SEQUENCE ALIGNMENT ALGORITHMS

Why compare sequences?

- Reconstructing long sequences from overlapping sequence fragment
- Searching databases for related sequences and subsequences
- Storing, retrieving and comparing sequences in databases
- Exploring evolutionary relationships among species
- Finding informative elements in sequences
- Comparing sequences for similarities

Biological Motivations Lots of sequences with unknown structure/function. A few sequences with known structure/function

- If they align, they are similar, maybe due to common descent
- If they are similar, then they may have same structure/function
- If one of them has known structure/function, then alignment to the other yields insight about how structure/function works.

Similarity and Difference

All contemporary genetic material have one common ancestral DNA

Differences between families of contemporary species are due to *local mutations* during the course of evolution

Local mutations :

- *Insertion* of a base or several bases to the sequence
- *Deletion* of a base or more from the sequence
- *Substitution* replacing a sequence base by another

Insertion and deletion are reverse of each other. They are called *indel*

Distance given two sequences: the minimal sum of weights for a set of mutations transforming one into the other.

Similarity given two sequences: the maximal sum of weights corresponding to resemblance of the two sequences

Simplest Model: Edit Distance

Definition The minimal number of edit operations (insertions, deletions and substitutions) needed to transform one sequence into the other. Edit distance can be used to roughly measure the number of DNA replications that occurred between two DNA sequences

Example Given `a c c t g a` and `a g c t a`, the minimal number of edit operations required to transform one into the other is 2:

```
a c c t g a
a g c t g a
a g c t a

a g c t g a
a g c t - a
```

Remark The definition of edit distance implies that all operations are done on one sequence only and the representation shown above might make false impression that the order of the changes is known

What is Alignment?

An alignment of two sequences S and T is obtained by first inserting chosen spaces, either into, at the ends of or before S and T, and then placing the two resulting sequences one above the other so that every character or space in either sequence opposite a unique character or a unique space in the other sequence

A *scoring function* is used to evaluate the *goodness* of the alignment

An algorithm searches for the best alignment, representing the *minimal difference* or the *maximum similarity* between S and T

Biological models consider the significance of each mutation and score the alignment accordingly. The alignment distance can be used to estimate the *biological difference* of two sequences

Example The aligned sequences

```
SEQ 1 GTAGTACAGCT-CAGTTGGGATCACAGGCTTCT

SEQ 2 GTAGAACGGCTTCAGTTG---TCACAGCGTTC-
```

Distance 1: match 0, substitution 1, indel 2 ⇒dis=14
Distance 2: match 0, d(A,T)=d(G,C)=1, d(A,G)=1.5, indel 2, ⇒dis=14.5

Similarity: match 1, substitution 0, indel -1.5 ⇒sim=16.5

Global Pairwise Alignment

Given two sequences S and T of roughly the same length. What is the maximum similarity between them? Find an optimal alignment. The term *global* emphasizes that for each sequence, the entire sequence is involved.

Naive approach Generate all possible alignments and then pick the best one (the one that with maximum similarity value. Very slow algorithm since the number of alignments between two sequence is exponential, that is $O(2^{2n})$ for length n. Below is faster method

Dynamic Programming consists of solving an instance of a problem by taking advantage of already computed solutions for smaller instance of the same problem. Given S and T, instead of determining the similarity between S and T as whole sequences only, we build up the solution by determining all similarities between arbitrary *prefixes* of the two sequences. We start with the shorter prefixes and use previously computed results to solve the problem for larger prefixes

A prefix of S is any substring of S of the form $S_{1\ldots j}$ for $0 \leq j \leq |S|$. We admit $j = 0$ and define $S_{1\ldots 0}$ as being the empty string, which is a prefix of S as well.

Global Alignment by Dynamic Programming

Given S and T, with $|S| = n$ and $|T| = m$
(i.e. $S = S_{1\ldots n}$ and $T = T_{1\ldots m}$).

