A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words

J. R. Ullmann*

Division of Computer Science, National Physical Laboratory, Teddington, Middlesex, TW11 0LW

An n-gram is an n-character subset of a word. Techniques that are already well known use n-grams for detecting and correcting spelling errors in words. This paper offers three basic contributions to n-gram technology. First, a method of reducing storage requirements by random superimposed coding. Second, an n-gram method for finding all dictionary words that differ from a given word by up to two errors. Third, an n-gram method for correcting up to two substitution, insertion, deletion and reversal errors without doing a separate computation for every possible pair of errors.

(Received November 1975)

1. Introduction

Human writers and keyboard operators sometimes introduce

substitution errors, in which one character is erroneously substituted for another,

insertion errors, in which a character is erroneously inserted,

deletion errors, in which a character is erroneously omitted, and

reversal errors, in which the positions of adjacent characters are erroneously interchanged.

Optical character recognition (OCR) machines do not introduce reversal errors, and usually do not introduce insertion and deletion errors. They do introduce substitution errors and also reject errors. A reject error is usually represented by a special symbol which indicates that the OCR machine has been unable to recognise the corresponding input character confidently.

Suppose we have a dictionary that lists all correctly spelt words which ever occur in a given text processing application. Spelling or recognition errors may change one word in the dictionary into another word in the dictionary, and we shall not be able to detect such errors just by using the dictionary. Very much more commonly, errors will change a word into a word that is not in the dictionary. To correct such errors it seems sensible to

(*a*) find the dictionary word differing from the given word by fewest errors, and

(*b*) replace the given word by the dictionary word found in (*a*).

To carry out step (*a*) we can use a string matching procedure for determining the number of errors by which two given words differ. If more than one word is found in (*a*), then syntactic or semantic disambiguation is required.

The literature on spelling correction and string matching is so big and diffuse that we cannot hope to review it usefully here in a few paragraphs. Section 8.1 of Ullmann (1973) provides an elementary introduction, and further references are given by Riseman and Hanson (1974). The following preliminary paragraphs are confined to dynamic programming techniques that provide the most sophisticated of the known techniques of string matching.

The string matching algorithm of Wagner and Fischer (1974) is similar to the dynamic programming algorithms of Vintsyuk (1968), Velichko and Zagoruyko (1970), and Sakoe and Chiba (1971). To match an m_1-character word $x_1, \ldots, x_i, \ldots, x_{m_1}$ against an m_2-character word $x'_1, \ldots, x'_j, \ldots, x'_{m_2}$, the algorithm of Wagner and Fisher processes $x_1, \ldots, x_i, \ldots, x_{m_1}$ in turn. Corresponding to x_i the algorithm computes m_2 scores $D_{ij}, j = 1, \ldots, m_2$, according to

$$D_{ij} := \min(f_1, f_2, f_3) ,$$

where

$$f_1 := D_{i-1,j-1} + \text{mismatch score for } x_i \text{ versus } x'_j;$$

$$f_2 := D_{i-1,j} + \text{penalty score for deleting } x_i;$$

$$f_3 := D_{i,j-1} + \text{penalty score for inserting } x'_j .$$

The overall mismatch score for the two words is taken to be

$$\min_j \{D_{m_1 j}\} .$$

This procedure is superior to that proposed by Ellis (1969) and developed by Warren (1972) in that it does not fallaciously assume that if x_{i-1} corresponds to x'_{j-1} in an optimal matching then the most similar of the three pairs (x_{i-1}, x'_j), (x_i, x'_{j-1}), (x_i, x'_j) correspond. Similarly, in the Markovian case, the Viterbi algorithm (Forney, 1973) is superior to the contextual algorithm of Denes (1959) in that it does not fallaciously assume that x_i should be optimally assigned to the r_ith recognition class such that $P(r_i/r_{i-1})$. $P(x_i/r_i)$ is maximal when x_{i-1} has been assigned to the r_{i-1}th recognition class. Abend (1968) pointed out theoretically, and Hanson *et al.* (1974) have confirmed experimentally, that errors tend to propagate when an algorithm such as that of Denes (1959) is used. The Viterbi algorithm is a simple dynamic programming algorithm that does not have this disadvantage, just as the algorithm of Wagner and Fischer does not have the disadvantage of the algorithm of Ellis (1969).

The algorithm of Wagner and Fischer, and Lowrance and Wagner's (1975) extension of it to cope with reversal errors, takes a time proportional to the product $m_1 . m_2$. This may be too slow if the dictionary has many thousands of entries and many successive input words have to be processed. The method of Szanser (1973) may reduce the computing time at the expense of increasing the storage area occupied by the dictionary.

The correction process may be speeded up by the use of special purpose hardware. To introduce the main work of the present paper we now outline a very simple hardware technique for determining the number of errors by which two given words differ.

2. A simple comparator for words

By way of example we shall throughout the remainder of this paper consider a problem in which

(*a*) every word in the dictionary has exactly six letters,

(*b*) if two words differ by exactly one substitution, insertion, deletion, or reversal error, we shall say that these two words differ by exactly one error. If two words differ by two

*Now at: Department of Applied Mathematics and Computing Science, The University, Sheffield S10 2TN.

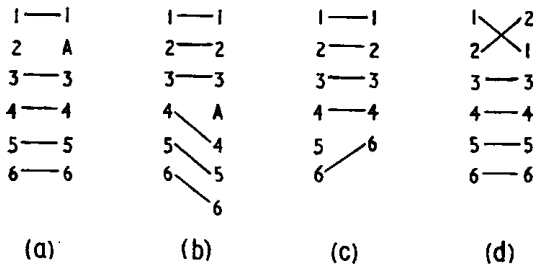

Fig. 1 Examples of one-error linesets

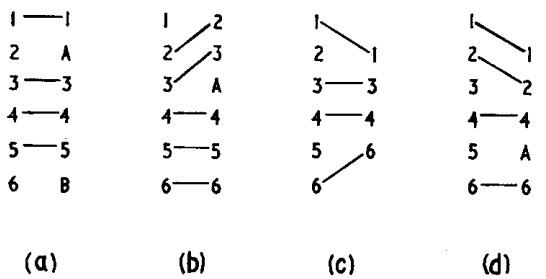

Fig. 2 Examples of two-error linesets

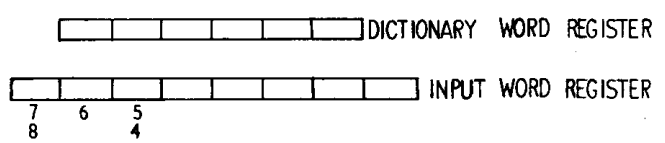

Fig. 3 Subdivision of word registers into character fields

errors that may be substitution, insertion, or deletion errors, or any combination of two such errors, then we shall say that two words differ by exactly two such errors. If two words differ by a reversal error and also by any other error whatsoever, then we shall say that the two words differ by more than two errors (because, in a sense, one reversal error is equivalent to two substitution errors).

