A Fast Algorithm for the Optimal Alignment of Three Strings.

Lloyd Allison,
Department of Computer Science,
Monash University,
Australia 3168.
email: lloyd@cs.monash.edu.au

1992

Partly supported by an Australian Research Council grant.

J. Theor. Biol. 164(2) 261-269 Sept 1993

Abstract. Ukkonen's (pair-wise) string alignment technique is extended to the problem of finding an optimal alignment for three strings. The resulting algorithm has worst-case time-complexity $O(nd^2)$ and space-complexity $O(d^3)$, where the string lengths are ~n and d is the three-way edit-distance based on tree-costs. In practice the algorithm usually runs in $O(n+d^3)$ time. The algorithm is particularly fast when the strings are similar in which case d<<n.

Three-way alignment is an important special case in string alignment. Each internal node in an unrooted, binary evolutionary-tree has three neighbours. The algorithm presented can be used as an iterative step in a heuristic multiple-alignment program for more than three strings.

Keywords: alignment, edit-distance, multiple-alignment, sequence, string, three.

1. Introduction.

Alignment algorithms are used to compare strings such as biological macro-molecules - DNA and proteins. Given k strings, As, Bs, Cs, ... , over an alphabet Σ, a k-way alignment is formed by padding the strings out with zero or more *null* characters, denoted by '-', so that they have the same length and writing the strings out in rows, as for example in **figure 1**, so that corresponding characters lie in columns. Each column of the alignment forms a k-tuple from $(\Sigma \cup \{\text{'-'}\})^k$. The all-null tuple is banned. Each tuple can be given a cost to reflect the amount of change or editing that it represents; this can be done in a variety of ways - see later. The cost of an alignment is the sum of the cost of its tuples. An *optimal alignment* is one having the minimum possible cost. This minimum cost is the (k-way) *edit-distance* of the strings.

Given k strings that are thought to be related through evolution or by function, an alignment indicates portions of the strings that are similar and portions that are dissimilar or that have mutated. This can give clues to the history and function of the strings. A group of strings that has a small edit-distance can reasonably be assumed to be closely related and this can be used to reconstruct an evolutionary-tree for a collection of strings. An evolutionary hypothesis for k strings can be modelled by a tree where the given strings are leaves. Each internal node represents an ancestral string. Each edge represents the evolution of one string from another. Usually rooted or unrooted binary-trees are considered.

The alignment problem has been extensively studied for the special case k=2. The well-known dynamic programming algorithm (DPA) and variations have time and space complexity $O(n^2)$, where n is the average string length, for a wide variety of cost-functions[$Goto82]. Using a divide-and-conquer technique due to Hirschberg[$Hirs], the space-complexity can be reduced to O(n). Ukkonen[$Ukko] devised an algorithm that has O(nd) worst-case time-complexity. In practice the algorithm usually runs in $O(n+d^2)$ time. It has space-complexity of O(d) if only the value of the edit-distance is required and $O(d^2)$ if an alignment is also required. Ukkonen's algorithm is very efficient on pairs of strings that are similar in which case d<<n.

Here, Ukkonen's technique is extended to the optimal alignment of three strings. The resulting algorithm has $O(nd^2)$ worst-case time-complexity. In practice it usually runs in $O(n+d^3)$ time. It has space-complexity $O(d^3)$ if an alignment is required and $O(d^2)$ if only the value of the edit-distance is required. An

```
As:     a c g t a c g t a c - g t
Bs:     a c - t a g c t a c t g t
Cs:     a c t t a c a t a c - g t
costs:  0 0 2 0 0 1 2 0 0 0 1 0 0

Total cost, d=6
```

Figure 1: A 3-way Alignment.

alignment can also be recovered in O(d^2) space if Hirschberg's technique is used. For comparison, Murata et al[$Mura] and Gotoh[$Gotoh86] gave O(n^3) time algorithms to align three strings under particular cost functions. k=3 is an important special-case of the alignment problem. Each internal node in an unrooted binary-tree has three neighbours. Three-way alignment can be applied at internal nodes as an iterative step in a heuristic method to align more than three strings related by such a tree[$Sank73].

In what follows we consider the simplest edit cost function. When one string evolves into another, the allowed *point-mutations* are to change a character, to delete a character or to insert a character. Given an alignment of two strings, a pair <x,x>, or just xx, represents no mutation or a match, xy represents change x to y, x- represents delete x and -x represents insert x. All mutations are considered equally likely and are given the cost 1. Given an alignment of k>2 strings related by an unrooted binary-tree, each tuple is given the minimum possible cost, summing over the implied mutations on each edge of the tree. These are known as *tree-costs*. When k=3 the tree is a three-pointed star with a single unknown "ancestor" Ps, see **figure 2**. As an example, xxy is given the cost 1 because the ancestral character being x implies a single change and this is the minimum possible. More realistic models allow the edges of a tree to be of different *lengths* to represent different periods of time or different amounts mutation, but this does complicate algorithms considerably[$Sank75][$Alli1].

Carrillo and Lipman[$Cari] gave a multiple-alignment algorithm for use when tuple costs are based on the sum of the pair-wise costs of the characters in a tuple. Altschul and Lipman[$Alts] extended this to tree-costs although apparently the algorithm was not implemented for those costs (private communication). Those algorithms use pair-wise alignments to calculate a restricted volume in the edit-distance array to which attention can be safely restricted. The algorithm described in the present paper does not have such a pre-processing phase and adapts dynamically as it runs; it is not a heuristic and an optimal alignment is guaranteed.

2. Ukkonen's Algorithm.

The well-known dynamic programming algorithm (DPA) to compute the edit-distance of two strings, As and Bs, is given in **figure 3**. It uses an edit-distance array D[0..|As|, 0..|Bs|] where D[i,j] contains the edit-distance of As[1..i] and Bs[1..j], 0≤i≤|As|, 0≤j≤|Bs|. It has O(n^2) time and space complexity.

A brief and informal description of Ukkonen's algorithm[$Ukko] for two strings is given below for completeness. Ukkonen made use of the following observations: (i) the entries of each diagonal of D are

```
As
  .
   .
    .
     Ps--------Cs
    .
   .
  .
Bs

Cost  Triple Patterns
----  ---------------
 0:   xxx
 1:   xxy xyx yxx  xx- x-x -xx  x-- -x- --x
 2:   xyz          xy- x-y -xy
```

Figure 2: Tree and Costs, k=3.

```
{Boundary Conditions:}

   D[0,0] = 0
   D[i,0] = i, i=1..|As|
   D[0,j] = j, j=1..|Bs|

{General Step, i=1..|As|, j=1..|Bs|:}

   D[i,j] = min(D[i-1,j-1]
                + if As[i]=Bs[j]
                  then 0        {same   xx}
                  else 1,       {change xy}
                D[i,  j-1] + 1, {insert -x}
                D[i-1,j]   + 1  {delete x-})
```

Figure 3: Basic Dynamic Programming Algorithm (DPA).

non-decreasing, (ii) only the entry D[|As|,|Bs|] and those entries that it directly depends on through selection by some minimisation step of the DPA are needed to get an optimal alignment, (iii) a dual-representation U can be used instead of D. The array U, indexed by diagonal-number and cost, is used to hold the position in D of *contours* of constant cost. Loosely, U[ab,d] is the maximum *length* along diagonal ‘ab’ of D that can be reached for a cost of ‘d’. Ukkonen’s algorithm (UA) is given in **figure 4**. A *row* in U corresponds to a *diagonal* in D and it is convenient to refer to points and diagonals in D in the discussion below, although the algorithm works only with U and not with D.

Part−1 of the general step relies on the fact that a lower bound on U[ab,d] can be set from the length that can be reached along diagonals ab+1, ab and ab−1, for cost d−1. For example, if U[ab−1, d−1] can be reached on diagonal ab−1 for a cost d−1 then a further deletion means that at least U[ab, d−1]+1 can be reached on diagonal ab for cost d.

Part−2 improves on the lower bound provided by part−1 if it leads to a position on diagonal ab which is at the start of a run of matches - which carries zero further cost.

The general step is iterated for increasing cost d, until U[|As|−|Bs|,d] covers the full strings. Algorithm UA uses $O(d^2)$ space if all of the U array is kept. An alignment can be recovered by tracing back through U. The forward computation is a *wave-front* computation and the value of the edit-distance can be recovered using just O(d) space. An alignment can also be recovered in O(d) space by using

```
{U[ab,d] = max a s.t. D[a,b]=d where ab=a-b
         = -infinity if no such a exists.}

{Boundary Condition:}

   U[0,0]  = max a s.t. As[1..a] = Bs[1..a]
   U[ab,d] = -infinity,  if |ab|>d

{General Step, iterated until U[|As|-|Bs|,d]=|As|:}

   U[ab,d] = max(U[ab+1, d-1],
                 U[ab,   d-1] + 1,            -- part-1
                 U[ab-1, d-1] + 1);

   while As[U[ab,d]+1] = Bs[U[ab,d]-ab+1] do
      U[ab,d] += 1                            -- part-2
```

Figure 4: 2-String Ukkonen’s Algorithm (UA).

Hirschberg's technique[$Hirs].

The general step is iterated O(d^2) times and this is a lower bound on the time-complexity of UA. In addition, part−2 must be iterated at least n−(#part−1) times *in total* so that each character of the strings is examined at least once. The overall complexity is worse if part−2 is iterated more frequently. It can be seen that the computation over U sweeps out an equivalent area in D of O(nd) cells on 2d+1 diagonals. Thus the worst-case time-complexity is O(nd). In practice, part−2 is iterated infrequently and the algorithm runs in O(n+d^2) time on average[$Myer86]. This means the algorithm effectively skips across most areas of D that have a constant value.

3. Three-Way Alignment.

The basic dynamic programming algorithm can be extended to align three strings, As, Bs and Cs, in the obvious way. A three-dimensional array D[0..|As|, 0..|Bs|, 0..|Cs|] is used where D[i,j,k] holds the 3-way edit-distance of As[1..i], Bs[1..j] and Cs[1..k]. The algorithm takes O(n^3) time and space which is not practical for n much over 100. The space-complexity of recovering an alignment can be reduced to O(n^2) by Hirschberg's technique.

Here Ukkonen's technique is applied to three strings. A dual array U is used. U[ab,ac,d] holds the maximum distance that it is possible to travel along diagonal (ab,ac) in D for a cost of at most d. As before, a row in U corresponds to a diagonal in D and it is convenient to refer to D and to points and diagonals in it although the algorithm operates only on U and not on D. Note that the diagonals of a cube lie in a two-dimensional, triangular lattice of diagonals.

The new algorithm, A3, is given in **figure 5**. A lower-bound on U[ab,ac,d] can be found from the results for costs d−1 and d−2 in row U[ab,ac] and the six surrounding rows by adding a new triple of non-zero cost. This forms part−1 of the new general step. Part−2 can improve on this lower bound if it leads to a run of matches. The general step is iterated until the three strings are fully covered.

Function *Step* calculates at what point one could step directly onto a diagonal in D, for a given cost, by a single step from the same diagonal or from a neighbouring diagonal; it is given in **figure 6**. The three variables da, db and dc, are given values 0 or 1 and indicate the direction of the step in the D array relative

```
{U[ab,ac,d] = max a s.t. D[a,b,c] <= d  where ab=a-b, ac=a-c
            = -infinity if no such a exists }

{Boundary Condition:}

   U[0,0,0]    = max a s.t. As[1..a] = Bs[1..a] = Cs[1..a]
   U[ab,ac,d] = -infinity if max(|a-b|,|a-c|,|b-c|)>d where ab=a-b, ac=a-c

{General Step, iterated until U[|As|-|Bs|, |As|-|Cs|, d] = |As|:}

   a = -infinity;                                          ]
   for neighbour = 1..7 do
   begin set ab1,ac1, da,db,dc for neighbour of ab,ac
         a = max(a, Step(ab1,ac1, d-1, da,db,dc))          } part-1
   end;
   b = a-ab; c = a-ac;                                     ]

   while As[a+1] = Bs[b+1] = Cs[c+1] do
      begin a+=1; b+=1; c+=1 end;                          -- part-2

   U[ab,ac,d] = a
```

Figure 5: Fast 3-String Alignment Algorithm (A3).

to the As, Bs and Cs axes. They cannot all be 0 but can all be 1. The step is from (a1,b1,c1) in D in direction (da,db,dc) to (a2,b2,c2) and corresponds to a triple of characters in an alignment. The sum da+db+dc equals the number of non-null characters in the triple. If the cost of the triple is 1 then it is appropriate to return U[ab1,ac1,d]+da. Something else must be done if the cost of the triple is 2; it is certainly necessary to return at least U[ab1,ac1,d−1]+da. If the triple contains three non-null characters it is sufficient to return the larger of U[ab1,ac1,d] and U[ab1,ac1,d−1]+da. If the triple contains exactly two non-null characters and U[ab1,ac1,d] > U[ab1,ac1,d−1], it is possible that some triple between those two points on the (ab1,ac1) diagonal costs 1 when one of its characters is dropped to step off-diagonal, e.g. da=db=1, dc=0, xxx→xx- or xxy→xx-. A loop backwards along the (ab1,ac1) diagonal checks for the first such position if any; the body of the loop is labelled part−3. This loop is also terminated if the step is obviously worse than any preceding relevant step and it is arranged that steps involving exactly two non-null characters are tested last out of the seven so as to minimise the number of times that part−3 is iterated. The maximum number of times that part−3 could be iterated in total during a run of A3 is the same as for part−2, ie. $O(nd^2)$.

An alignment can be reconstructed by tracing back through the U array recovering the 'max' decisions that lead to the final d value.

Algorithm A3 has space-complexity $O(d^3)$ if an alignment is required and $O(d^2)$ if only the value of the edit-distance d is needed. The time-complexity is determined by the number of times that part−1, part−2 and part−3 are iterated. Part−1 is executed $O(d^3)$ times. Part−2 and part−3 are each iterated at most $O(nd^2)$ times. The best-case time-complexity is $O(n+d^3)$ and the worst-case time-complexity is $O(nd^2)$ by an argument similar to that for the case of two strings. In practice the algorithm usually runs in $O(n+d^3)$ time - see next section. Part−1 is usually executed less than $d^3/3$ times. Part−2 is usually iterated a little more than n times. Part−3 is usually iterated approximately $d^3/3$ times.

On a pragmatic note, a production version of A3 would need to take some care over storage management. Simply to declare a maximum-size array U[-Maxd..Maxd, -Maxd..Maxd, 0..Maxd], as the prototype does, is wasteful in space; space should be requested as needed. Secondly, the computation is a wave-front computation and the cells in the wave-front are likely to lie in different pages on a virtual-memory computer, making for a large working-set. To cure these problems, U should be mapped onto a linear array or onto heap memory in such a way as to keep dependent elements close to each other as far as

```
function Step({off-diagonal}ab1,ac1, d, {direction}da,db,dc)
   a1=U[ab1,ac1,d]; b1=a-ab1; c1=a-ac1;         {step off}
   a2=a1+da {=a1 or =a1+1}; b2=b1+db; c2=c1+dc; {step on }
   case da+db+dc {number of non-null chars in triple} of
     1: {e.g. x--, must cost 1}
        return a2;
     2: {e.g. xx- costs 1 or xy- costs 2}
        a3 = U[ab1,ac1,d-1]+da;
        while this_is_as_good_as_any_other_Step and
           a2>a3 and TripleCost(a2*da,b2*db,c2*dc)=2 do
           begin a2 -= 1; b2 -= 1; c2 -= 1 end;           -- part-3
        {NB. a2=a3 or TripleCost(e.g. xx-)=1 or beaten}
        return a2;
     3: {xxy, xyx, yxx cost 1 or xyz costs 2}
        if TripleCost(a2,b2,c2)=2 then {NB. da=db=dc=1}
           return max(a1, U[ab1,ac1,d-1]+1)
   end
```

