Sequence Comparison: Two Sequence Alignment

Chin Lung Lu

Institute of Bioinformatics
National Chiao Tung University

September 27, 2005

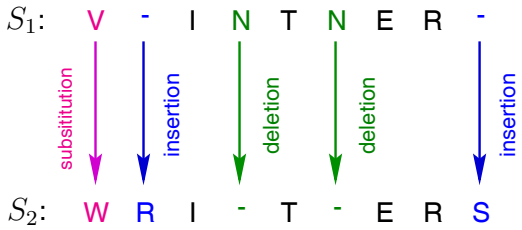
Biological Sequence Comparison

- Types of biological sequences:
  1. Nucleic acid sequences over 4-letter alphabet (e.g., DNA/RNA sequences)
  2. Amino acid sequences over 20-letter alphabet (e.g., protein sequences)
- Why sequence comparison?
  - High sequence similarity usually implies functional or structural similarity.
- Measurements of sequence comparison: edit distance (difference) and alignment (silimarity)

Edit Distance of Two Strings

- Edit distance: the minimum number of edit operations needed to transform S_1 to S_2
  - Three kinds of edit operations:

    ❶ Substitution ❷ Insertion ❸ Deletion
- **Example**: $S_1 =$ VINTNER, $S_2 =$WRITERS



Alignment of Two Strings

- Alignment of strings S_1 and S_2: a pair of strings (S_1', S_2') obtained by insertion of spaces in S_1 and S_2 such that
  1. $|S_1'| = |S_2'|$, and
  2. for each i, $S_1'[i]$ is aligned with $S_2'[i]$ and either $S_1'[i]$ or $S_2'[i]$ is not a space
- **Example**: $S_1 =$ VINTNER, $S_2 =$WRITERS

S_1: V - I N T N E R - S_2: W R I - T - E R S ❶: An alignment	S_1: V - - I N T N E R - S_2: W R - I - T - E R S ❷: Not allowed

Similarity of Two Strings

- $\sigma(x, y)$: score of aligning char. x with char. y
  - If $x = y$, then $\sigma(x, y) \geq 0$;
    otherwise, $\sigma(x, y) < 0$
  - Emphasize matches, penalize mismatches or inserted spaces
- $\sigma(S'_1, S'_2)$: score of an alignment (S'_1, S'_2)
  - $\sigma(S'_1, S'_2) = \sum_{1 \leq i \leq |S'_1|} \sigma(S'_1[i], S'_2[i])$
- $Sim(S_1, S_2)$: similarity of two strings S_1 and S_2
  - $Sim(S_1, S_2) = \underset{\text{all alignments } (S'_1, S'_2)}{\mathcal{MAX}} \{\sigma(S'_1, S'_2)\}$

Scoring Matrices for Nucleotides

	A	T	C	G
A	1	0	0	0
T	0	1	0	0
C	0	0	1	0
G	0	0	0	1

Identity Matrix

A	T	C	G
5	-4	-4	-4
-4	5	-4	-4
-4	-4	5	-4
-4	-4	-4	5

BLAST Matrix

A	T	C	G
1	-5	-5	-1
-5	1	-1	-5
-5	-1	1	-5
-1	-5	-5	1

Transition Transversion Matrix

Scoring Matrices for Amino Acids

- Similarity-based matrices:
  - Based on observed chemical/physical similarity: hydrophobicity, charge, size, etc.
  - Based on genetic code: minimum substitutions for converting a codon to the other
- Log odds matrices:
  - PAM: obtained by observing the substitutions occuring in alignments between similar sequences
  - BLOSUM: obtained from ungapped alignments of related proteins which are grouped using statistical clustering techniques

Types of Two-String Alignment

- Global alignment:
  - To find the optimal alignment between two entire strings S_1 and S_2
- Local alignment:
  - To find the optimal alignment between a substring of S_1 with a substring of S_2
- Semiglobal alignment:
  - To find the optimal alignment between suffix (prefix) of S_1 with prefix (suffix) of S_2

Grobal Alignment Problem slide 10 / 63

Global Alignment Problem

- Global alignment problem:
  - To find the best alignment between two strings S_1 and S_2

