have n symbols in q). As we descend one of the subtrees below q, say at node u, we use the α's computed for node q in one additional step of the recursive formula for computing α and we get the corresponding α's at node u. This way the overhead for computing α's is minimal since for each node in the HMM2-tree, we apply one step of the recursive formula for computing α for each symbol in the node, and the entire procedure is performed only once per node, i.e., we do not re-evaluate the α's for a node more than once.

In order to prune the subtrees accessed at insertion time, we use $\alpha^m_{i,j}$ to compute a new function φ^m_i, which is the probability that a symbol $O[i]$ appears at step i of the computation (i.e., φ^m_i is independent of the information about the state of the HMM). This can be achieved by summing $\alpha^m_{i,j}$ over all possible states j. Then,

$$\varphi^m_i = Prob[O[0]O[1]\cdots O[i]|H_m \text{ is at step } i] \tag{5.17}$$

$$\varphi^m_i = \sum_{j=0}^{N-1} \alpha^m_{i,j} \tag{5.18}$$

φ^m_i is computed for each symbol in a node and is compared against a threshold value. The subtree corresponding to a symbol is accessed only if its corresponding value of φ^m_i exceeds the threshold. In other words, the pruning function for each node is set to be:

$$f^m(l, q, O_q, o) = \varphi^m_l \tag{5.19}$$

The Upper-Bounds Method

The Viterbi algorithm [For73] is an efficient way to compute the probability that a sequence of outputs is explained by a particular model.

The upper-bounds method is an approximation of the pruning function φ^m_l. The computations for φ^m_l are exact and hence may be expensive to evaluate for each input pattern and each tree path that is accessed by the insertion algorithm. The upper-bounds method tries to overcome this problem by approximating φ^m_l so that it is dependent only on the level of a node q and not on the entire tree path that leads to q.

Define $p^m_k(s)$ to be the computed probability (or an estimate of it) that a model puts the output symbol s in the kth stage of executing the HMM H_m. Then, $p^m_0(s)$ is the probability of finding output symbol s in the first step. According to [Bar93], $p^m_k(s)$ can be estimated as follows (the derivations can be found in [Bar93]):

$$p^m_k(s) = \sum_{j=0}^{N-1} A_{T-k+1,j} \tag{5.20}$$

where $A_{T-k+1,j}$ is an upper bound of $\alpha_{i,j}$, and can be estimated as follows:

$$A_{T-k+1,j} = (0.5)^T(\binom{T-K+1}{j} b_0(O[k]) + \quad (5.21)$$

$$\binom{T-K+1}{j-1} R_1 b_1(O[k]) + \ldots + \quad (5.22)$$

$$\binom{T-K+1}{j-k+1} R_j b_j(O[k])) \text{ for } k \leq j \leq N-1 \quad (5.23)$$

where R_r is the number of paths that one can take to get to state r in $k-1$ steps and is evaluated as follows:

$$R_r = \binom{k-1}{r-1} \quad (5.24)$$

The values A and $p_k^m(s)$ can be computed by the following procedures [Bar93]:

Procedure solve_recurrence(k, j)
begin
    $A_{T-k+1,j} = 0$
    for $i = j$ to 0
        $A_{T-k+1,j} = A_{T-k+1,j} + \binom{T-k+1}{i} R_i b_i(O[k])$
    $A_{T-k+1,j} = (0.5)^T A_{T-k+1,j}$
    return$(A_{T-k+1,j})$
end

Function $p(k, m, s)$
begin
    $p = 0$
    for $(j = 0$ to $N-1)$
        $p = p+$ solve_recurrence(k, j)
    return(p)
end

5.1.3 Reducing the Space Complexity - The HMM-Tree. The problem with the HMM2-tree is its exponential storage complexity. The typical values of the number of samples in a pattern (T) and the number of possible output symbols (n) are 50 and 256, respectively [LT92b, LT92a]. As a result, the number of leaf nodes in the HMM2-tree is $256^{50} = 2^{400} \approx 10^{120}$, which is almost intractable. In this section, we describe a new data structure (termed the HMM-Tree) which is an enhancement over the HMM2-tree in terms of its storage complexity.

The basic idea of the HMM-tree is that we use the pruning function not only to prune the insertion time but also to prune the amount of space occupied by the tree. We use Figure 5.4 for illustration. Assume that we want to insert model H_m into the HMM-tree. Given the pruning function (any of the ones given in Section 5.1.2), we compute the two-dimensional matrix P^m where each entry $P^m[i][o]$ corresponds to the probability that H_m produces symbol o at step i of its execution. Notice that P is of size $n \times T$. From $P^m[i][o]$, we generate a new vector L^m where each entry in L^m, say $L^m[i]$, contains only the high probable symbols that may be generated by H_m at step i of its execution. In other words, each entry of L^m is a list of output symbols such that:

$$L^m[i] = [o | P^m[i][o] > \epsilon_i \text{ for all } 0 \leq o < n] \quad (5.25)$$

For example, Figures 5.4a, 5.4b, and 5.4c give the vectors L^1, L^2, and L^3 which correspond to the HMMs H_1, H_2, and H_3. respectively. Initially the HMM-tree is empty (Figure 5.4d). Figure 5.4e shows the result of inserting H_1 into the HMM-tree. Notice that the fanout of each node in the tree

