Using Hirschberg's Algorithm to Generate Random Alignments of Strings.

Lloyd Allison,

Department of Computer Science,
Monash University,
Australia 3168.

email: lloyd@cs.monash.edu.au

submitted 1993, revised April 1994, accepted May 1994

Abstract. Hirschberg gave an alignment algorithm for the longest common subsequence problem that uses $O(n^2)$ time and O(n) space for two strings of length ˜n. A simple modification of the algorithm can sample string alignments at random according to their probability distribution. This is useful for statistical estimation of evolutionary distances of a family of strings eg. DNA strings. The algorithm's time and space complexity are unchanged.

Keywords: alignment, edit distance, Gibbs sampling, LCS, LCSS, Monte Carlo method, random.

1. Introduction.

An optimal alignment algorithm returns an optimal way of relating two strings according to some criterion. In some situations it is natural to model the differences between strings, or the "noise" in strings, or both, as being due to a random process. This paper describes a method of sampling not optimal but *random* alignments of two strings. Statistics gathered from such samples can be better than those gathered from optimal alignments.

The longest common subsequence (LCS), the edit-distance[Sell] or Levenshtein metric[Leve] and similar notions are used to compare two strings in situations where letters may be inserted, deleted or changed. This occurs in DNA and protein strings and in other applications[Sank]. Figure 1 shows the result of such a comparison. If we consider string A to be the parent and string B to be the child, there have been at least three mutations in the evolution of B from A. An *alignment* is a way to display such results: Zero or more special *null* characters, denoted ‘–’, are inserted into A and into B to give modified strings A’ and B’ such that |A|=|B|. A’ and B’ are written out one above the other as shown. It is then easy to see if a letter has been subject to a *copy*, a *change*, an *insert* or a *delete*. The *pairs* of letters in the columns of the alignment indicate a way of editing string A into string B. A pair <x,x> represents copy x (no mutation), a pair <x,y> represents change x into y, a pair <x,–> represents delete x and a pair <–,y> represents insert y. The only restriction on forming A’ and B’ is that the all-null pair <‘–’,‘–’> is not allowed to appear in the alignment.

There are many alignments, or ways of editing A into B, so the question arises what is an optimal one, given some criterion for comparing alignments? The LCS problem is to maximise the number of copies. The edit-distance problem is to minimise the number of *mutations*, ie. changes, inserts and deletes. The well known dynamic programming algorithm (DPA) [Sell, Sank] can find an optimal alignment for the LCS problem or for the edit-distance problem in O(|A|.|B|) time and space; see figure 2 (i) and (ii).

The DPA uses a matrix D. In the case of the edit-distance, D[i,j] holds the edit distance of substrings A[1..i] and B[1..j]. The boundary conditions for D follow from i being the edit-distance of the empty string and a string of length i. In general, to edit A[1..i] into B[1..j] there are three possibilities: (i) edit A[1..i-1] into B[1..j-1] and copy or change the character A[i] into B[j] as appropriate, (ii) edit A[1..i] into B[1..j-1] and insert B[j] or (iii) edit A[1..i-1] into B[1..j] and delete A[i]. The algorithm chooses the minimum cost option. Note that D[i,j] is computed from three neighbours, to the N, W and NW, which leads to an iterative algorithm. The edit-distance of A and B is given by D[|A|,|B|]. An optimal alignment can be found by tracing *backwards* through D, from D[|A|,|B|] to D[0,0], retracing the choices that were made in the minimisation steps that resulted in the value D[|A|,|B|]. The trace-back requires O(|A|.|B|) space. However, a row of D can be calculated from the previous row so space for just two rows is sufficient to calculate the numerical value of the edit-distance. Hirschberg[Hirs] used this fact to recover an optimal alignment in O(|A|.|B|) time and in just O(|B|) space.

It is possible to give a probability to each alignment in a natural way. First, each type of mutation is given a probability. This might, for example, come from a knowledge of typical errors in the replication process of DNA, or of properties of a scientific instrument that is used to read the strings. The probability of a copy is one minus the sum of the probabilities of the mutations. The mutation probabilities induce a probability distribution on the alignments of two given strings. For example, if P(copy) is high then alignments containing many copies are highly probable. In what follows, Hirschberg’s algorithm is modified to generate not an optimal alignment but a *random* alignment, sampled from the probability distribution of alignments. This is useful in statistical estimation of the relatedness of a family of strings where we want *typical* alignments rather than optimal ones.