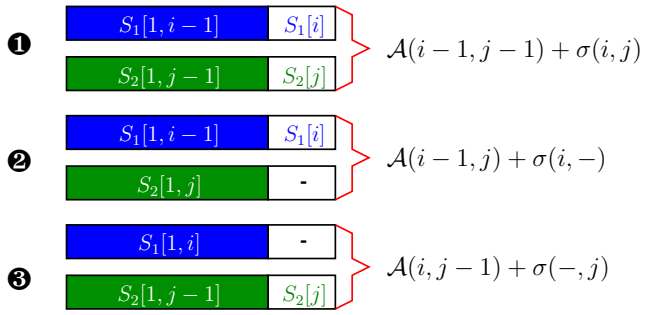
1. Exhaustive method:
   - Generate all possible alignments and then pick the best one (# of alignments is exponential)
   - **Time-commplexity**: exponential time
2. Dynamic programming:
   - Proposed by Needleman & Wunsch [NW70]
   - **Time-commplexity**: $\mathcal{O}(|S_1| \times |S_2|)$

Number of Global Alignments

- The number of ways of merging two sequences of lengths m and n into a signle sequence, while perserving the order of the characters in each sequence, is $\binom{m+n}{m}$.
- There is a one-to-one correspondence between the gapped alignment of two sequences and the merged sequence (as described above).
- The number of all possible alignment between two sequences of length n is $\binom{2n}{n} = \frac{(2n)!}{(n!)^2} \simeq \frac{2^{2n}}{\sqrt{\pi n}}$.
  - Stirling's formula: $x! \simeq \sqrt{2\pi} x^{x+0.5} e^{-x}$

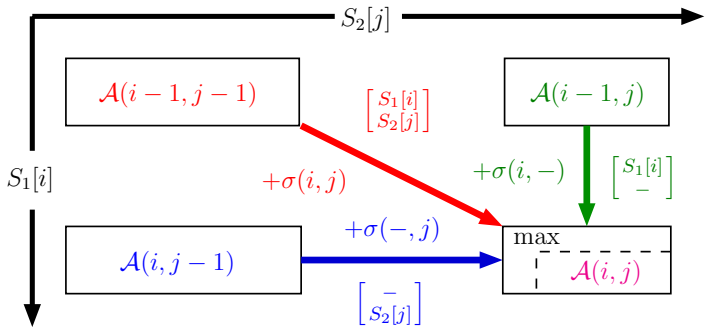
Needleman & Wunsch Method

- $\mathcal{A}(i,j)$: score of an optimal alignment of $S_1[1,i]$ and $S_2[1,j]$ (i.e., $Sim(S_1[1,i], S_2[1,j])$)
  - $Sim(S_1, S_2) = \mathcal{A}(|S_1|, |S_2|)$
- How to compute $\mathcal{A}(i,j)$?

