

Optimization of cursive words recognition and nearest neighbors search in metric spaces

Xavier Dupré

University of Paris V, laboratory SIP-CRIP5,
45, rue des Saint Pères 75005 Paris, France
dupre@math-info.univ-paris5.fr

Abstract

This paper deals with cursive words recognition by hidden Markov models. Starting with an image of a word, the recognition process returns the most probable words included in a list which is called the dictionary. The cost of this process is proportional to the dictionary size but could be reduced with an acceptable lack of performance by merging two ideas. The first one consists in reducing the number of words for which the probability must be computed. A recognition without dictionary or unconstrained returns the best sequence of letters which is rarely a word. Nevertheless, its neighborhood allows us to find the most probable words of the dictionary. The second idea, based on ascendant hierarchical classification, consists in reducing the number of computations of edit distances that the building of this neighborhood needs.

1. Introduction

Hidden Markov Models (HMM) are now well known. Since [4], [14], they have become more and more popular. Currently used for speech recognition, they were adapted to handwritten recognition as in [8] associated to neural networks. Like in [3] or [10], we are interested in cursive words recognition using a dictionary. This article aims at improving the speed process without making the performance decrease too much. The system presented in section 2 links together the recognition without dictionary and the search for neighbors in any metric space. This search is used to find the closest words from a word x included in a dictionary D . The purpose is to compute probabilities only for the words which are close to the solution, these words are supposed to be included in the neighborhood of the best sequence of letters found by a Viterbi's algorithm (see [9]).

This particular task involves many computations of an edit distance such as Levenstein's one (see [11], [18]). Several optimizations have already been proposed, they are classified in two main ways. The first approach consists in an optimization of distance computation and the second is the optimization of the neighbors search, both reviewed in [13] and [6]. In brief, the first optimization is based on suffix trees (see [1], [12]). The second one selects some particular words called pivots (see [17], [15]) to avoid exploring all the dictionary. The main problem of such methods lies in a good choice of those points (see [5] or [19]). Section 3 presents a new method to get them, based on an ascendant hierarchical classification (AHC, [16]). Section 4 describes how to use the resulting tree. Section 5 gives the results obtained by the whole system realising a faster word recognition process.

2. Cursive words recognition with a dictionary

The recognition process which is mentioned in this paper starts with an image of a cursive handwritten word and tries to recognize it among a list called the dictionary. Let's denote this list as $D = (m_1, \dots, m_n)$. The image is preprocessed into a sequence O of vectors called observations. A emission probability is then computed for every hidden Markov model (HMM) M_i associated to every word m_i in the dictionary. These probabilities are denoted as $\mathbb{P}(O | M_i)$. The word "read" by the system is given by M^* :

$$M^* = \arg \max_{1 \leq i \leq n} \mathbb{P}(O | M_i) \quad (1)$$

Getting M^* defined in (1) obviously implies the computation of $\mathbb{P}(O | M_i)$ for every word in D . The aim of this article is to avoid computing emission probabilities for all words. But before describing how to do faster, let's go on deeper into the models associated to the words. In fact, both O and M_i are sequences. In our application, O is a sequence of observations coming from an image processing and each observation corresponds to a letter or a piece of letter. Figure 1 shows a segmented image, every small piece or graphem is described by a vector - an observation - which contains statistics about it (average number of black pixel, presence of a loop, ... see [3]). M_i is a sequence of models and each one is able to recognize a letter. Figure 2 illustrates an assembly of letter models for the word "charles".

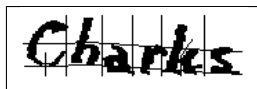


Figure 1. A example of an image segmentation leading to pieces as big as characters or smaller (letter C contains two pieces), every one is then described by a vector called observation.

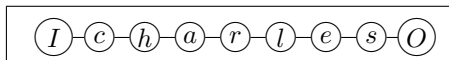


Figure 2. Word model for "charles" built from letter models "c", "h", "a", "r", "l", "e", "s", the symbols "I" and "O" means the input and ouput of the model.