(*c*) if two words differ by more than two errors we shall not determine the number of errors,

(*d*) each of the letters in a word may be one of the 26 letters of the alphabet, represented for our purposes by a 5-bit integer in the range 0 through 25. We shall also use numerals as symbols standing for letters.

In **Fig. 1**(*a*), (*b*), (*c*), (*d*), lines show which characters correspond in pairs of words differing by one error. We call the set of lines in Fig. 1(*a*) a *lineset*, and we use the same term for sets of lines such as those shown in Fig. 1(*b*), (*c*), and (*d*). **Fig. 2**(*a*), (*b*), (*c*) and (*d*) illustrate linesets for pairs of words differing by two errors. (A *lineset* differs from a *trace* in Wagner and Fischer (1974) in that no line in a lineset links nonidentical characters, whereas this is not true of a trace). Let L_1 be the set of all possible linesets of not more than one error, and let L_2 be the set of all possible linesets of not more than two errors. By enumeration we find that L_1 and L_2 have 25 and 174 members respectively.

In hardware we propose to enter a six-character dictionary word into a register that comprises six 5-bit character fields, as indicated in **Fig. 3.** An input word to be compared with this dictionary word is stored in a register that comprises eight 5-bit character fields. We reject any input word of more than eight or less than four characters because it must contain more than two errors. Otherwise we position the leftmost character of the input word in one of the eight fields selected in accordance with the number of characters in the word. The numerals under the input register fields in Fig. 3 show the word lengths for which the leftmost character is in the indicated field. Adjacent characters are entered into adjacent fields. This arrangement has been chosen so that if (i, j) belongs to any one-error or two-error lineset, then $i = j$ or $i = j \pm 1$.

For each $i = 1, \ldots, 6$, three separate logic circuits determine respectively whether $x'_i = x_{i-1}$, whether $x'_i = x_i$, and whether $x'_i = x_{i+1}$. The outputs of these 3×6 circuits provide inputs to two sets, L'_1 and L'_2 of *and* gates. The set L'_1 contains one *and* gate corresponding to each member of L_1, and there is a similar 1:1 correspondence between L'_2 and L_2. The connections are such that an *and* gate in L'_1 is activated iff the input word differs from the dictionary word by not more than one error; and an *and* gate in L'_2 is activated if the input word differs from the dictionary word by not more than two errors.

To find all dictionary words differing by not more than one or by not more than two errors from a given input word, we could apply each dictionary word in turn to the simple hardware comparator. This procedure might take too long if the dictionary were large and if many input words were to be processed successively. The main aim of our exploratory work is to reduce this time by using binary *n*-grams instead of a full dictionary.

3. Binary *n*-grams

Suppose that a dictionary word is held in the 6-field register in Fig. 3. A subset comprising n of these fields is an n-tuple. An n-gram is an assignment of one character to each of the n fields that constitute an n-tuple. For instance, 1234, 1356, 1245, are 4-grams of 123456. (This conforms with Shannon's (1951) usage of the term n-gram, but not with that of Riseman and Hanson (1974).)

In n-gram techniques of Thomas and Kassler (1967), Riseman and Hanson (1974), and Balm (1975), a collection of n-tuples are preselected. Data derived from a dictionary is stored as follows in a storage area that is initially cleared (i.e. set to all 0's). For each dictionary word in turn, the n-grams on the preselected n-tuples are determined. For each of these n-grams in turn a '1' is entered (inclusively *or*ed into) a bit location whose address can be computed by

$$26^n \mu + 26^{n-1} x_1 + 26^{n-2} x_2 + \ldots + x_n \qquad (1)$$

where $x_1, x_2, \ldots, x_n$ are 5-bit integers representing respectively the n characters of the n-gram. For the value of μ there are two principal possibilities. One is to set $\mu = 0$ for all n-tuples, and Riseman and Hanson (1974) call this the *non-positional* case. The other is to set $\mu = 0$ for the first n-tuple, $\mu = 1$ for the second, $\mu = 2$ for the third, and so on, so that a separate array of 26^n bits of store is associated with each different n-tuple. Riseman and Hanson call this the *positional* case.

Riseman and Hanson assume that any given 6-letter word that is not in their dictionary is misspelt. Following this assumption, we could test for spelling errors in a word by looking it up in the dictionary. Alternatively we can test whether a given word is *not* in the dictionary by using the stored n-gram data as follows. For each preselected n-tuple, obtain the n-gram from the given input word. For each of these n-grams, use (1) to address one bit. If any such bit is 0 then the given input word cannot possibly be in the dictionary. Riseman and Hanson (1974) claim that their n-gram methods for correcting detected substitution errors are faster than dictionary methods.

To cope with deletions and insertions, Hanson *et al.* (1974) propose to try deleting each character in turn, to try inserting characters in various positions, and to apply n-gram techniques for each such trial separately. This whole process could be slower than the simple technique outlined in Section 2 of the present paper.

To gain speed, at least in hardware implementation, Section 7 of this paper introduces a technique in which, loosely speaking, the method of Section 2 is applied separately to n-tuples that

are fitted together so as to cope with whole words. The possibility of time saving arises when the number of words in the dictionary is very much greater than the number of distinct n-grams per n-tuple that occur in dictionary words. The substantially increased logical complexity is to some extent offset by the fact that, for example, the number (actually 91) of 1-error and 2-error linesets of a 4-tuple is less than 174, the number of linesets in L_2. Sections 5 and 6 of this paper introduce preliminary steps towards Section 7, and Section 4 describes a storage economy technique that has made it possible for us to work experimentally with 4-grams in computer simulation.

