Comput. Chem. Vol. 12, No. 1, pp. 21–25, 1988
Printed in Great Britain.

0097-8485/88 $3.00 + 0.00
Pergamon Journals Ltd

AN EFFICIENT ALGORITHM FOR COMPARING TWO PROTEIN SEQUENCES: IMPLEMENTATION FOR MICROCOMPUTERS

MITSUO MURATA
Department of Biochemistry, University of Georgia, Athens, GA 30602, U.S.A.

(*Received* 10 *March* 1987)

Abstract—An algorithm which can be used on microcomputers to compare two protein sequences is presented. It is based on the algorithm developed by Needleman & Wunsch (*J. Mol. Biol.* **48**, 444). The original algorithm requries memory space of mn and computing time proportional to mn^2 where m and n are the lengths of the two sequences. Because of these space and time requirements, the algorithm is of limited use on microcomputers. The modified algorithm presented here reduces the computing time proportional to mn and the size of directly accessed memory to $2n$, thus making this algorithm usable by many currently available microcomputers. The time saving part of the algorithm will also improve the efficiency of computations on larger computers.

INTRODUCTION

There are several algorithms for comparing two protein sequences (Needleman & Wunsch, 1970; Gibbs & McIntyre, 1970; McLachlan, 1971; Sankoff, 1974; Sellers, 1974). Different methods are reviewed and described by Waterman (1984), who also presented a rigorous mathematical account of the subject (Waterman *et al.*, 1976). Among those algorithms the one developed by Needleman & Wunch served as a pioneering work. Their algorithm is efficient in finding the alignment of two sequences with the maximum number of matched residues. It restrains over-generous gaps by imposing a penalty. Statistical tests can be implemented to assess the relatedness of the two sequences. In the actual computation, however, the algorithm requires a large amount of computer memory and a long CPU time as it involves calculations on a two dimensional array of mn elements where m and n are the lengths of the two sequences. The requirement of the long CPU time is a serious problem if statistical analysis is to be carried out based on a number of comparisons (i.e. 100 or more) between the real and scrambled sequences.

The memory requirement of the algorithm is a more immediate problem, if it is to be implemented on a microcomputer. Although a large amount of RAM (random access memory) is now offered by many microcomputers, with high level languages like BASIC or Pascal under ubiquitous operating systems (e.g. IBM PC-DOS and Microsoft MS-DOS) the usable RAM for array data is limited to 64K bytes. In the algorithm of Needleman & Wunsch, each arrray element usually requires 2 bytes of memory. Therefore, with the usable RAM, only two sequences up to about 170 residues each can be compared, which is hardly satisfactory as many proteins are much larger than this size.

In recent years, there has been a huge increase in the number of newly determined protein and nucleotide sequences. Also increased is the number of low cost microcomputers used in chemistry and biochemistry laboratories. The purpose of this paper is to describe modifications to the algorithm of Needleman & Wunsch so that it can be used with microcomputers or can be run more efficiently on larger computers. Many time and memory saving concepts in the method described here were conceived during the development of an algorithm for comparing three protein sequences simultaneously (Murata *et al.*, 1985), when similar space and time problems were encountered with a more powerful minicomputer.

DESCRIPTION OF THE ALGORITHM

Let the two sequences A and B have lengths m and n and denote by $A(i)$ and $B(i)$ the i-th residues in the respective sequences. Assign to every possible amino acid pair from the two sequences a weight, κ, which reflects the similarity between the two amino acids;

$$\kappa(i,j) = w[A(i), B(j)].$$

The best alignment of the two sequences is envisaged as a string of the amino acid residue pairs along which the sum of κ is maximum. In the algorithm of Needleman & Wunsch the summation of κ is carried out in a two-dimensional array, here denoted by λ, in which the cell (i,j) corresponds to the residue pairs $A(i)$ and $B(j)$. Thus, finding the above string of matched residues becomes finding a path or the so called "maximum match pathway" through the matrix, along which the sum of κ is maximum. Gaps are

allowed in the alignment, but the matching of gaps themselves is not. In addition, the orders of residues in the respective sequences must be maintained. Because of these restrictions, one rule applies here; upon moving forward along the path from one cell to the next, the indices of the residue pair must increase at least by one, but one of them by exactly one.

The matrix summation is carried out recursively in such a way that the cell value $\lambda(i,j)$ is the sum of its κ and the maximum of already calculated cells which can be next to it, that is, either from $(x, j+1)$ for $i < x \leqslant m$ or from $(i+1, y)$ for $j < y \leqslant n$. (This follows the above mentioned rule.) The gap penalty is introduced at this point; a pre-determined penalty, γ, is subtracted from λ of all the cells except for the one at the diagonal position, $(i+1, j+1)$:

$$\lambda(i,j) = \kappa(i,j) + \max\{\lambda(i+1,j+1), \lambda(x,j+1) - \gamma, \lambda(i+1,y) - \gamma\} \quad (i < x \leqslant m, j < y \leqslant n). \tag{1}$$

The definition of $\lambda(i,i)$ becomes clear; as a result of the summation operation, each $\lambda(i,j)$ represents the maximum score for the path which starts from that cell.

