Note on approximate comparison of sequences with normalization

Liisa Räihä
Department of Computer Science
University of Tampere
FIN-33101 Tampere, FINLAND

August 10, 1995

Note on approximate comparison of sequences with normalization

Liisa Räihä
Department of Computer Science
University of Tampere
FIN-33101 Tampere, FINLAND

Abstract: The edit distance is a measure e.g., to classify a sequence to a correct word or concept. Often, particularly with editing errors, the edit distance measures are linearly dependent on the lengths of their parameter sequences. The normalized distances are an attempt to remove this dependency. In this work we show adjustments to the algorithms of Marzal and Vidal for the computation of the normalized distances.

Keywords: sequence comparison, algorithms

Chapter 1

Introduction

Sequence comparison has many application areas, for instance in character recognition, speech recognition, computer vision, pattern matching in biological databases, spelling correction in user interfaces and parser recovery in compilers. The general framework either for the algorithms or data structures or similarity measures or all of them is offered in the book edited by Sankoff and Kruskal [76] and in the overviews of Waterman [93], and Myers [61], in the articles of Galil and et all. [27, 26], Chang and Lampe [13], Eppstein [22], Rosenfeld and Pfaltz [74], Findler and van Leeuwen [23] and Basseville [9], in the paper for classification by Li et all.[50]. One of the first surveys are from 1980 by Hall and Dowling in various application areas [31] and from 1978 by Toussaint [83] in pattern recognition. However, for spelling correction there is an older survey of Peterson [68] and this is extended by K. Kukich [46].

In the sequel we concentrate on the editing model approach with the distance measures derived, projected or extended from the Levenstein distance. Although not listing, there have been algorithms also for two-dimensional data, good if the data is periodic, or of good average properties. The Levenstein distance measure [48] that is the number of the editing operations: insertions, deletions, substitutions . The most often used model introduces costs to the editing operations, and the edit distance is the minimum of costs of the editig sequences that may transform a sequence (string) to another. The algorithms developed by Wagner, Sellers and others have been based on this measure [90, 77, 78, 55, 86, 95, 29, 32, 38, 64].

The polygon matching in geometric data structures has been a more complex application area which have had several approaches, see [4, 6, 7, 5, 10, 11, 12, 20, 43, 81, 92]. In many of these sequence matching procedures using the angles and lengths of the sides, the sides may be normalized, thus allowing scaling for the object. Stereo vision has its own string-correspondence-problem, see e.g., [36, 44, 45, 51, 57, 91]. Handwritten-character matching may use the basic approach with the symbols consisting of two attributes, see on-line recognition e.g. , [41, 54, 49]. Moreover, for Chinese Character Recognition other operations such as merging and splitting have been introduced, [84, 85, 42]. The sequence comparison algorithm based on partial sorting tested for the histograms is presented [71]. The color histograms are used as the inputs for the similarity comparison also in the object recognition in [80], and in that paper the difference is normalized with by the sum of weights of the symbols in the model sequence. The normalization is done after processing the value-by-value differences. Marzal and Vidal introduce a concept called normalized edit distance [54]). They normalize by the length of the edit path, not by the lengths of the sequences. They propose on-line algorihm, which requires $O(m^2)$ space and $O(nm^2)$ time, if m is the length of the shorter sequence and n is the length of the longer sequence. In experiments of Marzal and Vidal done in the character classification the new similarity measure seems to be more reliable. Since the costs of the operations are defined in the similarity matrix (application dependent measurements), and this gives a non-metric measure, the normalization makes the distance to fulfill the triangle inequality property.

In this paper we extend and modify the basic algorithms, to implement the recurrence of Marzal and Vidal to cope with the normalization that allows more error in longer sequences, and also more minor mistakes but less big mistakes.

Chapter 2

Editing models

When two sequences are compared, there is also a measure to say how similar or dissimilar these two are. Often we have a distance measure that describes, how far they are from each other with the given metric or a cost function. The distance function is not always metric. Some research and practice use what they call a similarity measure: how similar the structures are is measured with some computational measure. We assume the following: If the distance $D(X, Y)$ is to be computed between sequences X and Y with lengths n and m, respectively, the distance is the minimum sum of costs of the edit operations to transform a sequence to another. If the cost function is symmetric, it makes $D(X, Y) = D(Y, X)$.

We dot not expect the distance to be metric, we request only that the editing sequence fulfills the following restrictions: all the editing operations can be done in parallel and the editing sequence is monotonic. The editing sequence is monotonic, if there are no two editing operations $\psi(X_i, Y_k)$ and $\psi(X_j, Y_h)$ such that $i < j$ but $k > h$. On the other hand, if the distance is metric, these properties will be got to the minimum editing sequence in the computation.

The distance is computed into a matrix $D = (d_{ij})$ of size $n \cdot m$. However, if the editing operations are not needed, due to the different constraints by cost functions , denoted by γ, the computation needs only two rows of the matrix, the most recently computed. The basic model is given by Equation (1).

Equation 1. Basic model:

$$\begin{aligned} D(0,0) &= 0, \\ D(i,j) &= min \quad \{D(i-1, j-1) + \gamma(x_i \rightarrow y_j), \\ &\qquad\qquad D(i-1, j) + \gamma(x_i \rightarrow \epsilon), \\ &\qquad\qquad D(i, j-1) + \gamma(\epsilon \rightarrow y_j)\} \end{aligned}$$

For the basic model with the restricted set of cost functions we have the bounds by Wong and Chandra [94].

Theorem 2.0.1 (Wong and Chandra(1976)) *If the editing costs are restricted so that each insertion costs C_I, each deletion costs C_D, and each substitution costs C_S (if the symbols are the same, the cost is zero), the alphabet is arbitrarily large and it holds that $0 \leq v = C_S/(C_I + C_D) \leq 1$, then the minimal number of entries to be computed, i.e.*

symbol equality comparisons, to be made in the worst case is $M(n,m)$. *The upper bound and the lower bound is for* $M(n,m)$: $m(n-m+1)+2+\sum_{1\leq k}^{k\leq m-1, m-(k/v)\geq 0} \lceil m-k/v \rceil \leq M(n,m) \leq mn - \lfloor m(1-v) \rfloor \lfloor m(1-v)+1 \rfloor$

For a general set of cost functions we can bound the computation by the minimum of the deletion and insertion costs, δ. The very idea is about the same as in the papers [72, 86]. But recall, the sets U_k and V_k used in the proofs of Wong and Chandra are in fact the same diagonals that are met in the approximation algorithms. And from there can be found the idea verifying the most distant entry (entries) from the main diagonal as can be with restrictions set by the cost function.

A diagonal $p, -n \leq p \leq n$ (when $p \geq 0$ starting from the entry $d_{0,p}$ in the matrix and ending at $d_{n-p,n}$) is the ordered set of entries $d_{i,j}$, where $i = j+p, 0 \leq j \leq n$. A group of diagonals is a band $b = \{p| - b \leq p \leq b\}$. Similarly a frontier $k, 0 \leq k \leq 2n$ (when $k < n$ starting from the entry $d_{k,0}$ in the matrix and ending at $d_{0,k}$) is the ordered set of entries $d_{i,j}$ where $k = j + i$, $1 \leq i,j \leq n$. A frontier is a concept used also in parallel processing. If we can bound the computation with the number of insertions and deletions that may take place, or with the threshold, the computation can be narrowed only to the narrow band of diagonals of the distance matrix, e.g., to diagonals $-p, -p+1, \ldots, 0, \ldots, p$. On the other hand, examining the values on the frontiers $k = 0, 1, \ldots$ we can bound with the threshold.