There are fast algorithms for optimal alignment problems. For example, Ukkonen’s algorithm[Ukko] runs in O(n.d) time where d is the edit-distance; this is fast if d<<n. Unfortunately such algorithms cannot be used for sampling because some are restricted to mutation costs being small integer values and because we need to take all possible alignments into consideration.

2. Alignment as an Inference Problem.

Alignments are used to compare biological macro-molecules such as DNA and proteins. DNA consists of a string of the four nucleic acids denoted {A, C, G, T}. DNA strings mutate so comparing them can indicate if they are (probably) related, how closely and in what way.

The different character mutations occur with different frequencies in different data sets. Therefore we should not use fixed costs for mutations but should base costs on the probabilities of the mutations that suit the given situation. A particular alignment is (just) a hypothesis about two strings; it indicates one possible relationship between them. If we know the probabilities of copy, change, insert and delete, denoted P(<x,x>), P(<x,y>) etc., the probability of an alignment is the product of the probabilities of the pairs in the alignment. (Strictly, this assumes that the length of the alignment is known in advance, but all plausible alignment lengths have very similar probabilities in practice and, as we only require the alignments' *relative* probabilities, length can be ignored here.) The DPA of figure 2 (iii) finds the probability of a most probable alignment; an actual alignment can be found by a trace-back as before. Conversely, given an alignment we can estimate the probabilities of the character mutations from the frequencies of pairs in the alignment. If the probabilities of character mutations are not known in advance, they can be optimised so as to maximise the probability of the best alignment say [AlWY, Bish, Thor], but that possibility is not examined further in this paper. Note that it is actually more convenient for the algorithms to work with the -log of probabilities rather than with probabilities directly, otherwise underflow would soon occur with strings of any significant length. (In this light the mutation cost used in the edit-distance DPA, figure 2 (ii), can be taken to be a crude approximation to the -log of a mutation probability.)

That two strings are related is itself (just) a hypothesis, called the *r-theory* for short. The probability of strings A and B assuming the r-theory, denoted P(A&B), can be found by summing the probabilities of *all* alignments of A and B [AlWY, Bish, Thor]. (A and B must be related by some alignment given that assumption.) Fortunately P(A&B) can be calculated in O(|A|.|B|) time, as in figure 2 (iv). The number of alignments is exponential in the lengths of the strings but that does not cause any problem because alignments are considered in groups not individually. D[i,j] holds P(A[1..i]&B[1..j]) in this version of the DPA. It is the sum of the probabilities of all alignments of A[1..i] and B[1..j] ending in either (i) copy or change A[i] to B[j], (ii) insert B[j] or (iii) delete A[i], ie. the sum over all alignments. Note that different alignments are exclusive hypotheses and their probabilities can be added.

3. Sampling Alignments.

Hirschberg's algorithm[Hirs] recovers an optimal alignment for one of the string comparison problems by a recursive divide and conquer procedure. The algorithm is first summarised and then modified so as to sample random alignments; the aim of the latter is that the chance of sampling a particular alignment should be proportional to its probability.

Given A[Alo..Ahi] where Ahi>Alo and B[Blo..Bhi] where Bhi≥Blo, the mid-point of A, i=(Alo+Ahi)div 2, is calculated and a corresponding point, j, is found at which to partition B into B[Blo..j] and B[j+1..Bhi]. To find j, A[Alo..i] is compared with B[Blo..Bhi] giving D[i,j] for all j in Blo−1..Bhi, and the reversed strings A[Ahi..i+1] and B[Bhi..Blo] are compared giving D'[i+1,j'] for all j' in Bhi+1..Blo. D[i,] contains the results of comparing A[Alo..i] with all prefixes of B[Blo..Bhi]. D'[i+1,] contains the results of comparing, in a reversed sense, A[i+1..Ahi] with all suffixes of B[Blo..Bhi]. This can be done in O(|B|) space using the DPA, figure 2 (i), (ii) or (iii). Adding corresponding elements of D[i,] and D'[i+1,] gives costs conditional on each possible partition of B and a best one is chosen. In the case of the DPA of figure 2 (iii), D[i,] and D'[i+1,] hold probabilities and corresponding elements are multiplied rather than added. Given j, two smaller problems comparing A[Alo..i] to B[Blo..j] and A[i+1..Ahi] to B[j+1..Bhi] are then solved recursively. The base-cases of the algorithm, when Alo=Ahi or when either string is empty are trivially solved. The overall time complexity remains O(|A|.|B|) because the DPA is used on one full-sized O(|A|.|B|) problem, two problems whose total size is half that of the original problem, four problems whose total size is quarter that of the original problem, etc.

