Lazy Dynamic-Programming can be Eager.

L. Allison,
Department of Computer Science,
Monash University,
AUSTRALIA 3168.

email: lloyd@cs.monash.edu.au

December 1991
Revised May 1992

Abstract. Lazy-evaluation in a functional language is exploited to make the simple dynamic-programming algorithm for the edit-distance problem run quickly on similar strings: being lazy can be fast.

Keywords: dynamic-programming, edit-distance, functional programming, lazy evaluation.

1. Introduction.

The *edit-distance* problem[$Sell] is to find the minimum number of point-mutations, D A B, required to edit one given string *A* into another given string *B*. A *point-mutation* is one of the following: *change* a letter, *insert* a letter, or *delete* a letter. Sometimes one also wants to find a minimal set of mutations that edits A into B. The problem, and variations on it, are important in Molecular-Biology for comparing linear macro-molecules for similarity[$Sank][$Bish]. It also arises in spelling correction, file comparison and other "computing" problems[$Wagn]. It is closely related to the longest-common-subsequence problem. There is a large body of work in the computing and biological literature on algorithms, in imperative languages, for the edit-distance problem and its relatives.

This paper discusses solving the edit-distance problem in a lazy functional language[$Peyt]. It begins with a simple but inefficient algorithm. Removing the cause of the inefficiency leads to a functional version of the well-known (imperative) dynamic-programming algorithm (DPA)[$Sell][$Wagn], which has O(|A|*|B|) time and space-complexity. A simple modification of this second algorithm and an obvious observation lead to an O(|A|*(1+D A B)) algorithm. This is fast for the important special case of similar strings. It exploits lazy-evaluation to gain its speed. It is as simple as the basic DPA but, thanks to lazy-evaluation, effectively implements an eager or greedy strategy[$Mill][$Myer] which requires added complication when written in an imperative language. This may give hope to the less energetic among us: being lazy can be fast.

```
A = acgtacgtacgt
B = acatacttgtact

A = acgtac  gtacgt
    || |||  |||| |
B = acatacttgtac t
      ^   ^^    ^
      |   ||    |
      |   ||    delete
      |   ||
      |   insert*2
      |
      change

D A B = 4
```

An Example.

Given strings A and B, a natural way to begin to compute D A B is to compare their first letters, hd A and hd B. (Equivalently, a start can be made with the last letters.) If they are equal, no mutation is required here, so the edit-distance of the remainder of the strings, As=tl A and Bs=tl B, is computed. If they are not equal a mutation is required and there are three possibilities: (i) hd A can be changed into hd B and As edited into Bs, (ii) hd A can be deleted and As edited into B, or (iii) hd B can be inserted and A edited into Bs. The least costly alternative is chosen. If either string is null, the edit-distance is the length of the other string. This leads to the first algorithm:

```
D [] B  = length B  ||         {A is null}
D A  [] = length A  ||         {B is null}
D A  B  = let As=tl A, Bs=tl B
          in  if hd A = hd B then D As Bs
              else 1 + min3 (D As Bs) (D As B) (D A Bs)
```

Algorithm 1: Inefficient.

This and other algorithms are written in lazy ML (LML)[$Augu]. Note that '||' separates *cases* in the definition of a function and that '[ ]' denotes the empty list or string.

If A=B, the algorithm runs in O(|A|) time, checking equality one character at a time, and it is clearly impossible to do better. However, the worst-case complexity is exponential in the length of the strings, as the function is triply-recursive; consider A=aaa... and B=bbb... .

The cause of the inefficiency is that intermediate results are calculated more than once. For example, D As Bs can be calculated three times. In such cases, the intermediate results should be calculated once and stored in a data-structure for re-use.

2. An O(|A|*|B|) Algorithm.

A data-structure is used to remove the source of inefficiency in the previous algorithm. It holds intermediate results D A[1..i] B[1..j], i,j>=0. From the previous section we know the following equations:

```
D A[1..i] "" = i,   i>=0
D "" B[1..j] = j,   j>=0

if i>0 & j>0:
D A[1..i] B[1..j]
   = D A[1..i-1] B[1..j-1],             if A[i]=B[j]
   = 1 + min((D A[1..i-1] B[1..j-1]),
             (D A[1..i-1] B[1..j]  ),
             (D A[1..i]   B[1..j-1]))   otherwise
```

Useful Equations.