The normalized distance measure adds a third dimension: the editing path length to the distance table. Thus the following recurrence equation exists [54]:

Equation 2. Normalized-with-the-path-length model:

$$D(i,j,k) = \begin{cases} min & \{D(i-1,j-1,k-1) + \gamma(x_i \ \rightarrow \ y_j), \\ & D(i-1,j,k-1) + \gamma(x_i \ \rightarrow \ \epsilon), \\ & D(i,j-1,k-1) + \gamma(\epsilon \ \rightarrow \ y_j)\} \forall k : max(i,j) \leq k \leq i+j \\ \infty & \forall k : k \leq max(i,j) \bigvee k \geq i+j \end{cases}$$

When the cost function is metric, and the distance function is the sum of the costs, there is no single deletion followed by a single insertion or vice versa in the editing sequence. This follows from the property: $\gamma(x_i \ \rightarrow \ y_j) < \gamma(x_i \ \rightarrow \ \epsilon) + \gamma(\epsilon \ \rightarrow \ y_j)$.

With the normalized distance measure this statement is not anymore true.

A simple example would be a case, where we have a constant cost for every operation - an indel costs 1 and a mismatch costs 2 - and the example sequences to be compared are AB and BB: with the path of length 2 we have a normalized distance 2/2, and with the path of length 3 we have 2/3.

The basic model can be extended with a transposition [53], where in the simplest case two adjacent symbols have exchanged their positions without any gaps between in either of the sequences [21].

Similarly the computation can be bound on bands and frontiers of the distance matrix, if we have information on the classification thresholds. For the equation(3), we have to check only the entries of the frontier $k, k+1, k+2, k+3$ for values greater than or equal to t. And the band is at most $p+1$, where $p <\min(n, t/\delta)$, where $\delta =$ minimum of positive insertion and deletion costs, see for instance [73].

Chapter 3

Computation

Computing the editing distance is similar to the finding the shortest path in the grid graph. The very idea in the algorithm developed in [54] is that for all possible lengths of paths the minimum cost is computed, and the minimum cost path of length k is to be normalized by dividing its cost by the length of the path. The distance is the minimum of the normalized minimum path costs.

Note that the edit path length is greater than or equal to the number of the editing operations. The edit path length includes also the number of the perfect matches.

3.1 Basic algorithm

The very basic algorithm based on the recurrence Equation 2 can be simply written as Algorithm 3.1.1. It cuts some computation with the threshold.

Algorithm 3.1.1 *Test-and-compute (X, n, Y, m, t, s)*

```
Output: s; /* a distance * or rejection if s ≥ t*/
Begin
T = t(n + m);
D[0,0,0] = 0;
for (k = l; k < n + m + 1); k + +) {
      minDist = T;
      for (i = max(0, k − m); i <= min(k, n); i + +) {
            for (j = min(k, k − i); i <= min(k, m); j + +) {
                  delD = D[i − 1, j, k − 1] + del(X(i));
                  insD = D[i, j − 1, k − 1] + ins(Y(j));
                  subsD = D[i − 1, j − 1, k − 1] + subs(X(i), Y(j));
                  D[i, j, k] = minOf(delD , insD, subsD);
                  minDist = min(minDist,D[i, j, k]);
            };
      };
      if (minDist >= T) {
            return(reject); /* all paths of length k have certainly distances greater than threshold */
            }
};
for (kk = n; (kk < k); kk + +)
      s:= min(s, D[m, n, k]/k);
      if s ≥ t return(reject); else return(s);
End
```

What we try to do is to limit the number of computations using the information used in the classification. First we set the stopping criterion by the knowledge collected on frontiers, f that contain entries $d_{i,j,\ldots}$, $i + j = f$.

Lemma 3.1.1 *In the distance matrix computed by equation(2) if the entries of the frontiers f and $f+1$ have values greater than or equal to t, then no entry on succeeding frontiers $f+2, f+3, \ldots,$ is less than t.*

.

Proof. By the equation(2) the entry $d_{i,j,\ldots}$ on frontier f depends on the entries $d_{i-1,j,\ldots}$, $d_{i,j-1,\ldots}$, and $d_{i-1,j-1,\ldots}$, and both indel costs are in the frontier $f+1$ and the substitution cost is in f. Since these values are greater than or equal to t, $d_{i,j,\ldots}$ must contain values greater than or equal to t. □

Next we get a relation between different subpaths. We show that it is possible for a given subsequence pair, the optimal costs of paths of different lengths do not increase or decrease monotonically.

Lemma 3.1.2 *If at any point of the computation of $D(i,j,*)$, the following is true for some k:*
$cost(x_i \rightarrow y_j)/k > (\gamma(x_i \rightarrow \epsilon)+\gamma(\epsilon \rightarrow y_j))/(k+1)$ and $D(i-1,j-1,k-2)/(k-2) < D(i,j,k-1)/(k-1)$, and $D(i-1,j-1,k-2))/(k-2)$
is the normalized distance between the subsequences $X_{1\ldots i-1}$ and $Y_{1\ldots j-1}$ then the optimal editing sequence with the normalized distance between subsequences $X_{1\ldots i}$ and $Y_{1\ldots j}$ is at least of length $k+1$.

Proof. We add the expressions on the right hand side and expressions on the left hand side of the comparisons, and we deduce the result. □

On the other hand, if the input sequence is to be classified to one of the candidate classes, its distance from the candidate sequence cannot be very large. Thus if we take a parameter to bound the distance computation, the algorithm can be modified to compute less entries. Let us assume that the normalized distance cannot be greater than t. Furthermore let us denote, that $T = t(n+m)$. Then the following lemma provides another stopping criterion.

Lemma 3.1.3 *If in the distance matrix computed by Equation(2) all the entries of the paths of length k have values $d_{i,j} \geq T$, then no entry on paths of length $k+1, k+2, \ldots$ cannot have a cost that would made the sequence acceptable.*

Now we can state formally that the basic algorithm works correctly.

Theorem 3.1.1 *The algorithm 3.1.1 computes correctly the normalized distance between two sequences, if this is under the given threshold. The computation succeeds in time $O(min(n+m, n+t(n+m)/\delta)nm)$.*

Proof. The algorithm is a straightforward implementation of the recurrence. By Lemma 3.1.3 (and also by 3.1.1) we can cut computation at certain point, if all entries fulfill the given condition. □

An another simple approach would be to start computing from the entries (i,j) and think k as an index in vector value of the entry (i,j) and limit the computation in that dimension.

To be more precise we can cut the computation to a band of only possible usable entries in the computation. With the edit distance in the basic model we get an upper bound for the bands of the maximum width, approximately s/δ, if s is the distance. With the normalized distance we do adjusting during the computation. The following lemmas can be proven from the dependencies of the entries in the distance table. First we check the right (upper) area .