So, a word model is made by the concatenation of the HMM models associated to every letter it contains. That is why this system allows us to compute the best sequence of models L^* for a given sequence of observations O . L^* is the model which maximizes $\mathbb{P}(O | L)$ where L is any sequence of letter models. This sequence is the result of a Viterbi's algorithm (see [9]). Getting L^* is only a first step to a recognition without dictionary because the linked word l^* is rarely a word. For example, the result for the image of Figure 1 is "ciharlis". Nevertheless, l^* is not so far from the dictionary D and it is possible to compute the neighborhood of l^* inside D according to an edit distance like Levenstein's one. If we denote d as this distance, then we have to search for the set $N^*(s)$ defined by :

$$N^*(s) = \{m \in D \mid d(m, l^*) \leq s\} \quad (2)$$

We would like to expect that M^* defined in (1) verifies:

$$M^* = \arg \max_{M \in N^*(s)} \mathbb{P}(O | M) \quad (3)$$

The set $N^*(s)$ should be small to avoid too many computations of emission probabilities but it should be high enough to be sure that it contains the right answer. The choice of s is a compromise between the speed optimization and the loss of performance. It first depends on the distance which is used to define the neighborhood. The recognition models are also important: a high accuracy of the recognition models makes the best sequence of letters be closer to the correct answer and in this case, we may use a small value of s . The optimized system consists of three steps:

1. We compute the most probable sequence of letters using the Viterbi's algorithm.

2. We find the neighborhood $N^*(s)$ of this sequence included in the dictionary.
3. We compute the probability for each word in the neighborhood $N^*(s)$.

We finally built a speed optimization system (see Figure 3) able to recognize a word in an image. The last problem is to compute the set $N^*(s)$ which is the purpose of the next sections.

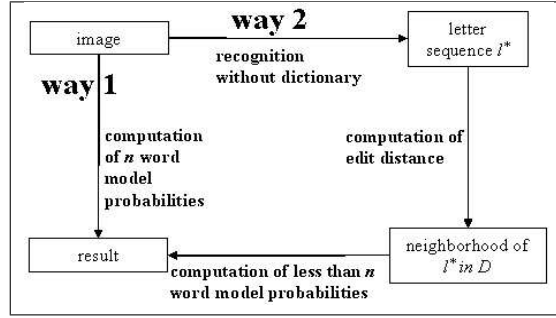


Figure 3. Speed optimization system : way 2 is faster than way 1

3. Partitioning tree

Before computing the search for such a neighborhood, we need to organize the dictionary D into a tree in order to avoid exploring all words in D every time we want the neighborhood of a word. The building of this tree is based on a ascendant hierarchical classification (AHC). But first, we need for this to define the radius and the center of a finite subset. Let $D = (y_1, \dots, y_N) \subset E$ be a finite subset of a metric space E , the center $C(D)$ and the radius $R(D)$ of D are defined by:

$$C(D) \in \arg \min_{x \in D} \left[\max_{y \in D} d(x, y) \right] \quad (4)$$

$$R(D) = \max_{x \in D} d(C(D), y) \quad (5)$$

$d(x, y)$ is the distance between the elements x and y . It is obvious that for any $(x, y) \in E^2$, $R(\{x, y\}) = d(x, y)$. The tree is built with a version of the AHC presented by Algorithm 1. At the beginning, the tree only contains N nodes linked to the N words the dictionary contains. At every iteration, two nodes are merged to form the part with the smaller radius.

Algorithm 1 Let $D = (y_1, \dots, y_N)$ be a finite subset of (E, d) . Let $N(n_1, n_2, C, R)$ be a node which is linked to two predecessors n_1, n_2 and which defines a part whose center is C and whose radius is R . L represents a set of nodes. If x is a node, $P(x)$ refers to the part pointed out by the node which is the union of the centers of the node N and all its ancestors.

initialisation:

for each $y \in D$, we add the node $N(\emptyset, \emptyset, y, 0)$ in L

step 1:

let (x, y) be in $\arg \min_{x, y \in L, x \neq y} R(P(x) \cup P(y))$

the node z is created such as $z = N(x, y, C(P(x) \cup P(y)), R(P(x) \cup P(y)))$

$L \leftarrow L \cup z - \{x, y\}$

step 2:

if L contains more than one node, return to step 1

The result of this algorithm is a graph which is illustrated in Figure 4 for five elements. Every node of it satisfies one or several of three cases. It has no predecessor, and the part pointed by this node is a singleton whose radius

is null. It has two predecessors, and the part pointed by this node contains more than one element, its radius is strictly positive if at least two elements are different. It has no successor, the part pointed by this node is the subset D .