4. Reduction of n-gram storage requirements by random superimposed coding

To work with 15 positional 4-tuples using formula (1) we would have required $15 \times 26^4 = 6{,}854{,}640$ bits. To reduce this we have adapted Mooers' (1951) technique of random superimposed coding. The adaptation is similar in principle, but different in detail, from that described by Ullmann (1971).

Instead of 15×26^4 bits, we use $14\lambda + 876$ 48-bit words, where λ is an experimentally determined integer less than 877. We denote the ith of these 48-bit words by $S(i)$. We also use a further 840 48-bit words and denote the ith of these by $Z(i)$. Initially each of these 840 words is set to contain b 1's and 48-b 0's, the 1's being in randomly chosen positions, except that no two of the 840 words are identical. We use 48-bit words because this is the word width of the KDF9 computer used in our experiments.

The store S is cleared initially, and is then set up as follows for use in error detection and correction. For each 6-letter word in the dictionary, for each of the 15 4-tuples do

$$S(\mu\lambda + 32x_1 + x_2 + x_3 + x_4) :=$$
$$S(\mu\lambda + 32x_1 + x_2 + x_3 + x_4) \vee Z(\mu + 32x_3 + x_4) \quad . \quad (2)$$

In this the symbol '$\vee$' denotes the inclusive *or* operation on 48-bit words. For the first, second, . . ., fifteenth n-tuples we set $\mu = 0, 1, \ldots, 14$ respectively.

Subsequently the computer decides that a given 6-letter word is in the dictionary iff for every one of the 15 4-grams in the given word it is found to be true that

$$Z(\mu + 32x_3 + x_4) = Z(\mu + 32x_3 + x_4) \,\&$$
$$S(\mu\lambda + 32x_1 + x_2 + x_3 + x_4) \quad (3)$$

where '&' denotes collation of 48-bit words.

In the S address, $x_3 + x_4$ is added to ensure that the number of bits per word is roughly the same for all words in S. If $x_3 + x_4$ were not added and if $\lambda > 875$ then some words of S would remain all 0's because some x_1x_2 digrams never occur in dictionary words, and because x_1 is multiplied by 32 instead of 26. In the Z address the addition of μ ensures that a different Z word is used for each n-tuple in which x_3 and x_4 are the same two characters. Experimentally we find that omission of μ slightly worsens the failure rate of the system, and an analogous result is reported in (Ullmann, 1971). In the S and Z addresses multiplication by 32 is used instead of multiplication by 26 because it can be implemented by a 5-bit shift. Because of this use of shift, the computing time for implementing (2) or (3) may not be much greater than that for (1).

To evaluate this technique we used the 2,755-word dictionary of Riseman and Hanson (1974) to set up S using (2). We then generated six test-words from each of the 2,755 words as follows. The first of the six test-words was generated by replacing the first character by a character randomly chosen except that it was different to the original first character, and the rest of the word was unchanged. The other five test-words were generated by doing the same thing with the remaining five characters in the dictionary words. Since six test-words were