Let σ be the scoring function. We will use
$\sigma(a,a) = +1$, for a match
$\sigma(a,b) = -1$, for a substitution
$\sigma(a,-) = \sigma(-,a) = -2$, for an indel

Define $V(i,j)$ as the similarity value of an optimal alignment of $S_{1\ldots i}$ and $T_{1\ldots j}$

Problem is to compute $V(n,m)$ by dynamic programming

Base cases: given $i,j > 0$ we have

$$\begin{aligned} V(0,0) &= 0 \\ V(i,0) &= V(i-1,0) + \sigma(S_i,-) = \textstyle\sum_{k=0}^{k=i} \sigma(S_k,-) \\ V(0,j) &= V(0,j-1) + \sigma(-,T_j) = \textstyle\sum_{k=0}^{k=j} \sigma(-,T_k) \end{aligned}$$

The basis for $V(i,0)$ says that if i characters of S are to be aligned with 0 characters of T, then they must all be matched with spaces. The basis for $V(0,j)$ is analogous

Global Alignment by Dynamic Programming

Let us consider an optimal alignment of the prefixes $S_{1\ldots i}$ and $T_{1\ldots j}, i, j > 0$. In particular, observe the last aligned pair of characters in such an alignment. This last pair must be one of the following three cases

1. $(S_i, -)$: The score in this case is the score $\sigma(S_i, -)$ of aligning S_i with a space plus the score $V(i-1, j)$ of aligning the prefixes $S_{1\ldots i-1}$ and $T_{1\ldots j}$

2. (S_i, T_j): The score in this case is the score $\sigma(S_i, T_j)$ of aligning S_i with T_j plus the score $V(i-1, j-1)$ of aligning the prefixes $S_{1\ldots i-1}$ and $T_{1\ldots j-1}$

3. $(-, T_j)$: The score in this case is the score $\sigma(-, T_j)$ of aligning a space with a T_j plus the score $V(i, j-1)$ of aligning the prefixes $S_{1\ldots i}$ and $T_{1\ldots j-1}$

The optimal alignment of $S_{1\ldots i}$ with $T_{1\ldots j}$ chooses whichever among these three possibilities has the greatest similarity value

To summarize, we have the following recurrence relation for $i, j > 0$

$$V(i,j) = \max \begin{cases} V(i-1,j) & + \ \sigma(S_i, -) \\ V(i-1,j-1) & + \ \sigma(S_i, T_j) \\ V(i,j-1) & + \ \sigma(-, T_j) \end{cases}$$

Tabular Computation of Optimal Alignment

Let $|S| = n$ and $|T| = m$. There are $n+1$ and $m+1$ prefixes of S and T, respectively (including the empty string). We can arrange the calculations in an $(n+1) \times (m+1)$ array V where entry $V(i,j)$ contains the similarity between $S_{1\ldots i}$ and $T_{1\ldots j}$

We compute $V(i,j)$ for all possible values of i and j. We start from smaller i,j and increasing them, filling in the table in a row-wise (or column-wise) manner. Finally, $V(n,m)$ is the required maximum similarity between S and T.

```
Algorithm:  Similarity
  Input:  $\sigma$, $S$ and $T$
  Output:  similarity between $S$ and $T$
  For $i = 0$ to $n$ do
    For $j = 0$ to $m$ do
    begin
        $V[i,j] \leftarrow \max \begin{cases} V[i-1,j] & + \quad \sigma[S_i,-] \\ V[i-1,j-1] & + \quad \sigma[S_i,T_j] \\ V[i,j-1] & + \quad \sigma[-,T_j] \end{cases}$
    end
    Return $V[n,m]$
```

Time complexity is $O(nm)$

Space complexity is $O(nm)$

Complexity is quadratic in both time and space

Recovering the Optimal Alignments

We know how to compute the similarity between S and T. But how to determine the optimal alignment itself, and not just its similarity value?

Solution: Retrace/Backtrack `Algorithm Similarity` from entry $V[n, m]$ back to entry $V[0, 0]$, determining which entries were responsible for the current one. Given matrix V, an optimal alignment can be constructed from right to left.

```
Recursive Algorithm:  Align
  Input:  indices $i$, $j$, $\sigma$ and matrix $V$ of Algorithm Similarity
  Output:  optimal alignment between $S$ and $T$ in align-S, align-T
  If $i = 0$ and $j = 0$ then
    $l \leftarrow 0$
  Else
    if $i > 0$ and $V[i, j] = V[i-1, j] + \sigma(S_i, -)$ then
      Align$(i-1, j, l)$
      $l \leftarrow l + 1$
      align-S$[l] \leftarrow S[i]$
      align-T$[l] \leftarrow -$
    Else
      if $i > 0$ and $j > 0$ and $V[i, j] = V[i-1, j-1] + \sigma(S_i, T_j)$ then
        Align$(i-1, j-1, l)$
        $l \leftarrow l + 1$
        align-S$[l] \leftarrow S[i]$
        align-T$[l] \leftarrow -T[j]$
      Else // has to be $j > 0$ and $V[i, j] = V[i, j-1] + \sigma(-, T_j)$
        Align$(i, j-1, l)$
        $l \leftarrow l + 1$
        align-S$[l] \leftarrow -$
        align-T$[l] \leftarrow T[j]$
```

Time complexity is $O(n + m)$

Space complexity is $O(nm)$

Complexity is linear in time

Local Pairwise Alignment

Given two sequences S and T. What is the maximum similarity between a substring of S and a substring of T. Find most similar substrings

In many applications, two sequences may not be highly similar as a whole, but may contain sections with high resemblance. Some biological motivations:

- Ignore stretches of non-coding DNA: introns are more likely to accumulate mutations than exons. When searching for a *local alignment* between two stretches of DNA (from 2 different species), finding a best match between is likely to be between two exons
- Protein domains: proteins of different kind/species, often exhibit local similarities called *homeoboxes*. These homeoboxes can be found by local alignment.

Example Consider the two sequences

```
S = g g t c t g a g
T = a a a c g a
```

If match = 2 and indel/substitution = -1, then the best local alignment is

```
c t g a (∈ S)
c - g a (∈ T)
```

Computing Local Pairwise Alignment

Naive approach Align by, dynamic programming, every substring of S with every substring of T and then pick the alignment that yields the maximum similarity. Time complexity is $O(n^3m^3)$ and hence such approach is too slow

Setup for local alignment by dynamic programming

1. The *empty string* λ is the string with $|\lambda| = 0$
2. U is a *prefix* of S iff $U = S_{1\ldots k}$ or $U = \lambda$, for some $1 \leq k \leq n$
3. U is a *suffix* of S iff $U = S_{k\ldots n}$ or $U = \lambda$, for some $1 \leq k \leq n$
4. $V(i,j)$ denotes the similarity value of an optimal (global) alignment of α and β over all suffixes α of $S_{1\ldots i}$ and all suffixes β of $T_{1\ldots j}$

Dynamic Programming Fill in a table with the values of $V(i,j)$, with increasing i,j. That is, each entry $V(i,j)$ holds the highest score of an alignment between a suffix of $S_{1\ldots i}$ and a suffix of $T_{1\ldots j}$. Since a suffix of a prefix is just a substring, we find the optimal pair of substrings by maximizing $V(i,j)$ over all possible pairs (i,j). The value of the optimal local alignment can be any entry, whichever contains the maximum

Local Alignment by Dynamic Programming

Bases cases: given any i, j we have

$$\begin{aligned} V(i,0) &= 0 \\ V(0,j) &= 0 \end{aligned}$$

Since the optimal suffix to align with a string of length 0 is the empty suffix

Consider an optimal alignment A of a suffix α of $S_{1\ldots i}$ and a suffix β of $T_{1\ldots j}$. There are four possible cases

1. $\alpha = \lambda$ and $\beta = \lambda$: in which case the alignment has value 0

2. $\alpha \neq \lambda$ and the last matched pair in A is $(S_i, -)$: in which case the remainder of A has value $V(i-1, j)$

3. $\alpha \neq \lambda$ and $\beta \neq \lambda$: in which case the remainder of A has value $V(i-1, j-1)$

4. $\beta \neq \lambda$ and the last matched pair in A is $(-, T_i)$: in which case the remainder of A has value $V(i, j-1)$

The optimal alignment chooses whichever of these cases has greatest value

Local Alignment by Dynamic Programming

To summarize, we have the following recurrence relation for $i, j > 0$

$$V(i,j) = \max \begin{cases} 0 & \\ V(i-1,j) & + \ \sigma(S_i, -) \\ V(i-1,j-1) & + \ \sigma(S_i, T_j) \\ V(i,j-1) & + \ \sigma(-, T_j) \end{cases}$$

The maximum local similarity is given by $\max_{1\leq i\leq n, 1\leq j\leq m} V(i,j)$

Observe that the recurrence for computing local alignment is almost identical to the one used for computing global alignment. The only difference is the new meaning of $V(i,j)$ and the inclusion of 0 in the case of local suffix alignment. We can reconstruct optimal alignments by retracing from any maximum entry to any zero entry

Time complexity is $O(nm)$ for similarity, and $O(n+m)$ for alignment

Space complexity is $O(nm)$ for similarity, and $O(n+m)$ for alignment

There is a modification of the dynamic programming algorithm (for both global and local) that computes the maximum similarity in $O(n+m)$ space and still runs in $O(nm)$ time. Reconstructing an alignment can be done in $O(n+m)$ space and $O(nm)$ time with a divide and conquer approach [Hirschberg, Myers and Miller]

End-Space Free Alignment

Given two sequences S and T, possibly of different lengths. Find a best alignment between substrings of S and T when at least one of these substrings is a prefix of the original sequence and one (not necessarily the other) is a suffix

Alignments are scored ignoring some of the end spaces in the sequences. *End spaces* are those that appear before first or after the last character in a sequence.

Biological Motivations

- Searching for regions of a long sequence that approximately the same as a given short sequence
- Searching for overlapping sequences in DNA fragment assembly

Example Consider the two alignments below

```
SEQ 1 CAGCA-CTTGGATTCTCGG
SEQ 2 ---CAGCGTGG--------

SEQ 1 CAGCACTTGGATTCTCGG
SEQ 2 CAGC-----G-T----GG
```

With (global) scores -19 and -12, respectively. The second is an optimal global alignment, however it is not so interesting. If we ignore end spaces, the first is pretty good, with score 3

End-Space Free Alignment Algorithm

The *end-space free alignment* algorithm is again a variation of the *global alignment* algorithm

Base cases: $V(0,0) = V(i,0) = V(0,j) = 0$

This allows zero weight to leading indels in (at most) one of the sequences

Recurrence relation for $i, j > 0$

$$V(i,j) = \max \begin{cases} 0 & & \\ V(i-1,j) & + & \sigma(S_i, -) \\ V(i-1,j-1) & + & \sigma(S_i, T_j) \\ V(i,j-1) & + & \sigma(-, T_j) \end{cases}$$

This is exactly the same recurrence relation as in global alignment algorithm

Search for

- i^* such that $V(i^*, m) = \max_{1<i<n,m} V(i,j)$
- j^* such that $V(n, j^*) = \max_{n,1<j<m} V(i,j)$

We search for the largest value in last column and last row. Thus allowing (at most) one sequence to end before the other, with zero weight for all indels from there on

The score of the alignment will be $\max \begin{cases} V(n, j^*) \\ V(i^*, m) \end{cases}$ We can reconstruct the optimal alignment by retracing from the largest value (between last row and column) to first row or column

Optimal Alignment with Gaps

Given two sequences S and T, possibly of different lengths. Find a best alignment between the two sequences using a gap penalty function

Definition A *gap* is any maximal, consecutive run of spaces in a single sequence of a given alignment. The *length* of a gap is the number of indels in it

Example Consider the alignment

```
SEQ 1 attc--ga-tggacc
SEQ 2 a--cgtgatt---cc
```

This alignment has four gaps and eight indels.

Biological Motivations

1. Insertion or deletion of large substrings often occurs as a single mutational event, and are as likely as insertion or deletion of a single base
2. Two proteins sequences might be relatively similar over several intervals but differ in intervals where one contains a protein subunit that the other does not.
3. cDNA matching
4. Need to score a gap as a whole when trying to align two sequences as to avoid assigning high cost to these mutations

Affine Gap Penalty Model

The total penalty (weight) for a gap of length $q > 1$ is

$$W_T = W_g + qWs$$

where W_g is the penalty for starting the gap, and W_s is the penalty for extending the gap.

We want to find an optimal global (local, or end-space free) alignment of S and T such that

$$\sum_{i=1}^{l} \sigma(S'_i, T'_i) + W_g(\#gaps) + W_s(\#spaces)$$

is maximal. Where S' and T' are S and T with spaces inserted, and $|S'| = |T'| = l$

Notations Define the following variables:

1. $V(i, j)$: the value of an optimal alignment of $S_{1...i}$ and $T_{1...j}$
2. $G(i, j)$: the value of an optimal alignment of $S_{1...i}$ and $T_{1...j}$ whose last pair matches S_i with T_j
3. $F(i, j)$: the value of an optimal alignment of $S_{1...i}$ and $T_{1...j}$ whose last pair matches S_i with a space
4. $E(i, j)$: the value of an optimal alignment of $S_{1...i}$ and $T_{1...j}$ whose last pair matches a space with T_j

Affine Gap Penalty Algorithm

To align S and T, consider the prefixes $S_{1\ldots i}$ and $T_{1\ldots j}$. Any alignment of these prefixes is one of the following three type:

1. S———i
   T———j
   alignment where S_i and T_j are matched

2. S———$i - - - -$
   T——————j
   alignment where S_i is matched with a character strictly to the left of T_j. Alignment ends with a gap in S.

3. S——————i
   T———$j - - - -$
   alignment of where S_i is matched with a character strictly to the right of T_j. Alignment ends with a gap in T.

Base cases: given $i, j > 0$ we have

$$\begin{aligned} V(0,0) &= 0 \\ V(i,0) = E(i,0) &= W_g + iW_s \\ V(0,j) = F(0,j) &= W_g + jW_s \end{aligned}$$

Look at indels and assign the correct values: not only the weight of spaces (qW_s), but also the weight of initiating a gap (W_g)

Affine Gap Penalty Algorithm

We will define three recurrence relations, one for each $G(i,j)$, $E(i,j)$ and $F(i,j)$. Each will be calculated from previously computed values. Take $E(i,j)$ for example. We are looking at alignments in which S ends to the left of T, there are two possible cases for the previous alignment:

1. It looked the same, i.e. S ended to the left of T. In this case, we only need to add another "extension weight" to the value, forming the new weight $E(i,j-1)+W_s$
2. S and T ended at the same place (type 1 alignment). In this case, we need to add both the gap "opening weight" and the gap "extension weight", forming the new weight $V(i,j)+W_g+W_s$

Taking the maximum of the two yields the value $E(i,j)$. Calculating $F(i,j)$ and $G(i,j)$ is done using similar arguments. $V(i,j)$ is calculated by simply taking the maximum of the three. As in *global alignment*, we search for the value $V(n,m)$, and trace the alignment back using pointers created while filling the table

Recurrence relation for $i,j>0$

$$\begin{aligned}
V(i,j) &= \max\{E(i,j), \quad F(i,j), \quad G(i,j)\} \\
G(i,j) &= G(i-1,j-1)+\sigma(S_i,T_j) \\
E(i,j) &= \max\{E(i,j-1)+W_s, \quad V(i,j-1)+W_g+W_s\} \\
F(i,j) &= \max\{F(i-1,j)+W_s, \quad V(i-1,j)+W_g+W_s\}
\end{aligned}$$

Multiple Sequence Alignment

Definition A *multiple alignment* of strings $S_1, S_2, \ldots, S_k$ is a series $S'_1, S'_2, \ldots, S'_k$ such that

1. $|S'_1| = |S'_2| = \cdots = |S'_k|$
2. S'_i is an extension of S_i, obtained by insertions of spaces

Example Given `accdb, cadbd, abcdad` we have

```
a c - c d b -
- c - a d b d
a - b c d a d
```

Biological Motivations

- protein databases are categorized by *protein families* (collection of proteins with similar structure, function, or evolutionary history)
- Comparing a new protein with a family requires to construct a representation of the family and then compare the new protein with the family representation

Scoring Metrics How to score a multiple alignment?

- *Consensus Distance*: the number of characters in all strings S_i that differ from the consensus string C: $\sum_i D(S_i, C)$
- *Evolutionary tree Distance* the weight of the lightest phylogenetic tree that can be constructed from the sequences
- *Sum of Pairs Distance*: the sum of pairwise distances between all pairs of sequences $\sum_{i<j} D(S_i, S_j)$

Multiple Alignment by Dynamic Programming

Generalization of global pairwise alignment algorithm:

Let $|S_1| = n_1, |S_2| = n_2, \ldots, |S_k| = n_k$, fill in a k-dimensional table of size $(n_1 + 1) \times (n_2 + 1) \times \cdots \times (n_k + 1)$ (that is $O(\Pi_{i=1}^{k} n_i)$). Computation of each entry depends on $2^k - 1$ adjacent entries, corresponding to the possibilities for the last match in an optimal alignment: any of the 2^k subsets of the k strings could participate in that match, except for the empty subset

Time and space complexity: assume for simplicity that $n_1 = n_2 = \cdots = n_k = n$ then

1. Space complexity: $O(n^k)$
2. Time complexity: $O(k2^k n^k)$
   (a) Fill in entire table: $O(n^k)$
   (b) Compute each entry: $O(2^k)$
   (c) Computation of σ: $O(k) \rightsquigarrow O(k^2)$

If $n \approx 350$ (typical length for proteins), the dynamic programming algorithm for MSA problem is practical only for small values of k (perhaps 3 or 4)

MSA problem is *NP-complete*: no polynomial time algorithm exists (and will never be found) to solve the problem

Solution: use heuristics (genetic algorithms, neural networks, simulated annealing, tabu search, hill-climbing, . . .) to find good approximate solutions to MSA problem

Center Star Heuristic for MSA

Define distance $D(S,T) = \sum_{i=1}^{l} \delta(S'_i, T'_i)$ the score of a multiple alignment of S and T (where S' and T' are S and T with spaces inserted)

Star alignment method consists in building a multiple sequence alignment M_c based upon the optimal global pairwise alignments between a fixed sequence S_c (the *center*) and all others. Each pairwise alignment is added to M_c using S_c as a guide

```
Algorithm:  Center Star
  Input:  $\delta$,  $S = \{S_1, S_2, \ldots, S_k\}$
  Output:  optimal multiple alignment of $S_1, S_2, \ldots, S_k$
  Find the center $S_c$ such that $\sum_{i \neq c} D(S_c, S_i)$ is minimal
  $S = \{S_1, S_3, \ldots, S_{k-1}\}$
  For $i = 1$ to $k-1$ do
    Find an optimal global alignment $[S'_c, S'_i]$ between $S_c$ and $S_i$
  Let $M_c = \{\text{best } S'_c\}$
  For $i = 1$ to $k-1$ do
  begin
    Add $S'_i$ to $M_c$
    If needed, add spaces to all pre-aligned strings
  end
  Return $M_c$
```

Time Complexity: $O(k^2 n^2)$, n is maximum length

Approximation Analysis: Center Star Heuristic produces a multiple alignment whose sum-of-pairs (SP) value is less than twice that of the optimal SP alignment

Multiple Alignments With Profile

Definition Given a multiple alignment M of a set of strings. a *profile* for M specifies for each column the *frequency* that each character appears in the column

Profile is used to represent a family of proteins and to identify the family of an unknown sequence

Example Consider the following MSA

```
a b c - a
a b a b a
a c c b -
c b - b c
```

Its corresponding profile is

	C_1	C_2	C_3	C_4	C_5
a	75%		25%		50%
b		75%		75%	
c	25%	25%	50%		25%
$-$			25%	25%	25%

Aligning a string S to a profile P will tell us how well S (or some substring of it) fits P. Given the column positions C of P, the alignment consists of inserting spaces into S and C as in pure string alignment. For instance, an alignment of `aabbc` to P is:

```
a a b - b c
1 - 2 3 4 5
```

String-to-Profile Alignment Algorithm

How to score a string to profile alignment?

1. Scoring a column j: equivalent to aligning S_j to each character at column C_j. Then
   $\sigma(j) = \sum_{i=1}^{i=k} \sigma(S_i, i_j) p_{i_j}$
   p_{i_j} is frequency of i-th character in column j,
   k is number of characters in P

2. Score of an alignment: $\sum_{j=1}^{j=l} \sigma(j)$
   l is length of alignment

How to find the optimal global string to profile alignment? By dynamic programming

- Let the score for aligning a character x with column j be $\delta(x, j) = \sum_y \sigma(x, y) p_{y_j}$

- Let $V(i, j)$ denote the value of the optimal alignment of $S_{1\ldots i}$ with the first j columns of C. The recurrences for computing optimal global string to profile alignment are

$$\begin{aligned}
V(0,0) &= 0 \\
V(i,0) &= \textstyle\sum_{k \le i} \sigma(S_k, -) \\
V(0,j) &= \textstyle\sum_{k \le j} \delta(-, k) \\
V(i,j) &= \max \begin{cases} V(i-1, j) & + \quad \sigma(S_i, -) \\ V(i-1, j-1) & + \quad \delta(S_i, j) \\ V(i, j-1) & + \quad \delta(-, j) \end{cases}
\end{aligned}$$

Other MSA Heuristics

Iterative Pairwise Alignment First align two string whose alignment score is the best over all pair. Then iteratively find a string with the smallest distance to any of the already aligned strings and add it to (align it with) the growing alignment

```
Algorithm:  Iterative Alignment
  Find a best pair and align it
  While (not done)
    Find the nearest string to the aligned set
    Align with the previously aligned group
```

Alignment is either done as in Center Star or as in String-to-Profile alignments methods

Progressive Alignment Algorithms There might be case in which some of the strings are very *near* to each other and form *clusters*. It might be an advantage to align strings in the same cluster first, and then merge the clusters of strings. The problem is how to define *near* and *cluster*. It also require the use of a clustering algorithm

Consensus Alignment Given a MSA M of $S_1, S_2, \ldots, S_k$, the *consensus character* of column i of M is the character c_i that minimizes the sum of distances, $d_i = \sum_{j=1}^{j=k} \sigma(S'_{i_j}, c_i)$ to it from all the characters in column i; that c_i is the most common character in column i. The *consensus string* is the concatenation $c_1 c_2 \ldots c_l$ of all consensus characters, where l is the length of M. The *alignment error* of the consensus (or M) is $\sum_{i=1}^{i=l} d_i$. Problem is to find a M with the smallest alignment error

Sequence Database Search

Alignment algorithms and heuristics discussed so far cannot be used when searching a database of size 10^9–10^10 for the closest match to a query string of length 200–500 (for proteins) or length 1000–1500 (for DNAs or RNAs)

Solutions:

1. Implement dynamic programming in hardware
2. Use parallel or distributed hardware
3. Design efficient search heuristics:
   Heuristics are algorithms that give only *approximate* solutions to given problems. Solutions are not guaranteed to be optimal. However, heuristics are much faster than exact algorithms and are relatively cheap and available to any researcher. They are based on the observations that
   (a) Even linear time is slow for a huge database of size over 10^9
   (b) Preprocessing of the database is desirable
   (c) Substitutions are much more likely than indels
   (d) We expect similar sequences to contain lots of segments with matches or substitutions, but without indels or gaps. These segments can be used as starting points for further searching

Professional Sequence Database Searcher

How to search a database of more than 13 millions sequences (such as GenBank)?

Given a new unknown sequence

1. Compare the new sequence with PROSITE and BLOCKS databases for sequence motifs

2. Search the DNA and protein sequence databases (GenBank, Swiss-Prot, . . .) for sequences highly similar to the new sequence (usually a local similarity)

   - Using approximate heuristics such as BLAST or FASTA

3. If needed, compute optimal similarity on BLAST or FASTA results using any algorithm/heuristic

4. Refinement by using profiles (substitution matrices) such as PAM and BLOSUM matrices

Searcher may use many different heuristics whose results will be combined in some way

BLAST: Basic Local Alignment Search Tool

Motivations: increase search speed by finding fewer and better *hot spots*

Finds regions of high local similarity in alignments without gaps, using a profile

Definitions:

- *Segment pair*: pair of same length substrings of S_1 and S_2 aligned without gaps
- *Locally maximal segment (LMS)*: segment whose alignment score (without gaps) cannot be improved by extending or shortening it
- *Maximum segment pair (MSP* in S_1 and S_2: segment pair with the maximum score over all segment pairs in S_1 and S_2
- *High scoring pair*: a MSP above a cut-off score σ

BLAST heuristic

1. Break query sequence into words of length w
2. Given threshold τ, find all *hits*: w-length words of database strings that align with the query words with alignment score higher than τ
3. Extend each hit to a LMS, and check if its score is above cut-off σ, i.e. if the hit is a HSP. The extension of a hit terminates when the score falls below a drop-off threshold
4. Return all HSPs

Parameters:

1. $w \rightarrow$ speed
2. $\tau \rightarrow$ speed and sensitivity (most critical parameter)