Figure 6: Function Step.

possible.

An alignment can also be recovered in just O(d^2) space by using Hirschberg's divide-and-conquer technique[$Hirs].

Ancestor Length	#changes, #indels requested per edge: 0,0	1,1	2,2	4,4	8,8
50	0	6	11	19	30
	0,50,0	76,47,50	431,75,343	1887,115,1761	7408,253,7715
	0.0 1.0	1.04 0.89	0.97 0.91	0.82 0.81	0.82 0.85
100	0	6	10	24	43
	0,100,0	76,100,41	318,101,252	3778,188,3452	20705,573,21003
	0.0 1.0	1.04 0.89	0.95 0.88	0.82 0.80	0.78 0.80
400	0	6	12	24	48
	0,400,0	90,404,46	532,431,380	3732,487,3646	28320,1018,28584
	0.0 1.0	1.23 0.99	0.92 0.87	0.81 0.82	0.77 0.78
1600	0	6	12	24	48
	0,1600,0	76,1599,46	558,1629,419	3874,1680,3560	28782,2149,28505
	0.8 1.0	1.04 0.99	0.97 0.95	0.84 0.84	0.78+ 0.79+
					(9.7 secs *)
					(+ std. dev. <0.03)

Len'	16,16	32,32 ----> #changes, #indels /edge
100	53	76
	38K,1146,42K	110K,3697,130K
	0.77 0.82	0.75 0.83
400	91	166
	192K,3612,198K	1.1M,25K,1.2M
	0.77 0.79	0.75 0.79
1600	92	186
	200K,4242,200K	1.6M,17K,1.7M
	0.77 0.78	0.76 0.79
	(78 secs *)	(660 secs *)

KEY: d — where p1=counter for Part-1 etc.
p1, p2, p3 — $r1=p1/(1+d^3/3)$
r1 r2 — & $r2=(p1+p2+p3)/(n+2d^3/3)$

(* sparc-station)
NB. space-usage, #U-matrix cells = p1+1

Table 1: Empirical Complexity.

4. Results.

Algorithm A3 was coded in Pascal and run on a sparc-station. The pattern of evaluation of the U entries is moderately complex and the simplest control mechanism is a recursive one based on the recurrence for U given in figure 5. U is implemented as a memo-array so that each value of U that is needed is only computed once. Time and space usage were measured empirically on data formed by generating a random ancestor string Ps and mutating it to give descendants As, Bs and Cs. The number of change and indel mutations on each edge of the tree could be specified. An alphabet of four was used as the algorithm is most likely to be used on DNA or RNA. Small alphabets often cause the most trouble for an alignment algorithm as they increases the chance of "noise" matches. Mutation sites on Ps were chosen at random from a uniform distribution and each indel had a 50/50 chance of being an insert or a delete. **Table 1** gives typical results. It shows the actual edit-distance d and the number of steps (part−1, part−2 and part−3) for various lengths of ancestor and various amounts of mutation. Note that the number of cells used in the U-matrix is one more than the number of times that part−1 is executed and that *two* edges of the tree separate any two of As, Bs and Cs.

The maximum possible edit-distance for three strings of length n is 2n: consider $As=x^n$, $Bs=y^n$ and $Cs=z^n$. However the edit-distance of three *random* strings over an alphabet of four characters is typically about n. Values of d greater than n/2 indicate considerable divergence.