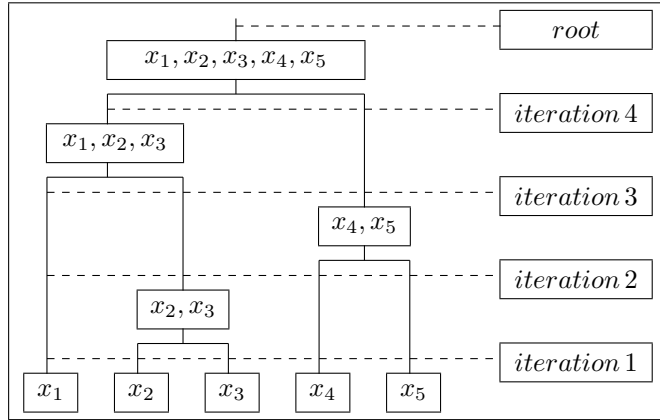


Figure 4. Dendrogram: the initial state contains one part per element in D (the leaves), the first iteration groups together the closest elements, the second iteration groups together the second couple of closest elements, the third iteration builds the part with the smaller radius, the choice is between $\{x_1, x_2, x_3\}, \{x_1, x_4, x_5\}, \{x_2, x_3, x_4, x_5\}$, the fourth iteration builds the last node or root.

The purpose of this algorithm is to group together the closest elements or parts at every iteration. The expected consequence is that the neighborhood of one word is concentrated in one branch of that tree which contains exactly $2N - 1 = 2 * card(D) - 1$ nodes. We can also prove that any node n of this tree satisfies $R(n) \leq R(s(n))$ if $s(n)$ exists.

However, the main drawback of the algorithm 1 is its cost. If we suppose the cost of the distance can be approximated by a constant c , we denote n as the dictionary size, the algorithm has a $O(c n^5)$ complexity. This cost can be reduced. For example, from the iteration k to the next one, the list L keeps $n - k - 1$ nodes unchanged and this remark allows us to compute the center and the radius for only one new node at each iteration. In the same way, it is possible to keep for every node the best neighbor which is included in L , it does not change unless it is one the two nodes which has been merged at the previous iteration or unless the new node is a better candidate. Finally, the approximative cost of Algorithm 1 is $O(c n^3)$.

Nevertheless, for dictionaries of several thousands words, Algorithm 1 still lasts several hours. A faster way consists in using a version of the k-means algorithm to divide the dictionary into k smaller parts. This step minimizes the sum of the radius of the k parts. Then, Algorithm 1 is applied on every of them, it builds k sub-dendograms which are also grouped together by Algorithm 1. This method has approximatively a $O\left(k \left[\frac{n}{k}\right]^3\right)$ complexity and it can be applied to large dictionaries in a reasonable time.

4. Optimization of neighborhood search

This optimization uses the graph built with Algorithm 1. Every node of it defines a part described by a center and a radius. The problem to solve here consists in finding, for one element m , the list of neighbors included in the subset $D = (y_1, \dots, y_N)$ for which the distance to x is below a given threshold $s > 0$. Let $P \subset E$ be a part whose center is $C(P)$ and whose radius is $R(P)$, the optimization is based on the triangular inequality verified by any distance:

$$d(m, C(P)) > s + R(P) \implies \forall w \in P, d(m, w) > s \tag{6}$$

$$d(m, C(P)) + R(P) \leq s \implies \forall w \in P, d(m, w) \leq s \tag{7}$$

Therefore, we deduce of (6) that m has no neighbor inside the part P if the first inequality is verified. We also

deduce of (7) that every element of P is part of the neighborhood of m if the second inequality is verified. In both cases, there is no need to compute the distances between the word m and any element of P . These observations leads to the following algorithm:

Algorithm 2 *Let r be the root of the graph A obtained by the algorithm 1. N is a node set and B is the list of elements $w \in D$ verifying $d(w, m) \leq s$.*

initialisation:

$N \leftarrow r$ and $B \leftarrow \emptyset$

step 1:

let $n \in N$ and p the part defined by n

n is removed from N : $N \leftarrow N - \{n\}$ and

if $d(m, C(p)) \leq s + R(p)$ then
 | if $p = \{w \in D\}$ then $B \leftarrow B \cup \{w\}$
 | else $N \leftarrow N \cup \{p_1(n), p_2(n)\}$ where $\{p_1(n), p_2(n)\}$ are the two predecessors of n
 else if $d(m, C(p)) + R(p) \leq s$ then
 | $B \leftarrow B \cup p$

step 2:

if $N \neq \emptyset$, return to step 1

The list B produces all the neighbors of m . During the first step of Algorithm 2, it is necessary to compute many distances between the element m and some centers. These centers belong to the subset D and several parts may have the same center if they include each other. As a result, it could be useful to keep the distances in memory once they are computed. This also means that this optimization cannot involve more than n distance computations if D has n elements. Vectorial spaces allow more precise expression of the average cost of such an algorithm (see [2]). Otherwise, in any metric space, [7] gives the cost of the search for the nearest neighbor.

5. Experimental result

Several experiences measured the improvements due to the method which was presented in the previous sections. All use a recognition system which takes an image such as the one which is shown in Figure 1 and which returns a probability for every word in the dictionary. This system is a combination between Hidden Markovs Models and Neural Networks (see [3]). It was trained on a database of 38000 French first names and all the results given in the following tables are estimated on a test database of 12000 first names. Without any speed optimization, Table 1 shows the performances of this system for two dictionaries, the first one contains 2100 words, the second one 11000 words.

| dictionary size | recognition rate | reading rate for 1% substitution | processing time |
|-----------------|------------------|----------------------------------|-----------------|
| 2100 | 91.9 % | 72.9 % | 101 ms |
| 11000 | 83.8 % | 46.9 % | 244 ms |

Table 1. Performance of the recognition system. For the first dictionary, the reading rate for 1% substitution is 72.9 %, it means than the system can process 72.9 % of the test database with less than 1% of errors. The time is not linear with the number of words in the dictionary because the emission probabilities are factorised (see [9]), $t \sim 0.015n + 70$ ms where t is the average time per document and n the dictionary size. This measures was obtained on a Pentium IV 1 GHz.

Table 1 presents two scores, the first one - recognition rate - counts the number of times the answer of the recognition system is the correct one whatever its probability. The second rate - reading rate for 1% substitution - counts the number of times the answer is the correct one when its probability is over a threshold chosen to have less than 1% of errors. The purpose of the next experience is to show the evolutions of these rates and the processing

time. Using the speed optimization system, the average time spent per document can be decomposed the following way:

$$t_{doc} \sim \underbrace{C_{dico} + N_{nn}(s) t_{word}}_{\text{recognition time}} + \underbrace{C_{viterbi} + C_{nn} + N_{dist}(s) t_{dist}}_{\text{optimization time}} \quad (8)$$

| <i>parameter</i> | <i>meaning</i> | <i>estimated value</i> |
|------------------|---|------------------------|
| C_{dico} | recognition constant, see the caption of Table 1 | 70 ms |
| $N_{nn}(s)$ | average size of the neighborhood, function of s | |
| t_{word} | average recognition time per word of the dictionary | 0.015 ms |
| $C_{viterbi}$ | average time for the Viterbi's algorithm used to find the most probable sequence of letters | 0.01 ms |
| C_{nn} | constant of the neighborhood search | 1.8 ms |
| $N_{dist}(s)$ | average number of computed edit distances, function of s | |
| t_{dist} | average time to compute an edit distance | 0.004 ms |

Equation (3) delayed the choice of the threshold s which determines the size of the neighborhood. For some values of s , Table 2 shows that the added time due to the optimization part is less than the reduction of the recognition part. For this experience and if s is equal to 4, the optimized system can recognize a word with eight milliseconds less and a loss of performance which is insignificant.