Lemma 3.1.4 *If in the distance matrix computed by Equation(2) for some k for the entries $d_{i+1,j}(k)$ and for all $h > j-2$ $d_{i,h}(k)$ it holds that these entries all have costs greater than or equal to T, then no entry $d_{i+1,j+d}(k+1)$, where $1 \leq d \leq m$ cannot have a cost that would made the sequence acceptable.*

Next we check the (lower) left area.

Lemma 3.1.5 *If in the distance matrix computed by Equation(2) for some k and the entries $d_{i,j}(k)$ and for all $h < j$ it holds that $d_{i,h}(k) \geq T$, then no entry $d_{i+1,j-d}(k+1)$, where $1 \leq d \leq j$ cannot have a cost that would made the sequence acceptable.*

.

Last two lemmas give us the simple means to reduce computation. The algorithms have been written in C++.

Algorithm 3.1.2 *Test-and-compute* (X, n, Y, m, t, s)

Output: s; /* a distance * or rejection if $s \geq t$*/
Begin
$T = t(n+m)$;
minDist := ∞; D[0,0,0]=0;
for $(j = 1; (j < m+1)\&\&(D[0, j-1, j-1] < T); j++)$ D[0,j,j] = D[0,j-1,j-1] + ins(Y(j));
if $(j < m+1)$ lastPrevious = j-1; else lastPrevious = m+1;
for $(i = 1; (i < n+1)\&\&(D[i-1, 0, i-1] < T); i++)$ D[i,0,i] = D[i-1,0,i-1] + ins(X(i));
if (i ¡ n+1) firstThresholded = i-1; else firstThresholded = n+1; firstCountable = 1;
for $(i = 1; i < n+1; i++)$ {
    if (i < firstThresholded)
    minDistR = D[i,0,i]; else minDistR = T; pre=0;/*
    for $(j = firstCountable; j < lastPrevious\&\&pre < T; i++)$ {
        $minl$ = min(i, j); pre = minDist; minDist = T;
        for $(k = minl; k < (i+j+1); k++)$ {
            $delD = D[i-1, j, k-1] + del(X(i))$;
            $insD = D[i, j-1, k-1] + ins(Y(j))$;
            $subsD = D[i-1, j-1, k-1] + subs(X(i), Y(j))$;
            $D[i, j, k]$ = minOf(delD , insD, subsD);
            minDist = min(minDist,$D[i, j, k]$); };
        if (minDist <= T) {
            if (minDistR == T) firstCountable++;
            if (lastPrevious == j-1) {
                lastPrevious = j;
                break; /* from the current row */
            }
            if (lastPrevious == j+1) {
            lastPrevious = j;
            break; /* from the current row */
    };}
    minDistR= min(minDistR,minDist);
    if (minDistR == T) return(reject);
}
for $(k = n; (k < (n+m)); k++)$
    s:= min$(s, D[m, n, k]/k)$;
if $s \geq t$ return(reject); else return(s);
End

The space can be reduced as done in [33, 87, 54] by keeping only record on the contents of the current and the previous row. This algorithm can be used on-line if the rows from 3 to 6 are inserted inside the loop transforming at the same time them from for-loops into if-statements. Since the tracking of minimum values increases the constant time slice for each entry computed, this kind of optimization is useful for sequences of certain length.

Theorem 3.1.2 *The Algorithm 3.1.2 computes the distance between two sequences or rejects, if the distance is not less than the given threshold. It runs in time* $O(n \cdot (min(t/\delta + 1, m))^2)$ *and in space* $O(m^2)$.

Proof. The first row of the matrix $D[0, *, *]$ is computed for suitable indices in vectors of length $O(n + m)$. Suitable means that there is at least a path of length k when inserting as is done k symbols. It is done in $O(min(m, mt/\delta)$ time, where the variable δ is the minimum indel cost in the sequences to be compared. If a path of length exceeding the threshold is met, then the computation is stopped and the variable lastPrevious is set accordingly to be the column to hold the last acceptable subpath. Similarly the first values in the first column, $D[*, 0, *]$ are computed in $O(min(n, nt/\delta)$ time. The firstThresholded is set to point the first row, where there are no acceptable paths in the first column. FirstCountable tells us the first column in the next row where there is a path under the given threshold. At row 17 it is increased only if all the entries sofar in the row reach the threshold, i.e., when there have been no suitable paths. Variable minDistRow computes the minimum of the row: in the beginning of the row it is set to be either the value of the first column or the threshold value. The variable lastPrevious is updated to point the right or to the left accordingly, depending on if the threshold met in the current row is in the earlier column or in the column to the right. Thus we discard the upper right corner unnecessary to compute. By lemmas the lines 7-28 are correct. The last lines compute minimum normalized distance in the set of paths in $O(n)$. □

The special cases of the cost function may bring the worst cases to the front.

Lemma 3.1.6 *If for all pairs* (x_i, y_j) *the following is true:* $cost(x_i \rightarrow y_j)/2 > (\gamma(x_i \rightarrow \epsilon) + \gamma(\epsilon \rightarrow y_j))$ *then the optimal editing sequence with the normalized distance between subsequences* $X_{1...n}$ *and* $Y_{1...m}$ *is of length* $m + n$ *and it does not have any substitutions.*

Proof. It is easy to see that for every case it is less expensive to use the detour, unless there is a perfect match. □

The algorithm for the case with no substitutions is as follows: we change the lines 13 -14 to the following lines:

```
if X(i) == Y(j) {
subsD =D[i-1,j-1,k-1];
D[i,j,k] = minOf(delD , insD, subsD);}
else D[i,j,k] = minOf(delD , insD);
```

3.2 Priority queue

If on the other hand we are sure that the only a very narrow band is actually checked we could set the algorithm differently. Instead of the third dimension as a vector we could declare it as a priority queue maintained partially ordered. The queue would maintain all the paths of different lengths computed in the given band. The rough outline would be: use a modified Dijkstra's algorithm on a graph containing a node for the entry D(i,j,k) and edges to that node, from nodes corresponding D(i-1,j,k-1), D(i,j-1,k-1) and D(i-1,j-1,k-1) with weights of the costs of the editing operations divided by k.

This approach would be useful only if it captures a band $b << m$. Otherwise it takes considerably more space. Let use declare

```
class q
{real av;
public:
real val; //the distance of the subsequences with a path of
int length; //length k
q next;
q(real v, unsigned int l);
q min(); // returns the minimal element from the queue
setnext(*q p); //returns the next element in the queue from the current
    cursor
insert(*q p); //insert an element in the sorted queue, but still with different
    l:s } ;
```

with obvious implementations.

