Bioinformatics Algorithms

(Fundamental Algorithms, module 2)

Zsuzsanna Lipták

Masters in Medical Bioinformatics
academic year 2018/19, II. semester

Pairwise Alignment 3

Optimal pairwise alignment in linear space

Given two sequences s, t of length n:

- DP algorithm for global alignment: $O(n^2)$ time and space
- if we only want the score of an optimal alignment $sim(s, t)$ (problem variant 1), then we can do this in $O(n^2)$ time and $O(n)$ space (space-saving variant)
- But that algo does not give us the optimal alignment itself (problem variant 2)
- Now: algorithm for computing an optimal alignment itself in time $O(n^2)$ but space $O(n)$

There are several algorithms achieving this, e.g. Hirschberg (1975) a.k.a. Myers-Miller (1988). Here we present the divide-and-conquer algorithm from the book by Durbin, Eddy, Krogh, Mitchison: *Biological Sequence Analysis*, 1998 (ch. 2.6).

$s =$ GAAGA, $t =$ CACA match: 2, mismatch: -1, gap: -1

$D(i,j)$			C	A	C	A
		0	1	2	3	4
	0	0	−1	−2	−3	−4
G	1	−1	−1	−2	−3	−4
A	2	−2	−2	1	0	−1
A	3	−3	−3	0	0	2
G	4	−4	−4	−1	−1	1
A	5	−5	−5	−2	−2	1

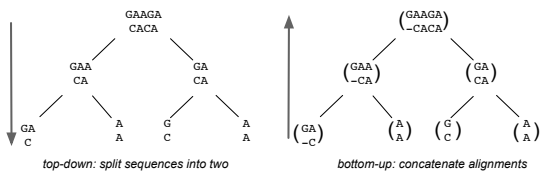
The optimal alignments are:

1. $\binom{\text{GAAGA}}{\text{-CACA}}$
2. $\binom{\text{GAAGA}}{\text{CA-CA}}$
3. $\binom{\text{GAAGA}}{\text{C-ACA}}$
4. $\binom{\text{GAAGA}}{\text{CAC-A}}$

Consider the first optimal alignment $\binom{\text{GAAGA}}{\text{-CACA}}$:

Idea: Divide-and-conquer

We divide the two sequences s, t in two parts, left and right, align left with left, right with right, and then concatenate the two alignments:



Why does this work?

Generalization of the theorem on which the DP recursion for pairwise alignment is based (see p. 18 of "Pairwise alignment 1"):

Theorem

Let alignment $\mathcal{A}$ be the concatenation of two alignments $\mathcal{B}$ and $\mathcal{C}$, i.e. $\mathcal{A} = \mathcal{B} \cdot \mathcal{C}$. If $\mathcal{A}$ is optimal, then so are $\mathcal{B}$ and $\mathcal{C}$.

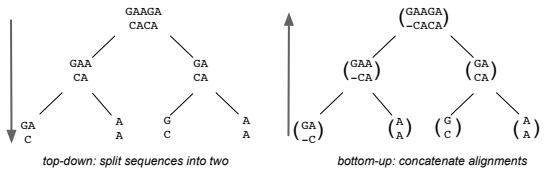
Proof

Again, we prove the claim by contradiction. Let $\mathcal{A}$ be an alignment of s and t, $\mathcal{B}$ one of s' and t', and $\mathcal{C}$ one of s'' and t''. (Thus $s = s's''$ and $t = t't''$.) Assume that $\mathcal{B}$ is not optimal, then $\mathcal{B}$ can be replaced by some alignment $\mathcal{B}'$ of the same strings s', t' with higher score than $\mathcal{B}$. Define $\mathcal{A}' = \mathcal{B}' \cdot \mathcal{C}$. Then $\mathcal{A}'$ is also an alignment of s, t, and

$$score(\mathcal{A}') = score(\mathcal{B}') + score(\mathcal{C}) > score(\mathcal{B}) + score(\mathcal{C}) = score(\mathcal{A}),$$

a contradiction to the optimality of $\mathcal{A}$.—The case where $\mathcal{C}$ is not optimal is analogous. □

So it's okay to align optimally the left and the right parts, and then to concatenate them:



Question

But how do we know where to divide them?

The problem is: The reverse of the theorem is not true!
Concatenating two optimal al's does not always yield an optimal al.:
e.g. $\binom{\texttt{GA}}{\texttt{G-}} \cdot \binom{\texttt{-C}}{\texttt{AC}}$ yields $\binom{\texttt{GA-C}}{\texttt{G-AC}}$, which is not optimal.

The problem is: The reverse of the theorem is not true!
Concatenating two optimal al's does not always yield an optimal al.:
e.g. $\binom{\texttt{GA}}{\texttt{G-}} \cdot \binom{\texttt{-C}}{\texttt{AC}}$ yields $\binom{\texttt{GA-C}}{\texttt{G-AC}}$, which is not optimal.

Definition

A cut is a pair of positions (n', m'), where $1 \le n' \le n$, and $1 \le m' \le m$ (with $|s| = n, |t| = m$).

We are looking for a good cut, i.e. one for which there is an optimal alignment passing through it.

- $(3, 2)$ is a good cut: the optimal alignments $\binom{\texttt{GAAGA}}{\texttt{-CACA}}, \binom{\texttt{GAAGA}}{\texttt{CA-CA}}, \binom{\texttt{GAAGA}}{\texttt{C-ACA}}$ all pass through the cell $(3, 2)$, aligning GAA with CA.
- $(3, 3)$ is a good cut: the optimal alignment $\binom{\texttt{GAAGA}}{\texttt{CAC-A}}$ passes through the cell $(3, 3)$, aligning GAA with CAC.
- $(3, 1)$ is not a good cut, since no optimal alignment passes through cell $(3, 1)$, i.e. no optimal alignment aligns GAA with C.

Computing a good cut

1. In sequence 1, we will always take the middle cut position $n' = \lceil n/2 \rceil$.
2. In sequence 2, we will remember where the middle row $n' = \lceil n/2 \rceil$ was crossed.
3. For this, we will need to compute another matrix M (again, in space-saving manner!).

Matrix M

- Definition: For $i \ge n'$, cell $M(i, j)$ contains an index r s.t. there exists an optimal alignment with score $D(i, j)$ passing through cell (n', r).
- Computation of $M(i, j)$:
  - for $i = n'$ and $j = 1, \ldots, m$: $M(n', j) = j$;
  - for $i > n', 0 \le j \le m$:
    $M(i, j) = M(i', j')$, where $D(i, j)$ derives from cell (i', j')
    —if there is more than one, choose acc. to priority (e.g. *left-diag-top*)
- Note that by definition $(i', j') \in \{(i-1, j), (i-1, j-1), (i, j-1)\}$.
- Then $M(n, m) = r$ s.t. there is an optimal alignment of s and t which passes through cell $(\lceil n/2 \rceil, r)$.
- Thus, we can use the cut $(n', r) = (\lceil n/2 \rceil, M(n, m))$ in the divide-step and recurse with $s_1 \ldots s_{n'}$ and $t_1 \ldots t_r$ on the left, and $s_{n'+1} \ldots s_n$ and $t_{r+1} \ldots t_m$ on the right.

Back to the example (p. 4): Here $n = 5$, thus $n' = \lceil n/2 \rceil = 3$. We compute the matrix M according to the priority *left-diag-top*:

$D(i,j)$		0	C 1	A 2	C 3	A 4
	0	0	−1	−2	−3	−4
G	1	−1	−1	−2	−3	−4
A	2	−2	−2	1	0	−1
A	3	−3	−3	0	0	2
G	4	−4	−4	−1	−1	1
A	5	−5	−5	−2	−2	1

$M(i,j)$	0	1	2	3	4
3	0	1	2	3	4
4	0	0	2	2	4
5	0	0	0	2	2

So we have to recurse with $r = 2$, i.e. GAA,CA (left) and GA,CA (right).

For the left part GAA,CA, we have $n' = \lceil n/2 \rceil = 2$, and we get

$D(i,j)$			C	A
		0	1	2
	0	0	−1	−2
G	1	−1	−1	−2
A	2	−2	−2	1
A	3	−3	−3	0

$M(i,j)$	0	1	2
2	0	1	2
3	0	0	1

Thus, $r = 1$ and we have to divide these at cut $(2, 1)$, yielding GA,C and A,A.

Algorithm PWA(s,t)

1. if $\max(|s|, |t|) \leq 2$, then return an optimal alignment computed with N-W-algorithm
2. else
3. for $i = 0$ to $n' - 1$ compute i'th row of D (space-saving manner, row-wise)
4. for $i = \lceil n/2 \rceil$ to n, compute i'th row of D and i'th row of M (space-saving manner, row-wise)
5. $r \leftarrow M(n, m)$
6. return $PWA(s_1 \dots s_{\lceil n/2 \rceil}, t_1 \dots t_r)$ concatenated with $PWA(s_{\lceil n/2 \rceil + 1} \dots s_n, t_{r+1} \dots t_m)$.

Analysis (1)

Space

- all matrix computations are space-saving (row-wise), they all need linear space in the number of columns, which is always $\leq m$
- at any given time, there are the two matrices D and M to be computed
- nothing needs to be stored for later, once we have computed $r = M(n, m)$
- thus for the matrix computations we need space $O(m)$;
- we need to store the partial alignments, whose total length is the length of the final alignment, thus $O(n + m)$
- altogether space $O(n + m)$

Analysis (2)

Time

- in the first iteration, we compute the entries of the two matrices D and M, each in constant time: $\underbrace{(n+1)(m+1)}_{D} + \underbrace{\lceil n/2 \rceil (m+1)}_{M}$ entries, so $O(nm)$ time
- In each iteration, we are exactly halving the problem size (wherever we cut t, string s is always cut in the middle), thus we get:

$$nm + \frac{1}{2}nm + \frac{1}{4}nm + \ldots \leq nm \sum_{k=0}^{\infty} \frac{1}{2^k} = 2nm \in O(nm).$$

Thus we doubled the time (asymptotically the same: $O(nm)$), but reduced the space from quadratic to linear.