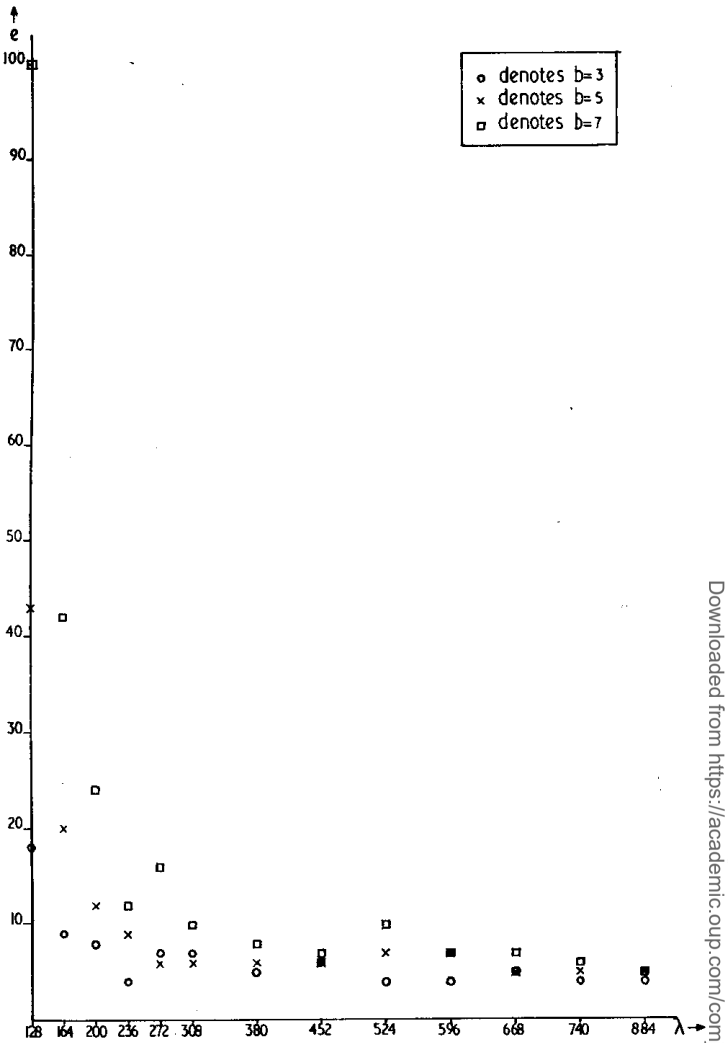

Fig. 4(*a*) Plot of e versus λ

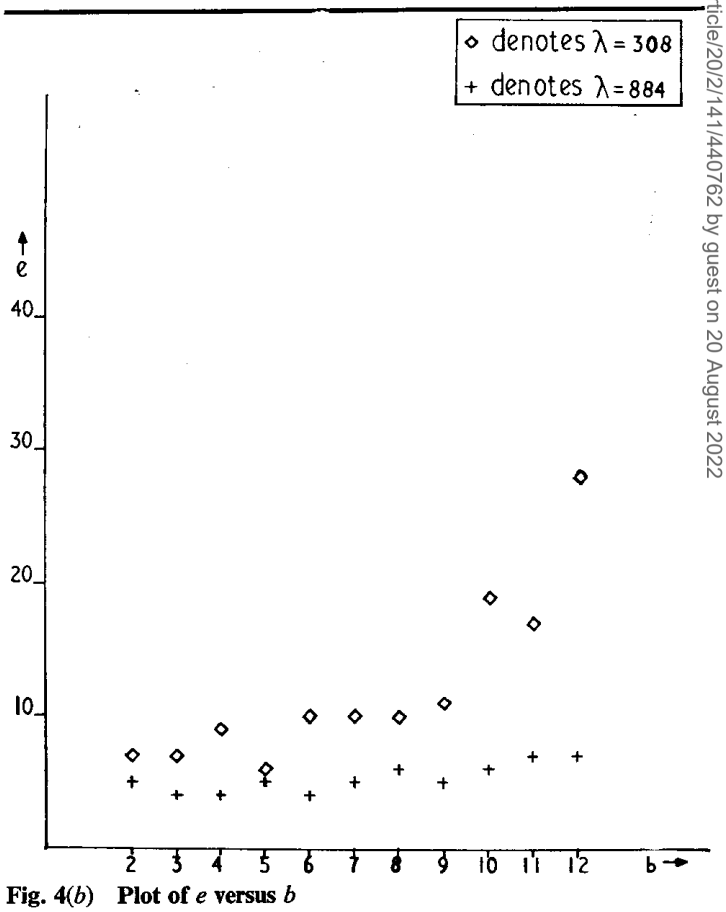

Fig. 4(*b*) Plot of e versus b

generated from each of the 2,755 dictionary words, the total number of test-words was $2,755 \times 6 = 16,530$. Of these, 113 were found to be identical to words in the dictionary.

For each of the remaining $16,530 - 113 = 16,417$ words, we tested condition (3) for all 4-grams. Let e be the number of words in this set for which (3) was true for all 4-grams. The system erroneously decided that these e words were in the dictionary. **Fig. 4**(*a*) plots e versus λ for $b = 3$, 5, and 7 bits per word in Z; and Fig. 4(*b*) plots e versus b for $\lambda = 308$ and $\lambda = 884$. For the 20 positional 3-grams, Riseman and Hanson used $20 \times 26^3 = 351,520$ bits, which is more than the $48 \times (14 \times 308 + 875 + 840)$ bits that our system uses when $\lambda = 308$, and in this case our error detection rate when $b = 3$ is

$$100 - \frac{100 \times (113 + 7)}{16,530} = 99{\cdot}2\%$$

c.f. Table IV in Riseman and Hanson (1974). In our system the tradeoff between error detection and storage area can be quite finely controlled by the choice of λ, as Fig. 4(*a*) shows.

5. Assembling words from confusion sets

Let $C_i = \{x_{i1}, \ldots, x_{ij}, \ldots, x_{i|C_i|}\}$ be a set of $|C_i|$ alternative characters for the ith character-position of a 6-letter word. C_i is called a *confusion set* (Ehrich and Koehler, 1975; Balm, 1975). For later reference we now consider the problem of using minimal computing time in determining the set $W(C_1, \ldots, C_i, \ldots, C_6)$ of all 6-letter words that belong to a given dictionary and whose first letter is in C_1, second letter is in C_2, and so on for all six letters. There are $|C_1|.|C_2|.|C_3|.|C_4|.|C_5|.|C_6|$ ways of choosing one character from each of $C_1, C_2, \ldots, C_6$ (the order of $C_1, C_2, \ldots, C_6$ is fixed). Instead of looking up each such combination in the dictionary, we use n-grams to ensure that many of the combinations not in $W(C_1, \ldots, C_6)$ are not actually generated (*c.f.* Cherry and Vaswani, 1961).

Let N_i be the set of n-tuples that include the ith character field. Let us say that the character fields constituting an n-tuple in N_i are indexed $i, k_2, k_3, \ldots, k_n$. On an n-tuple in N_i let $W(x'_{ij}, C_{k_2}, \ldots, C_{k_n})$ be the set of all n-grams that

(*a*) have occurred in at least one dictionary word. (The method of Section 4, using (3), can be used to determine whether any given n-gram has occurred in at least one dictionary word). And

(*b*) have x'_{ij} as one character, the other $n - 1$ characters being in $C_{k2}, \ldots, C_{k_n}$ respectively.

Let X' be any word in $W(C_1, \ldots, C_6)$ that contains x'_{ij}. From our construction it readily follows that the n-gram in X' on an n-tuple in N_i belongs to $W(x'_{ij}, C_{k_2}, \ldots, C_{k_n})$ for that n-tuple. Therefore x'_{ij} belongs to at least one word in $W(C_1, \ldots, C_6)$ only if:

$W(x'_{ij}, C_{k_2}, \ldots, C_{k_n})$ is non-empty for every n-tuple in N_i . (4)

If x'_{ij} does not satisfy this condition, then there is no advantage in our considering combinations of letters of $C_1, \ldots, C_6$ that include x'_{ij}. Instead we can remove x'_{ij} from C_i, thus reducing the number of combinations that need be processed subsequently in the search for the members of $W(C_1, \ldots, C_6)$.

The *refinement procedure* is a subroutine that applies (4) to every character in each of $C_1, \ldots, C_6$ and removes any character for which (4) is false. An Appendix, below, gives implementational details. If removal of characters leaves any of $C_1, \ldots, C_6$ empty, then the refinement procedure takes its FAIL exit, and otherwise it takes its SUCCEED exit. **Fig. 5** is a flowchart for an algorithm that uses the refinement procedure in the determination of $W(C_1, \ldots, C_6)$ for given initial confusion sets $C_1, \ldots, C_6$. When characters are removed from C_i we still use the symbol C_i to denote the remaining set of characters.

