Architectures for Efficient Scribble Matching

Dave Reynolds, Dipankar Gupta, Richard Hull
Office Appliance Department
HP Laboratories Bristol

HPL-94-62
July, 1994

handwriting, matching, recognition, scribble

Scribble matching is a form of partial handwriting recognition that allows like scribbles to be identified. In a companion paper [1] we describe a set of three algorithms which achieve high accuracy on this task. In this paper we address the issue of increasing the computational efficiency of these algorithms by over an order of magnitude to the point where interactive use on a modest platform such as Intel 80386 is possible.

1 Introduction

In [1] Hull, Reynolds and Gupta describe a set of algorithms which support the accurate matching of different samples of electronic ink. This technique of *scribble matching* is very attractive for applications such as document retrieval and personal information management due to its high accuracy, its ease of use and its language independence. However, the algorithms described in [1] are computationally expensive. Furthermore, to achieve the high accuracy figures quoted several algorithms must be run in parallel and their results combined, thus adding to the computational cost.

In this paper we discuss a variety of approaches to speeding up the scribble matching algorithms in order to be able to delivery their high match accuracy on relatively modest hardware such as Intel 80386 class processors. We first briefly describing the scribble match algorithms, which are all based on the string edit distance metric. We then examine three different classes of technique which can be used to speed up such computations. We finally exhibit an example architecture which uses some of the successful techniques to offer a 33-fold speed increase over the original unoptimised algorithm set.

2 Summary of scribble matching

The scribble matching algorithms described in [1] all take the same basic form illustrated in figure 1:

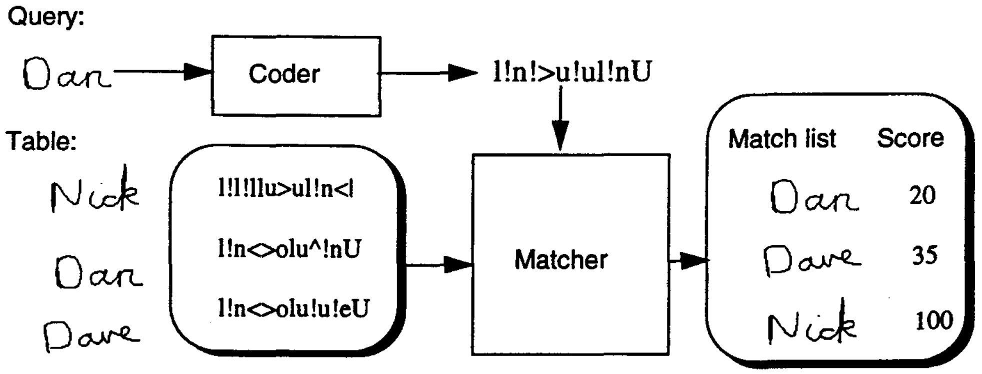

Figure 1 - Performing a query search

1. The query scribble is preprocessed to detect velocity minima [3][4][5], remove hooks and (optionally) polygonally resample [6] the ink.
2. For each significant point in the preprocessed ink (velocity minimum knot point or polygonal vertex) we generate a symbolic or numeric label describing some aspect of the ink at that point. These labels are concatenated together into a code vector.
3. The resulting code is compared to each entry in a table of known codes using a weighted string edit distance metric implemented by dynamic programming [7][8]. An ordered list of the entries in the table which most closely match the query is returned.

Of the approaches explored three algorithms proved to be particularly successful. They differ in the nature of the coding (and thus the string edit weights) applied. They are described in more detail in [1]:

- The *syntactic matcher* codes each pen velocity minimum by one of a small set of symbolic labels.
- The *word shape matcher* codes each velocity minimum by its relative height above a median line.
- The *elastic matcher* codes each polygonal sample point using height, angle and point class labels.

By combining the results of all three algorithms we achieve very high match accuracies. On a set of 200 word searches (using data collected from 31 authors as described in [1]) we obtain a match accuracy of 97% and in better than 99% of the cases the correct match is in the top 5 elements of the match list.

