Local Sequence Alignment
FASTA

Local Sequence Alignment Against a Database Problem

Local Sequence Algignment. The local sequence alignment problem is defined as follows:

> *Given two strings $S = s_1 \ldots s_n$ and $T = t_1 \ldots t_m$, a substitution matrix $Score$ and an insertion/deletion penalty δ, find a pair of substrings $s_i \ldots s_{i+k}$ of S and $t_j \ldots t_{j+l}$ of T that have the best overall alignment score, and return the best alignment for them.*

Local Sequence Algignment against a database. The local sequence alignment against a database problem extends the local sequence alignment problem by introducing multiple strings against which a single *query string* is a aligned:

> *Given a query string $S = s_1 \ldots s_n$, and a collection of strings $D = \{D_1, \ldots D_M\}$ (usually referred to as a sequence database), a substitution matrix $Score$ and an insertion/deletion penalty δ, find the best local alignments of S with any/all strings from D.*

Note. Smith-Waterman algorithm for local sequence alignment has runtime complexity $O(nm)$. When extended to a database D of M strings of average length m, the runtime complexity of applying Smith-Waterman to each pair of strings S, D_j becomes $O(nmM)$. This is unacceptable for practical applications.

Idea. Smith-Waterman algorithm guarantees the correct solution - i.e., always finds the best local alignment between a pair of strings. We can *sacrifice the accuracy* in lieu of *efficiency* by requiring our algorithm to find *pretty good* local alignments against a database *fast*.

Approximation algorithms. We consider two approximation algorithms: FASTA and BLAST. Both algorithms are faster than Smith-Waterman. Both do not guarantee that they return the best possible alignments. However, for both algorithms, the probability of an error is computable, and can be kept to be very small.

FASTA

FASTA [1] compares a query sequence with each sequence from a database in search of local alignments using a five-step process described below.

- **Step 1.Dot-plot construction.** Determine occurrences of all subsequences of length k (k-tuples) (for some given k) in the query string S and in each string D_1 through D_M. (Construct implicit k-tuple dot-plots of S with all D_is).
- **Step 2. Diagonal scoring.** For each pair of strings S and $D_i \in D$, score the diagonals of the dot-plot matrix for total number of matching positions. Select the best diagonals.
- **Step 3. Rescoring of the diagonals.** For each diagonal selected in **Step 2**, use the substitution matrix $Score$ to rescore them. Identify subregions of S and D_is with the highest scores.
- **Step 4. Region Join.** Using a given gap penalty scoring process, join selected diagonals using gaps.
- **Step 5. Local algignment.** For each region produced in **Step 4**, perform a full local alignment search using Smith-Waterman algorithm.

Step 1. Dot-plot Construction

k-tuple dot plot. Given two strings $S = s_1 \dots s_n$ and $T = t_1 \dots t_m$, and a number $k < \min(n, m)$, a k-tuple dot plot of S and T, denoted $P_k(S, T)$ is a function

$$P_k : [1..n - k + 1] \times [1..m - k + 1] \longrightarrow \{0, 1\},$$

such that

$$P_k(i, j) = 1, \textbf{iff} s_i \dots s_{i+k-1} = t_j \dots t_{j+k-1},$$

i.e., $P_k(S, T)$ *marks all locations* in S and T, where the two strings have an exact match of length k.

Naïve k-tuple dot plot construction. Given two strings, $S = s_1 \dots s_n$ and $T = t_1 \dots t_m$, a naïve straightforward way of constructing their k-tuple dot plot is to compare every k-tuple from S to every k-tuple from T. Assuming $k << \min(n, m)$, this procedure requires $O(nm)$ time.

Efficient, implicit k-tuple dot plot construction. We observe two things:

- k-tuple dot plots of two strings is not expected to have too many entries, i.e., it is a sparse matrix. We do not need to instantiate the matrix explicitly.
- What we really need is to know *which k-tuples are found in each string* and *where they are found.*

- A string of length n contains $n-k+1$ k-tuples, starting at positions $1, 2, \ldots n-k+1$.
- Given an alphabet $\Sigma = \{a_1 \ldots a_K\}$, there are $|Sigma|^k = K^k$ *different possible* k-tuples. If Σ is fixed, and k is kept constant, then K^k is a constant.