Hirschberg's algorithm is modified to generate a *random* alignment, sampled from the probability distribution of alignments, as follows. Given A[Alo..Ahi] where Ahi>Alo and B[Blo..Bhi] where Bhi≥Blo, i=(Alo+Ahi)div 2, rather than choose an optimal point j to partition B[Blo..Bhi], instead j is sampled *at random* from the probability distribution of possible values taken over *all* alignments. Once j is chosen, two smaller resulting problems are solved recursively.

Consider A[i]. It must occur somewhere in every alignment. Consider an arbitrary alignment. A[i] either occurs as <A[i],'-'> which we call type (i) or as <A[i],B[j]> which we call type (ii). Let j equal the largest j'≥Blo such that B[j'] occurs with or before A[i] in the alignment, and let j equal Blo−1 if no such B[j'] exists; Blo−1≤j≤Bhi. There is some probability distribution on possible values for j when we now consider *all* possible alignments: The probability of a particular value for j is proportional to the sum of the probabilities of all those alignments in which it corresponds to i; these events are summarised in figure 3 (i) and (ii) for types (i) and (ii) respectively. Now P(A[p..q]&B[r..s]) can be calculated for all s by the DPA, figure 2 (iv), as described in section 2. Therefore two calls to the DPA, one on A[Alo..i−1] and B[Blo..Bhi] and one on the reversed A[Ahi..i+1] and B[Bhi..Blo], yield P(A[Alo..i−1]&B[Blo..j]), P(A[Alo..i−1]&B[Blo..j−1]) (trivially) and P(A[Ahi..i+1]&B[Bhi..j+1]) for all j in Blo−1..Bhi. These values allow the probabilities of events of types (i) and (ii) for every possible j to be calculated and one is sampled accordingly. Given a value for j, alignments of A[Alo..i−1] with B[Blo..k] and of A[i+1..Ahi] with B[j+1..Bhi] are sampled recursively where k=j for event type (i) and k=j−1 for event type (ii). Note that the subproblems are conditional on j, as they should be. The probabilities of events in the base-cases of the algorithm are easily calculated for sampling; see figure 4 (i) and (ii). If either string is empty there is only one possible event. The sampling algorithm takes O(|A|.|B|) time and O(|B|) space for the same reason that Hirschberg's original algorithm does.

An optimal or most-probable alignment has more chance of being sampled by the procedure than has any one sub-optimal alignment. However there are a great many sub-optimal alignments and it is in general unlikely that the sampled alignment will be a most-probable one.

4. Applications.

It is not feasible to extend the DPA to K>2 strings, unless they are short, due to rapidly increasing computational complexity. However, the DPA and the alignment sampling procedure described above can be used as iterative steps in procedures for K strings that are related by an evolutionary tree or in some other way[AlWa]. The algorithms have been extended to pairwise alignment of alignments, rather than of strings. In this way they can be used to sample random multiple-alignments of K strings. This gives a Gibbs sampling or Monte-Carlo method[Hast] for estimating P(copy), P(change), P(insert) and P(delete) on each edge of the tree, and also gives a simulated-annealing search for a (near) optimal multiple-alignment.

5. References.

[AlWa] Allison L. & C. S. Wallace. An information measure for the string to string correction problem with applications. Australian Comp. Sci. Comm. **16**(1) 659-668 1994.

[AlWY] Allison L., C. S. Wallace & C. N. Yee. Finite-state models in the alignment of macromolecules. J. Mol. Evol. **35** 77-89 1992.