These equations form the basis of the well-known dynamic-programming algorithm (DPA) for the edit-distance problem[$Sell]. A matrix, Dmatrix[i,j], would be used to hold D A[1..i] B[1..j], i≥0, j≥0, in an imperative language:

```
     a c a t ... -> B
   0 1 2 3 4 ...
a: 1 0 1 2 3 ...
c: 2 1 0 1 2 ...
g: 3 2 1 1 2 ...
t: 4 3 2 2 1 ...
.  ....
|
v A
```

Dmatrix.

A list of rows can be used in a functional program, each row being a list of ints. It can be seen that a row can be calculated from left to right, given the previous row. An element depends only on the elements to the west, north-west and north. This leads to the following functional version:

```
D A B =
  let rec
      Rows = [0 .. length B]         {the first row }
           . (EachRow A (hd Rows))   {the other rows}

  and EachRow []       lastrow = []  ||
      EachRow (Ach.As) lastrow =
         let rec
             DoRow []        NWs    W = []  ||       {NW N }
             DoRow (Bch.Bs)  (NW.N) W =              {W  me}
                let me = if Ach = Bch then NW
                         else 1 + min3 W NW (hd N)   {NB. N is a list}
                in  me . (DoRow Bs N me)
```

```
        and firstelt = 1 + hd lastrow
        and thisrow = firstelt . (DoRow B lastrow firstelt)

        in thisrow . (EachRow (tl A)  thisrow)

   in last (last Rows)
```

Algorithm 2: O(|A|*|B|).

Note that '.' is the list constructor, also known as 'cons'. A lazy functional language, such as LML[$Augu], is needed to run this algorithm. Function DoRow calculates one row, except for the first element. A row is recursively defined, the current element 'me' depending on the previous element, to the west, W. Me becomes the previous element for next element. The current element also depends on two elements in the previous row, to the north-west and the north, NW and hd N. Note that N is a list beginning with the northern element. This is an example of a *circular program*[$Bird][$Alli], using a recursively defined data-structure. It runs in O(|A|*|B|) time and space. This is both its best-case and its worst-case complexity, so while a lot has been gained in the general case, compared with algorithm 1, something has been lost for the particular case of identical strings.

3. An O(|A|*(1+D A B)) Algorithm.

The previous algorithm used a list of rows to store intermediate results. If one thinks of the values as "heights" in a contour-map, they represent a valley climbing from 0=D "" "" in the north-west to a height of D A B in the south-east, with a peak D "" B to the north-east and another peak D A "" to the south-west. D A B is the maximum height of the valley floor in the south-east. We are not interested in the peaks. This suggests reorganising the data-structure as diagonals; apart from that the data-structure contains the same values as before. An entry still depends on three neighbours which lie on the diagonal below, the current diagonal and the diagonal above. Each diagonal therefore depends on the diagonal below and the diagonal above where a row depends only on the row above. The code needs a little modification to accommodate this:

```
D A B =
   let rec
   and MainDiag = OneDiag  A B (hd Uppers) (-1 . (hd Lowers))
   and Uppers   = EachDiag A B (MainDiag.Uppers)  {upper diagonals}
   and Lowers   = EachDiag B A (MainDiag.Lowers)  {lower diagonals}
                                                  {note swap B A !}

   and OneDiag A B diagAbove diagBelow =
         let rec
             DoDiag []        B        NW N W = []  ||
             DoDiag A         []       NW N W = []  ||
             DoDiag (Ach.As)  (Bch.Bs) NW N W =
               let me = if Ach = Bch then NW
                        else 1 + min3 (hd W) NW (hd N)    {see below}
               in me . (DoDiag As Bs me (tl N) (tl W))   {along diag}
                           {hope these   ^^^^   ^^^^  not evaluated}
```

```
        and firstelt = 1 + hd diagBelow
        and thisdiag =
              firstelt . (DoDiag A B firstelt diagAbove (tl diagBelow))

        in thisdiag

  and EachDiag A []       Diags            = []  ||
      EachDiag A (Bch.Bs) (LastDiag.Diags) =
        let NextDiag = hd(tl Diags)
        in  (OneDiag  A Bs NextDiag LastDiag) {one diag &}
          . (EachDiag A Bs Diags)             {the others}

  and LAB = (length A) - (length B)

  in last (if      LAB=0 then MainDiag
           else if LAB>0 then select   LAB  Lowers
           else   {LAB<0}     select (-LAB) Uppers )
```

Algorithm 3: O(|A|*(1+D A B)).

The algorithm defines all (|A|+1)*(|B|+1) entries in all diagonals but they are not necessarily all evaluated. It prints the last entry in the appropriate, LAB = |A|−|B|, diagonal. The worst-case complexity is O(|A|*|B|) as before, eg. when A=aaa... and B=bbb... . However, if A=B the complexity is now O(|A|) because only the main diagonal is evaluated. What happens in other cases crucially depends on the definition of min3.

A standard implementation of min3 is as follows:

```
{standard} min3 X Y Z = min X (min Y Z)
```

This requires all three of X, Y and Z to be evaluated. The reason that min3 is called is that a run of matches has come to an end and a change may be needed on the diagonal. Given the above definition of min3, if A and B are unequal but similar, the complexity is still O(|A|*|B|); consider A=cc...cca and B=cc...ccb. Because the last characters differ, min3 is called and all of the upper and lower triangular areas of the data structure are evaluated to calculate X and Z.

A little thought reveals that if X<Y in our situation then Z≥X, so there is no need for min3 to evaluate Z in that case. A better coding of min3 for our purposes is:

```
{special} min3 X Y Z = if X < Y then X else min Y Z
```

Now Z is only evaluated if X≥Y. The new min3 is biased to examine elements closer to the main diagonal, ie. X and Y, in preference to the the one further from it, Z. This is a good thing because it means that diagonals are only evaluated up to a "height" of D A B. Only parts of at most 1+2*D A B *central* diagonals are evaluated at all. This algorithm therefore runs in O(|A|*(1+D A B)) time and space. eg. Only 3 diagonals are evaluated if A=cc...cca and B=cc...ccb. Note that it is important to compare X and Y first, and to try to avoid evaluating Z, in order to gain the efficiency. A similar technique can also be used in the first two algorithms where it brings a small improvement but does not change their underlying complexities.

The algorithms were coded in λ-calculus and run on an instrumented interpreter, and also coded in LML and timed to confirm their expected behaviour.

A = acgtacgtac...
B mutated from A by point mutations,
change : insert : delete in ratio 2:1:1,
uniformly spread through length of string.
length = |A| = |B|

```
             0    10    20  → D A B
length   ----------------
1000:    0.11  0.72  1.47  CPU time (secs)
2000:    0.29  1.59  3.17      for
4000:    0.62  3.42  6.35  LML, Dec-5000/125

          Using special min3


             0    10    20  → D A B
length   ----------------
1000     0.12    30    30
2000     0.28   112   130  CPU time (secs)
4000     0.62   524   491

         Using standard min3
```

Sample Timing Results for Algorithm 3.

Although it is lazy and driven by the demand to print the last entry of the LAB diagonal, algorithm 3 evaluates just the entries that an eager, imperative algorithm does[$Mill]. Briefly, the latter tries to "push" along the LAB diagonal from the north-west to the south-east for as far as it can with a run of matches. If the run terminates in a change, it pushes the two neighbouring diagonals to the end of a run. This gives enough information to advance the LAB diagonal to the next "height" and so on. This strategy requires major changes from the simple DPA in an imperative language. It all happens automatically in the final lazy functional DPA.

If a minimal set of mutations to edit A into B is needed, they can be recovered by tracing back from the south-east of the data-structure to the north-west following the chain of dependencies selected by min3.

Further reflection can reveal that because each diagonal is monotonic increasing, a "dual" data-structure which stores the positions of *contours* in the previous data-structure can be used. However, this requires major surgery on the algorithm. It leads to a functional version of Ukkonen's algorithm[$Ukko], having the same time-complexity as above but $O((D\ A\ B)^2)$ space-complexity for the data-structure.

4. References.

[$Alli] Allison L. Circular programs and self-referential structures.
Software Practice & Experience **19**(2) 99-109 Feb 1989.

[$Augu] Augustsson L. & T. Johnsson. Lazy ML user's manual.
Programming Methodology Group, University of Goteborg & Chalmers University of Technology, 1990.

[$Bird] Bird R. S. Using circular programs to eliminate multiple traversals of data.
Acta Informatica **21**(3) 239-250 Oct 1984.

[$Bish] Bishop M. J. & C. J. Rawlings (eds). *Nucleic Acid and Protein Sequence Analysis, a Practical Approach*.
IRL Press 1987.

[$Mill] Miller W. & E. W. Myers. A file comparison program.
Software Practice & Experience **15**(1) 1025-1040 Nov 1985.

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

[$Peyt] Peyton Jones S. L. *The Implementation of Functional Programming Languages*.
Prentice-Hall Int. Series in Computer Science 1987.

[$Sank] Sankoff D. & J. B. Kruskal. *Time Warps, String Edits and Macromolecules: the Theory and Practice of Sequence Comparison*.
Addison Wesley 1983.

[$Sell] Sellers P. H. An algorithm for the distance between two finite sequences.
Journal of Combinatorial Theory **16** 253-258 1974.

[$Ukko] Ukkonen E. On approximate string matching.
Proceedings of the International Conference on Foundations of Computation Theory, M. Karpinski (ed) Springer-Verlag LNCS **158** 487-495 Aug 1983.

[$Wagn] R. A. Wagner R. A. & M. J. Fischer. The string-to-string correction problem.
Journal of the A.C.M. **21**(1) 168-173 1974.

-- For the record, the full program for algorithm 3:

```
let takeDNA  ('>'.title) =
      let rec
         skipline 1 ('0.dna) = getDNA dna ||
         skipline N ('0.dna) = skipline (N-1) dna ||
         skipline N (a.b) = skipline N b

      and getDNA (Ch.dna)&(mem Ch ['0; ' ']) = getDNA dna ||
         getDNA (Base.dna)&(mem Base ['a';'c';'g';'t';'A';'C';'G';'T']) =
           let Bases,rest = getDNA dna
           in  (Base.Bases),rest       ||
         getDNA x = [],x

      in  skipline 2 title

-------------------------------------------------------------------------------
and

D A B =
   let rec
      MainDiag = OneDiag  A B (hd Uppers) ( -1 . (hd Lowers))

   and Uppers   = EachDiag A B (MainDiag.Uppers)

   and Lowers   = EachDiag B A (MainDiag.Lowers)

   and OneDiag A B diagAbove diagBelow =
         let rec
            DoDiag [] B NW N W = [] ||
            DoDiag A [] NW N W = [] ||
            DoDiag (A.As) (B.Bs) NW N W =
              let me = if A=B then NW
                      else 1+min3 (hd W) NW (hd N)
              in  me.(DoDiag As Bs me (tl N) (tl W))

         and firstelt = 1+(hd diagBelow)
         and thisdiag =
              firstelt . (DoDiag A B firstelt diagAbove (tl diagBelow))

         in thisdiag

   and min3 X Y Z =
         -- min X (min Y Z)               -- makes it O(|A|*|B|)
         if X < Y then X else min Y Z   -- makes it O(|A|*D(A,B))

   and EachDiag A [] Diags = [] ||
      EachDiag A (B.Bs) (LastDiag.Diags) =
         let NextDiag = hd(tl Diags)
         in  (OneDiag A Bs NextDiag LastDiag).(EachDiag A Bs Diags)

   and LAB = (length A)-(length B)

   in last( if      LAB = 0 then MainDiag
          else if LAB > 0 then select   LAB  Lowers
          else                select (-LAB) Uppers )

-------------------------------------------------------------------------------
```

```
in let rec
   L = choplist takeDNA input
and A = hd L
and B = hd(tl L)

in "D A[" @ (itos(length A)) @ "] B[" @ (itos(length B)) @ "] = "
  @ (itos(
    D A B
   ))
  @ "0
```

-- **O(|A|*D(A,B)) Edit Distance.**