As a check, a naive 3-string DPA using Hirschberg's technique was also coded and run on shorter strings only. No attempt was made to optimise either the DPA or A3 to any great extent. The algorithms took similar amounts of time on random strings. A3 was much faster on similar strings and d, rather than n, is the important parameter for all practical purposes.

5. Conclusions.

Ukkonen's technique was extended to the optimal alignment of three strings. The resulting algorithm, A3, has worst-case time-complexity $O(nd^2)$, where the strings are of length ˜n and d is the 3-way edit-distance. In practice, the algorithm typically runs in $O(n+d^3)$ time. The space-complexity is $O(d^3)$ if an alignment is required and $O(d^2)$ if only the value of d is required. The algorithm is particularly efficient for similar strings when d<<n. An alignment can be recovered in just $O(d^2)$ space by means of Hirschberg's technique.

The above applies to the simplest edit costs where each mutation costs 1. Linear indel (gap) costs[$Goto82] of the form $w(k)=a+b\times k$ for a run of inserts or deletes of length k are popular. The constants a and b are usually small integer values. Ukkonen's technique has been applied to such costs over two strings[$Myer89][$Alli2]. It remains to do this for linear costs on three strings. It also remains to apply the technique to k>3 strings, hopefully giving an $O(n+2^k d^k)$ average time, $O(d^k)$ space algorithm. However this may be of theoretical rather than practical value for k greater than say 4 or 5 unless d is very small. Possibly of more practical value is to use algorithm A3 as an iterative step in a heuristic alignment algorithm for k>3 strings.

There is an obvious parallel version of algorithm A3 suitable for a grid-connected, multi-processor computer. Each processor would be allocated to a row of matrix U (˜diagonal of D), or to a small number of rows. The algorithm would run in O(n) time, in the worst-case, given $O(d^2)$ processors. It seems likely that this could be further improved in the average-case.

The source-code of the Pascal version of algorithm A3 is available by email only from lloyd@cs.monash.edu.au.

6. References.

[$Alli1] Allison L., C. S. Wallace & C. N. Yee. Minimum message length encoding, evolutionary trees and multiple alignment. 25th Hawaii Int. Conf. Sys. Sci. **1** 663-674 Jan 1992.

[$Alli2] Allison L., C. S. Wallace & C. N. Yee. Finite-state models in the alignment of macro-molecules. J. Mol. Evol. **35**(1) 77-89, July 1992.

[$Alts] Altschul S. F. & D. J. Lipman. Trees, stars and multiple biological sequence alignment. SIAM. J. Appl. Math. **49**(1) 197-209, Feb 1989.

[$Cari] Carrillo H. & D. Lipman. The multiple sequence alignment alignment problem in biology. SIAM J. Appl. Math. **48** 1073-1082 1988.

[$Goto82] Gotoh O. An improved algorithm for matching biological sequences. J. Mol. Biol. **162** 705-708 1982.

[$Goto86] Gotoh O. Alignment of three biological sequences with an efficient traceback procedure. J. Theor. Biol. **121** 327-337 1986.

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

[$Mura] Murata M., J. S. Richardson & J. L. Sussman. Simultaneous comparison of three protein sequences. Proc. Natl. Acad. Sci. USA. **82** 3073-3077 1985.

[$Myer86] Myers E. W. An O(nd) difference algorithm and its variations. Algorithmica **1** 251-266 1986.

[$Myer89] Myers E. W. & W. Miller. Row replacement algorithms for screen editors. ACM Trans. Prog. Lang. & Sys. **11**(1) 33-56 1989.

[$Sank73] Sankoff D., C. Morel & R. J. Cedergren. Evolution of 5S RNA and the non-randomness of base replacement. Nature New Biology **245** 232-234 1973.

[$Sank75] Sankoff D. Minimal mutation trees of sequences. SIAM J. Appl. Math. **28** 35-42 1975.

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