Combining these three observations, we obtain the following efficient procedure for *implicitly* constructing the dot plot of two strings:

1. Construct a k-tuple lookup table $L_S[]$ for string S and another one, $L_T[]$ for string T. Enumerate all possible k-tuples in alphabet Σ. $L_S[]$, $L_T[]$ will have indexes $1 \ldots K^k$ - each index representing a specific k-tuple in alphabet Σ.

   Given a k-tuple $V = v_1 \ldots v_k$, $L_S[V]$, a.k.a. $L_S[index(V)]$ will contain the list of positions in string S, where V is found. Similarly, $L_T[V]$, a.k.a. $L_T[index(V)]$ will contain the list of positions in T, where V is found.
2. Fill the lookup table $L_S[]$ by scanning S and adding i to $L_S[s_i \ldots s_{i+k-1}]$ on each step.
3. Fill the lookup table $L_T[]$ in a similar way.

Example. Let $S = ATCGTATCG$ and let $k = 3$, and $\Sigma = \{A, T, C, G\}$. There are 4^3 possible 3-tuples in Σ: `AAA`, `AAC`,...,`TTT`. S contains the following seven 3-tuples in it:

```
    ATCGTATCG
1   ATC
2    TCG
3     CGT
4      GTA
5       TAT
6        ATC
7         TCG
```

The lookup table $L_S[]$ looks as follows (we show only non-empty entries):

```
L[ATC] = {1,6}
L[CGT] = {3}
L[GTA] = {4}
L[TAT] = {5}
L[TCG] = {2,7}
```

k-tuples in FASTA. FASTA uses $k = 6$ for the alphabet of nucleotides and $k = 2$ for the alphabet of amino acids. As such, the lookup tables have the following sizes:

Alphabet	Number of characters	Typical k	Size of lookup table
Nucleotide	4	6	$4^6 = 4096$
Amino Acid	21 (stop codon)	2	$21^2 = 441$

Step 2. Diagonal scoring

Idea. An alignment between two substrings $s_i \dots s_{i+l-1}$ and $t_j \dots t_{j+l-1}$ (of the same length l), is *represented by a diagonal* on the dot plot $P_k(S,T)$, that passes through the cells $[i,j], [i+1,j+1], \dots [i+l-1]$.

The more matches are found on the diagonal, the closer the alignment will be.

Example. Let $S = ATCGTATCG$ and $T = CAGATCGTCTCGAT$ and $k = 3$. The (explicit) dot plot $P[S,T]$ can be represented as follows:

```
  CAGATCGTCTCGAT
  --------------
A|   *
T|    *    *
C|     *
G|
T|
A|   *
T|    *    *
C|
G|
```

The diagonal originating at $[1,4]$ contains four total matches on it: $[1,4], [2,5], [3,6], [7,10]$. This corresponds to the following alignment between the substrings in S and T:

```
   ATCGTATCG
   ||||| |||
cagATCGTCTCGat
```

Step 2 of FASTA is computing the total number of matches on each diagonal.

Number of diagonals. The dot plot $P_k[S,T]$ for strings S of length n and T of length m contains $n + m - 1$ diagonals.

Computing diagonal scors. Naïve way: go through the $P_k[S,T]$ matrix. Runtime: $O(nm)$.

A better way uses the lists $L_S[]$ and $L_T[]$ constructed on **Step 1** of FASTA.

Idea. Initialize all diagonal scores to 0. Traverse through non-empty entries in $L_S[]$. For each k-tuple V occurring in S, check if it occurs in T by accessing $L_T[V]$. For each pair of positions i, j, where there is a match, *determine which diagonal it is on* and update the diagonal score.