[Bish] Bishop M. J., A. E. Friday & E. A. Thompson. Inference of evolutionary relationships. In *Nucleic Acid and Protein Sequence Analysis*. M. J. Bishop & C. J. Rawlings (eds), IRL Press, Oxford 359-385 1987.

[Hast] Hastings W. K. Monte-Carlo sampling methods using Markov chains and their applications.

Biometrika **57** 97-109 1970.

[Hirs] Hirschberg D. S. A linear space algorithm for computing maximal common subsequences. Comm. A.C.M. **18**(6) 341-343 1975.

[Leve] Levenshtein V. I. Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics Doklady **10**(8) 707-710 1966.

[Sank] Sankoff D. S. & J. B. Kruskal (eds). *Time Warps, String Edits and Macro-Molecules*. Addison Wesley 1983.

[Sell] Sellers P. H. An algorithm for the distance between two finite sequences. J. Comb. Theory **16** 253-258 1974.

[Thor] Thorne J. L., H. Kishino & J. Felsenstein. Inching towards reality: an improved likelihood model of sequence evolution. J. Mol. Evol. **34** 3-16 1992.

[Ukko] Ukkonen E. On approximate string matching. In *Proc. Int. Conf. on Foundations of Computation Theory*. M. Karpinski (ed) Springer Verlag LNCS **158** 487-495 1983.

```
string A: ACGTACGTAGT
string B: AGTATGTACGT

Possible alignment:
   ACGTACGTA-GT
   | ||| ||| ||
   A-GTATGTACGT
    |   |   |
    |   |   insert
    |   change
    delete

Edit-distance = 3
length of LCS = 9
```

Figure 1: Example Alignment.

```
Boundary Conditions:
   D[0, 0] = v
   D[i, 0] = f( D[i-1, 0], g(A[i], '-') ),  i = 1..|A|
   D[0, j] = f( D[0, j-1], g('-', B[j]) ),  j = 1..|B|

General Step, for i = 1..|A|, j = 1..|B|:
   D[i, j] = h( f(D[i-1, j-1], g(A[i], B[j])),     {NW}
                f(D[i-1, j  ], g(A[i], '-' )),     {N}
                f(D[i  , j-1], g('-' , B[j])) )    {W}

Return: D[|A|,|B|]

(i) longest common subsequence:
       v = 0
       f = +
       g(x,x)=1, g(x,y)=g(x,'-')=g('-',y)=0
       h = max

(ii) edit-distance:
       v = 0
       f = +
       g(x,x)=0, g(x,y)=g(x,'-')=g('-',y)=1
       h = min

(iii) most probable alignment:
       v = 1
       f = *
       g(x,x)   = P(<x,x>),   g(x,y)   = P(<x,y>),
       g(x,'-') = P(<x,'-'>), g('-',y) = P(<'-',y>)
       h = max

(iv) r-theory, P(A&B):
       v, f and g as for (iii)
       h = +
```

Figure 2: Dynamic Programming Algorithm (DPA).

```
(i)    A[Alo..i-1]&B[Blo..j]; <A[i], '-'>;  A[i+1..Ahi]&B[j+1..Bhi]
                               delete


(ii) A[Alo..i-1]&B[Blo..j-1]; <A[i], B[j]>; A[i+1..Ahi]&B[j+1..Bhi]
                               copy or
                               change
```

Figure 3: Divide and Conquer Cases of A[Alo..Ahi] with B[Blo..Bhi].

```
(i)  A[i]  -      ...  -       or   -     A[i]  -       ...  -         etc.
      -   B[Blo] ... B[Bhi]       B[Blo]  -   B[Blo+1] ... B[Bhi]
     del  ins    ... ins          ins    del  ins      ... ins

(ii) A[i]    -        ...  -      or   -     A[i]     ...  -           etc.
     B[Blo] B[Blo+1] ... B[Bhi]       B[Blo] B[Blo+1] ... B[Bhi]
     copy/                                   copy/
     chng   ins      ... ins          ins    chng     ... ins
```

Figure 4: Base-Cases of A[i] with B[Blo..Bhi].