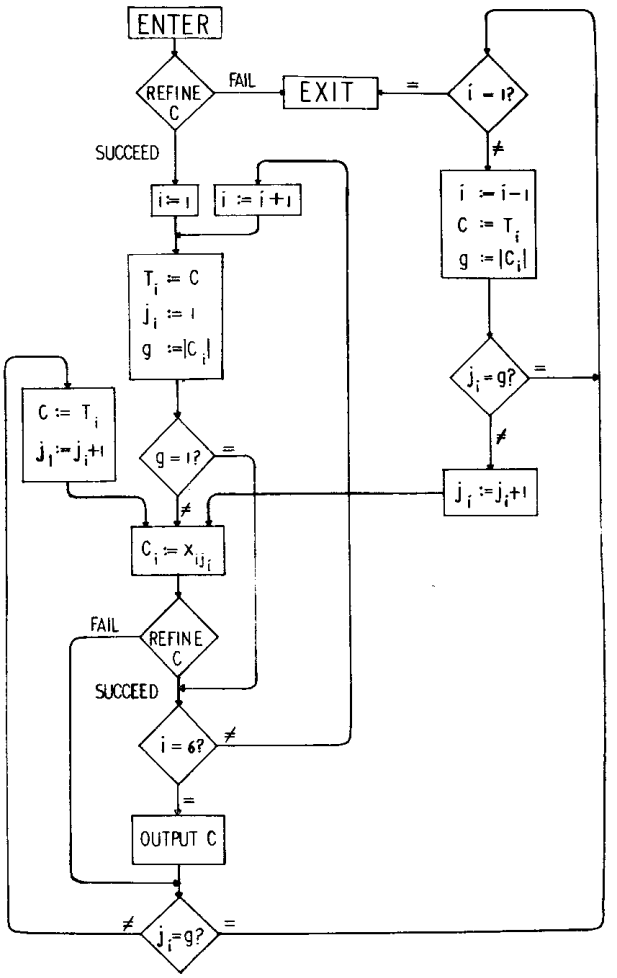

Fig. 5 Backtrack algorithm for finding words

The Fig. 5 algorithm starts by applying the refinement procedure to the initial $C_1, \ldots, C_6$. Then it eliminates all but the first character in C_1, applies the refinement procedure to remove further characters, then eliminates all but the first remaining character in C_2, applies the refinement procedure, and so on until possibly $C_1, \ldots, C_6$ each contain one character. In this case, if the refinement procedure is applied and reaches its SUCCEED exit, then the six characters in $C_1, \ldots, C_6$ constitute one of the words that we are seeking, and this word should be output. On the other hand, if the FAIL exit of the refinement procedure is reached at some stage, then the algorithm eliminates all but a different one of the members of the most recently processed C_i; except that if all members of this C_i have already been tried in this way, then the algorithm backtracks to C_{i-1}, as can be seen in Fig. 5.

In Fig. 5, $j_1, \ldots, j_i, \ldots, j_6$ are integer pointers that point to characters in $C_1, \ldots, C_i, \ldots, C_6$ respectively. The assignment $C_i := x_{ij_i}$ means 'eliminate from C_i all characters except x_{ij_i}'. At any time, C is the set $\{C_1, \ldots, C_6\}$. The assignment $T_i := C$ means 'store C in a storage area T_i'; and altogether the algorithm uses six such areas $T_1, \ldots, T_6$.

In the non-iterative refinement procedure, condition (4) is applied once to each character in each of $C_1, \ldots, C_6$. When a character is eliminated, this may cause (4) to be falsified for further characters. Therefore further characters may be eliminated if we apply (4) to all surviving characters a second time. In the iterative version of the refinement procedure, condition (4) is applied successively to all surviving characters in turn over and over again until no more are eliminated. The iterative refinement procedure takes more time but may eliminate more characters than the non-iterative version, and

therefore be executed fewer times in the course of the Fig. 5 algorithm. (In fact we must insist that the non-iterative version does two iterations in the special case where $i = 6$ and all confusion sets contain exactly one character.)

To find whether the iterative or the non-iterative version gave the fastest overall timing, we applied the Fig. 5 algorithm to randomly generated confusion sets $C_1, \ldots, C_6$, using n-gram data obtained from the 2,755-word dictionary of Riseman and Hanson (1974). We applied the Fig. 5 algorithm for fifty different choices of $C_1, \ldots, C_6$ with $|C_i| = 5$ for all $i = 1, \ldots, 6$. The iterative version (*c.f.* Ullmann, 1976) took 88 seconds on the KDF9 and the non-iterative version (*c.f.* Ehrich and Koehler, 1975) took 85 seconds. In a further experiment we applied the Fig. 5 algorithm for five different choices of $C_1, \ldots, C_6$ with $|C_i| = 10$ for all $i = 1, \ldots, 6$. The iterative version took 512 seconds, and the non-iterative version took 311 seconds.

(This result prompted us to test a non-iterative version of a similar algorithm that determines graph isomorphism (Ullmann, 1976). For 20-point isomorphic random graphs with average edge density 0·5, the non-iterative version was about 7% faster than the iterative version.)

The iterative and non-iterative methods erroneously output some words not in the dictionary, owing to the use of n-grams instead of straightforward dictionary look-up, and due to the use of random superimposed coding. An experimental error rate is quantitatively reported in the following section where we are concerned with confusion sets that are more realistic than the pseudorandom confusion sets used in the present section. Such errors can be remedied by checking each output word for membership in the dictionary.

6. Using n-grams to find all dictionary words differing by not more than two substitution errors from a given input word

The methods of Riseman and Hanson (1974) only correct errors that have been detected by the use of n-grams. To guard against errors that change one dictionary word into another, we may wish to find all dictionary words that differ by not more than some given number of errors from a given input word. To select just one of these words, we suggest that syntactic and semantic techniques should subsequently be used, but we shall not pursue this suggestion in the present paper, which is concerned only with the use of a dictionary or n-gram data derived from it.

Deletion, insertion and reversal errors will be dealt with in Section 7. As a preliminary step, in the present section we only find dictionary words that differ from an input word by up to two substitution errors. We use the Fig. 5 algorithm with $C_1, \ldots, C_6$ each initially containing all of the 26 letters of the alphabet, and with condition (4) replaced by condition (5). Condition (5) is a necessary condition for x'_{ij} to belong to a dictionary word that differs from a given input word by not more than two substitution errors.

To derive condition (5), let X' be any word in $W(C_1, \ldots, C_6)$ that contains x'_{ij} and differs from a given input word X by not more than two substitution errors. On any given n-tuple, the n-grams in X and X' necessarily match in at least n-2 characters. For instance 1*AB*4, *A*2*B*4, 123*A* match the 4-gram 1234 in at least two characters, but 1*ABC* does not. From the definition of $W(x'_{ij}, C_{k_2}, \ldots, C_{k_n})$ it follows that the n-gram in X' on an n-tuple in N_i belongs to the respective $W(x'_{ij}, C_{k_2}, \ldots, C_{k_n})$ and matches the n-gram in X in at least n-2 characters. Therefore x'_{ij} belongs to at least one word in $W(C_1, \ldots, C_6)$ that differs from X by not more than n-2 substitution errors only if:

For every n-tuple in N_i, $W(x'_{ij}, C_{k_2}, \ldots, C_{k_n})$ contains at least one n-gram that matches X in at least n-2 characters. (5)

Although, using (5), the Fig. 5 algorithm necessarily finds and outputs all the required words, it also outputs a few further words incorrectly, owing to the use of n-grams instead of a dictionary. To assess this failure (i.e. false output) rate quantitatively we applied the algorithm to 100 different input words (which were actually the 1st, 20th, 40th, 60th, . . . words in the dictionary). To the same words we also applied the straightforward dictionary method that does not produce any false output. For the 100 input words, the dictionary method outputted a total of 853 words, all of which were also outputted by the Fig. 5 method. The total number of words outputted by the Fig. 5 method was 898, so the failure rate was $\frac{45 \times 100}{853} = 5{\cdot}3\%$.

This result was obtained using $\lambda = 524$, $b = 3$, and the 2,755-word dictionary of Riseman and Hanson (1974).

7. Finding all dictionary words that differ by not more than two errors from a given input word

We now extend the technique of the previous section so as to cope with insertion, deletion, and reversal, as well as substitution errors. By way of example we consider the problem of finding words that differ by not more than two errors. To find words differing by not more than one error the same techniques can be used, but with L_1 instead of L_2.

In a lineset we denote any line $(i, i-1)$ by α, any line (i, i) by β, and any line $(i, i+1)$ by γ. For example the linesets in Fig. 1 are respectively β-$\beta\beta\beta\beta$, $\beta\beta\beta\gamma\gamma\gamma$, $\beta\beta\beta\beta$-α, $\gamma\alpha\beta\beta\beta\beta$, where '-' indicates a field touched by no line. The linesets in Fig. 2 are 'β-$\beta\beta\beta$-, -$\alpha\alpha\beta\beta\beta$, γ-$\beta\beta$-α, $\gamma\gamma$-β-β, and the set L_2 can now be regarded as a list (or dictionary) of six-character strings such as these.

Instead of using the term n-gram for an n-character subset of a lineset, we use the term n-line, in the hope of avoiding confusion. For instance, $\gamma\ \beta$ -α is a 4-line of γ-$\beta\beta$-α. We say that two words *match* in a given lineset iff all the lines in this lineset link pairs of identical characters. For instance TOLVES matches STOVES in the lineset -$\alpha\alpha\beta\beta\beta$, Fig. 2(*b*), and ROCKEN matches BROKEN in this same lineset. We say that an input word X matches an n-gram in a given n-line (on the same n-tuple as the n-gram) iff all the lines in this n-line link pairs of identical characters. For instance, WARENT matches A R AN in the 4-line $\gamma\ \beta$ -α.

For use in the following work we record positionally, for each n-tuple, which n-lines occur in at least one lineset in L_2, and we call such n-lines *stored* n-lines. The set L_2 is used only for determining the stored n-lines, and it is then discarded.

In Section 2 we used linesets to determine whether any given pair of words differ by not more than two errors. As a further preliminary step, we shall now use n-lines, instead of linesets, for this purpose. The input word is $X = \{x_1, \ldots, x_m\}$, and the dictionary word is $X' = \{x'_1, \ldots, x'_6\}$. Corresponding to the six dictionary word characters, let us initially construct subsets $E_1, \ldots, E_i, \ldots, E_6$ of α, β, γ as follows:

$$\alpha \in E_i \text{ iff } x'_i = x_{i-1} ,$$
$$\beta \in E_i \text{ iff } x'_i = x_i ,$$
$$\gamma \in E_i \text{ iff } x'_i = x_{i+1} .$$

Thus $\{E_1, \ldots, E_6\}$ is essentially the same as Alberga's (1967) coincidence matrix. Let $L_2(E_1, \ldots, E_6)$ be the set of all linesets in which X matches X', such that the ith line (if any) of any lineset in $L_2(E_1, \ldots, E_6)$ belongs to E_i, $i = 1, \ldots, 6$.

Corresponding to any n-tuple in N_i, let $\{E_i, E_{k_2}, \ldots, E_{k_n}\}$ be the n-member subset of $\{E_1, \ldots, E_6\}$. Let $L_2(y_{ij}, E_{k_2}, \ldots, E_{k_n})$ be the set of all *stored* n-lines that have y_{ij} as one line, their other $n-1$ lines being in $E_{k_2}, \ldots, E_{k_n}$ respectively. If a line $y_{ij} \in E_i$ belongs to a lineset in $L_2(E_1, \ldots, E_6)$, then in this lineset the n-line on any n-tuple in N_i belongs to the set $L_2(y_{ij}, E_{k_2}, \ldots, E_{k_n})$ which corresponds to this n-tuple. Therefore y_{ij} belongs to a lineset in $L_2(E_1, \ldots, E_6)$ only if:

Corresponding to each n-tuple in N_i, $L_2(y_{ij}, E_{k_2}, \ldots, E_{k_n})$ is non-empty. (6)

We remove from E_i any line y_{ij} that does not satisfy this condition. We process all lines in $E_1, \ldots, E_6$ in this way in turn iteratively, until there is a pass through $E_1, \ldots, E_6$ when no further lines are removed. Experimentally we always find that if $E_1, \ldots, E_6$ are non-empty when this procedure converges (terminates) then X matches X' by a lineset in L_2; and the Fig. 5 backtrack algorithm need not be used. Our procedure only works correctly when space characters are placed each side of the input word and the dictionary word, and we process all of the 70 possible 4-tuples of the eight fields. Intuitively, the idea is that if two words differ by three or more errors, then at least one 4-tuple will, so to speak, span three errors, so that the stored 2-error n-lines required by (6) will not all be found. This is why we have used 4-tuples and not 3-tuples. The space characters deal with insertions and omissions of characters at beginnings and ends of words.

One might perhaps think that the simpler procedure:

{decide that X differs from X' by not more than two errors iff for every n-tuple $L_2(E_{k_1}, E_{k_2}, \ldots, E_{k_n})$ is non-empty

would be adequate, but it is not. For instance this procedure erroneously decides that X differs from X' by not more than two errors when X' is 123456 and X is the three-error word 122445A6. Our iterative procedure does not make this mistake.

We now finally turn to the problem of using n-grams to find all dictionary words that differ from a given input word by up to two errors. We use the Fig. 5 algorithm, starting with $E_i = \{\alpha, \beta, \gamma\}$, and C_i comprising the 26 letters of the alphabet, for each $i = 1, \ldots, 6$. We redefine $L_2(E_1, \ldots, E_6)$ to be the set of all linesets in L_2 in which the input word $\{x_1, \ldots, x_m\}$ matches at least one word in $W(C_1, \ldots, C_6)$, such that the ith is in E_i, for all $i = 1, \ldots, 6$.

It is easy to see that y_{ij} belongs to a lineset in $L_2(E_1, \ldots, E_6)$ only if:

for each n-tuple in N_i there is an n-gram in $W(C_i, C_{k_2}, \ldots, C_{k_n})$ such that $x_1, \ldots, x_m$ matches this n-gram in an n-line in $L_2(y_{ij}, E_{k_2}, \ldots, E_{k_n})$ (7)

Furthermore, let X' be a word in $W(C_1, \ldots, C_6)$ that contains x'_{ij}, such that $X = x_1, \ldots, x_6$ matches X' in a lineset in $L_2(E_1, \ldots, E_6)$. X matches the n-gram in X' on an n-tuple in N_i in a lineset that belongs to $L_2(E_i, E_{k_2}, \ldots, E_{k_n})$ for that n-tuple. Therefore x'_{ij} belongs to a word X' only if:

for each n-tuple in N_i, X matches an n-gram in $W(x'_{ij}, C_{k_2}, \ldots, C_{k_n})$ in an n-line in $L_2(E_i, E_{k_2}, \ldots, E_{k_n})$. (8)

In the Fig. 5 algorithm we now use a version of the refinement procedure in which each line in $E_1, \ldots, E_6$ is removed unless it satisfies condition (7), and each character in $C_1, \ldots, C_6$ is removed unless it satisfies condition (8). Our programming technique is outlined below in the Appendix.

To implement conditions (7) and (8) in hardware it would be possible to process sets of n-grams on all n-tuples in parallel, and this might be faster than scanning through a dictionary as in Section 2. In the course of the Fig. 5 algorithm it may be necessary to execute the refinement procedure many times, whereas the method of Section (2) scans through the dictionary just once. Our method is unlikely to have any advantage over the dictionary method except possibly when the dictionary is very large and contains millions of words of different lengths. Our method can easily be extended to cope with dictionary words of varying length by employing special symbols to represent spaces, and treating these simply as characters. Although in the large dictionary case our algorithm may be advantageous from the point of view of speed, this is certainly not true of its software implementation on KDF9, and it was not practical to assess the failure rate as in Section 6.

8. Conclusion

To make the automatic correction of substitution, insertion, deletion, and reversal errors fast enough for practical purposes it may be necessary to employ special purpose hardware instead of relying purely on a general purpose computer. Finding the principles of operation of this hardware is still a basic research problem.

The present paper has worked towards a method of processing a set of n-tuples in parallel, in order to attain speed in the realistic case where the dictionary contains millions of words and names. The main original contribution of this paper is a method of applying n-gram techniques to linesets in order to capitalise the fact that the number of distinct n-grams on a given n-tuple that occur in at least one dictionary word is less than the number of words in the dictionary if n is quite small (e.g. $n = 4$). This paper does not disclose a new technique that has already undergone a process of commercial development. Instead it contributes to the pool of basic ideas from which commercially useful systems will eventually be developed.

Acknowledgements

The author is very grateful to Dr. E. M. Riseman for kindly providing a copy of his 2,755-word dictionary.

Appendix

In the refinement procedure in Section 5 it is not necessary to apply condition (4) separately to each character in each of $C_1, \ldots, C_6$ in turn. Instead we gain speed by implementing the refinement procedure as follows. We process each n-tuple in turn, and by way of example let us now consider the 4-tuple that consists of the second, fourth, fifth and sixth character fields. We define four sets C'_2, C'_4, C'_5 and C'_6 to be empty initially. We generate in turn every possible 4-gram whose first character is in C_2, second in C_4 third in C_5 and fourth in C_6. Using the method of Section 4 we test whether each such 4-gram has occurred in at least one dictionary word. For each n-gram that passes this test, the first character is entered into C'_2, the second into C'_4, third into C'_5 and fourth into C'_6. Repetitions of the same characters are removed from C'_2, C'_4, C'_5, C'_6. When every 4-gram for this 4-tuple has been processed, C_2 is replaced by C'_2, C_4 by C'_4, C_5 by C'_5 and C_6 by C'_6. After all 4-tuples have been processed in this way, any character x'_{ij} that remains in C_i necessarily satisfies condition (4). To see why this is the case, note for example that a character x'_{2j} in C'_2 necessarily belongs to at least one n-gram in $W(x'_{2j}, C_4, C_5, C_6)$.

In Section 6 we have used a similar technique to implement (5). It is never necessary to scan through 26^4 possible 4-grams on a given 4-tuple. Consider for instance the 4-tuple that consists of the first four character fields. If x_1 is in C_1 and x_2 is in C_2, then we process in turn all 4-grams that consist of x_1, x_2, one character from C_3 and one from C_4, applying the method that we have just introduced above in the previous paragraph. If x_1 is in C_1 and x_3 is in C_3, then we do the same thing with all 4-grams taken from $x_1C_2x_3C_4$. We process $x_1C_2C_3x_4$, $C_1x_2C_3x_4$, $C_1C_2x_3x_4$, and $C_1x_2x_3C_4$ similarly, and then replace C_1 by C'_1, C_2 by C'_2, C_3 by C'_3, and C_4 by C'_4. At most we process at total of 6×26^2 4-grams, not 26^4, for a 4-tuple.