| s | recognition rate | reading rate for 1% substitution | average size of the neighborhood $N_{nn}(s)$ | average number of computed edit distances $N_{dist}(s)$ | time (ms) t_{doc} |
|-----|------------------|----------------------------------|--|---|---------------------|
| 0 | 38.0 % | - | 0.4 | 281 | 83 |
| 1 | 59.6 % | 53.8 % | 2.0 | 508 | 84 |
| 2 | 78.3 % | 65.8 % | 11.0 | 836 | 86 |
| 3 | 87.1 % | 70.8 % | 63.3 | 1227 | 88 |
| 4 | 90.6 % | 72.3 % | 266.6 | 1587 | 93 |
| 5 | 91.7 % | 72.9 % | 707.1 | 1792 | 101 |
| 6 | 91.8 % | 72.9 % | 1241.1 | 1753 | 110 |
| 7 | 91.9 % | 72.9 % | 1724.4 | 1490 | 116 |

Table 2. Loss of performance and speed optimization for several values of the threshold s (see Equation (3)) when the dictionary contains 2178 words. The best value for s seems to be 4, the system is about 8 ms faster than the version without speed optimization and the reading rate loses only 0.6 points.

With the dictionary of 11000 words, the best value for s is 3, the reading rate loses 1.6% (see Table 3) and the processing time per document is divided by two. For bigger dictionaries, the speed optimization works better but the main drawback of the method remains the building of the tree. For 2100 words, the tree is built in ten minutes, and for 11000 words, in two hours.

The recognition part and the other one which processes the search for the neighborhood are independant. Consequently, the second part have been replaced by another algorithm called LAESA (Linear Approximating Eliminating Search Algorithm, see [15]). Instead of a tree, this algorithm needs to keep in memory many edition distances between the whole dictionary and a subset called the pivots which are randomly chosen. They play the same role as the nodes of the tree which was previously used. Table 4 shows the results obtained for the dictionary with 2100 words. The recognition and the reading rates are the same, the neighborhood size and the processing time only change.

According to Table 4, the association of Algorithms 1 and 2 gives better results than Algorithm LAESA. Furthermore, the tree built by Algorithm 1 contains only $2N + 1$ nodes (N is the dictionary size) whereas Algorithm

| s | recognition rate | reading rate for 1% substitution | average size of the neighborhood | average number of computed edit distances | time (ms) |
|-----|------------------|----------------------------------|----------------------------------|---|-----------|
| 2 | 71.8 % | 40.1 % | 36 | 4045 | 111 |
| 3 | 79.7 % | 44.6 % | 281 | 6366 | 127 |
| 4 | 82.9 % | 45.7 % | 1305 | 8398 | 154 |

Table 3. Loss of performance and speed optimization for several values of the threshold s (see Equation (3)) when the dictionary contains 11000 words. The best value for s seems to be 3, the system is about two times faster than the version without speed optimization and the reading rate loses 1.6 points.

| s | LAESA 10 pivots average number of computed edit distances | LAESA 10 pivots time (ms) | LAESA 500 pivots average number of computed edit distances | LAESA 500 pivots time (ms) |
|-----|---|---------------------------|--|----------------------------|
| 2 | 878 | 103 | 83 | 78 |
| 3 | 1400 | 131 | 345 | 107 |
| 4 | 1770 | 155 | 886 | 190 |
| 5 | 1941 | 172 | 1321 | 287 |
| 6 | 1950 | 180 | 1397 | 318 |

Table 4. Time reduction obtained with the LAESA algorithm and 10 or 500 pivots. Compare to Table 2, the number of computed edit distances is higher for 10 pivots and lower for 500 pivots. However, in both cases, the processing time is worst because, at every iteration, the algorithm AESA needs to update arrays whose dimensions are proportional to the dictionary size and the number of pivots. Compare to the cost of the edit distance, this step is significant.

LAESA needs to memorise the results of many edit distances between the dictionary and the pivots. However, the search for the neighborhood with LAESA needs less distance computations than Algorithm 2. This remark suggests the method could be improved if both algorithms were associated. Moreover, it could be interesting to study the behavior of Algorithm LAESA if its pivots were picked in the hierarchy built by Algorithm 1.

6. Conclusion

This article presents a way to reduce the computation time in order to recognize a word among a dictionary. Instead of computing the recognition probabilities for every word in the dictionary, the system selects the most interesting part of it which corresponds here to the neighborhood of the most probable sequence of letters. With a insignificant loss of performance, it is possible to reduce the recognition time in a significant way for dictionaries whose size is over few thousands of words. The search for neighborhood, based on a ascendant hierarchical classification, gives encouraging results which can still be improved. That is our first direction of research. Furthermore, the Levenstein’s edit distance does not make any difference between letters and the performance could certainly be improved by modifying it to take into account the confusion made by the recognition models. Building such a distance adapted to a recognition problem is our second direction of research.