3 The performance issue

Our typical applications for scribble matching involve searching a table of a few hundred entries in interactive time. We set ourselves a target of searching a table of 200 entries in times on the order of one second.

The original scribble matching algorithms described in [1] were first developed on HP 9000/735 UNIX workstations on which the performance of any one matcher without any optimisation at all meets such a target though a brute force parallel combination of all three matchers does not.

Table 1: Search times for 200 word scribble table

Algorithm	Time on HP 735
Syntactic matcher	0.5s
Word shape matcher	0.5s
Elastic matcher	1.8s
Combination matcher	2.8s

Ideally, we would like to apply the scribble matching techniques on lower compute power platforms such as Intel 80386 and 80486 based personal computers. In particular, interactive performance on an 80386 class processor would open up the application of scribble matching to hand-held pen devices. The raw performance ratio between this platform and the development workstations is roughly a factor of 10.[1]

Considering the components of the scribble matching algorithms described above the preprocessing and coding steps (1 and 2) are relatively efficient. By use of integer based algorithms for filtering, velocity estimation and polygonal sampling the cost of these two stages for all of the algorithms combined is around 2% of the above combined match time. Further reductions in preprocessing time could be obtained should this become a limitation but it is clear that the matching step (3) is currently the bottleneck.

One option would be to simply replace the matching algorithms by some fundamentally cheaper approach which did not use the edit distance metric. However, the matching algorithms are the result of extensive comparative tests on alternative matching schemes and the edit distance approach demonstrated substantially better accuracy than other schemes tried.

Retaining the basic edit distance algorithms, we can consider three approaches to performance improvement:

1. Reduce the cost of each edit distance measurement.
2. Reduce the number of table entries visited by the edit distance matchers using branch and bound techniques.
3. Index the table using some alternative, cheap feature measurements and only apply the edit distance measures to a reduced set of table entries.

In the remainder of this paper we will explore these alternatives and their combination.

4 Reducing the cost of a single edit distance measurement

The standard algorithm for string edit distance [9] is most easily thought of in terms of a two dimensional array of costs (see table 2). Given two strings $a_{1 \dots n}$ and $b_{1 \dots m}$ then entry $d(i,j)$ in the table gives the lowest cost so far, i.e. the cost of editing the substring $a_{1 \dots i}$ into $b_{1 \dots j}$. We can then simply calculate the $d(i,j)$ entries in one pass of the table using the formulae:

$$d(i,j) = min \begin{cases} d(i-1,j) + del(a_i) \\ d(i-1,j-1) + subs(a_i, b_j) \\ d(i,j-1) + ins(b_j) \end{cases}$$

$$d(0,0) = d(\perp,\perp) = 0$$

1. In fact for general processing the difference in platform power is greater. However, the scribble matching algorithms are designed around 16 bit integer operations which are well suited to the x86 platforms. In addition, a greater level of compiler optmisation was used in testin g on the x86 platform.

Where $\perp$ represents the null string and $del/ins/subs$ are the deletion/insertion/substitution cost functions respectively.

Table 2: Example edit distance computation on strings acbb/abbbc

	$\perp$	a	b	b	b	c
$\perp$	0	1	2	3	4	5
a	1	0	1	2	3	4
c	2	1	2	3	4	5
b	3	2	2	2	3	4
b	4	3	3	2	2	3

The final edit distance is then, $d(n, m)$, the bottom right hand corner of this table.

In practice, since we only want the final cost, rather than the edit path, we do not store the whole matrix but only one row. Even so this direct algorithm requires the equivalent of a full scan of the matrix and thus has cost $O(n^2)$ where n is the length of the strings. In the case of scribble matching n is on average 22 for velocity minima coding and 70 for polygonal vertex coding.