Algorithm 3.2.1 *Test-and-compute* (X, n, Y, m, t, s)

```
Output: s; /* a distance * or rejection if s ≥ t*/
Begin
1.  T = t(n + m);
2.  minDist := ∞; D[0,0] = new q(0,0);
3.  for (j = 1; (j < m + 1)&&(D[0,j-1]. min() ¡ T);j++)
4.  D[0,j] .insert(new q(D[0,j-1] .min().v+ ins(Y(j),j));
5.  for (i = 1; (i < n + 1)&&(D[i − 1, 0].min(l) < T); i + +)
6.        D[i,0]. insert(new q(D[i-1,0].min(l).v+ ins(X(i)), i);
7.  for (i = 1; i < n + 1; i + +) {
8.        for (j = 1; j < m + 1; j + +) {
9.        for (delD = (D[i − 1, j].min(); delD.v/(delD.l + n + m − i − j) < t;
                    delD = (D[i − 1, j].setnext(); )
10.             D[i,j] .insert(new q(delD.v+ del(X(i),delD.l+ 1) ;
11.       for (insD = (D[i, j − 1].min(); insD.v/(insD.l + n + m − i − j) < t;
                    insD = (D[i, j − 1].setnext(); )
12.             D[i,j] .insert(new q(insD.v+ ins(Y(j),insD.l+ 1) ;
13.       for (subsD = (D[i, j].min(); subsD.v/(subsD.l + n + m − i − j) < t;
                    subsD = (D[i − 1, j − 1].setnext(); )
14.             D[i,j] .insert(new q(subsD.v+ subs(X(i),Y(j)),subsD.l+ 1) ;
15. } }
16. for (kdist = D[m, n].min(); k! = nil; kdist = D[m, n].setnext())
17.       s:= min(s,kdist.v/kdist.l);
18. if s ≥ t return(reject); else return(s);
End
```

Lemma 3.2.1 *For the queue of paths at node d(i,j) it is sufficient and necessary to hold only elements for which it holds: if (s, l) is the minimum element, then for all other entries (s_i, l_i) we have $s < s_i$ and moreover, $l < l_i$.*

Proof. The minimal element has the smallest (unnormalized) distance, sofar, between two subsequences X_i and Y_j: $s > s_i$ by definition. Assume we have an element (s', l') such that $s' > s$ and $l' < l$. Let us denote the optimal path using this path of length l' by k' and for it holds: the path k' consists of two parts: the prefix subpath of length l' and the suffix subpath starting from $d(i, j)$ is of length $p : k' = p+l'$. Its (unnormalized) distance is $s'+t'$. But the same suffix subpath can also be chosen with the path l, which makes the following constraint true: $\frac{s+t'}{l+p} < \frac{s'+t'}{l'+p}$. Thus this element (s', l') cannot be a part of the winning path. It is unnecessary. The other paths form an sufficient base to the further computation. $\square$

Lemma 3.2.2 *For the queue of paths at node $d(i, j)$ it is sufficient to maintain an increasingly monotonic queue: if (s_i, l_i) is an element, then for all other entries $(s_j, l_j), i < j$ we have $s_i < s_j$ and moreover, $l_j > l_i$.*

Proof. A similar argumentation as in 3.2.1 can be used to drop the unnecessary elements from the queue. □

Lemma 3.2.3 *For a minimal path at node $d(i, j)$ the distance computed so far fullfills the condition $s/(l + n + m - i - j) < t$.*

Proof. The required minimal path must have a normalized distance less than t. Assuming that the other edit operations do not increase notably the sum s, we can approximate the final normalized distance by dividing s by the longest possible path length containing this minimal path as a prefix path. □

The number of paths in the maintained set is limited to be at most $m + 1$. A Fredericson-Dijkstra method for the shortest path computation in planar graphs would solve this in $O(N\sqrt{\log N})$, where N is the number of vertices [24], and in this case N is nm. The number of paths $L(n, m)$ in the grid graph can be approximated by the expression $\log L(n, m) = -\log(\sqrt{2}-1)(n+m)-0.5\log n+O(1)$, if $n-m$ is $O(\sqrt{n+m})$. The expected length of the longest set of matches is bound by cn, where c depends on the size of the alphabet, (see Deken [19], and also work of Dancik[18]). For instance, with the alphabet size 15 we have for c a lower bound constant 0.327 and an upper bound constant 0.465. Thus we have an ecpected number of mismatches to be greater than $(1 - c)n$. Therefore we can state the following:

Theorem 3.2.1 *The algorithm 3.2.1 computes correctly the normalized distance between two sequences, if this is under the given threshold. The computation succeeds in time $O(min(n, m)nm)$. If there is a constant c such that the expected number of the matches is less than cn, then we can bound the expected computation to $O(nm^{\frac{(n+m)t-(1-c)n\delta}{\delta}})$, where δ is the minimum positive editing cost.*

Proof. The algorithm is a straightforward implementation of the recurrence except that the third dimension is covered only by the essential values. If the distances are not in the order of the path lengths we can drop some of the elements by Lemma 3.2.2. By Lemma 3.1.3 we can cut the computation at certain point, if all entries fulfill the given condition: less than $(n + m)t$. The expected number of the matches is less than the length of the shortest path. Therefore the paths have an expected cost greater than $(1 - c)n\delta$. The expected number of distinct paths in the priority queues cannot be greater $\frac{(n+m)t-(1-c)n\delta}{\delta}$, where δ is the minimum positive edit cost. □

3.3 Experiments

We have implemented the basic off-line algorithm and the priority queue algorithm. The implementation of the latter algorithm avoids two out of three unnecessary insertions to the queues by comparing the values first and making it possible to insert only the best possible node in the sets. Even more, in one case the old node, the value of which is used, can be directly reused without freeing and reallocating. We have run the algorithms without the threshold testing and with a reasonable threshold. The C++ coding of the algorithms is in the appendix A.

In many experiments, some shown in the appendix B, we have got the picture that the algorithm using the priority queue is not significantly slower than the simple algorithm. In some cases where rejection is the final result, the latter is clearly slower, see APPENDIX B, TABLE 2, row 6.

From the test runs we have collected the number of nodes computed in the distance matrix, the processing time and also in TABLE 1 we see the intervals in which the running times are. Moreover, there are some data about the shortest path length in matrix and the cost of the shortest editing sequence in the average for certain input descriptions. In the column showing the number of the entries/nodes computed in the distance matrix we see that the k-path method has sometimes smaller numbers. One case is when the probability of the indels is set to very high: 0.9. (See TABLE 3 in the APPENDIX B). Then over thousand entries have been left out from the computation. If we compare tables TABLE 1 and TABLE 2, we see that a reasonable threshold can speed up the computation and in the worst case do not increase the running time considerably.

Why the time values are not exact? I have used the getrusage system function of SunOS system. The accuracy of the time taking depends for instance, on the number of interrupts done during the run. Of course it depends on the accuracy of the clock, too. The times shown in the table include the user mode time and the system mode time of the process.

The space requirements for the implemented algorihms are very high. The base method uses vectors of size the maximum path length, and the number of vectors is quadratic. The k-path candidate method uses two vectors of linear size and a set of nodes needed to compute paths. The set is divided between different entries of vectors. With efficient storage allocator, this set cannot grow larger than mn. In fact this set can be as small as $n + 1$.

