Algorithms for pairwise alignment of biological sequences

Peter Sestoft
sestoft@dina.kvl.dk
Department of Mathematics and Physics, KVL

2000-04-05

Literature:

Durbin et al: *Biological Sequence Analysis*, Cambridge University Press 1998, chapter 2

Altschul et al: Gapped BLAST and PSI-BLAST: a new generation of protein database search programs, *Nucleic Acids Research* 25 (1) 1997.

Algorithms for pairwise alignment

Consider two amino acid sequences HEAGAWGHEE and PAWHEAE.

Call them x and y, and denote their lengths by $n = 10$ and $m = 7$.

We discuss two alignment problems:

- global alignment: all of x must be aligned with all of y (Needleman-Wunsch):

  ```
  HEAGAWGHE-E
  --P-AW-HEAE
  ```

- local alignment: a subsequence of x must be aligned with a subsequence of y (Smith-Waterman):

  ```
  AWGHE
  AW-HE
  ```

As we can see, an alignment may contain gaps. We consider two kinds of gap costs:

- simple linear gap costs: a gap of length g has score $-d \cdot g$
- affine gap costs: a gap of length g has score $-d - e \cdot g$

Warming up: Similarity of two strings; substitution matrices

Consider two strings, e.g. $x =$ RLKAE and $y =$ KNKGE of the same length $n = m = 5$.

In an *ungapped alignment*, an amino acid x_i in x must be matched by an amino acid y_i in y.

The score of a match between amino acids x_i and y_i is $score[x_i][y_i]$, given by a *substitution matrix*.

It describes the likelihood that amino acid x_i was replaced (substituted) by amino acid y_i by an evolutionary event.

A high score means 'likely' and a low one means 'unlikely'.

An amino acid is likely to remain the same, so the diagonal of the substitution matrix has high numbers.

The similarity of the strings x and y is just the sum of the scores:

$$sim(x, y) = score(x_1, y_1) + score(x_2, y_2) + score(x_3, y_3) + score(x_4, y_4) + score(x_5, y_5)$$

Various substitution matrices are used in alignment algorithms, e.g. BLOSUM50 and the PAM matrices.

So far so good. The fun begins when we allow gaps in either x or y.

The BLOSUM50 substitution matrix

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	5	-2	-1	-2	-1	-1	-1	0	-2	-1	-2	-1	-1	-3	-1	1	0	-3	-2	0
R	-2	7	-1	-2	-4	1	0	-3	0	-4	-3	3	-2	-3	-3	-1	-1	-3	-1	-3
N	-1	-1	7	2	-2	0	0	0	1	-3	-4	0	-2	-4	-2	1	0	-4	-2	-3
D	-2	-2	2	8	-4	0	2	-1	-1	-4	-4	-1	-4	-5	-1	0	-1	-5	-3	-4
C	-1	-4	-2	-4	13	-3	-3	-3	-3	-2	-2	-3	-2	-2	-4	-1	-1	-5	-3	-1
Q	-1	1	0	0	-3	7	2	-2	1	-3	-2	2	0	-4	-1	0	-1	-1	-1	-3
E	-1	0	0	2	-3	2	6	-3	0	-4	-3	1	-2	-3	-1	-1	-1	-3	-2	-3
G	0	-3	0	-1	-3	-2	-3	8	-2	-4	-4	-2	-3	-4	-2	0	-2	-3	-3	-4
H	-2	0	1	-1	-3	1	0	-2	10	-4	-3	0	-1	-1	-2	-1	-2	-3	2	-4
I	-1	-4	-3	-4	-2	-3	-4	-4	-4	5	2	-3	2	0	-3	-3	-1	-3	-1	4
L	-2	-3	-4	-4	-2	-2	-3	-4	-3	2	5	-3	3	1	-4	-3	-1	-2	-1	1
K	-1	3	0	-1	-3	2	1	-2	0	-3	-3	6	-2	-4	-1	0	-1	-3	-2	-3
M	-1	-2	-2	-4	-2	0	-2	-3	-1	2	3	-2	7	0	-3	-2	-1	-1	0	1
F	-3	-3	-4	-5	-2	-4	-3	-4	-1	0	1	-4	0	8	-4	-3	-2	1	4	-1
P	-1	-3	-2	-1	-4	-1	-1	-2	-2	-3	-4	-1	-3	-4	10	-1	-1	-4	-3	-3
S	1	-1	1	0	-1	0	-1	0	-1	-3	-3	0	-2	-3	-1	5	2	-4	-2	-2
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	2	5	-3	-2	0
W	-3	-3	-4	-5	-5	-1	-3	-3	-3	-3	-2	-3	-1	1	-4	-4	-3	15	2	-3
Y	-2	-1	-2	-3	-3	-1	-2	-3	2	-1	-1	-2	0	4	-3	-2	-2	2	8	-1
V	0	-3	-3	-4	-1	-3	-3	-4	-4	4	1	-3	1	-1	-3	-2	0	-3	-1	5

Global alignment (Needleman-Wunsch 1970)

Consider two strings, e.g. $x =$HEAGAWGHEE and $y =$PAWHEAE of lengths $n = 10$ and $m = 7$.

In a *gapped alignment*, an amino acid x_i is matched either by an amino acid y_j, or by a *gap*.

The score of a match between amino acids x_i and y_j is $score[x_i][y_j]$, given by e.g. the BLOSUM50 matrix.

The score of a match between an amino acid and a gap is $-d$, where d may be 8.

We want to find an *optimal* global alignment of x and y: one that has the maximal sum of scores.

Naive attempt:

Enumerate all possible alignments of x and y, compute their scores, then choose one with maximal score.

But . . . the number of possible matches for two sequences is very large (may be 10^{100}).

The naive approach would be much too slow on even the largest existing computers.

Useful observation 1:

> Any prefix of the optimal alignment between x and y
> is an optimal alignment between a prefix $x_{1\ldots i}$ of x and a prefix $y_{1\ldots j}$ of y.

So an optimal alignment can be computed by scanning x and y from left to right, recording only the optimal alignments between prefixes of x and y, and forgetting all the non-optimal ones.

More precisely, we can build a table F in which

$$F(i, j) = \text{ the maximal score for an alignment between } x_{1\ldots i} \text{ and } y_{1\ldots j}$$

Then, by definition, $F(n, m)$ is the maximal score for a global alignment between x and y.

(Because $x_{1..n}$ is the entire string x, and $y_{1..m}$ is the entire string y).

Useful observation 2:

> The value $F(i, j)$ depends only on the values
> $F(i-1, j-1)$, $F(i-1, j)$, and $F(i, j-1)$.

This is because an optimal alignment between $x_{1\ldots i}$ and $y_{1\ldots j}$ consists of either

- an optimal alignment between $x_{1\ldots(i-1)}$ and $y_{1\ldots(j-1)}$ extended with a match between x_i and y_j; or
- an optimal alignment between $x_{1\ldots(i-1)}$ and $y_{1\ldots j}$ extended with a match between x_i and a gap; or
- an optimal alignment between $x_{1\ldots i}$ and $y_{1\ldots(j-1)}$ extended with a match between a gap and y_j.

So we can fill in the F table from left to right and top to bottom.

This 'filling in the table' is called *dynamic programming* (Bellman 1955).

Table F gives us the maximal *score*. How find a corresponding optimal *alignment*?

When filling in $F(i, j)$, we record the traceback $B(i, j)$ from (i, j):

The traceback points at the cell that led to the maximal score: $(i-1, j-1)$ or $(i-1, j)$ or $(i, j-1)$.

When we are finished we find an optimal alignment just by following the traceback from (n, m) to $(0, 0)$.

Filling in the F matrix for $x =$ HEAGAWGHEE and $y =$ PAWHEAE

$x \backslash y$		H	E	A	G	A	W	G	H	E	E
P											
A											
W											
H											
E											
A											
E											

The filled-in F matrix for global alignment of $x =$ HEAGAWGHEE and $y =$ PAWHEAE

$x \backslash y$		H	E	A	G	A	W	G	H	E	E
	0	-8	-16	-24	-32	-40	-48	-56	-64	-72	-80
P	-8	-2	-9	-17	-25	-33	-41	-49	-57	-65	-73
A	-16	-10	-3	-4	-12	-20	-28	-36	-44	-52	-60
W	-24	-18	-11	-6	-7	-15	-5	-13	-21	-29	-37
H	-32	-14	-18	-13	-8	-9	-13	-7	-3	-11	-19
E	-40	-22	-8	-16	-16	-9	-12	-15	-7	3	-5
A	-48	-30	-16	-3	-11	-11	-12	-12	-15	-5	2
E	-56	-38	-24	-11	-6	-12	-14	-15	-12	-9	1

The traceback is recorded in a matrix B with the same shape as F.

Implementing global alignment: Initialization

Upper border: position $(i, 0)$ represents the alignment of $x_{1\ldots i}$ to the empty prefix of y.

That is, the prefix $x_{1\ldots i}$ has been matched with i gaps in y.

With simple linear gap costs, the score is $-d \cdot i$.

The traceback pointer at $(i, 0)$ points to $(i - 1, 0)$.

The left-hand border is similar.

Hence we initialize the borders as follows:

```
for (int i=1; i<=n; i++) {
  F[i][0] = -d * i;
  B[i][0] = new Traceback2(i-1, 0);
}
for (int j=1; j<=m; j++) {
  F[0][j] = -d * j;
  B[0][j] = new Traceback2(0, j-1);
}
```


Implementing global alignment: Filling in the matrix

Position (i, j) may be reached

- from $(i - 1, j - 1)$ with a match, adding $score[x_i][y_j]$ to the score;
- from $(i - 1, j)$ with a gap in y, subtracting d from the score; or
- from $(i, j - 1)$ with a gap in x, subtracting d from the score.

The traceback $B(i, j)$ points to the source of the maximal resulting score $F(i, j)$. Thus:

```
for (int i=1; i<=n; i++)
  for (int j=1; j<=m; j++) {
    int s = score[seq1.charAt(i-1)][seq2.charAt(j-1)];
    int val = max(F[i-1][j-1]+s, F[i-1][j]-d, F[i][j-1]-d);
    F[i][j] = val;
    if (val == F[i-1][j-1]+s)
      B[i][j] = new Traceback2(i-1, j-1);
    else if (val == F[i-1][j]-d)
      B[i][j] = new Traceback2(i-1, j);
    else if (val == F[i][j-1]-d)
      B[i][j] = new Traceback2(i, j-1);
  }
B0 = new Traceback2(n, m);
```

The start `B0` of the traceback is cell (n, m).

Local alignment of $x =$ HEAGAWGHEE and $y =$ PAWHEAE (Smith-Waterman 1981)

A subsequence of x must be aligned with a subsequence of y:

```
AWGHE
AW-HE
```

Requirement: the expected score of a random match must be negative.

If the score of a random match extension were positive, then any local alignment could be profitably extended to a 'better' (but probably biologically meaningless) one.

New interpretation of $F(i, j)$:

$F(i, j) =$ the maximal score for an alignment between a suffix of $x_{1\ldots i}$ and a suffix of $y_{1\ldots j}$

Implementing local alignment: Initialization

Upper border: position $(i, 0)$ represents the alignment of a suffix of $x_{1\ldots i}$ to an empty sequence.

An empty match, with score 0, is the best we can do (because gaps have negative scores).

Then $(i, 0)$ is the start of a new local alignment, and the traceback pointer at $(i, 0)$ points nowhere.

The left-hand border is similar.

Hence we initialize the border cells to 0 and the traceback to `null` (this is the default value in Java).

Implementing local alignment: Filling in the matrix

Position (i, j) may be reached

- from nowhere, with score 0, because we can always start a new local alignment;
- from $(i-1, j-1)$ with a match, adding $score[x_i][y_j]$ to the score;
- from $(i-1, j)$ with a gap in y, subtracting d from the score; or
- from $(i, j-1)$ with a gap in x, subtracting d from the score.

The traceback $B(i, j)$ points to the source of the maximal resulting score $F(i, j)$, if any. Thus:

```
for (int i=1; i<=n; i++)
  for (int j=1; j<=m; j++) {
    int s = score[seq1.charAt(i-1)][seq2.charAt(j-1)];
    int val = max(0, F[i-1][j-1]+s, F[i-1][j]-d, F[i][j-1]-d);
    F[i][j] = val;
    if (val == 0)
      B[i][j] = null;
    else if (val == F[i-1][j-1]+s)
      B[i][j] = new Traceback2(i-1, j-1);
    else if (val == F[i-1][j]-d)
      B[i][j] = new Traceback2(i-1, j);
    else if (val == F[i][j-1]-d)
      B[i][j] = new Traceback2(i, j-1);
} }
```

The start `B0` of the traceback must be set to some cell (i, j) in F that has maximal score.

Reducing the space consumption of global alignment

All algorithms require time $O(nm)$ to fill in the tables and space $O(nm)$ in the computer to store the tables.

However, column i of F depends only on column $i-1$.

So only two columns of F (and the traceback) need to be stored at the same time.

Hence we can compute the best score using only space $O(n+m)$.

How reconstruct the optimal global alignment in this case?

When $n \leq 1$ or $m \leq 1$, use the standard algorithm (in this case it uses little space anyway).

Otherwise, let $u = n/2$ and assume the optimal alignment passes through (u, v).

(We can determine v while filling in F).

Recursively determine

- the optimal global alignment z_1 between $x_{1\ldots u}$ and $y_{1\ldots v}$
- the optimal global alignment z_2 between $x_{(u+1)\ldots n}$ and $y_{(v+1)\ldots m}$

Then the optimal alignment between x and y is the concatenation of z_1 and z_2.

Reducing the space consumption of local alignment

Fill in F using only space $O(n+m)$ as above.

Keep track of the starting point (s_1, s_2) and the ending point (e_1, e_2) of the local alignment with highest score.

Compute the optimal *global* alignment between the subsequences $x_{s_1\ldots e_1}$ and $y_{s_2\ldots e_2}$ in space $O(n+m)$.

The result is also the optimal *local* alignment between x and y.

Affine gap costs

Until now we used *linear* gap costs $g(k) = -dk$, where $d = 8$.

Thus a gap of length $k = 4$ has 4 times the cost of a gap of length 1.

This is unrealistic; too expensive, biologically speaking.

A gap arises by an evolutionary event, and a long gap is nearly as likely to arise as a short one.

Better use *affine gap costs* of the form $g(k) = -d - ek$ where $d = 12$ and $e = 2$.

Hence it is expensive to open a gap (-12) but inexpensive to extend it (-2).

Alignment with affine gap costs is done by dynamic programming using three matrices F_1, F_2, F_0 instead of one.

The matrices have the following meanings:

$F_0(i,j)$ = max score for alignment between $x_{1\ldots i}$ and $y_{1\ldots j}$ ending with a match between x_i and y_j

$F_1(i,j)$ = max score for alignment between $x_{1\ldots i}$ and $y_{1\ldots j}$ ending with a match between x_i and a gap in y

$F_2(i,j)$ = max score for alignment between $x_{1\ldots i}$ and $y_{1\ldots j}$ ending with a match between a gap in x and y_j

Observation about the F matrix in dynamic programming

- When the strings x and y are *identical*, the traceback will follow the diagonal of the F matrix.
  In that case, only the diagonal needs to be filled in (which takes much less time).
- When the strings x and y are *very similar*, the traceback will follow a band along the diagonal.
  In that case, only the elements of that band of F need to be filled in.
- We may speed up dynamic programming by filling in *only* a band along the diagonal of F.
  But this may overlook a good (high-scoring) alignment whose traceback would go outside the band.
  Hence this gives only an approximation to the optimal alignment.

Database searches

A sequence database contains a large number of sequences (e.g. 100.000).

When searching a database we a given short *query string*, e.g. of length $n = 500$.

We then seek the best local, gapped alignment between the query string and each of the database sequences.

So we might use the Smith-Waterman algorithm for each sequence in the database. Too slow in practice.

Database search programs (Blast and Fasta) do use dynamic programming, but only after some preliminary work.

The Blast 2 database search algorithm (Altschul et al. 1997)

Let a query string be given.

- Find *hits* between 3-letter substrings of the query string and 3-letter substrings of the database strings.
  A hit must have a score of at least T, e.g. $T = 11$.
- Find two non-overlapping hits on the same diagonal that are close to each other (distance less than A).
  Such neighbour hits are probably part of the same (ungapped) local alignment.
- Extend a hit in both directions to get an *ungapped* local alignment (just add up scores).
  Stop the extension when the score of the alignment has fallen more than X below the maximum attained.
- If the resulting local alignment is good (score at least S_g) then try to make a *gapped* alignment:
  Extend the alignment in both directions, using dynamic programming.
  Do not fill in F matrix elements if their score would fall more than X_g below the maximum attained.
  This will give a variable-width band along the diagonal.

Blast 2 is 100 times faster than Smith-Waterman on the computer, and nearly as sensitive and selective.