We considered four techniques for improving upon this basic algorithm:

1. Submatrix approach
2. Suffix tree matching
3. Error correcting codes
4. Beam limiting

4.1 Submatrix approach

Several techniques for speeding up string-edit computation to asymptotically $O(n^2/(\log n))$ exist. A good example is [10] which uses the so called Four Russian's algorithm. It splits the edit matrix into submatrices, computing the cost on each submatrix using precomputed submatrices then combining the results. Unfortunately such techniques are only superior for rather large n (numbers in the range 200,000 are used in [10]) and do not appear appropriate for scribble matching.

4.2 Suffix tree matching

A second approach is applicable when we are expecting to repeatedly match the same query code vector against multiple target vectors. The idea is to precompile the query code into some form of approximate matching automaton. This concept is similar in principle the well known Boyer-Moore algorithm for exact string matching. After examination of the literature in this area the most promising algorithm which we selected for experimentation is the suffix tree approach described in [12].This is asymptotically faster than the full search but experimentation demonstrated that for our relatively short query codes the cost of this algorithm exceeds that of the beam limited matcher described below and was not pursued further.

4.3 Error correcting codes

In [13] Dolev, Harari and Parnas describe a set of algorithms which use error correcting codes to index a table of approximate substring matches. The idea is to partition the code strings in k parts and chose a k large enough (based on the maximum expected edit distance) that at least some subparts must match directly. Error correcting codes can then be used to index the and match partitions. Our attempts to apply this approach to scribble matching (using Golay codes) were unsuccessful. This is due to the relatively large number of insertion/deletions which can occur in the scribble match problem which leads to a combinatorial growth in the number of partitions needed. It may be that future developments of this approach, perhaps using different error correcting code schemes could be more successful. However, we conclude that in its current form it is not applicable to the scribble match problem.

4.4 Beam limiting

The final alternative in this class of approaches is to use a metric which only considers edit paths where the maximum number of substitutions or deletions is less than some beam limit b. This is only an approximation to the full edit metric but in some applications is a more appropriate metric and can sometimes lead to better match performances. This computation corresponds to a scan of the diagonal of the edit cost matrix $\pm b$ cells and thus has a cost $O(nb)$. For the syntactic matcher, running on the 11 author development data set described in [1] ,we obtain the results shown in table 3 (similar figures are obtained for the other matching algorithms).

Table 3: Effect of beam limiting on the syntactic matcher

Beam limit	Search time	Accuracy	In top 5
None	35.6	90.8%	98.0%
6	19.3	90.8%	98.0%
4	15.5	91.3%	98.0%
2	9.9	91.3%	98.3%
1	7.4	86.5%	95.2%

Thus by using a beam limit of 2 to 4 we can obtain a threefold speed up without loss of accuracy (indeed with a slight gain). A further refinement is to make the beam width dependent upon the length of the query code. This offers marginally better accuracy at the lower beam settings without a significant increase in average cost.

4.5 Summary

Of these approaches, only the use of a beam-limited search seems to be effective for scribble matching.

5 Bounding the table matches

The simple linear table search implied by our earlier description of scribble matching is clearly inefficient. The second class of approaches to performance improvement make use of the branch-and-bound search strategy [14]. The essence of this strategy is to use a cheap lower bound function to quickly determine if a candidate match would yield a worse result than the best match found so far. Given a good lower bound function this can be an effective method of reducing a search space.

Two algorithms in this class have been examined:

1. The metric space algorithm
2. Algorithm specific bounds

5.1 The metric space algorithm

Scribble matching can be seen as a special case of nearest neighbour search. Several algorithms for fast nearest neighbour search exist [15][16] which take advantage of the metric properties of a discrimination function. Each entry in the search table can be characterised by a vector of distances from some set of basis points. Once the distance of the query from some of these basis points is known then a lower bound on its distance from each of the table entries can be found by use of the metric triangle inequality. The table can then be efficiently searched using branch and bound techniques [14]. For example [15] describes such an algorithm which has linear space complexity and asymptotically constant time complexity.

Empirical tests on the scribble matchers showed that the metric triangle inequality is obeyed to a very good approximation and so the metric space algorithm is admissible.

For the syntactic matcher early tests indicated that for code tables with 200 entries only 20% of table needs to be probed to find the best match, giving a five-fold speed up. Though in this application the algorithm operates far below the constant time region and search times still increase linearly with table size.

However, an important feature of using scribble matching in retrieval applications is that we can offer very high confidence that the correct match will occur in some short list of near misses (typically around 5) even if it is not the top match. If we adapt the metric space approach to generate a list of the top N matches then the performance advantage degrades rapidly as N increases:

Table 4: Effect of returning multiple matches on search efficiency

Return set size	1	3	5	10
Fraction of table searched	20%	40%	45%	51%

This is not necessarily fatal since we could imagine a system solution in which a fast search for the best match is run and presented to the user and then the search for the remaining top N matches proceeds in the background. However, this problem coupled to our failure to obtain similar speed ups when applying the same technique to the other scribble matchers caused us to examine other approaches.

5.2 Branch and bound pruning

It is possible to devise other lower bound estimates based on specific features of the detailed scribble matching algorithms which are more accurate than the very general metric space approach. Such techniques can give good branch and bound performance and one promising example is described in [17]. However, these approaches suffer from the same difficulty as the metric space algorithm when returning the remainder of the top N match list and will not be considered further here.

5.3 Summary

Given our interest in returning the top N matches we do not currently utilize branch and bound pruning though it remains important for some applications.

6 Table pruning

The final set of approaches ignore the edit distance metrics completely and uses alternative, cheaper, metrics to reduce the size of the search table. In the both cases the general form of the algorithm is:

1. For each entry in table, measure its distance to the query scribble using some cheap distance measure.
2. Sort the table on the distance measure values.
3. Retain only those entries whose distance d is less than $k \cdot d_0$ where d_0 is the lowest distance found and k is a heuristically chosen parameter typically around 2.

We consider two such approaches:

1. Linear time approximations to edit-distance.
2. Constant time feature vectors.

6.1 Linear time approximations

The computational cost of string-edit comparisons derives from the need to attempt many different alignments of the variable length code vectors. A cheaper, though less accurate, approach would be to use similar feature measures but to stretch the feature codes to fill a constant length vector which can then be tested in linear time.

After some experimentation we developed three such algorithms. These three linear algorithms, the profile, angle and histogram matchers, use knot height, segment angle and ink density respectively to generate a fixed length code. Since the matchers are all similar in concept and differ only in the numeric feature used we will only describe one of them, the profile matcher.

The profile matcher uses the pattern of ascender and descender information in a scribble in a similar fashion to the word shape coder. However, rather than performing a string edit comparison the profile matcher reduces the pattern of knot height differences to a simple graph which can then be summed.

Profile coder

The coding part of the profile algorithm works as follows. It uses the same median line as found by the word shape code as the basis for deciding which knot points are ascenders and which are descenders.

```
profile coder:
        for each knot point in the scribble {
                if knot.y > median      add (knot.x, knot.y) to upper-graph
                        else            add (knot.x, knot.y) to lower-graph
        }
        resample upper graph to N equidistant points
        resample lower graph to N equidistant points
        return 2N element integer vector contain the two resampled graphs
```

Where N is a parameter which trades off match time (and storage cost) against accuracy. We use N=20, on our development data set lower values lose too much accuracy whereas values higher than 20-25 show now extra increase in accuracy but increase the match time.

The graphing procedure is shown diagrammatically in figure 2. For each of the knot points above the median line we add a point on the upper profile graph with the same (x, y) coordinate values as the knot. In resampling we then pick N x-values equally spaced between the start and end of the scribble and calculate the y-value by linear interpolation between the two nearest knot points. This process is repeated separately for those knot

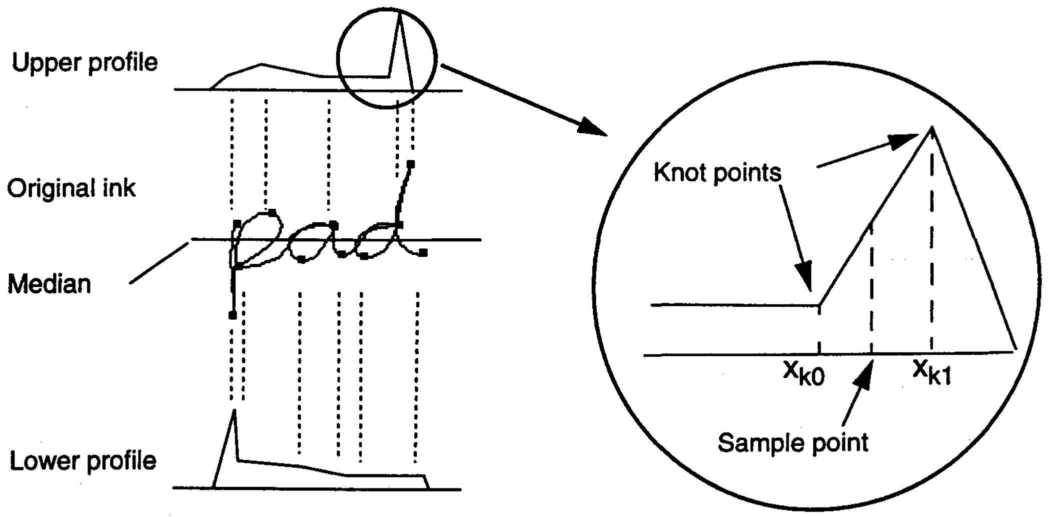

Figure 2 - Profile graph resampling

points lying below the median line and the two sets of N integer y-values are concentrated into a single length 2N feature vector.

Profile matcher

When comparing two scribbles with the profile matcher we use a simple linear distance measure - the summed absolute differences in the feature vector. If S1 and S2 are the length 2N feature vectors for the two scribbles then the distance between them is simply defined as: $d_p = \sum_{i=0}^{2N-1} |S1_i - S2_i|$.

The profile metric (and other linear metrics) run in approximately 10% of the time of the beam limited edit-distance metrics and allow us to prune the search table by a factor of around 5.

6.2 Constant feature vector

The final possibility is to chose some simple global numeric features which characterise the scribbles and thus offer the potential for indexing into the search table. For example, we might imagine that the number of strokes (pen ups) used to write a given word might be quite stable so that indexing the matching table on a feature like "number of strokes" would permit a very efficient search.

In practice, the number of strokes is not stable due to variability in ligatures and diacritical marks, however a systematic empirical search revealed features which are sufficiently stable to be usable as indices. These are, in decreasing order of stability:

- length (defined as the total number of velocity minima knot points, summed across strokes)
- aspect ratio of the scribble's bounding box
- first and second order horizontal moments
- finger print derived from the syntactic match code.[1]

None of the features is stable enough to act as the sole index into the table. However, by treating the total set of features as a feature vector we can estimate the similarity of each table entry to the query using some appropriate vector distance measure. Experiments with weighted versions of linear (city block metric), Euclidean and fuzzy distance measures indicated than simple variance weighted city-block distance was the most accurate.

Computing such vector distances requires only 2-3% of the time for a full beam limited edit-distance match and allows us to prune the search table by a factor of 2.

6.3 Summary

Both the linear and constant time pruners are effective in our problem domain. They can also be successfully combined to provide greater pruning power than either approach alone.

7 Putting it all together

We have explored and described a wide variety of methods for reducing the computational cost of scribble matching. The software components implementing these methods may be combined into different system designs according to application requirements. In this section, we describe one such system design and report on the resulting improvement in performance.

We assume an application in which it is useful to return, say, the top 5 matches (e.g. to support error recovery). Hence, we do not incorporate branch and bound pruning into the system design for the reasons given in section 5. Instead, the system design is based on a cascade of matchers separated by pruning stages as illustrated in figure 3. The first matcher utilizes the constant time feature vector of section 6.2. Next come

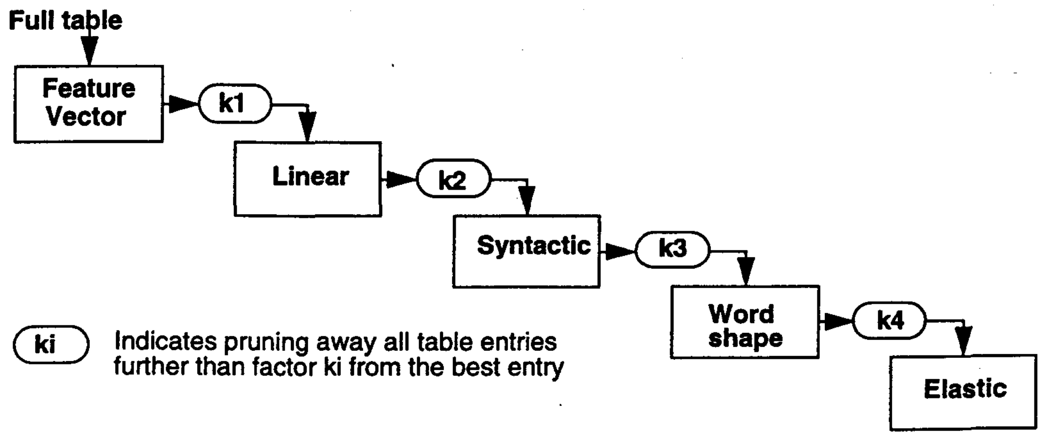
Full table
Feature Vector
k1
Linear
k2
Syntactic
k3
Word shape
k4
Elastic
ki
Indicates pruning away all table entries further than factor ki from the best entry

Figure 3 - Cascade pruning architecture

the linear matchers of section 6.1, followed by the original edit distance matchers using a beam-limited dynamic programming search as described in section 4. The edit distance matchers are themselves cascaded to reflect the better pruning performance of the syntactic matcher, and the greater computational cost of the elastic matcher.

1. This was a weighted sum of the counts of those symbolic labels found to be both stable and discriminating, specifically symbols 'y', 'o', 'p' and 'u' as described in [1].

Finally, the overall matching distance of the codes remaining in the table at the end of the cascade is generated by a variance-weighted linear sum of the three edit distance scores. Again experiments with Euclidean and fuzzy based score combinations were less successful than the simple linear approach.

Clearly the pruning after each stage may introduce errors which cannot be recovered. Hence, there is a balance between accuracy and speed which can be chosen to suit a given application. Our approach of scaling the pruning threshold by the best match distance (as described in section 6) means that the performance is relatively robust against changes in the k_i coefficients. The algorithm automatically adjusts to the problem by spending additional processing time on the ambiguous cases. Choosing a reasonable set of pruning parameters for our development data set, the pruning levels achieved when applied to the total test datasets (T2+T3) described in [1] are shown in table 5:

Table 5: Performance of the cascade pruning implementation

	Fraction passed on	Cumulative error	Relative time in stage[a]
Preprocessing	100%	0%	35%
Const	50%	0.15%	7%
Linear	18%	0.42%	13%
Syntactic	4.3%	0.7%	25%
Word shape	2.5%	0.75%	10%
Elastic	2.5%	0.75%	10%

a. Timings measured on an HP 735

In absolute terms this test data requires an average search time of 83ms on the development workstations, corresponding to a search time of 0.83s on a 386/33MHz processor. This represents a 33-fold speed increase over the unoptimised match cost described in section 3. The final average accuracy figures measured on this test dataset are 97.1% correct match and 98.8% top 5 match which compares well to the corresponding figures of 97.2% and 99.3% obtained without pruning.

8 Conclusions

Efficient scribble matching turned out to be a deceptively subtle problem and many standard approaches to table indexing and fast nearest neighbour matching proved less effective than desired. Our final approach of using a cascade of cheaper matching algorithms to reduce the search space for the expensive matchers has turned out to be cheap, effective and quite simple to implement. As a side effect we now have flexible control over the speed/space/accuracy trade-offs for the cascade which can be adjusted to suit different applications. Certainly for our example target platform (a 33MHz Intel 80386 processor) we were able to reach our target of sub-second searches without significant loss of accuracy.

Several avenues of research remain open for further exploration. Firstly, the techniques used in the metric space algorithm may be adapted to provide a fast index into clusters of similar scribbles and so might achieve "top N" search performances closer to those obtained for "top 1" searches. Secondly, we have yet to fully explore alternative error correcting codes as described in section 4.3 which might provide an additional avenue of performance improvement. Finally, the use of branch and bound methods to rapidly return the best match and perform a slower "top N" search in the background remains a useful option for some applications.

9 References

[1] Richard Hull, Dave Reynolds and Dipankar Gupta. Scribble matching. *Submitted to 4th Int. Wkshp on Frontiers of Handwriting Recognition.* 1994

[3] Rejean Plamondon. An evaluation of motor models of handwriting. *IEEE transactions on systems, man and cybernetics.* 19(5):1060 - 1072, September/October, 1989.

[4] Hans-Leo Teulings and Lambert Schomaker and Gerben Abbink and Eric Helsper. Invariant segmentation of on-line cursive script. *Proceedings Sixth International Conference on Handwriting and Drawing, ICOHD'93, Paris,* pages 198 - 200. 1993.

[5] Hans-Leo Teulings and Lambert Schomaker and Pietro Morasso and Arnold Thomassen. Handwriting-analysis system. *Proceedings 3rd International Symposium on Handwriting Computer Applications,* pages 181 - 183. July, 1987.

[6] J. Sklansky and V. Gonzalez. Fast polygonal approximation of digitized curves. *Pattern Recognition.* 12327 - 331, 1980.

[7] David Sankoff and Joseph B. Kruskal (editor). *Time warps, string edits, and macromolecules: The theory and practice of sequence comparison.* Addison-Wesley, 1983.

[8] Okuda et al. Correction of garbled words based on Levenstein metric. *IEEE transactions on computing.* C-25(2):??, Feb, 1976.

[9] R.A. Wagner and M.J. Fischer. The string-to-string correction problem. *Journal of the Association for Computing Machinery.* 2. 168-173, 1974.

[10] William J. Masek and Michael S. Paterson. How to compute string-edit distances quickly. David Sankoff and Joseph B. Kruskal (editor), *Time warps, string edits, and macromolecules: The theory and practice of sequence comparison,* Addison-Wesley, 1983.

[12] Jorma Tarhio and Esko Ukkonen. Approximate Boyer-Moor String Matching. *SIAM Journal of Computing.* 22(2):243-260, April, 1993.

[13] D. Dolev, Y. Harai and M. Parnas. Finding the neighborhood of a query in a dictionary. 1993.

[14] Patrick H. Winston. *Artificial Intelligence.* Addison-Wesley, 1977.

[15] E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Patt. Recognition Letters.* (3):145 - 157, July, 1986.

[16] Andras Farago, Tamas Linder and Gabor Lugosi. Fast nearest-neighbor search in dissimilarity spaces. *IEEE PAMI.* 15(9):957-962, September, 1993.

[17] Richard Hull. *Lower bound pruning in computing string edit distance.* Gryphon discussion document Gryphon-dd-12a, Hewlett Packard Laboratories, Bristol, May, 1994.

Architectures for efficient scribble matching

Dave Reynolds, Dipankar Gupta,
Richard Hull
Office Appliance Department
HP Laboratories Bristol

HPL-94-62
July, 1994

handwriting, matching, recognition, scribble

Scribble matching is a form of partial handwriting recognition that allows like scribbles to be identified. In a companion paper [1] we describe a set of three algorithms which achieve high accuracy on this task. In this paper we address the issue of increasing the computational efficiency of these algorithms by over an order of magnitude to the point where interactive use on a modest platform such as Intel 80386 is possible.