Chapter 4

Summary

Computing the editing distance or string-to-string-correspondence is an identification or a classification method in many applications. If the editing distance takes into account besides the costs of the operations also the number of operations, the problem complexity increases from quadratic to cubic. We have extended the algorithms of Marzal and Vidal to be used with the given threshold and also presented the variation using only possible candidates in the set of paths. The algorithms run in time $O(min(n+m, n+t(n+m)/\delta)nm)$ and in space $O(m^2)$, where n and m are the lengths of sequences, t is the threshold given by the user , and δ is the minimum cost. If there is a constant c such that the expected number of the matches is less than cn, then we can bound the expected computation to $O(nm\frac{(n+m)t-(1-c)n\delta}{\delta})$. The experiments show that in the computation of edit distance the basic algorithm and the priority queue based algorithm perform equally well. In the case of the rejection the latter may be slightly faster. The total space requirements can be in the worst case very different: using only two rows in both algorithms the priority queue approach would require as much as the links require the basic entry size multiplied. The lower bound to the size of the priority queue sets is maximum of $n+1$ and $m+1$.

Bibliography

[1] K. Abrahamson. Generalized string matching. *SIAM Journal on Computing*, 16(6):1039–1051, December 1987.

[2] V.A. Aho, D.S. Hirschberg, and J.D. Ullman. Bounds on the complexity of the longest common subsequence problem. *Journal of the ACM*, 23(1):1–12, 1976.

[3] T. Akutsu. Approximate string matching with don't care characters. *Proc. of the 5th Annual Symposium on Combinatorial Pattern Matching, CPM 94, Springer-Verlag LNCS 807,*, pages 240–249, 1994.

[4] H. Alt, B. Behrends, and J. Blomer. Approximate matching of polygonal shapes. *Proceedings of the 6th ACM Conference on Computational Geometry*, pages 186–193, 1991.

[5] A. Amir and M. Farach. Efficient 2-dimensional approximate matching of non-rectangular figures. *Proc. of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 212–223, 1991.

[6] M.J. Atallah. Computing some distance functions between polygons. *Information Processing Letters*, 17:207–209, 1983.

[7] M.J. Atallah, C.C. Ribeiro, and S. Lifschitz. Computing some distance functions between polygons. *Pattern Recognition*, 24(8):755–781, 1991.

[8] R. Baeza-Yates and G. Gonnet. Fast string matching with mismatches. *Information and Control*, 108(2):187–199, 1994.

[9] M. Basseville. Distance measures for signal processing and pattern recognition. *Signal Processing*, 18(4):349–369, 1989.

[10] H. Bunke and U. Buhler. Applications of approximate string matching to 2d shape recognition. *Pattern Recognition*, 26(12):1797–1812, 1993.

[11] H. Bunke and G. Kaufmann. Jigsaw puzzle solving using approximate string matching and best-first search. *Computer Analysis of Images and Patterns, Proc. of 5th International Conference, CAIP'93, Springer-Verlag, Lecture Notes in Computer Science 719*, pages 299–308, 1993.

[12] Shi-Kuo Chang, Qing-Yun Shi, and Cheng-Wen Yan. Iconic indexing by 2-d strings. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-9(3):413–428, 1987.

[13] W.I. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. *Proc. of the 3rd Annual Symposium on Combinatorial Pattern Matching*, pages 172–181, 1992.

[14] W.I. Chang and E.L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):116–124, 1994.

[15] V. Chvatal and D. Sankoff. Longest common subsequences of two random sequences. *Journal of Applied Probability*, 12:306–315, 1975. cited by Chang and Lawler in proc. of FOCS'90.

[16] F.J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, March 1964.

[17] V. Dancik. Expected length of longest common subsequences. PhD thesis, University of Warwick, United Kingdom, 1994.

[18] V. Dancik and M. Paterson. Upper bounds for the expected length of a longest common subsequence of two binary sequences. *Proc. of the 11th Annual Symposium on Theoretical Aspects of Computer Science, STACS'94, Springer-Verlag, LNCS 775*, pages 669–678, 1994.

[19] J.G. Deken. Probabilistic behaviour of longest common subsequence length. *in the book of Sankoff and Kruskal*, 359–362, 1983.

[20] I. Dinstein and G.M. Landau. Parallel computable contour based feature strings for 2-d shape recognition. *Pattern Recognition Letters*, 12(5):299–306, May 1991.

[21] I. Durham and D.A. Lamb and J. Saxe. Spelling correction in user interfaces. *Communications of ACM*, 26(10):171–176, October 1983.

[22] D. Eppstein. Sequence comparison with mixed convex and concave costs. *Journal of Algorithms*, 11:85–101, 1990.

[23] N.V. Findler and J. van Leeuwen. A family of similarity measures between two strings. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1:116–118, 1979.

[24] G.N. Frederickson. Shortest Path Problems in Planar Graphs. *Proc. of IEEE 24th Annual Symposium on Foundations of Computer Science*, 1983, 242–247.)

[25] Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17(4):52–53, 1986.

[26] Z. Galil and R. Giancarlo. Data structures and algorithms for approximate string matching. *Journal of Complexity*, 1(4):33–72, March 1988.

[27] Z. Galil and K. Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science*, 92:49–76, 1992.

[28] J. Gregor and M.G. Thomason. Dynamic progamming alignment of sequences representing cyclic patterns. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(2):129–135, February 1993.

[29] D. Gries and B. Burkhardt. Presenting an algorithm to find the minimum distance. Technical Report 88–903, Department of Computer Science, Cornell University, Ithaca, NY, 1988.

[30] R. Grossi and F. Luccio. Simple and efficient string matching with k mismatches. *Information Processing Letters*, pages 113–120, 1989.

[31] P.A.V. Hall and G.R. Dowling. Approximate string matching. *Computing Surveys*, 12(4):381–402, December 1980.

[32] P. Heckel. A technique for isolating differences between files. *Communications of the ACM*, 21(4):264–268, April 1978.

[33] D.S. Hirschberg. Linear-space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, June 1975.

[34] D.S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM*, 24(4):664–675, October 1977.

[35] D.S. Hirschberg. An information-theoretic lower bound for the longest common subsequence problem. *Information Processing Letters*, 7(1):40–41, 1978.

[36] Z.D. Hua and B. Dubuisson. String matching for stereo vision. *Pattern Recognition Letters*, 9:117–126, 1989.

[37] L.C.Kwong Hui. Color set size problem with applications to string matching. *Proc. of the 3rd Annual Symposium on Combinatorial Pattern Matching*, pages 227–239, 1992.

[38] R.L. Kashyap and B. John Oommen. The noisy substring matching problem. *IEEE Transactions on Software Engineering*, SE-9:365–370, May 1983.

[39] J. Kececioglu and D. Sankoff. Exact and approximation algorithms for the inversion distance between two chromosomes. *Proc. of the 4th Annual Symposium on Combinatorial Pattern Matching, CPM 93*, pages 87–105, 1993.

[40] J. Kececioglu and D. Sankoff. Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement. *Algorithmica*, 13(1/2):180–210, 1995.

[41] B. Klauer and K. Waldschmidt. An object-oriented pen-based recognizer for handprinted characters. *Computer Analysis of Images and Patterns, Proc. of 5th International Conference, CAIP'93, Springer-Verlag, Lecture Notes in Computer Science 719*, pages 586–593, 1993.

[42] K. Kobayashi, F. Yoda, K. Yamamoto, and H. Nambu. Recognition of handprinted kanji characters by the stroke matching method. *Pattern Recognition Letters*, 1:481–488, 1983.

[43] M.W. Koch and R.L. Kashyap. Matching polygon fragments. *Pattern Recognition Letters*, 10:297–308, 1989.

[44] M.I. Kolesnik. Fast algorithm for the stereo pair matching with parallel computation. *Computer Analysis of Images and Patterns, Proc. of 5th International Conference, CAIP'93, Springer-Verlag, Lecture Notes in Computer Science 719*, pages 533–537, 1993.

[45] A. Koschan. Dense stereo correspondence using polychromatic block matching. *Computer Analysis of Images and Patterns, Proc. of 5th International Conference, CAIP'93, Springer-Verlag, Lecture Notes in Computer Science 719*, pages 538–542, 1993.

[46] K. Kukich. Techniques for automatically correcting words in text. *Computing Surveys*, 24(4):377–439, 1992.

[47] G.M. Landau and U. Vishkin. Fast string matching with k differences. *Journal of Computer and System Sciences*, 37:63–78, 1988.

[48] V.I. Levenstein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Phys.Dokl.*, 10:707–710, 1966. Cited in ([SK83],[Ukk85a]).

[49] C. Li, K. Fan, and F.Lee. On-Line recognition by Deviation-Expansion model on dynamic programming matching. *Pattern Recognition*, 26(2):259–268, 1993.

[50] X. Li, N.S. Hall, and G.W.Humpreys. Discrete distance and similarity measures for pattern candidate selection. *Pattern Recognition*, 26(6):843–851, 1993.

[51] S. Lloyd. Stereo Matching using Intra and Inter-row Dynamic Programming. *Pattern Recognition Letters*, 4:273–277, 1986.

[52] D.F. Logan and L.A. Shepp. A variational problem for random young tableaux. *Advances in Mathematics*, 26:206–222, 1977. cited by Paterson and Dancik in RR268.

[53] R. Lowrance and R. Wagner. An extension of the string-to-string correction problem. *Journal of the ACM*, 22(2):177–183, April 1975.

[54] A. Marzal and E. Vidal. Computation of normalized edit distance and applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):926–932, 1993.

[55] W.J. Masek and M.S. Paterson. A faster algorithm for computing string-edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980. Also appeared in chapter under the title "How to compute string-edit distances quickly" in [SaK83].

[56] W. Miller and E.W. Myers. A file comparison algorithm. *Software — Practice and Experience*, 15(11):1025–1040, November 1985.

[57] T.P. Monks and J.N. Carter. Improved stripe matching for color encoded structured light. *Computer Analysis of Images and Patterns, Proc. of 5th International Conference, CAIP'93, Springer-Verlag, Lecture Notes in Computer Science 719*, pages 476–485, 1993.

[58] H.L. Morgan. Spelling correction in systems programs. *Communications of the ACM*, 13(2):90–94, February 1970.

[59] A. Mukhopadhyay. A fast algorithm for the longest-common-subsequence problem. *Information Sciences*, 20:69–82, 1980. cited by Hsu and Du in 1984.

[60] E.W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[61] E.W. Myers. An overview of sequence comparison algorithms in molecular biology. Technical Report TR-1991/29, Department of Computer Science, University of Arizona, Tucson, Arizona, December 1991.

[62] E.W. Myers and W. Miller. Row replacement algorithms for screen editors. *ACM Transactions on Programming Languages and Systems*, 11(1):33–56, 1989.

[63] N. Nakatsu, Y. Kambayashi, and S. Yajima. A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica*, 18:171–179, 1982. cited by Ukkonen in 1985.

[64] T. Okuda, E. Tanaka, and T. Kasai. A method for correction of garbled words based on the levenstein metric. *IEEE Transactions on Computers*, C-25(2):172–177, February 1976.

[65] B.J. Oommen. String editing with substitution, insertion, deletion, squashing and expansion operations. Technical Report SCS-TR-194, School of Computer Science, Carleton University, Ottawa, Canada, September 1991.

[66] M. Paterson and V. Dancik. Longest common subsequences. *Proc. of MFCS'94, Springer-Verlag, LNCS 841*, pages 127–142, 1994.

[67] M. Paterson and V. Dancik. Longest common subsequences. Technical Report RR-268, Department of Computer Science, University of Warwick,Coventry CV4 7AL, United Kingdom, May 1994.

[68] J.L. Peterson. Computer programs for detecting and correcting spelling errors. *Communications of the ACM*, 23(12):676–687, April 1980.

[69] P.A. Pevzner and M. S. Waterman. Matrix longest common subsequence problem, duality and hilbert bases. *Proc. of the 3rd Annual Symposium on Combinatorial Pattern Matching*, pages 77–87, 1992.

[70] R.Y. Pinter. Efficient string matching with dont-care characters. *in Combinatorics on Words (edited by Apostolico et all.)*, pages 11–29, 1985.

[71] L. Räihä. Approximate sequence comparison: A study with histograms. Technical Report A-1988-2, Dept. of Computer Science, University of Tampere, Finland, January 1988.

[72] L. Räihä. Approximate sequence comparison: A study with histograms. *Pattern Recognition*,12(1/2):159–169, November 1990 .

[73] L. Räihä. Approximate sequence comparison: Computation of the edit distance. Technical Report C-1991/59, Department of Computer Science, University of Helsinki, Helsinki, Finland, October 1991.

[74] A. Rosenfeld and J.L. Pfaltz. Distance functions on digital pictures. *Pattern Recognition*, 1:33–61, 1968.

[75] V. Salari and J.K. Sethi. Feature point correspondence in the presence of occlusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(1):87–91, 1990.

[76] D. Sankoff and J.B. Kruskal, editors. *Time Warps, String Edits and macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley Publishing Company, 1983.

[77] P.H. Sellers. An algorithm for the distance between two finite sequences. *Journal of Combinatorial Theory, Series A*, 16:253–258, 1974.

[78] P.H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms*, 1:359–373, 1980.

[79] R.M. Sinha, B. Prasada, G.F. Houle, and M. Sabourin. Hybrid contextual text recognition with string matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):915–925, 1993.

[80] M.J. Swain. Color indexing. Technical Report Computer Science 360, University of Rochester, November 1990.

[81] E. Tanaka, H. Awano, and S. Masuda. A proximity measure of line drawings for comparison of chemical compounds. *Computer Analysis of Images and Patterns, Proc. of 5th International Conference, CAIP'93, Springer-Verlag, Lecture Notes in Computer Science 719*, pages 291–298, 1993.

[82] W.F. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, 1984.

[83] G. Toussaint. The use of context in pattern recognition. *Pattern Recognition*, 10:189–204, 1978.

[84] W.H. Tsai and S.S. Yu. Attributed string matching with merging for shape recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-7:453–462, 1985.

[85] Y.-.T. Tsay and W.-H. Tsai. Attributed string matching split-and-merge for on-line chinese character recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(2):180–185, 1993.

[86] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1–3):100–118, 1985.

[87] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.

[88] E. Vidal, H.M. Rulot, F. Casacuberta, and J-M. Benedi. On the use of a metric-space search algorithm (aesa) for fast dtw-based recognition of isolated words. *IEEE Transactions for Acoustics, Speech, and Signal*, 36(5):651–660, May 1988.

[89] R. Wagner. On the complexity of the extended string-to-string correction problem. *Proc. of the 7th ACM Symposium on Theory of Computing*, pages 218–223, 1975.