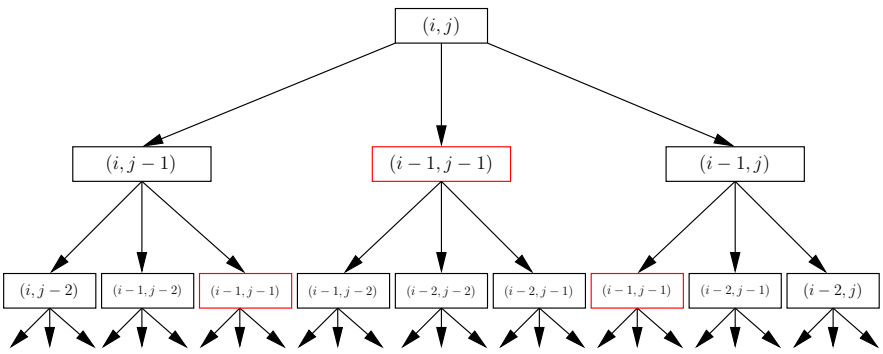


Recursive Function

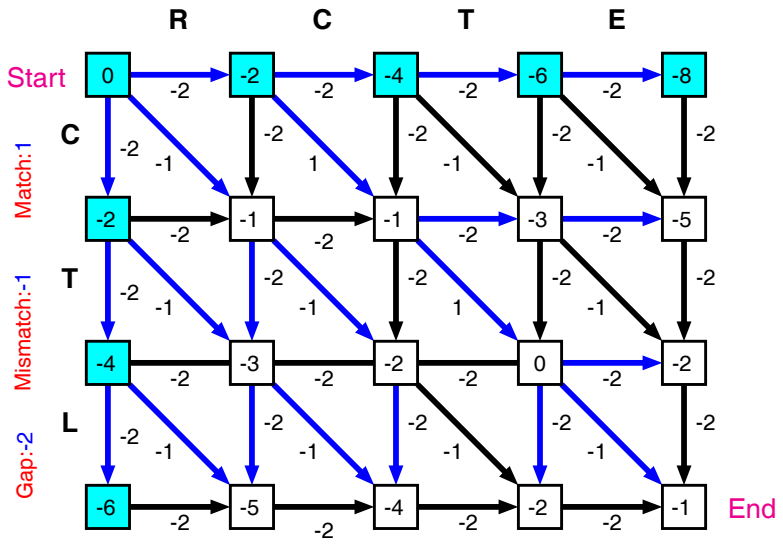
- $\mathcal{A}(i,j) = \max \begin{cases} \mathcal{A}(i-1,j-1)+\sigma(i,j) \\ \mathcal{A}(i-1,j)+\sigma(i,-) \\ \mathcal{A}(i,j-1)+\sigma(-,j) \end{cases}$



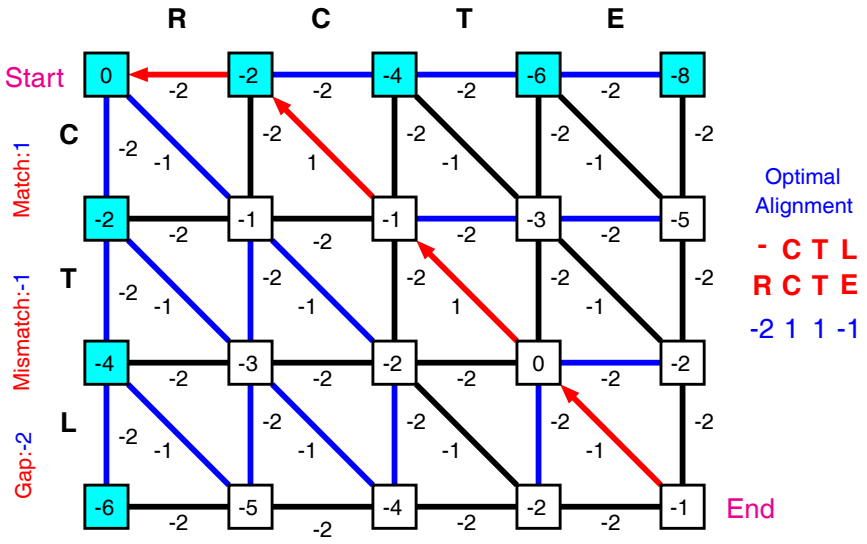
Overlap Between Subproblems



Computation of Optimal Score



Backtracking: Global Alignment



Longest Common Subsequence

- Subsequence of string S: $S[i_1]S[i_2]\cdots S[i_k]$ with $i_1 < i_2 < \cdots < i_k$
  - **Example**: LCS is a subsequence of LongestCommonSubsequence.
- Longest common subsequence problem:
  Given two strings S_1 and S_2, find the longest common subsequence of S_1 and S_2
  - Equal to the global alignment problem with match = 1, mismatch = 0, space = 0
  - **Ex**:

S_1' :	V	-	I	N	T	N	E	R	-
S_2' :	W	R	I	-	T	-	E	R	S

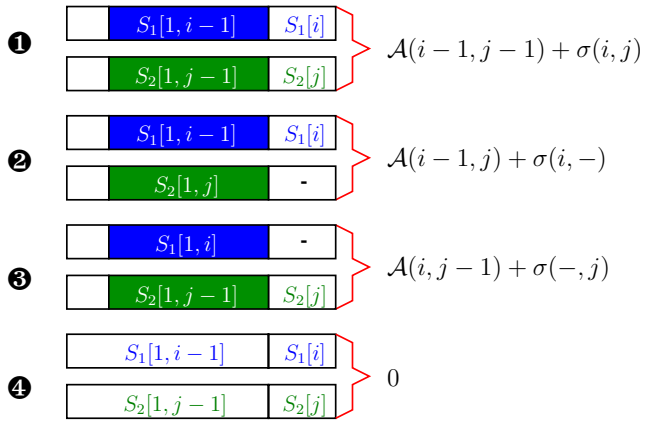
Local Alignment Problem slide 19 / 63

Local Alignment Problem

- Local alignment problem:
  - To find the best alignment between a substring of S_1 and a substring of S_2
- