3 FastA and the chaining problem

We will discuss:

- Heuristics used by the FastA program for sequence alignment
- Chaining problem

3.1 Sources for this lecture

- Lectures by Volker Heun, Daniel Huson and Knut Reinert, in particular last years lectures
- Dan Gusfield: Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology. Cambridge University Press, Cambridge, 1997. ISBN 0-521-58519-8
- David B. Mount: Bioinformatics. Sequence and Genome Analysis. Cold Spring Harbor Laboratory Press, New York, 2001. ISBN 0-87969-608-7
- File fasta3x.doc from the FastA distribution

3.2 Motivation

A local alignment of two sequences x, y can be computed by the Smith-Waterman algorithm, which requires $O(|x| \cdot |y|)$ time and space in its original form.

While the space requirement can be pushed down to $O(|x| + |y|)$ using a divide-and-conquer algorithm, the running time remains $O(|x| \cdot |y|)$, which is not feasible for many applications.

Therefore, heuristic approaches are of great importance in practice.

Heuristics

A *heuristic* is an—often problem-specific—algorithm that will yield reasonable results, even if it is not provably optimal or even lacks a performance guarantee.

In most cases, it is intuitively clear that the method works / is fast / is space efficient (in all / many / typical / special cases) but we do not (yet?) have a sound analysis for it.

3.3 FastA

FastA is a program for heuristic pairwise local alignment of nucleic and amino acid sequences.

FastA is much faster than the exact algorithms based on dynamic programming, and can be used to search a database of sequences for homologs to a query sequence. However its speed is based on heuristic considerations and it may fail to detect similar sequences in certain cases.

The name FastA is an abbreviation for "fast all". (There was a version for nucleic acids before.) The official pronounciation is "fast-ay".

First version by Pearson and Lipman, 1988; recent version (3.4) August 2002.

3.4 Main steps of the FastA algorithm

1. *Find hot-spots.* A hot-spot is a short, exact match between the two sequences.
2. *Find diagonal runs.* A diagonal run is a collection of hot-spots on the same diagonal within a short distance from each other.
3. *Rescore the best diagonal runs.* This is done using a substitution matrix. The best "initial region" is INIT1.
4. *Chain several initial regions.* This is where the chaining problem comes up. The result is INITN.
5. *Moreover, compute an optimal local alignment in a band around INIT1.* The result is called OPT.
6. *Use Smith-Waterman alignments to display final results.*

3.5 Hot-spots

Essentially, a *hot-spot* is "a dot in a dot plot".

Definition. Let the input sequences be $x = x[1 \dots m]$ and $y = y[1 \dots n]$. Then a *hot-spot* is a pair (i, j) such that the substrings of length k starting at the positions i and j are equal, i. e., $x[i \dots i + k - 1] = y[j \dots j + k - 1]$. Here $k = ktup$ (for "k-tuple") is a user-specified parameter.

The idea behind hot-spots is that in most cases, homologous sequences will contain at least a few identical segments. By default, FastA uses ktup = 2 for amino acids and ktup = 6 for nucleic acids. Larger values of ktup increase *speed* (because fewer hot-spots have to be considered in subsequent steps) and *precision*, but decrease *recall*.

3.6 Precision and recall

Precision and recall are concepts from information retrieval and can be defined as follows:

- *Precision:* The probability that a retrieved item is correct.

$$\text{precision} = \frac{TP}{TP + FP}$$

- *Recall:* The probability that a correct item is retrieved.

$$\text{recall} = \frac{TP}{TP + FN}$$

(where TP is "true positives", etc.). A perfect precision score of 1.0 means that every result retrieved by a search was relevant (but says nothing about whether all relevant items were retrieved) whereas a perfect recall score of 1.0 means that all relevant items were retrieved by the search (but says nothing about how many irrelevant items were also retrieved).

3.7 Diagonal runs

If there are similar regions, they will show up in the dot plot as a number of hot-spots which are roughly on a diagonal line. FastA gains efficiency by relying on the heuristic assumption that a local alignment will contain several hot-spots along the same diagonal line.

Definition. A *diagonal run* is a set of hot-spots $\{(i_1, j_1), \ldots, (i_\ell, j_\ell)\}$ such that $i_p - j_p = i_q - j_q$ for all $1 \leq p, q \leq \ell$. (They are on *exactly* the same diagonal.)

A diagonal can contain more than one diagonal run, and a diagonal run need not contain all hot-spots on the same diagonal.

Each diagonal run is assigned a *score*. Hot-spots score positive. The gaps between them score negative, depending on their length. At this stage, FastA does not yet use a substitution matrix; it just considers whether the sequence positions are identical or not.

For each diagonal, the hot-spots on it are joined by FastA into diagonal runs.

The crucial observation is that diagonal runs can be found without looking at the whole matrix. FastA does not create a dot plot!

Since *ktup* is small, we can *hash* all positions i of x into the *bins* of a lookup-table, using the substring $x[i \ldots i + k - 1]$ as a *key*. (There are $20^2 = 400$ bins for AA and $4^6 = 4096$ bins for NA, using the default values for ktup.)

Collisions in the hash table can be resolved, e.g., by linked lists. We can fill the hash table in a single pass over x. This way the collision lists are already in sorted order.

Example. We use a two-letter alphabet $\{a, b\}$ and ktup = 2 for simplicity.

sequence x
aababbaaab
0123456789

$\longrightarrow$

key	collision_list
aa	0, 6, 7
ab	1, 3, 8
ba	2, 5
bb	4

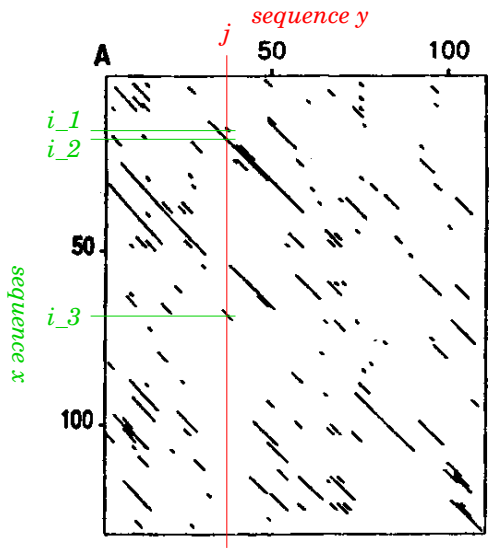
Using the hash we can compute the diagonal runs as we walk along sequence y:

When we are at position j of y, we look at the positions $i_1, i_2, \ldots$ of x which we find in the hash bin indexed by $y[j \ldots j + k - 1]$. Each pair $(i_1, j), (i_2, j), \ldots$ is a hot-spot, and we can update the corresponding diagonal runs (and their scores) accordingly.

Two hot-spots $(i, j), (i', j')$ are on the same diagonal iff $i - j = i' - j'$. Hence the "growing diagonal runs" (along the diagonals) can be adressed by an array of size $m + n - 1$.

Thus the time needed to compute the diagonal runs is proportional to the number of hot-spots—which is usually much smaller than $|x| \cdot |y|$.

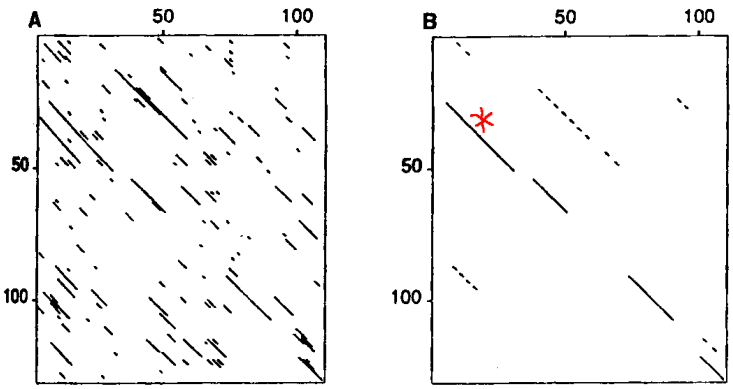
Figure A shows the diagonal runs and the hot-spots which are found using the hash table.



3.8 Rescoring the best diagonal runs

Next, the 10 highest-scoring diagonal runs are identified and *rescored* using a *substitution matrix*. Note that thus far, FastA still does not allow any indels in the alignment; it just counts matches and mismatches.

Definition. The best diagonal run is called *INIT1*. It is marked with a * in Figure B below.



3.9 Chaining of initial regions

To account for indels, FastA tries to *chain* several initial regions into a better alignment.

Definition. The resulting alignment is called *INITN*.

The chaining problem can be modeled as a purely graph theoretical problem. . .

3.10 The chaining problem

Chaining problem or **heaviest-path-in-DAG problem**
Find a *heaviest path* in a *directed acyclic graph* $G = (V, E)$ with *weights* $w : V \cup E \to \mathbb{Z}$ on the edges and vertices.

Two diagonal runs can be *chained* iff they do not overlap in one of the sequences. This is the case iff the lower right end of one diagonal run is to the left and top of the upper left end of the other one.

Formally, let

$$r = \big([i \dots i+p-1], [j \dots j+p-1]\big)$$

and

$$r' = \big([i' \dots i'+q-1], [j' \dots j'+q-1]\big)$$

be diagonal runs, then we define

$$r < r' \quad :\Leftrightarrow \quad i+p-1 < i' \quad \wedge \quad j+p-1 < j' \,.$$

Let $G = (V, E)$ be the directed graph whose vertex set V are all diagonal runs and whose edge set is $E := \{(r, r') \mid r < r'\}$.

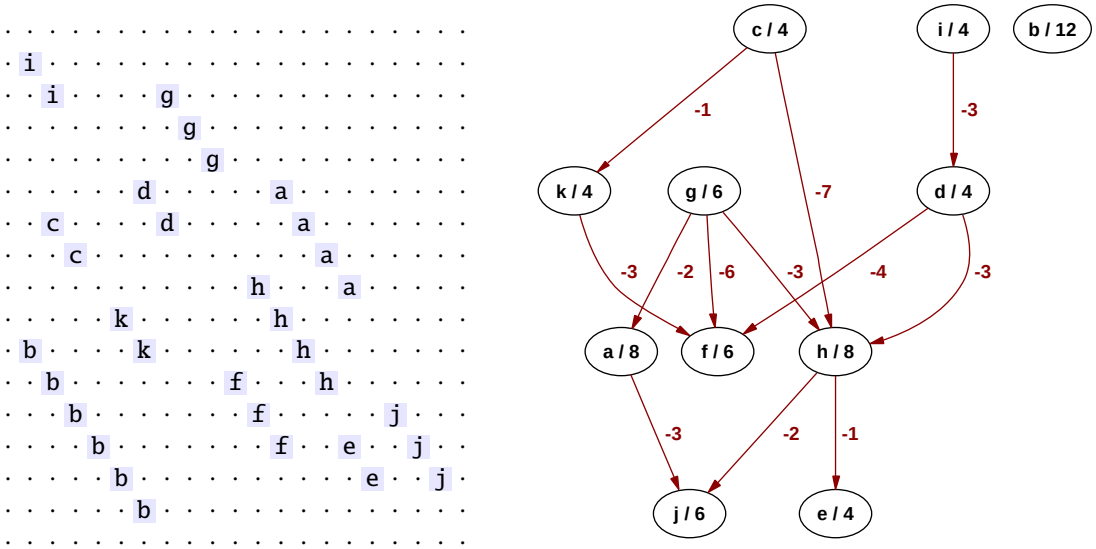
Clearly, G is acyclic, so we can find a topological ordering of G in $O(|V| + |E|)$ time. That is, we can write

$$V = \{r_1, \dots, r_N\} \quad \text{such that} \quad r_i < r_j \Rightarrow i < j \,.$$

Moreover, FastA assigns a score $s(r) > 0$ to every diagonal run $r \in V$ and a score $s(e) < 0$ to every edge $e \in E$.

Note that Dijkstra's algorithm is not applicable, because we have negative edge weights. However, we can use that the graph is acyclic.

Example. For simplicity, each "letter" scores 2 and all other columns of the (resulting) alignment score −1 (diagonal "shortcuts" allowed).



Formally, we have to compute a path

$$r_{\pi(1)} < r_{\pi(2)} < \ldots < r_{\pi(l)}$$

that maximizes

$$\sum_{i=1}^{l-1} \Big(s(r_{\pi(i)}) + s(r_{\pi(i)}, r_{\pi(i+1)}) \Big) + s(r_{\pi(l)}) \ .$$

In the following, we show how to use the algorithm **DAG_shortest_paths** for this task.

```
DAG_shortest_paths(G, w, s, d, π)
{
G = (V, E) : directed acyclic graph;
Adj : V → 𝒫(V) :  set of successor nodes, this implements E;
w : E → ℤ : edge weights;
d : V → ℤ : distance from s as computed by algorithm;
π : V → V : predecessor;
V = {s = v_1, v_2, ..., v_n} : topological ordering starting with s.
d[s] ← 0;
π[s] ← nil;
for i = 1 to n do
    foreach ( v ∈ Adj(v_i))do
        c = d[v_i] + w(v_i, v);
        if (c < d[v])
            d[v] ← c;
            π[v] ← v_i;
        fi
    od
od
}
```

Theorem
Let $G = (V, E)$ be a directed acyclic graph, let $s \in V$ be a vertex without predecessors and let $w : E \to \mathbb{R}$ be a weight function on the edges (which may take negative values). Let $v_1, \ldots, v_n$ be the topological ordering used by Algorithm DAG_shortest_paths.

Then the following holds: After the loop for v_i has been run through, $d[v_i]$ equals $\mathrm{dist}_w(s, v_i)$, the weight of a shortest path from s to v_i, and one such path is given by $s, \ldots, \pi(\pi(v_i)), \pi(v_i), v_i$.

Proof.
We apply induction on $i = 1, \ldots, n$. For $v_i = s$ there is nothing to be shown. Next we derive two inequalities from which together the claim will follow.

- Since backtracking from v_i yields *a* path $s, \ldots, \pi(\pi(v_i)), \pi(v_i), v_i$ of length $d[v_i]$, we have $d[v_i] \geq \mathrm{dist}_w(s, v_i)$.
- Now let $P = (s = u_0, u_1, \ldots, u_k = v_i)$ be *a shortest* path. Since u_{k-1} is a predecessor of v_i, and the loop runs through the vertices in a topological ordering, u_{k-1} was processed before v_i and by our inductive assumption, $d[u_{k-1}] = \mathrm{dist}_w(s, u_{k-1})$.

  Due to the relaxation step we have $d[v_i] = d[u_k] \leq d[u_{k-1}] + w(u_{k-1}, u_k) = \mathrm{dist}_w(s, u_{k-1}) + w(u_{k-1}, u_k) = \mathrm{dist}_w(s, v_i)$.

By combination of these inequalities, it follows that $d[v_i] = \mathrm{dist}_w(s, v_i)$. ■

3.11 Reduction of the problem

The chaining problem in FastA appears in a slightly different form from that expected by the DAG_shortest_path algorithm, so we need to transform the input before we can apply it.

There are three issues:

1. Allow *any* vertex to play the role of s and t
2. Eliminate the vertex weights
3. Reverse the optimization goal

Assume that $G = (V, E)$, $w : V \cup E \to \mathbb{Z}$ is an instance of the heaviest-path-in-DAG problem.

Arbitrary source and target.

- Add a new vertex s with outgoing edges to all other vertices.
  Let $w(s, x) = 0$, $w(s) = 0$.
- Add a new vertex t with incoming edges from all other vertices.
  Let $w(x, t) = 0$, $w(t) = 0$.

Eliminating the vertex weights.

- Replace $w(u, v)$ with $w'(u, v) := w(u, v) + w(v)$, for all edges (u, v).

Then any path $s = v_1, v_2, \ldots, v_k = t$ has weight

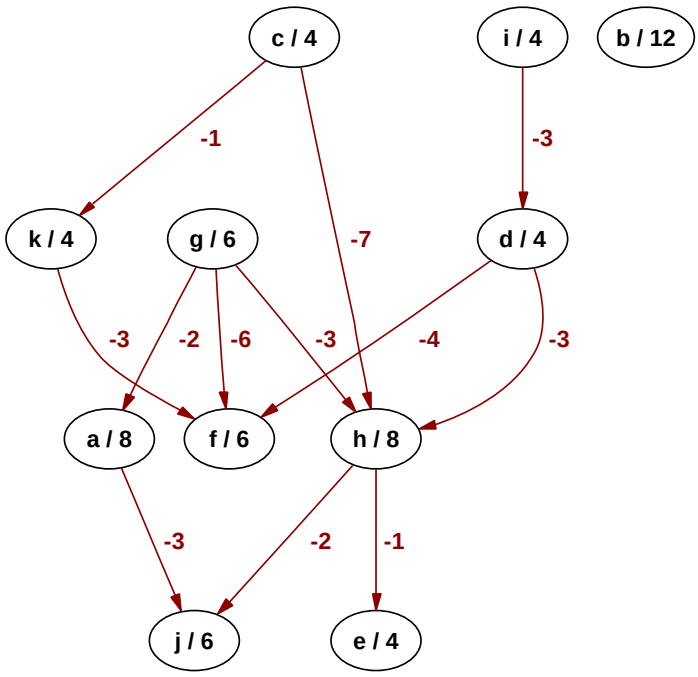
$$w(v_1) + w(v_1, v_2) + w(v_2) + w(v_2, v_3) + \ldots + w(v_k)$$

in both problems.

Reversing the optimization goal.

- Replace w' with $w'' := -w'$.
- Replace "max" with "min".

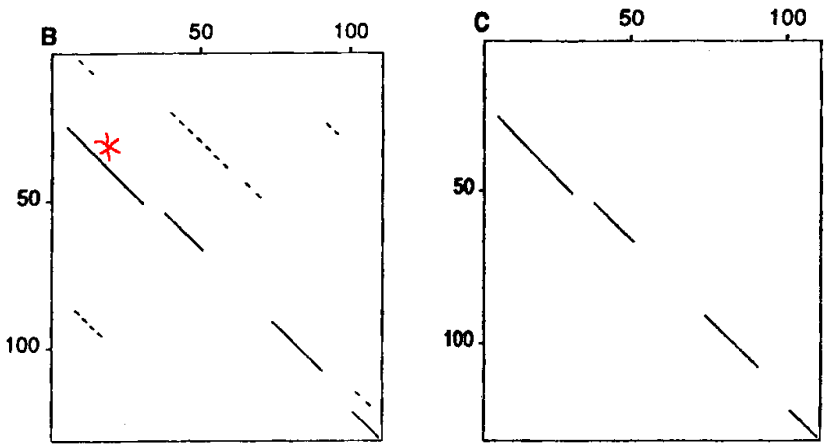
Example. (Finished on blackboard)



To illustrate the idea, consider the path $c \to k \to f$.

										total weight:
path before reduction:			c	$\to$	k	$\to$	f			
weights:			4	-1	4	-3	6			10
path after reduction:	s	$\to$	c	$\to$	k	$\to$	f	$\to$	t	
weights:		-4		-3		-3		0		-10

Figure C shows the result (INITN) of chaining the initial regions in Figure B.



3.12 Banded alignment

Apart from solving the chaining problem, FastA makes another heuristic attempt to improve INIT1.

Definition. A *banded alignment* is an alignment that stays within a certain diagonal "band" of the DP matrix.

FastA computes a banded alignment around the diagonal containing INIT1.

Definition. This alignment is called *OPT*.

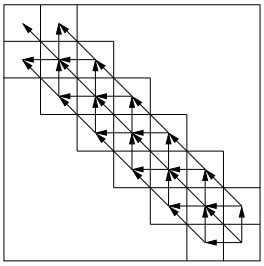
Note that OPT is not necessarily optimal! We use the (confusing) terminology from the original paper. %-)

To compute a banded alignment, we simply ignore entries of the DP matrix which are outside the band. Of course, the recurrence has to be modified in the obvious way for cells at the boundary of the band.

This reduces running time and space requirement.

A band can be specified by an interval I and the condition that computed entries (i, j) satisfy $i - j \in I$.

An example with $I = \{-1, 0, 1\}$:



3.13 Final application of Smith-Waterman

As is usual practice with heuristic alignment programs, FastA computes an exact local alignment using the Smith-Waterman algorithm (or some variant thereof) before it displays the final result. This is possible in reasonable time, because the search has been narrowed down to a small region.

3.14 Concluding remarks on FastA

- FastA is actually *a collection of programs* which are customized for various tasks (DNA, proteins, peptide collections, mixed input types).
- If FastA is used to search through a large database, one will often find a relatively good hit *simply by chance*. Therefore, FastA also computes a *statistical significance* score. This information is very important to assess the results—however, the underlying theory is beyond this lecture. . .