[90] R. Wagner and M. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(2):168–178, 1974.

[91] Y.P. Wang and T. Pavlidis. Optimal correspondence of string subsequences. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(11):1080–1086, 1990.

[92] N. Washio, E. Tanaka, and S. Masuda. A similarity measure between 3-d objects and its parallel computation. *Computer Analysis of Images and Patterns, Proc. of 5th International Conference, CAIP'93, Springer-Verlag, Lecture Notes in Computer Science 719*, pages 322–326, 1993.

[93] M.S. Waterman. General methods of sequence comparison. *Bulletin of Mathematical Biology*, 46(4):473–500, 1984.

[94] C.K. Wong and A.K. Chandra. Bounds for the string editing problem. *Journal of the ACM*, 23(1):13–16, 1976.

[95] S. Wu, U. Manber, G. Myers, and W. Miller. An O(np) sequence comparison algorithm. *Information Processing Letters*, 35, 1990.

Appendix A

```
copyright by University of Tampere, Liisa Raiha 14.7.1995
#include "normalized.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define del(a) a
#define ins(a) a
#define subs(a,b) (((a-b)¡0)? (b-a) : (a-b))
#define minof(a,b) ((a < b)?a : b)
#define maxof(a,b)((a < b)?b : a)
#define minof3(a,b,c) ((a < b)?((a < c)?a : c) : ((c < b)?c : b))
int normbasic(double *s1, double *s2,int n,int m, double threshold, int &kl,
double &sdistance)
{
double ***D;
int i,j,k;
double DelK, InsK, SubsK,T, pathK, minDist;
//preprocessing
    D = (double***) calloc ((n+1), sizeof(double**));
    T = (n+m)*threshold;
    minDist =T;
    for (i=0; i¡=n; i++)
        { D[i] = (double**) calloc((m+1), sizeof (double*));
        for (j=0; j¡=m; j++)
        {
            D[i][j] = (double*) calloc((n+m+1), sizeof(double));
        }
    };
//computation
    int tmax ;
    D[0][0][0] = 0.0;D[0][0][1] = T;
    for (j=1; j<=m;j++)
    {D[0][j][j] = D[0][j-1][j-1]+ del(s2[j]);
        D[0][j][j-1]= T;
        if (j¡m) D[0][j][j+1]= T;} ;
    for (i=1; i<=n; i++)
```

```
    {
        D[i][0][i] = D[i-1][0][i-1] +ins(s1[i]); D[i][0][i-1]= T;
        if (i>1) D[i][0][i-2]= T;
        if (i < n) D[i][0][i+1]= T;
        minDist = T;
        for (j=1; j<=m; j++)
        {
            tmax = maxof(i,j);
            D[i][j][tmax-1] =T;
            if (tmax > 1) D[i][j][tmax-2] =T;
            if (i+j < n+m) D[i][j][i+j+1] =T;
            for (k=tmax; k<=i+j; k++)
            {
                DelK= D[i][j-1][k-1]+ del(s2[j]) ;
                InsK= D[i-1][j][k-1]+ ins(s1[i]);
                SubsK= D[i-1][j-1][k-1]+ subs(s1[i], s2[j]);
                D[i][j][k]= minof3( DelK, InsK, SubsK);
                minDist = minof(minDist, D[i][j][k]);
            }
        }
        if (minDist >= T) { return -1;}
    }
if (n ¿ m)
        {kl = n;sdistance = D[n][m][n] / n;}
    else {sdistance = D[n][m][m] / m;kl = m;}
    for (k=maxof(n,m) +1; k<= n+m; k++)
        { pathK= (D[n][m][k] / k);
        if (sdistance > minof(sdistance,pathK) ) {
        kl = k; sdistance = pathK;
        }
    }
    if (sdistance < threshold) return 0;
    return -1;
};
```

```
/*normalized.C */
#include "normalized.h"
#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>
#include <float.h>
#include <limits.h>
```

```
pqueue *entrypaths::min()
{
    lastretrieval = paths;
    return paths;
};
void entrypaths::findrightspot(pqueue *p,pqueue *q)
{ pqueue *pre; //
    pre = NULL;
    while((q− >next != NULL) && (p− >length > q− >length) )
    {
        if (p− >val ¡=q− >val)
        {
            p− >next = q− >next;
            pre− >next = p− >next;
            delete q;
            q = p− >next;
        }
        else
        {           pre =q; q=q− >next;
        }
    }
    if ((p− >length ¿ q− >length) ) // at the end of queue
    {
        if (p− >val ¿ q− >val)
        {
        p− >next = NULL; q− >next = p; insertionpoint = p;
        }
        else if (pre!= NULL ) // remove q?
        {
        pre− >next = p; p− >next = q− >next; delete q; insertionpoint = p;
        }
        else {
        delete p;
        }
        }
        else if ((p− >val ¡ q− >val) && (p− >length ¡ q− >length)) // in the middle {
        pre− >next = p; p− >next =q; insertionpoint = p; }
    else delete p;
};
entrypaths::insert(pqueue *p)
{
    if (paths == NULL) {
        paths =p; p− >next = NULL;insertionpoint=paths; lastretrieval=paths;}
    else {
        if (p− >val ¡ paths− >val)
            {
```

```
        if (paths− >length ¡= p− >length)
        {
            p− >next = paths− >next; delete paths; paths = p; insertionpoint =p; }
        else {
            p− >next = paths;
            paths =p;
            insertionpoint = p;
        }
        }
        else if (p− >length <= paths-¿length)
        {delete p;
        return 0;}
        else
        {
        if (insertionpoint− >length ¡ p− >length)
        {
            findrightspot(p,insertionpoint);
        }
    else
    if (insertionpoint− >length ¿= p− >length)
        {
// try to find a spot that a length and a cost are suitable
            pqueue *temp= paths;
            if (insertionpoint− >val ¿ p− >val)
            findrightspot(p,temp);
            else
            {delete p;
            }
        }
    }
  }
  return 0;
}
pqueue *entrypaths::getnext()
{
  if (lastretrieval != NULL)
    lastretrieval = lastretrieval− >next;
  return lastretrieval; }
void entrypaths::setnull()
{ pqueue *a;
  a = paths;
  while (paths != NULL) {
    a = paths− >next; delete paths; paths = a;
  }
  lastretrieval=NULL;insertionpoint=NULL;
}
```

```
void entrypaths::init()
{insertionpoint=NULL; lastretrieval=NULL; paths = NULL;}




int ndistance( double *s1, double *s2,int n,int m, double threshold, double &sdistance, int &k) {
entrypaths *D, *Dcurr, *temp; //pointers to the rows
int i,j;
double T, minDist;
pqueue *Ddel, *Dins, *Dsubs, *ks; pqueue *first;
    T = (n+m)*t;
    D= (entrypaths *)calloc(m+1, sizeof(entrypaths));
    Dcurr= (entrypaths *)calloc(m+1, sizeof(entrypaths));
    D[0].init(); Dcurr[0].init();
    D[0].insert((new pqueue(0.0,0)));
    for (j=1; j<=m; j++)
        { Dcurr[j].init(); D[j].init();
        first =new pqueue(D[j-1].min()− >val + del(s2[j]), j);
        D[j].insert(first);};
    for (i=1; i<=n; i++)
        { Dcurr[0].setnull();
        first = new pqueue(D[0].min()− >val + ins(s1[i]), i);
        Dcurr[0].insert(first);
        minDist = T;
        for (j=1; j<=m; j++)
        { Dcurr[j].setnull();
            Dsubs = D[j-1].min();
            Ddel = Dcurr[j-1].min();
            Dins = D[j].min();
            while ( (Dsubs!= NULL) —— (Dins != NULL) —— (Ddel != NULL)
            {
                if ((Dsubs != NULL) && ((((Dins!= NULL) && (Dins− >length >= Dsubs− >length)) —— (Dins == NULL)) && (((Ddel!= NULL) && (Ddel− >length >= Dsubs− >length)) ——(Ddel == NULL))))
                    { first = Dsubs;
                    Dsubs = D[j-1].getnext();
                    D[j-1].paths = Dsubs;
                    first− >val = first− >val + subs(s1[i], s2[j]);
                    if ((Dins!= NULL) && (Dins− >length == first− >length))
                    { if (first− >val > (Dins− >val + ins(s1[i])))
                        first− >val = Dins− >val + ins(s1[i]);
                        Dins = D[j].getnext();
                    };
                    if ((Ddel!= NULL) && (Ddel− >length == first− >length))
                    {
                        if (first− >val > (Ddel− >val + del(s2[j])))
                            first− >val = Ddel− >val + del(s2[j]);
```

```
                    Ddel = Dcurr[j-1].getnext();
                };
                first- >length = first- >length +1;
            }                else
                { if ((Dins! = NULL) && (((Ddel! = NULL) && (Ddel- >length >= Dins- >length))——(Ddel == NULL))) {
                    first= new pqueue(Dins- >val+ins(s1[i]), Dins- >length+1);
                    if ((Ddel!= NULL) && (Ddel- >length == Dins- >length))
{
                        if (first- >val > Ddel- >val + del(s2[j]))
                            first- >val = Ddel- >val + del(s2[j]);
                            Ddel = Dcurr[j-1].getnext();
                        }
                        Dins = D[j].getnext();
                    }
                    else {
                        first= new pqueue(Ddel- >val+del(s2[j]), Ddel- >length+1);
                        Ddel = Dcurr[j-1].getnext();
                    }
                }; //else
                if (first- >val < T) {
                    Dcurr[j].insert(first);
                    minDist = minof(minDist, first- >val);
                }
                else delete first;
            }
        }
        if (minDist >= T) return -1;
        else {
        temp=Dcurr; Dcurr=D; D=temp; }
    }
  ks=D[m].min();
  sdistance= ks- >val/(double)ks- >length;
  k= ks- >length;
  for (ks=D[m].getnext(); ks!=NULL; ks=D[m].getnext())
      {if (setmin(sdistance, ks- >val/ks- >length)) k=ks- >length;
  };

  if (sdistance ¡ threshold)
      return (0);
  else return(-1);

};

short setmin(double &sdistance, double val)
{ if (sdistance ¿ val) { sdistance =val; return TRUE; }
return FALSE; }
```