When the summation is completed on all the cells, the maximum match pathway can be traced back starting at the cell with the largest value in λ. If, for example, (s, t) contains the maximum value, the first residues to be aligned are $A(s)$ and $B(t)$. (Note that either s or t (or both) must be 1 for the first residue pair.) The next residue pair is sought by finding the maximum from the cells $(i, t+1)$ for $s < i \leqslant m$ and $(s+1, j)$ for $t < j \leqslant n$. The process is repeated until s and/or t reaches the limiting value(s), m and/or n. As in the matrix summation the gap penalty is imposed for the cells except for the one at the diagonal position, in this case, $(s+1, t+1)$. In the implementation of the algorithm described in this paper, gaps at each end of the path were not penalized. Furthermore, the same penalty was applied to gaps regardless of their sizes.

The scanning of $\lambda(x, j+1)$ for $i < x \leqslant m$ and $\lambda(i+1, y)$ for $j < y \leqslant n$ in (1) is a very consuming process. Fortunately, however, it is not necessary to scan all the cells. Note that in (1) the maximum for $\lambda(x, j+1)$ and $\lambda(i+1, y)$ is also the maximum of $\lambda(x, y)$ for $i < x \leqslant m$ and $j < y \leqslant n$. If we have this maximum stored in another array μ, the calculation of $\lambda(i,j)$ becomes:

$$\lambda(i,j) = \kappa(i,j) + \max\{\lambda(i+1,j+1), \mu(i+1,j+1) - \gamma\}$$

and μ can be calculated as:

$$\mu(i,j) = \max\{\lambda(i,j), \mu(i+1,j), \mu(i,j+1)\}.$$

Although the time required to calculate λ is dramatically reduced by storing the maximum values in μ, the algorithm now demands additional mn storage of the computer memory. Notice, however, that array μ serves only as a temporary storage needed for the calculation of λ, and it is not necessary to allocate as much storage as mn cells. Thus the memory space can be saved by storing only the necessary part of μ. In the following algorithm, for example, only a single one-dimensional array is allocated for μ.

Algorithm 1.

```
for i := m downto 1 do
for j := n downto 1 do
begin
λ(i,j) := κ(i,j)
        + max{λ(i+1,j+1), μ1(j+1) − γ};
  if j < n then
μ1(j+1) := max{λ(i,j+1), μ1(j+1), μ1(j+2)}
end;
```

Initial values of zero are assumed to have been assigned for both $\lambda(i,j)$ and $\mu 1(j)$ for $i > m$ or $j > n$. Here $\mu 1(j+1)$ represents the maximum of $\lambda(x,y)$ for $i < x \leqslant m$ and $j < y \leqslant n$. The inclusion of $\lambda(i,j+1)$ in the calculation of the new $\mu 1(j+1)$ effectively eliminates the need of the i-term in $\mu 1$. In the original algorithm by Needleman & Wunsch, the matrix summation requires time proportional to mn^2. The above algorithm takes time proportional to mn at the small expense of the additional storage for $\mu 1$.

IMPLEMENTATION OF THE ALGORITHM TO A MICROCOMPUTER

Based on the above algorithm a program was written in Pascal (TURBO Pascal 2.0, Borland International) and run on the Toshiba T3100, an IBM AT compatible microcomputer. Two sequences with each up to 170 residues can be compared within 10 s. Although the running time is acceptable, the program is of limited use because of the maximum lengths of protein sequences which can be compared. This limit is due to the amount of usable RAM for array data, 64K bytes, which is set by the operating system (PC-DOS 3.2) and the Pascal compiler. The restriction is unwarranted because the computer itself, as in many other comparable machines, is equipped with 640K bytes of RAM.

With commonly used high level programming languages two methods are avbailable to handle data whose size exceeds the above limit; one is to use inaccessible RAM area as a RAM or virtual disk, and the other is to use the heap (a stack-like storage) which is provided by the Pascal for dynamic variables. (The size of the heap depends on the program size and the total memory available to the machine used—with the system described here, 528,800 bytes are available.) Both methods use the same principle. Notice that in Algorithm 1 the calculation of $\lambda(i,j)$ requires only $(i+1)$ th row of λ at a time. Indeed

the summation of the entire λ can be accomplished using 2 one-dimensional arrays, in this case, $(1,..,n)$ in the main RAM, and with the whole array $(1,..,m,1,..,n)$ stored either in the virtual disk or in the heap. In the following Algorithm 2, this approach is described; the calculation of the whole λ is carried out on a two-dimensional array (but consists of only two rows), β. (In the description of the following algorithms, the storing and retrieving processes are indicated by "*". (See in Appendix how data are handled in the RAM disk and in the heap.)

Algorithm 2.

```
for j := 1 to n do
  β(1,j) := κ(m,j) {calculation of the last row,
  λ(m,j)};
*store β((1,j),j = 1,n) {as λ(m,j),j = 1,n};
for i := m − 1 downto 1 do
begin
  for j := 1 to n do
    β(2,j) := β(1,j) {exchange rows of β};
  for j := n downto 1 do
  begin
    if j = n then
      β(1,j) := κ(i,j)
    else
      β(1,j) := κ(i,j)
                + max{β(2,j+1), μ2(j+1) − γ};
    μ2(j + 1) := max{β(1,j + 1), μ2(j + 1),
                μ2(j + 2)};
  end;

  *store β((1,j),j = 1,n) {as λ(i,j),j = 1,n};
end;
```

Again we assume that initial values of zero have been assigned to $\beta(1,j)$ and $\mu 2(j)$ for $j > n$. Since the entire array is needed only to trace back the maximum match pathway and if the statistical analysis of the similarity between two sequences is all that is required, and the actual alignment of the two sequences is of no interest, even the use of the virtual disk or the heap becomes unnecessary.

Tracing back of the maximum match pathway is carried out as follows:

Algorithm 3.

```
(s, t) := (1, 1) {starting location};
while s ≤ m and t ≤ n do
begin
  *get λ((s,j),j = t,n);
  (x, y) := (s, t) {best location yet};
  b := λ(s, t) {best score so far};
  d := 0 {shortest distance};
  for j := t + 1 to n do
    if λ(s,j) − γ > b then
    begin
      (x, y) := (s,j);
      b := λ(s,j) − γ;
      d := j − t
    end;
  for i := s + 1 to m do
  begin
  *get λ(i, t);
  if λ(i, t) − γ > b or
    [λ(i, t) − γ = b and i − s < d] then
  begin
    (x, y) := (i, t);
    b := λ(i, t) − γ;
    d := t − j;
  end;
  print (x, y);
  (s, t) := (x + 1, y + 1)
  end;
  end;
```

The original Pascal program was modified to implement Algorithms 2 and 3. As an example of the application of the program, the alignment of two hypothetical sequences are shown in Fig. 1. With 512K bytes of RAM allocated as a virtual disk or using a similar amount of storage in the heap, now two sequences up to 500 residues long can be compared. However, between the two alternative methods the one which uses the heap for data storage is recommended. As expected the methods of handling data in the RAM disk slows down the computation considerably; the time required for tracing the maximum match pathway is almost prohibitive. This is partly due to the tracing process itself, which, even if the whole matrix is in the main RAM, takes time proportional to mn. The problem is aggravated in the program because the data in a two-dimensional array, $(1,..,m,1,..,n)$, are stored in the RAM disk as a collection of one-dimensional arrays of $(1,..,n)$, which has to be accessed i times (in the actual program $m - i + 1$ times) for the alignment of i-th residue.

To shorten the tracing process in the RAM disk method, the program can be modified so that it scans only 50 elements at most along each row and column, that is, λ is treated as a pseudo-band matrix. The result by this method is identical if the size of each gap is of < 50 residues long. If a gap is more than that size, the trace will be on a wrong pathway. This can be easily tested as follows (a valid test applicable to other sequence comparison algorithms). By definition the best alignment in this case is the alignment which produces the maximum sum of the weights given to the aligned residue pairs minus number of gaps times the gap penalty. The program is designed to lists as shown in Fig. 1 for each of aligned residues, the weight, the sum from the start of the alignment (calculated during the tracing process) and the score from λ (calculated during the matrix summation process). If the correct pathway is traced back the weight sum and the maximum match score are equal. The size of residues to be scanned depends on the two sequences; if they are short, say < 150, this approach

```
Sequence alignment of sq1 and sq2

  sq1      sq2       score     weight   gap  wt.sum

           D   1
           N   2
           A   3
 1   A     A   4       199         10     0      10
 2   D     G   5       189          9            19
 3   N     K   6       180          9            28
 4   I     S   7       171          7            35
 5   Q     D   8       164         10            45
 6   L     L   9       154         14            59
 7   E     P  10       140          7            66
 8   I
 9   D     Q  11       139         10     6      70
10   S     S  12       129         10            80
11   I     G  13       119          5            85
12   V     L  14       114         10            95
13   K     K  15       104         20           115
14   Q     Q  16        84         12           127
           L  17
           V  18
           M  19
15   E     A  20        78          8     6     129
16   F     L  21        70         10           139
17   G     E  22        60          8           147
18   A     E  23        52          8           155
19   I     F  24        44          9           164
20   D     D  25        35         12           176
21   T     T  26        23         11           187
22   Q     Q  27        12         12           199
           A  28


sq1 (top) vs sq2 (bottom)

  Gap Penalty = 6
  Alignment Score =   199

   ADNIQLEIDSIVKQ   EFGAIDTQ
DNAAGKSDLP QSGLKQLVMALEEFDTQA
```

Fig. 1. Alignment of two hypothetical sequences, sq1 and sq2. Weights were taken from (Dayhoff *et al.*, 1978), and were adjusted by adding 8 to each of them to make them all non-negative.

is not necessary. If, on the other hand, comparing two sequences of very different lengths and a large gap is expected, then it may be necessary to increase the size for scanning at the expense of a longer computing time.

Acknowledgements—The author thanks Professor John Lee for reading the manuscript. This work was supported by a grant from NIH, GM28139 to J. W. Lee.

REFERENCES

Dayhoff M. O., Schwartz R. M. and Orcutt B. C. (1978) *Atlas of Protein Sequence and Structure* (Edited by Dayhoff M. O.), Natl. Biomed. Res. Found., Washington, D. C. Vol. 55, pp. 345–352.

Gibbs A. J. and McIntyre G. A. (1970) *Eur. J. Biochem.* **16,** 1.

McLachlan A. D. (1971) *J. Mol. Biol.* **61,** 409.

Murata M., Richardson J. S. and Sussman J. L. (1985) *Proc. Natl. Acad. Sci. U.S.A.* **82,** 3073.

Needleman S. B. and Wunsch C. D. (1970) *J. Mol. Biol.* **48,** 444.

Sankoff D. (1972) *Proc. Natl. Acad. Sci. U.S.A.* **68,** 4.

Sellers P. (1974) *SIAM J. Appl. Math.* **26,** 787.

Waterman M. S. (1984) *Bull. Math. Biol.* **46,** 473.

Waterman M. S., Smith T. F. and Beyer W. A. (1976) *Adv. Math.* **20,** 367.

APPENDIX

The following methods can be used to store a collection of one-dimensional array data, here denoted by ALPHA, in the RAM disk or in the heap. Note that in the RAM disk method, the size of the disk file component is not specified, therefore, the data are stored dynamically. In the heap method, on the other hand, the pointer variables are defined in a static array, a deviation from the concept of dynamic data storage, but it serves its purpose well here, since the same data must be accessed repeatedly in the path tracing.

1. The use of the heap:

```
{type and variable declaration}
const maxlen = 500;
type
  datapointer = ^ data;
  data = record
      alpha: array [1 .. maxlen] of integer;
    end;
```

```
var
  ptr: datapointer;
  rowptr: array [1 . . maxlen] of datapointer;
{Alg. 1, for example—relevant part only}
begin
ptr : = nil;
new (ptr);
for j : = 1 to n do
begin
  beta[1,j]: = weight[a.protein[m],b.protein[j]];
  ptr^ .alpha[j]: = beta[1,j];
end;
rowptr[m] : = ptr;
for i: = m − 1 downto 1 do
begin
  :
  new (ptr);
  for j: = 1 to n do
    ptr^ .alpha[j]: = beta[1,j];
  rowptr[i]: = ptr;
end;
end;
```

2. The use of the RAM disk under PC-DOS or MS-DOS with 640K system: The system must be configured first by creating a "CONFIG.SYS" file in the stem directory, which contains, for example, the following line;

DEVICE = VDISK 512 512 64

```
{type and variable declaration}
const maxlen = 500;
type
  data = record
      alpha:array[1 . . maxlen] of integer;
    end;
var
  datafile: file of data;
  datarec:data;
{Algorithm 1, relevant parts only}
begin
  assign (datafile, 'd:align.dta') {d: for RAM disk};
  rewrite (datafile);
  for i: = 1 to n do
    beta[1,j] : = weight[a.protein[m], b.protein[j]];
  with datarec do
  begin
    for j : = 1 to n do
      alpha[j] : = beta[1,j];
    write (datafile, datarec);
  end;
  for i: m − 1 downto 1 do
  begin
    .
    .
    with datarec do
    begin
      for j: = 1 to n do
        alpha[j] : = beta[1,j];
      write (datafile, datarec);
    end;
  end;
  close (datafile);
end;
```

{A total of m alpha will be stored in the locations from 0 to $m - 1$, and each of them, as a component, is accessed by SEEK (i), for $i = 0 . . m - 1$.}