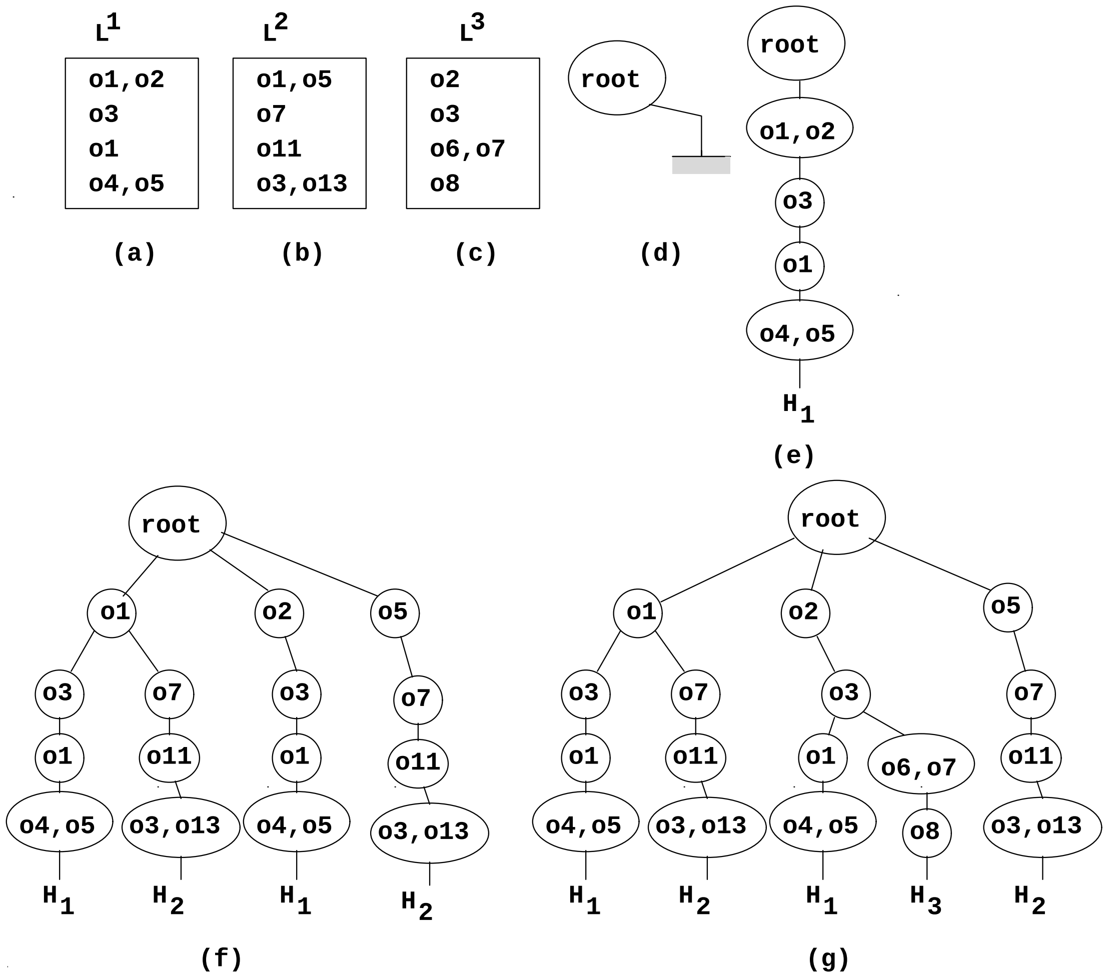

Figure 5.4. An example illustrating the savings of space achieved by the new HMM-tree.

is $\leq n$. The output symbols are added in the internal nodes only as necessary. Figures 5.4f and 5.4g show the the resulting HMM-tree after inserting H_2 and H_3, respectively. Notice how we expand the tree only as necessary and hence avoid wasting extra space.

The HMM-tree is advantageous since it has the nice features of both the HMM1-tree and the HMM2-tree while winning against both structures in terms of the space complexity. The HMM-tree has a searching time of $O(T)$ similar to the HMM1-tree, and uses the same pruning strategies for insertion as the HMM2-tree and hence reducing the insertion time.

5.2 The Handwritten Trie

The Trie structure [Fre60] is an M-ary tree, whose nodes have M entries each, and each entry corresponds to a digit or a character of the alphabet. An example trie is given in Figure 5.5 where the alphabet is the digits $0 \cdots 9$. Each node on level l of the trie represents the set of all keys that begin with a certain sequence of l characters; the node specifies an M-way branch, depending on the $(l+1)$st character. Notice that in each node an additional *null* entry is added to allow for storing two numbers a and b where a is a prefix of b. For example, the trie of Figure 5.5 can store the two words 91 and 911 by assigning 91 to the null entry of node A.

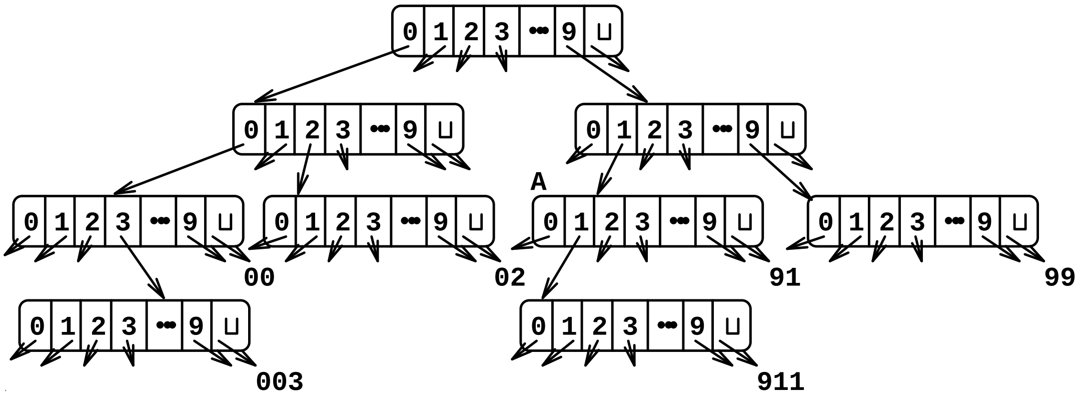

Figure 5.5. An Example trie data structure. The right-most entry of each node corresponds to a null symbol.

Searching for a word in the trie is simple. We start at the root and look up the first letter, and we follow the pointer next to the letter and look up the second letter in the word in the same way (see [Knu78] for a detailed description).

Notice that we can reduce the memory space of the trie structure (at the expense of running time) if we use a linked list for each node, since most of the entries in the nodes tend to be empty [dlB59]. This idea amounts to replacing the trie of Figure 5.5 by the forest of trees shown in Figure 5.6. Searching in such a forest proceeds by finding the root which matches the first letter in our input

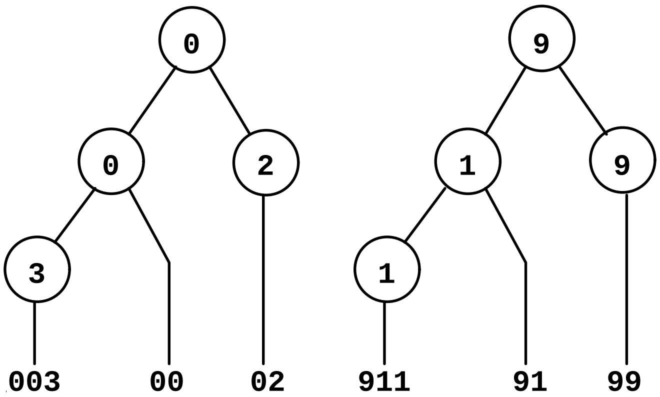

Figure 5.6. A forest of trees representing the trie of Figure 2.

word, then finding the son node of that root that matches the second letter, etc. It can be shown (see [Knu78]) that the average search time for N words stored in the trie is $\log_M N$ and that the "pure" trie requires a total of approximately N/lnM nodes to distinguish between N random words. Hence the total amount of space is MN/lnM.

Because of the high space complexity, the trie idea pays off only in the first few levels of the tree. It has been suggested in [Jr.63] that we can get better performance by mixing two strategies: using a trie for the first few characters of a word and then switching to some other technique, e.g., when we reach part of the tree where only, say, six or less words are possible, then we can sequentially run through the short list of the remaining words. As reported in [Knu78], this mixed strategy decreases the number of trie nodes by roughly a factor of six, without substantially changing the running time.

Consider a simple extension of the trie data structure where each letter is handwritten. Assume that we are given a handwritten cursive word w that is composed of a sequence of letters $l_1 l_2 \cdots l_L$, where L 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 best match between the letter l_i of w and the letters at level i of the tree. One problem with this approach is the difficulty in matching the individual letters in w with the letters in the trie. The reason is that it is difficult to handwrite a word twice in exactly the same way. As a result, a more elaborate matching method is needed.

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). As a result, in order to match and recognize a given input letter, we execute each of our alphabet HMMs and select the one that accepts the input letter with the highest probability. An example handwritten trie is given in Figure 5.7.

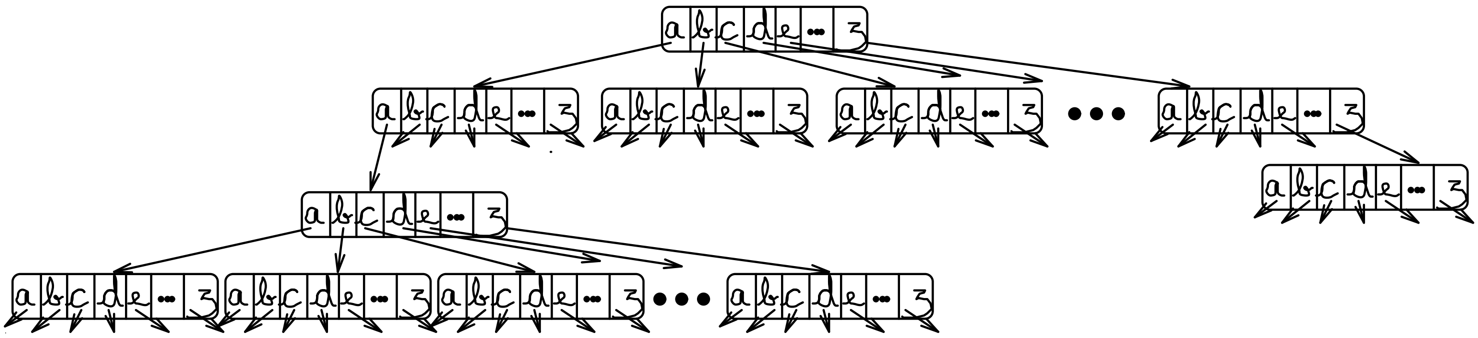

Figure 5.7. An example of a handwritten trie.

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, versus considering all possible letter combinations as in the level-building algorithm.

In order for the handwritten trie to function properly, two challenging issues have to be addressed:

1. **Cursive character segmentation**: since the input handwritten word is cursive, 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.
2. **Inter-character strokes**: 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.

In the following sections, we propose techniques to deal with each of these issues.

5.2.1 Cursive Character Segmentation. Given a cursive handwritten word w, our goal is to partition w into the point sequences $s_1 s_2 \cdots s_n$ so that each sequence can be used separately in the matching process while descending the handwritten trie. In this section, we provide several techniques that achieve this goal and hence has the same effect as character segmentation.

Using Counts of Minima and Maxima

One way to determine the point where one letter ends and another letter starts in a handwritten word is by counting the number of local minima and local maxima (in the values of the y coordinate

- the vertical direction) and the number of inflection points that are associated with each letter. For example (refer to Figure 5.8), the stroke information of the letter a contains three local minima and three local maxima and one inflection point. As a result, we store the number of local minima and maxima and inflection points with each letter in a given node of the trie. More specifically, (we

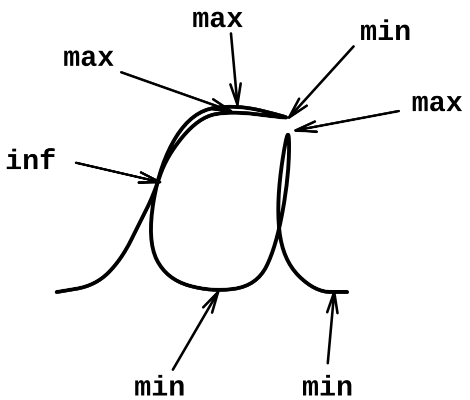

Figure 5.8. The handwritten letter "a" marked with the locations of the local maximum, local minimum, and inflection points.

use Figure 5.9 for illustration) a node n in the handwritten trie h contains f entries $e_1, e_2, \cdots, e_f$, where each entry e_i corresponds to a letter l_i and contains the following five fields: a pointer p_h to the HMM that corresponds to l_i, three values v_{min}, v_{max}, and v_{inf} that correspond to the number of local minima, local maxima, and inflection points in l_i, respectively, and a pointer p_c to a child node. The matching algorithm proceeds as follows:

1. given the input handwritten word w, start at the root node r,
2. for each entry e_i in r,
   a) scan w and retrieve into s the prefix of w that contains the same number of local minima and maxima and inflection points as in $e_i.v_{min}$, $e_i.v_{max}$, and $e_i.v_{inf}$.
   b) retrieve the HMM H pointed at by $e_i.p_h$.
   c) compute the probability$(s|H)$ and maintain the maximum.
3. Let e_k be the entry that corresponds to the maximum probability for all e_i in r.
4. assign to r the child node pointed at by $e_k.p_c$, i.e., $r \leftarrow e_k.p_c$.
5. discard the prefix s from w.
6. repeat the above process till all of w is consumed or a leaf node node is reached.
7. the letters that correspond to the path that is descended from the root node till the last visited node compose the resulting recognized word.

Figure 5.10 illustrates this procedure. Figure 5.10a shows an input word (the word "bagels") where it is segmented in Figure 5.10b into letters using the number of local minima, local maxima, and inflection points. Figures 5.10c and 5.10d shows how the trie is traversed where each level consumes a portion of the input word.

Utilizing the Most-likely HMM States

This method uses a modified version of the Viterbi algorithm [Vit67] to segment the characters in a cursive handwritten word. The modified Viterbi algorithm is used as a guide to partition the input pattern into character segments.

The main obstacle facing the Handwritten trie, is that we are not able to know the number of sample points T that we should consume by each HMM at any level in the tree. The main idea

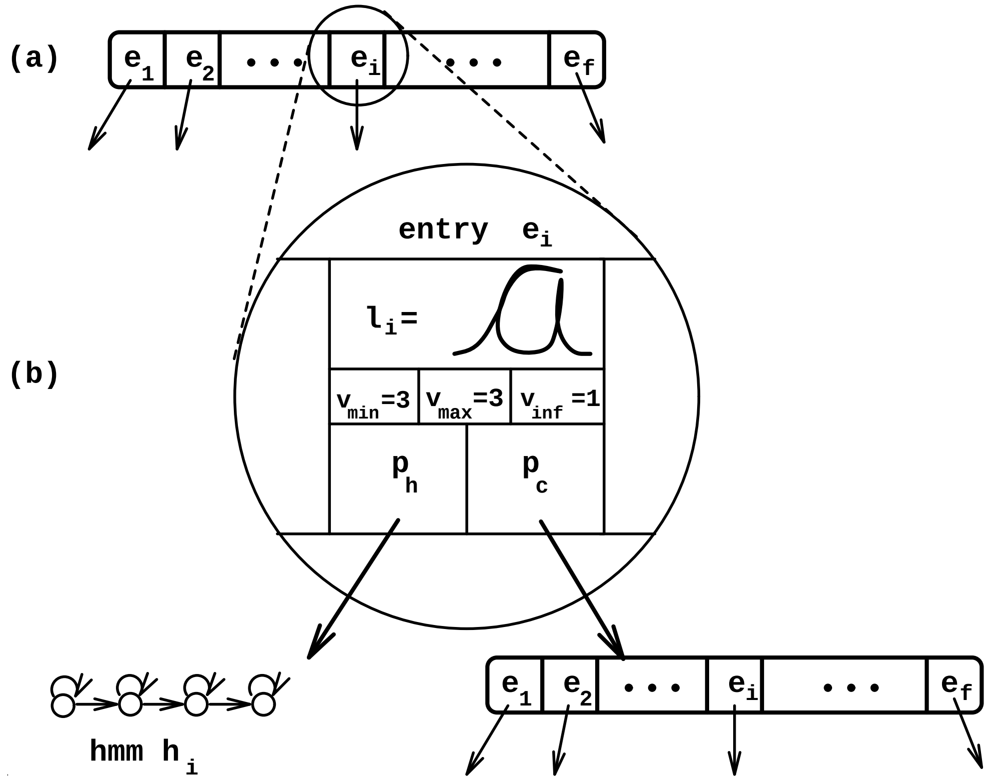

Figure 5.9. (a) A node of the handwritten trie (b) an example showing the fields in one entry of a node in the trie.

behind the new algorithm that we describe here is that we start at the root of the trie, and consider all possible pairs of letters that exist in the trie (refer to Figure 5.11 for illustration). For example, we combine the HMM of each letter in the root node with the HMM of each letter in the children nodes (as in Figure 5.12). For example, in Figure 5.11, we combine the HMMs of the root node r with the HMMs of the children nodes c_1, c_2, and c_4. This will result in the pairs:

$$
\begin{array}{l}
(H_{1a}, H_{2a}), (H_{1a}, H_{2b}), \cdots, (H_{1a}, H_{2z}) \\
(H_{1b}, H_{3a}), (H_{1b}, H_{3b}), \cdots, (H_{1b}, H_{3z}) \\
. \\
. \\
. \\
(H_{1z}, H_{4a}), (H_{1z}, H_{4b}), \cdots, (H_{1z}, H_{4z})
\end{array}
$$

Let H_l and H_r be two such pairs and H_{lr} be their combined version. We apply a variant of the Viterbi algorithm on H_{lr}, with w as input, in order to find out the two consecutive input points p_i and p_{i+1} of the input pattern w such that p_i is most likely to be the last point processed by the final state of H_l and p_{i+1} is most likely to be processed by the initial state of H_r. We save the index i as well as the probability *prob* associated with it. We apply this technique for all possible letter combinations in the root and the child nodes (as given above) and compute i and *prob* for each letter-pair. We follow that path that corresponds to the letter pairs with the highest probability.

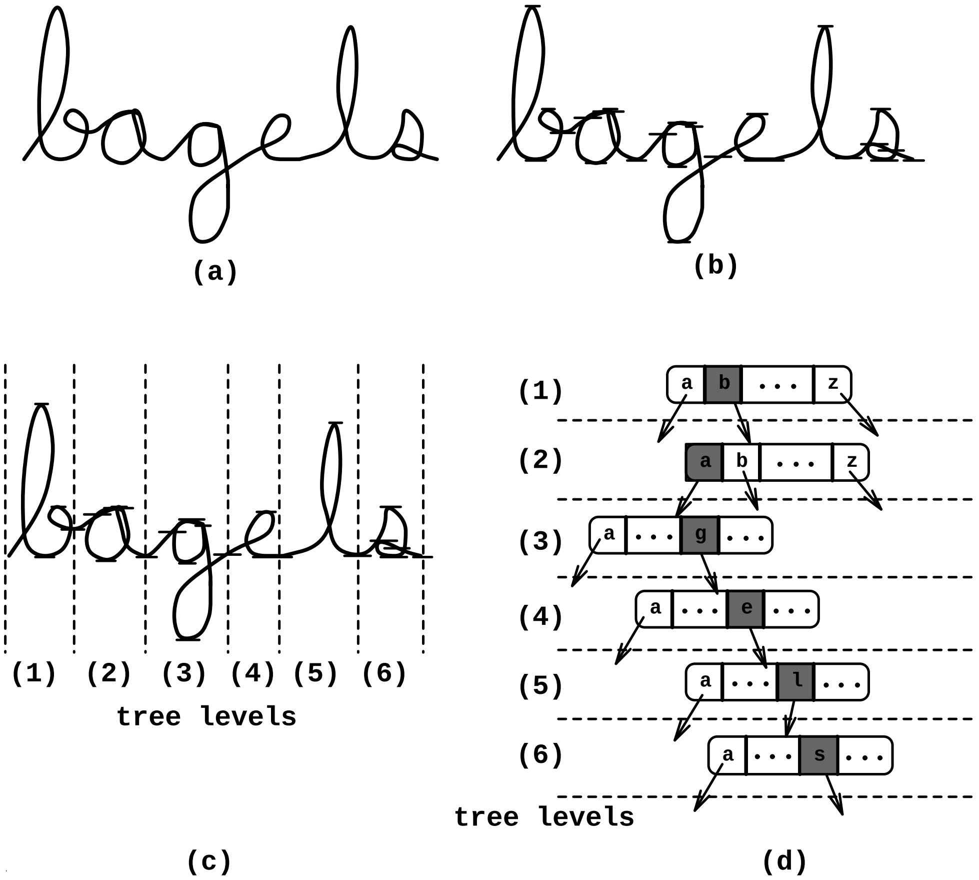

Figure 5.10. Example illustrating the execution of the algorithm. (a) an input word, (b) its segmentation, (c) the portions of the input word that are consumed by each level of the trie, and (d) the path from the root of the trie to the leaf during the recognition of the word "bagels".

For example, from Figure 5.11, if the pair (H_{1b}, H_{3a}) results in the highest probability value (*prob*), then we know that the first letter in the input word is most likely to be the handwritten letter *b* and we descend to node c_3 to repeat the same process after consuming the sample points in w that represent the letter *b*. These points are detected by the Modified Viterbi Algorithm (described in Section 5.2.3) and is maintained by the variable T_k in the procedure given below. In this case, we consume only the first T_k points of w, i.e., the ones that are generated by H_l only (in the example of Figure 5.11, H_l corresponds to H_{1b}). We repeat the same process starting with the child node that contains H_r, i.e., we descend to the child node that contains the second HMM in the HMM-pair that corresponds to the maximum *prob*. In the example of Figure 5.11, the algorithm proceeds to the child node c_3 that contains H_{3a} since the pair (H_{1b}, H_{3a}) corresponds to the maximum *prob*. The listing for the new recognition procedure using the handwritten-trie search is given below.

1. given the input handwritten word w, start at the root node r,
2. for each entry e_i in r,
   a) retrieve the HMM H_l pointed at by $e_i.p_h$
   b) let s be the child node pointed at by $e_i.p_c$
   c) for each entry $s.e_j$,
      i. retrieve the HMM H_r pointed at by $s.e_j.p_h$

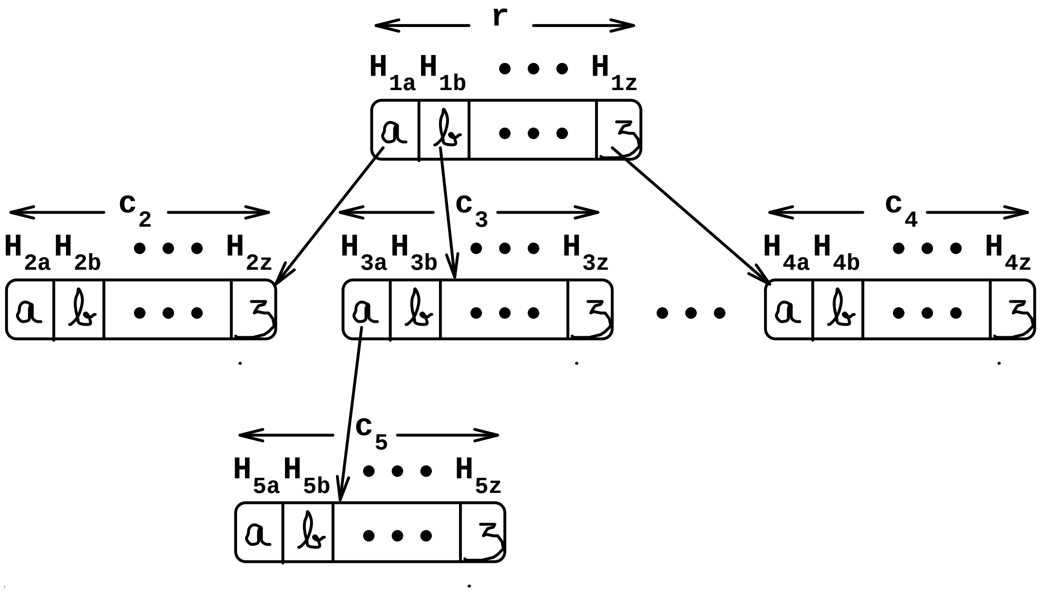

Figure 5.11. An example illustrating the new recognition procedure using the handwritten-trie and the modified-Viterbi algorithm.

ii. construct an HMM H_{lr} by simply concatenating the HMMs H_l and H_r (see Figure 5.12).
iii. apply the modified-Viterbi algorithm MV
$T_1, prob \leftarrow MV(H_{lr}, w)$, where T_1 is the number of points consumed from w (i.e., the prefix w_p of w of size T_1 points) and $prob$ is the probability that H_{lr} generates w_p.
iv. maintain the values s, j, and T_1 that correspond to max $prob$.

3. Let $r.e_k$ and $r.e_k.p_c.e_l$ be the two entries that corresponds to the maximum probability for all $r.e_i$ and all $r.e_i.p_c.e_j$, and let w_k be the prefix of w of length T_k that corresponds to the points consumed from w in this case.
4. assign to r the child node pointed at by $e_k.p_c$, i.e., $r \leftarrow e_k.p_c$.
5. discard the prefix w_k from w.
6. repeat the above process till all of w is consumed or a leaf node node is reached.
7. the letters that correspond to the path that is descended from the root node till the last visited node compose the resulting recognized word.

The modified-Viterbi algorithm remains to be explained. Before presenting the new modified-Viterbi algorithm, we start by a brief overview of the Viterbi algorithm.

5.2.2 The Viterbi Algorithm. The Viterbi algorithm [Vit67] is used to find the single best (or most likely) state sequence $q = (q_1 q_2 \cdots q_T)$, for the given observation sequence $O = (o_1 o_2 \cdots o_T)$. Define the quantity $\delta_t(i)$ to be the highest probability along a single path at time t which accounts for the first t observations and ends in state i, i.e.,

$$\delta_t(i) = \max_{q_1, q_2, \cdots, q_{t-1}} Prob(q_1 q_2 \cdots q_{t-1}, q_t = i, o_1 o_2 \cdots o_t) \tag{5.26}$$

This can be expressed recursively as:

$$\delta_t(j) = \max_{1 \leq i \leq N} [\delta_{t-1}(i) a_{ij}] b_j(o_t), \;\; 1 \leq j \leq N, \;\; 2 \leq t \leq T \tag{5.27}$$

$$\delta_1(i) = \pi_i b_i(o_1), \;\; 1 \leq i \leq N \tag{5.28}$$

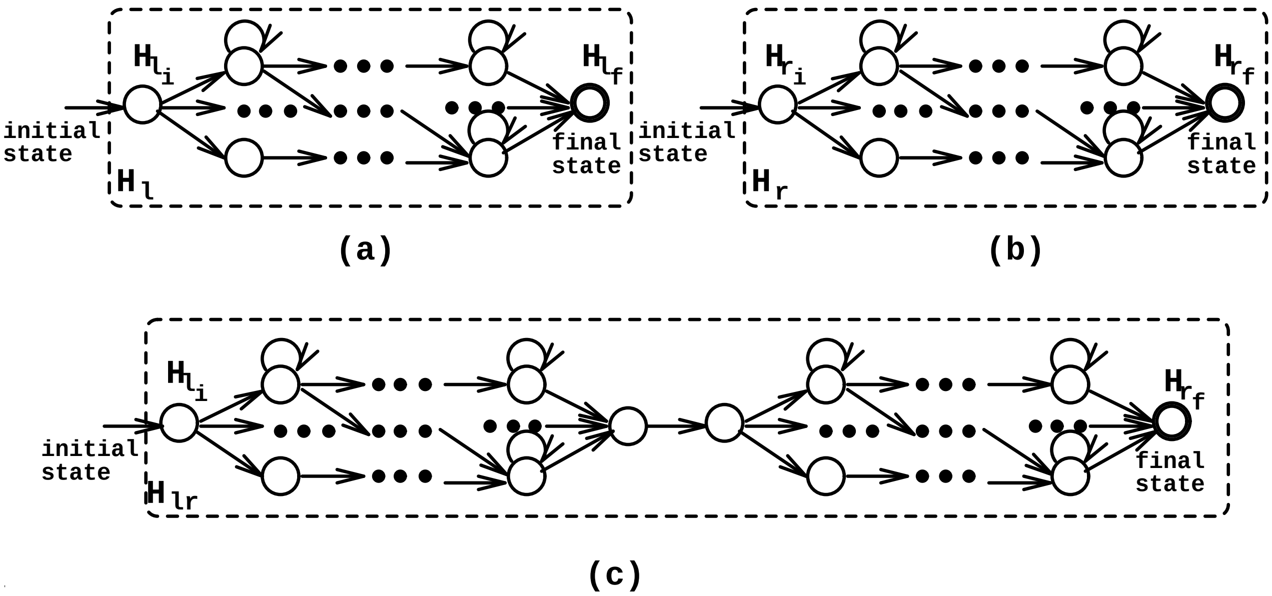

Figure 5.12. (a) The HMM H_l, (b) the HMM H_r, (c) the HMM H_{lr} is the concatenation of H_l and H_r.

To actually retrieve the state sequence, we use the array $\psi_t(j)$ to keep track of the state that maximizes Equation 5.27, Therefore,

$$\psi_t(j) = \arg \max_{1 \leq i \leq N} [\delta_{t-1}(i) a_{ij}], \; 1 \leq j \leq N, \; 2 \leq t \leq T \tag{5.29}$$

Now, in order to find the best state sequence, we first find the state with highest probability at the final stage, and then backtrack in the following way.

$$p^* = \max_{1 \leq i \leq N} [\delta_T(i)] \tag{5.30}$$

$$q_t^* = \arg \max_{1 \leq i \leq N} [\delta_T(i)] \tag{5.31}$$

$$q_t^* = \psi_{t+1}(q_{t+1}^*), \; t = T-1, T-2, \cdots, 1. \tag{5.32}$$

Figure 5.13 illustrates one application of the Viterbi algorithm for an HMM with four states and an input sequence of size 15 points. The dotted lines give the best state sequence at the intermediate stages, while the bold line gives the state sequence with the highest probability that is identified by the Viterbi algorithm.

5.2.3 The Modified-Viterbi Algorithm - Estimating *T*. Our goal is the following: We are given an HMM H_{lr}, which is a composition of the two HMMs H_l and H_r, and an input pattern w of length T points. H_l is assumed to be best at recognizing a prefix of w. The problem is that we do not know the length T_k of the prefix w_p For example, (refer to Figure 5.14), assume that we are given the input word "bagels" (Figure 5.14a), and a left-to-right HMM (Figure 5.14b) for recognizing the letter b (Figure 5.14c). Because the final state of the HMM (the right most state in the HMM of Figure 5.14b) contains a cyclic transition probability of value 1, all the input points past the letter b in the word "bagels" can be consumed in this state. The modified-Viterbi algorithm that we present in this Section addresses this problem. In other words, it identifies the point of the input word at which we stop, and hence isolating the letter b from the rest of the input word. It achieves this by appending another HMM (H_r) at the end of H_l (resulting in the HMM H_{lr}) and detect the point in

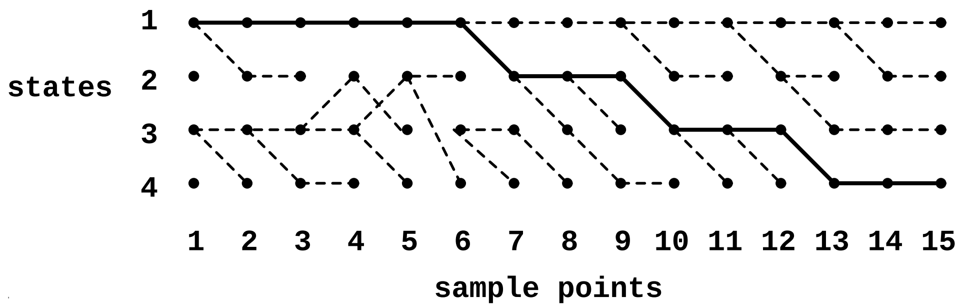

Figure 5.13. An example illustrating the execution of the Viterbi Algorithm. The dotted lines indicate the best state sequence at the intermediate stages, while the bold line indicates the overall best state sequence for the given input pattern.

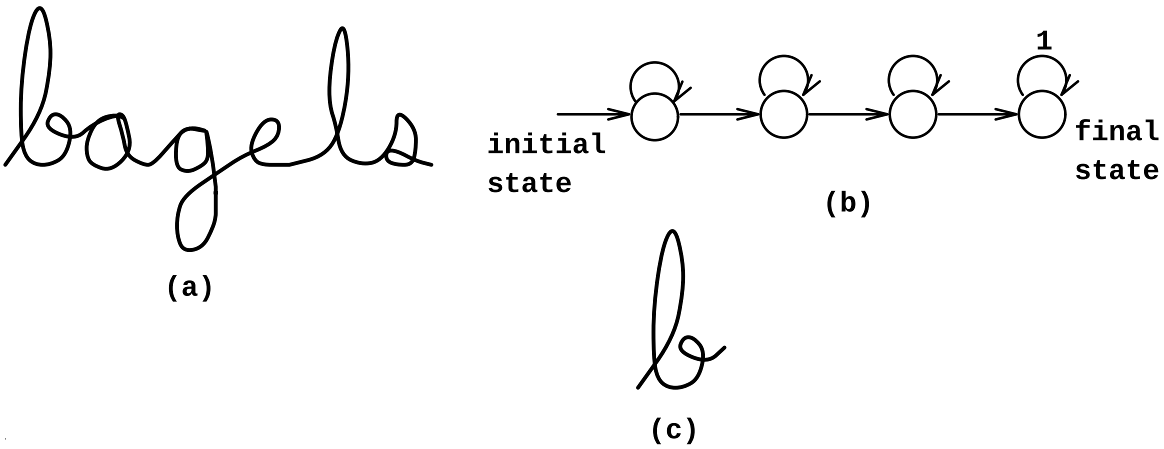

Figure 5.14. (a) An input word, (b) an HMM for recognizing the letter *b* given in (c).

time when it is best (according to the highest probability values generated) that a state transition (in H_{lr}) from the final state in H_l to the initial state in H_r takes place. This point is registered (T_k) and is returned by the algorithm. The procedure is applied repeatedly for the rest of the letters in the word, as explained in Section 5.2.1.

We make use of the following observations regarding the Viterbi algorithm.

1. the forward part of the Viterbi algorithm does not involve any backtracking, i.e., once a point t is processed and the best states were assigned to it, we do not backtrack at any point in computation to recompute or change these best-state assignments.
2. once we encounter the last input symbol of the tree, e.g., o_{T_k}, the way to figure out the best state sequence is to find the state with maximum probability and trace back the state sequences using the array ψ, as shown in Section 5.2.2.

Here, we show how we make use of these two properties of the Viterbi algorithm. Assume that we are given two HMMs H_l and H_r, where each HMM has one initial and one final states and that we concatenate the two HMMs to produce a new HMM H_{lr} (see Figure 5.12) by adding a transition from the final state of H_l to the initial state of H_r. Notice that some probability value has to be assigned to the newly added transition. For example, in left-to-right models, this can be achieved by changing the transition probability of the final state of the left HMM (H_l), as given in Figure 5.15. Let the final state of H_l be H_{l_f} and the initial state of H_r be H_{r_i}.

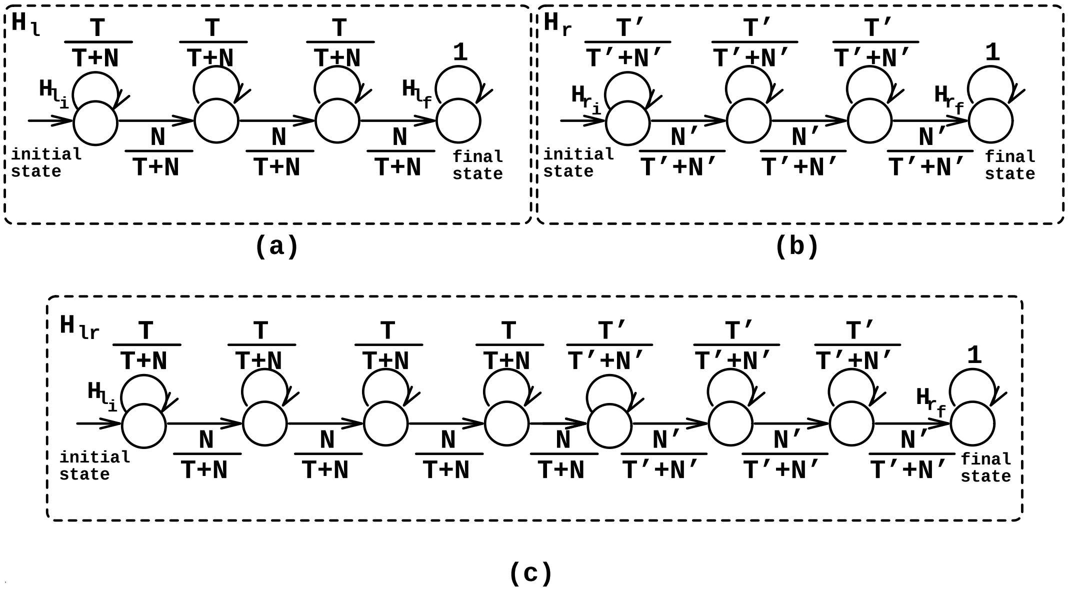

Figure 5.15. (a) The left-to-right HMM H_l, (b) the left-to-right HMM H_r, (c) the HMM H_{lr} is the concatenation of H_l and H_r.

In order to find the value of T_k, we apply the Modified Viterbi algorithm, which is the same as the Viterbi algorithm except that for each iteration that involves one of the T input symbols, we monitor the probability values of the two states H_{l_f} and H_{r_i} until for some $t = T_k, 2 \leq T_k l T$, the following conditions are satisfied (in the given order):

1.
$$H_{l_f} = \arg \max_{1 \leq i \leq N} [\delta_{T_k - 1}(i) a_{ij}], \ 1 \leq j \leq N \tag{5.33}$$

2. for $t = T_k + 1$ (i.e., the next input point),
$$H_{r_i} = \arg \max_{1 \leq i \leq N} [\delta_{T_k}(i) a_{ij}], \ 1 \leq j \leq N \tag{5.34}$$

Once these two conditions are satisfied for some $t = T_k$, then we stop the algorithm and return the value T_k which indicates that the input points $o_1, o_2, \cdots, o_{T_k}$ are the prefix of w that are supposed to be recognized by the HMM H_l.

We plan to investigate the performance of the two techniques for character segmentation, presented in this paper, in the implementation phase of the handwritten-trie.

5.3 Inter-character Strokes

In cursive writing, some additional strokes are introduced to interconnect the handwritten characters. The shape of these strokes depends on the letters that are to be connected, i.e., the letters to the left and to the left of the connecting stroke. We discuss briefly how we can deal with them in the handwritten trie.

One way of dealing with inter-character strokes is to allow for some input points (some constant number) to be skipped between the end of one letter and the start of the next letter. These skipped points will not be considered in the HMM probability computation.

A second approach to dealing with this problem is to change the nodes of the trie so that they reflect pairs of already connected characters instead of single characters. In addition, letters in children nodes overlap with their parent node in one character. E.g., the word bagels will be stored in the handwritten-trie nodes as: ba, ag, ge, el, and ls. This way, the inter-character strokes are incorporated into the tree search. We plan to investigate both of these two techniques in the implementation phase of the handwritten trie.