7. Acknowledgements

This work was carried out in the Laboratory of Perception Intelligent Systems (SIP-CRIP5) of the University of Paris V (France) under the direction of Pr. G. Stamon and with the collaboration of the company Artificial

Intelligence and Image Analysis (A2iA). I would like to thank Professors Suen and Stamon for their advice as well as Mr Augustin and Baret for their help.

References

- [1] A. Apostolico, "The Myriad virtues of subword trees", In *Combinatorial Algorithms on Words*, Springer-Verlag, 1985, pp 85-96
- [2] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, Angela Y. Wu, "An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions", *JACM* 45(6):891-923 (1998)
- [3] E. Augustin, *Handwritten Word Recognition using Hybrid Neural Network and Hidden Markov Systems*, PhD Thesis from R. Descartes University - Paris V, Mathematics and Computer Science Department, 2001
- [4] L. E. Baum, "An Inequality and Associated Maximization Technique in Statistical Estimation for Probabilistic Functions of a Markov Process", In *Inequalities 1972*, 3:1-8.
- [5] B. Bustos, G. Navarro, E. Chavez, "Pivot Selection Techniques for Proximity Searching in Metric Spaces", In *Proceedings of SCCC'01*, 2001, pp 33-40,
- [6] E. Chavez, G. Navarro, R. Baeza-Yates, J. Marroquin, "Searching in metric spaces", Technical report TR/DCC-99-3, Departement of Computer Science, University of Chile, 1999
- [7] A. Faragó, T. Linder, G. Lugosi, "Fast Nearest-Neighbor Search in Dissimilarity Spaces", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol 15, no 9, September 1993, pp 957-962
- [8] S. Knerr, Y. Tay, P. Lallican, M. Khalid, C. Viard-Gaudin, "An offline cursive handwritten word recognition system", In *Proceedings of IEEE Region 10 Conference*, 2001.
- [9] A. L. Koerich, R. Sabourin, C. Y. Suen, "Fast two-level viterbi search algorithm for unconstrained handwriting recognition", In *Proc. 27th International Conference on Acoustics Speech and Signal Processing*, Orlando, USA (2002).
- [10] A. L. Koerich, R. Sabourin, Y. Leydier, C. Y. Suen, "A Hybrid Large Vocabulary Handwritten Word Recognition System using Neural Networks with Hidden Markov Models", *8th International Workshop on Frontiers of Handwriting Recognition (IWFHR'8)*, pp. 99-104, Niagara-on-the-Lake, CA, August 6-8, 2002.
- [11] V. Levenstein, "Binary codes capables of corrections, deletions, insertions, and reversals", *Soviet Physics Doklady* 10(8):707-710, 1966.
- [12] S. Madhvanath, V. Krpasundar, V. Govindaraju, "Syntactic methodology of pruning large lexicons in cursive script recognition", *Pattern Recognition* 34 (2001), pp 37-46
- [13] G. Navarro, "A Guided tour to approximate string matching", *ACM Computing Survey* 33(1):31-88 (2001)
- [14] L. R. Rabiner, S.E. Levinson, N. M. Sondhi, "An introduction to the application of the theory of probabilistic functions of a Markov process to automatic speech recognition", *Bell System Technical Journal* Vol 52. No 4. April 1983 pp257-285
- [15] J. R. Rico-Juan, L. Mico, "Comparison of AESA and LAESA search algorithms using string and tree-edit-distances", *Pattern Recognition Letters* 24 (2003) pp 1417-1426
- [16] Gilbert Saporta, *Probabilités, analyse des données et statistique*, Editions Technip 1990
- [17] J. Uhlmann, "Implementing trees to satisfy general proximity / similarity queries with metrics trees" *Information Processing Letters*, 40, (1991), pp 175-179
- [18] R. A. Wagner, M. Fisher, "The string-to-string correction problem", *Journal of the ACM* 21:168-178, 1974
- [19] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces", In *Proc. of the 4th Annual ACM-SIAM Symp. on Discrete Algorithms*, 1993, pp 311-321