Appendix B

Sequences (lengths)	K-path(s) av. time	min time	max time	nodes computed	Base (s) av. time	min time	max time	nodes computed	av. distance	av. path	runs
5 & 6 (50,50)	1.29	1.24	1.38	44775	1.26	1.23	1.30	45477	0.865	64.72	25
5 & 7 (50,50)	1.34	1.30	1.43	45477	1.28	1.23	1.46	45477	0.443	60.12	25
5 & 8(50,50)	1.33	1.30	1.39	45477	1.27	1.25	1.35	45477	0.436	60.00	25
5 & 9 (50,46)	1.26	1.18	1.35	40181	1.14	1.10	1.20	40183	0.402	55.92	25
5 & 10(50,46)	1.17	1.14	1.20	40080	1.13	1.10	1.19	40183	0.520	59.36	25
5 & 11 (50,50)	1.27	1.22	1.38	43877	1.28	1.26	1.34	45477	1.39	66.6	25
12 & 13(50,50)	1.31	1.28	1.42	45477	1.28	1.23	1.47	45477	0.0958	50.00	20
12 & 14(50,50)	1.34	1.31	1.41	45477	1.28	1.25	1.31	45477	0.159	50.00	20
12 & 15(50,50)	1.33	1.27	1.41	45477	1.29	1.25	1.42	45477	0.0971	50.00	20
12 & 16(50,46)	1.33	1.28	1.42	40183	1.13	1.11	1.20	40183	0.253	50.05	20
12 & 17(50,46)	1.16	1.14	1.20	40183	1.14	1.12	1.18	40183	0.172	50.00	20
12 & 18(50,46)	1.19	1.17	1.24	40176	1.15	1.13	1.26	40183	0.320	50.05	20
12 & 19(50,50)	1.31	1.26	1.35	45168	1.31	1.25	1.41	45477	0.889	56.00	20
1(6,6)	0.005	0.0	0.01	135	0.005	0.00	0.01	135	0.733	9	4
2 & 3(100,100)	9.80	9.70	10.28	348452	10.19	9.82	11.32	348452	31.49	109.6	10

Table B.1: No Thresholds

Sequences	K-path av. time(s)	av. # of nodes	Base av. time(s)	. av. # of nodes	Comment
5 &6(50,50)	0.96	32109	1.36	45477	
5 &7(50,50)	1.23	40435	1.37	45477	
5 &8(50,50)	1.23	40507	1.38	45477	
5 &9(50,46)	1.10	36191	1.22	40183	
5 &16(50,46)	1.23	36548	1.22	40183	
5 &11(50,50)	0.83	**27859**	1.38	45477	dissimilar; rejected
12&13(50,50)	1.40	45477	1.26	45477	a worst case sequence
12 &14(50,50)	1.40	45477	1.27	45477	
12 &15(50,50)	1.41	45477	1.29	45477	
12 &16(50,46)	1.35	40183	1.13	40183	
12 &17(50,46)	1.23	40183	1.14	40183	
12 &18(50,46)	1.24	40183	1.14	40183	
12 &19(50,50)	1.39	45053	1.28	45477	

Table B.2: Threshold=(max+min)/2 ; Testruns = 5

Sequence	Description	Length	Possible Mult. Error in value	Prob. of indels
1	Handmade	6	0.00	0.00
2	Constant 50	100	0.00	0.00
3	Sequence 2	100	2.00	0.02
4	Sequence 2	100	2.00	0.00
5	sin x +1	50	0.00	0.00
6	Sequence 5	50	2.00	0.00
7	Sequence 5	50	0.20	0.00
8	Sequence 5	50	0.10	0.00
9	Sequence 5	46	0.20	0.10
10	Sequence 5	46	0.20	0.20
11	Sequence 5	50	0.20	0.90
12	Constant 1	50	0.10	0.00
13	Constant 1	50	0.10	0.00
14	Constant 1	50	0.20	0.00
15	Constant 1	50	0.10	0.00
16	Constant 1	46	0.20	0.10
17	Constant 1	46	0.02	0.010
18	Constant 1	46	0.20	0.20
19	Constant 1	50	0.20	0.90

Table B.3: Input description