5.4 Performance

We have build an initial prototype of the Handwritten Trie in main memory. Initial results show that we can accommodate up to 18,000 pictograms in less than one million bytes of memory (including the space taking by the index and the HMM representations), using an alphabet of 26 symbols (roman characters). The matching rate of the index is better than 90%.

Figure 5.16 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.

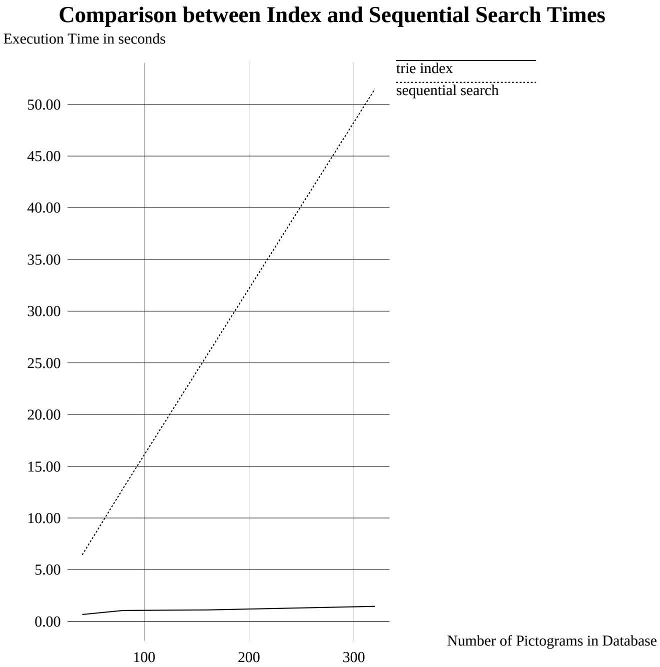

Figure 5.16. A comparison between the matching time using our indexing technique versus using a sequential algorithm. The x-axis corresponds to various database sizes.

6. Conclusions

We have presented several techniques of indexing large repositories of pictograms. Preliminary results show that the index helps drastically in reducing the search time, when compared to sequential searches. The results show search times on the order of 2 seconds for database sizes up to 150,000 words (running on a 40MHz NeXT workstation).

We are currently experimenting with these techniques to implement both main memory and disk-based implementations of ink databases. In doing so, we hope to obtain a better understanding of the issues involved in handling large volumes of pictograms.

7. Acknowledgments

The *Moby-Dick* text we used in our experiments was obtained from the Gutenberg Project at the University of Illinois, as prepared by Professor E. F. Irey from the Hendricks House edition.

TSW90

[AB94] 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.

[AS90] W. G. Aref and H. Samet. Efficient processing of window queries in the pyramid data structure. In *Proceedings of the 9th. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 265–272, Nashville, TN, April 1990. (also in Proceedings of the Fifth Brazilian Symposium on Databases, Rio de Janeiro, Brazil, April 1990, 15–26).

[AVB94] Walid G. Aref, Padmavathi Vallabhaneni, and Daniel Barbará. On training hidden markov models for recognizing handwritten text. In *Fourth International Workshop on Frontiers of Handwriting Recognition*, Taipei, Taiwan, December 1994.

[Bar93] Daniel Barbará. Method to index electronic handwritten documents. Technical Report MITL-TR-77-93, Matsushita Information Technology Laboratory, Princeton, NJ, November 1993.

[BK92] C.B. Bose and S. Kuo. Connected and degraded text recognition using Hidden Markov Models. In *International Conference on Pattern Recognition*, 1992.

[dlB59] Rene de la Briandais. File searching using variable length keys. In *Proceedings of the Western Joint Computer Conference*, pages 295–298, 1959.

[For73] G. D. Forney. The Viterbi Algorithm. *Proceedings of the IEEE*, 61, 1973.

[Fre60] E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–500, 1960.

[Jr.63] E. H. Sussenguth, Jr. Use of tree structures for processing files. *Communications of the ACM*, 6:272–279, 1963.

[Knu78] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, MA, 1978.

[LBG80] Yoseph Linde, Andres Buzo, and Robert M. Gray. An algorithm for vector quantizer design. *IEEE Transactions on Communications*, COM-28, No 1:84–95, 1980.

[LRS83] 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.

[LT92a] D. P. Lopresti and A. Tomkins. Applications of Hidden Markov Models to Pen-Based Computing. Technical Report MITL-TR-32-92, M.I.T.L, November 1992.

[LT92b] D. P. Lopresti and A. Tomkins. Pictographic Naming. Technical Report MITL-TR-21-92, M.I.T.L, August 1992.

[LT93a] 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.

[LT93b] Daniel Lopresti and Andrew Tomkins. Approximate matching of hand-drawn pictograms. In *Proceedings of the Third International Workshop on Frontiers in Handwriting Recognition*, pages 102–111, Buffalo, NY, May 1993.

[LT93c] Daniel Lopresti and Andrew Tomkins. Pictographic naming. In *Adjunct Proceedings of the 1993 Conference on Human Factors in Computing Systems (INTERCHI'93)*, pages 77–78, Amsterdam, the Netherlands, April 1993.

[LT94] Daniel P. Lopresti and Andrew Tomkins. On the searchability of electronic ink. In *Proceedings of the Fourth International Workshop on Frontiers in Handwriting Recognition (to appear)*, Taipei, Taiwan, December 1994.

[Mac67] J. MacQueen. Some methods for classification and analysis of multivariate observations. *Proceedings of the Fifth Berkeley Symposium on Mathematics, Statistics and Probability*, 1:281–296, 1967.

[Rab89] 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.

[Rub91] Dean Rubine. *The Automatic Recognition of Gestures*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1991.

[Sch92] Robert Schalkoff. *Pattern Recognition. Statistical, Structural and Neural Approaches*. John Wiley & Sons, Inc, 1992.

[Sla93] Slate Corporation, Scottsdale, AZ. *JOT: A Specification for an Ink Storage and Interchange Format (ver. 1.0)*, 1993.

[SM83] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1983.

[TP75] S. Tanimoto and T. Pavlidis. A hierarchical data structure for picture processing. *Computer Graphics and Image Processing*, 4(2):104–119, June 1975.

[TSW90] C. Tappert, C. Y. Suen, and T. Wakahara. The state of the art in on-line handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(8), August 1990.

[Tuc84] L. Tucker. *Computer Vision Using Quadtree Refinement*. PhD thesis, Polytechnic Institute of New York, Brooklyn, May 1984.

[Vit67] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, IT-13:260–269, 1967.

[WF74] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the Association for Computing Machinery*, 21(1):168–173, 1974.