Pseudocode. The pseudocode for this algorithm is shown below.

```
Algorithm DiagonalScores(S, T, L_S[], L_T[], k)
begin
  declare Diag[−n..m];        // array of diagonal scores
  for i = −n to m do
    Diag[i] := 0;        // initialize all diagonal scores to 0
  end for
  for each k such that L_S[k] ≠ ∅  and  L_T[k] ≠ ∅ do
                    // find all matches in the dot plot
    for each i ∈ L_S[k] do
      for each j ∈ L_S[k] do
        d := i − j;        // determine the diagonal for the match
        Diag[d] := Diag[d] + 1;       // update the diagonal score
      end for
    end for
  end for
  return Diag[];
end
```

Example. Consider the pair of strings $S = ATCGTATCG$ and $T = CAGATCGTCTCGAT$. The dot plot matrix $P_k[S,T]$ contains $9 + 14 - 1 = 22$ diagonals, from -14 to $+9$, although diagonals -13, -14, $+8$ and $+9$ can be excluded for $k = 3$.

The intersection of $L_S[]$ and $L_T[]$ lookup tables looks as follows:

3-tuple	L_S	L_T	Diagonals
ATC	$\{1, 6\}$	$\{4\}$	$1 - 4 = -3$; $6 - 4 = +2$
TCG	$\{2, 7\}$	$\{5, 10\}$	$2 - 5 = -3$; $2 - 10 = -8$; $7 - 5 = +2$; $7 - 10 = -3$
CGT	$\{3\}$,	$\{6\}$	$3 - 6 = -3$

The Diagonals column in the table above shows to which diagonal scores each dot plot match contributes. Using this table we can construct the diagonal scores table for this example (the table below shows only non-zero scores):

d	$Diag[d]$
-8	1
-3	3
$+2$	2

Filtering. **Step 2** of FASTA returns only **significant diagonals**, which are defined as diagonals on which the score **exceeds the expected score by at least two standard deviations**. If there are many significant diagonals in $P_k[S,T]$, FASTA further filters them by taking only the **top 10 significant diagonals** with the best diagonal scores.

Running time. The worst-case scenario for this step is $O(nm)$ (consider matching `AAAA...AAAA` vs. `AAA...AAAA`). However, since most DNA sequences show large divergence of occurrences of k-tuples, in practice, this step runs in almost linear time.

Step 3. Rescoring Diagonals

Step 3. On **Step 3** of FASTA, the significant diagonals selected on **Step 2** are **rescored** using the given substitution matrix.

This is a straightforward procedure:

- Each diagonal is responsible for a specific **gapless** alignment of subsequences in S and T.
- This alignment is scored using the substitution matrix provided as input to FASTA.

Example. Consider a substitution matrix $Score[]$ on the alphabet of nucleotides, such that $Score[\alpha, \alpha] = 5$ and $Score[\alpha, \beta] = -4$ for $\alpha \neq \beta$.

From the example above, consider two significant diagonals from the dot plot of strings $S = ATCGTATCG$ and $T = CAGATCGTCTCGAT$: diagonals -3 (with Step 2 score of 3) and $+2$ (with Step 2 score of 2).

Diagonal -3 yields the following alignment of S and T:

```
   ATCGTATCG
   ||||| |||
cagATCGTCTCGat
```

Scoring this alignment starting with the first algined character (`A`) and ending on the last aligned character (`G`), yields the score: $5+5+5+5+5+5-4+5+5+5 = 36$.

Diagonal $+2$ yields the following alignment of S and T:

```
atcgtATCG
     ||||
  cagATCGtctcgat
```

The score of this alignment is $5 + 5 + 5 + 5 = 20$.

Thresholding. FASTA retains the diagonals (regions) with the best scores.

Step 4. Region Join

Adding Gaps. On **Step 3** of FASTA good **ungapped** local alignments are found. However, sometimes, the best local alignments must include a gap.

Gapped alignments correspond to **joins of two or more** diagonal fragments from the dot plot. On **Step 4**, the gapped alignments between different diagonals are considered.

Gap penalties. There are two ways to approach gap penalties in local alignments:

- **Global alignment-style gap penalty.** A penalty of δ is assessed on every single insertion/deletion. A gap of length h has a gap penalty of $h \cdot \delta$.
- **Dampened gap penalties.** In practice, it may make sense to penalize longer gap a bit less stringently. A traditional gap penalty scheme used in local alignment methods is to have two values: δ and ϵ. Any time a gap is introduced, a penalty of $\delta + \epsilon$ is assessed. Extending the gap by a single

character, yields a penalty of ϵ. Thus a gap of length h has a gap penalty of $\delta + h \cdot \epsilon$.

In practice, $\epsilon < \delta$, i.e., the add-ons for extending the gap are smaller than the penalty for having it in first place.

Dampened gap penalties explained. Essentially, dampened gap penalties favor alignments that have fewer, but possibly longer, gaps over alignments that have more smaller gaps. Thus, this alignment:

```
ATAAAGAGAGA
||   ||||
AT___GAGA
```

is preferred to

```
AT_A_AAGAGAGA
|| | |
ATGAGA
```

despite the latter alignment having fewer misalinged characters.

The idea is that the top sequence is considered to originate from the bottom one with a single insertion of a codon `AAA` after the first two characters (`AT`).

Step 4. Given a theshold g on the size of the gap, for each diagonal l remaining from **Step 3**:

- for each diagonal $l-g, l-g+1, \ldots, l-1, l+1, \ldots l+g$, join l with it.
- score the resulting local alignment using the substitution matrix and the gap penalty.
- if any joined region improves the alignment score of l, replace l with it for **Step 5**.

Step 5. Smith-Waterman

Step 5. For each region (a single diagonal, or possibly a merger of two diagonals constructed on **Step 4**) find the best local alignment using the Smith-Waterman algorithm.

Efficiency

For a pair of strings $S = s_1 \ldots s_n$ and $T = t_1 \ldots t_n$:

Step 1. **Step 1** runs in $O(n + m + |\Sigma|^k$ steps. Assuming a constant alphabet, and a constant predefined k, this step is linear.

Step 2. Step 2, as observed above, has $O(nm)$ worst case complexity. However, under normal circumstances, it will run in linear or almost linear time.

Step 3. Assuming that only a constant number of diagonals is selected on **Step 2**, **Step 3** can run in $O(n + m)$ time.

Step 4. Assuming that only a constant number of diagonals is carried over from **Step 3**, **Step 4** can run in $O(m + n)$ time.

Step 5. This step takes, in worst case, $O(nm)$ time. Given a constant number l of regions to align, with average region lengths of x and y for strings S and T, the overall running time will be $O(lxy)$. If l is a constant, then this is $O(xy)$. If both x and y are significantly smaller than n and m, **Step 5** will run faster than the Smith-Waterman algorithm applied to the entirety of S and T.

Weaknesses of FASTA

Insensitivity to local alignments with short matches. FASTA will not find the best local alignment between two strings if that alignment does not contain at least one k-tuple match.

Insensitivity to large gaps. If the best local alignment has a gap that is bigger than the gap threshold g used on **Step 4**, then FASTA will not find the best local alignment.

Example 1. Consider the following two strings:

```
 CCCCCAATAATAATAATAAT
      || || || || ||
      AAGAAGAAGAAGAACCC
```

If $k = 3$, then this alignment won't be discovered. Instead, FASTA will find:

```
             CCCCCAATAATAATAATAAT
             |||
AAGAAGAAGAAGAACCC
```

Example 2. Consider the following two strings:

```
ATGTCCCCCTGAT
||||     |||
ATGT_____TGA
```

The best local alignment has a gap of 5. If $g = 4$, then FASTA will only find the alignment of the ATGT fragments of the two strings.

References

[1] Wilbur, W. J.; Lipman, D. J. (1983). Rapid similarity searches of nucleic acid and protein data banks, in *Proceedings of the National Academy of Sciences of the United States of America*, Vol. 80 (3), pages 726 — 730.