In our programming implementation of the Section 7 technique, each n-tuple is processed in turn, and let us for example consider again the 4-tuple that consists of the first four character fields. Initially, C'_1, C'_2, C'_3, C'_4 and four further sets E'_1, E'_2, E'_3 and E'_4 are empty. For each of the stored 4-lines on this 4-tuple, the programme executes a procedure that is best explained in terms of examples. If the first 4-line is $\beta\beta\beta\beta$

then $x_1x_2x_3x_4$ is tested as in Section 4 and entered into C_1', C_2', C_3', C_4' if (3) is satisfied. If (3) is satisfied then β is entered into E_1', E_2', E_3', E_4'. If the next 4-line is β--β, then all 4-grams that can be constructed from $x_1C_2C_3x_4$ are tested as in Section 4 and entered into C_1', C_2', C_3', C_4' if (3) is satisfied. If (3) is indeed satisfied for any such 4-gram, then β is entered into E_1' and E_4'. If the next 4-line is $\beta\beta\gamma\alpha$, then $x_1x_2x_4x_3$ is tested as in Section 4 and entered into C_1', C_2', C_3', C_4' if (3) is satisfied. If (3) is satisfied for this 4-gram, then β, β, γ, α are respectively entered into E_1', E_2', E_3' and E_4'. As a final example, if the next 4-line is γ-$\beta\gamma$ then all 4-grams that can be constructed from $x_2C_2x_3x_5$ are tested as in Section 4 and entered into C_1', C_2', C_3', C_4' if (3) is satisfied. If (3) is satisfied for any such 4-gram, then γ is entered into E_1', β into E_3', and γ into E_4'. When this procedure has been carried out for all stored 4-lines on this 4-tuple, C_1, C_2, C_3 and C_4 are replaced respectively by C_1', C_2', C_3' and C_4'; and E_1, E_2, E_3 and E_4 are replaced respectively by E_1', E_2', E_3' and E_4'. This whole process is repeated for the other fourteen 4-tuples of the six character fields. For the remaining 70-15 4-tuples of the eight fields obtained by enclosing the six fields within spaces, we apply a procedure, like that described in the first paragraph of this Appendix, which implements (6).

References

ABEND, K. (1968). Compound Decision Procedures for Unknown Distributions and for Dependent States of Nature, In *Pattern Recognition*, edited by L. N. Kanal, Thompson Book Co., Washington DC, pp. 207-249.

ALBERGA, C. N. (1967). String Similarity and Misspellings, *CACM*, Vol. 10, No. 5, pp. 302-313.

BALM, G. J. (1975). Apparatus for Identifying an Unidentified Item of Data, *UK Patent No.* 1,403,816.

CHERRY, C., and VASWANI, P. K. T. (1961). A New Type of Computer for Problems in Propositional Logic, with Greatly Reduced Scanning Procedures, *Information and Control*, Vol. 4, Nos. 2-3, pp. 155-168.

DENES, P. (1959). The Design and Operation of a Mechanical Speech Recogniser at University College London, *J. of the British Inst. of Rad. Eng.*, Vol. 19, No. 4, pp. 219-229.

EHRICH, R. W., and KOEHLER, K. J. (1975). Experiments in the Contextual Recognition of Cursive Script, *IEEE Transactions on Computers*, Vol. C-24, No. 2, pp. 182-194.

ELLIS, J. H. (1969). Algorithm for Matching 1-Dimensional Patterns in Multidimensional space with Application to Automatic Speech Recognition, *Electronics Letters*, Vol. 5, No. 15, pp. 335-336.

FORNEY, G. D. (1973). The Viterbi Algorithm, *Proc. IEEE*, Vol. 61, No. 3, pp. 268-278.

HANSON, A. R., RISEMAN, E. M., and FISHER, E. G. (1974). Context in Word Recognition, *COINS Technical Report*, 74C-6, Department of Computer and Information Science, University of Massachusetts, Amherst, Mass. 01002, USA.

LOWRANCE, R., and WAGNER, R. A. (1975). An Extension of the String-to-String Correction Problem, *JACM*, Vol. 22, No. 2, pp. 177-183.

MOOERS, C. N. (1951). Zatocoding Applied to Mechanical Organisation of Knowledge, *American Documentation*, Vol. 2, pp. 20-32.

RISEMAN, E. M., and HANSON, A. R. (1974). A Contectual Postprocessing System for Error Correction Using Binary *n*-Grams, *IEEE Transactions on Computers*, Vol. C-23, No. 5, pp. 480-493.

SAKOE, H., and CHIBA, S. (1971). A Dynamic Programming Approach to Continuous Speech Recognition, *Seventh International Congress on Acoustics*, Budapest, 20C13, pp. 65-68.

SHANNON, C. E. (1951). Prediction and Entropy of Printed English, *Bell System Tech. J.*, Vol. 30, pp. 51-64.

SZANSER, A. J. (1973). Bracketing Technique in Elastic Matching, *The Computer Journal*, Vol. 16, No. 2, pp. 132-134.

THOMAS, R. B., and KASSLER, M. (1967). Character Recognition in Context, *Information and Control*, Vol. 10, No. 1, pp. 43-64.

ULLMANN, J. R. (1971). Reduction of the Storage Requirements of Bledsoe and Browning's *n*-tuple Method of Pattern Recognition, *Pattern Recognition*, Vol. 3, pp. 297-306.

ULLMANN, J. R. (1973). *Pattern Recognition Techniques*, Butterworths, London, and Crane Russak, New York.

ULLMANN, J. R. (1976). An Algorithm for Subgraph Isomorphism, *JACM*, Vol. 23, No. 1, pp. 31-42.

VELICHKO, V. M., and ZAGORUYKO, N. G. (1970). Automatic Recognition of 200 Words, *Int. J. Man-Machine Studies*, Vol. 2, pp. 223-234.

VINTSYUK, T. K. (1968). Speech Discrimination by Dynamic Programming, *Kibernetika*, Vol. 4, No. 1, pp. 81-88 (in Russian). Complete translation into English published in *Cybernetics*, Vol. 4, No. 1, pp. 52-58.

WAGNER, R. A., and FISCHER, M. J. (1974). The String-to-String Correction Problem, *JACM*, Vol. 21, No. 1, pp. 168-173.

WARREN, J. H. (1972). A Dynamic Pattern Matching Algorithm, In *Machine Perception of Pattern and Pictures*. Conference Series No. 13, The Institute of Physics, London, pp. 141-150.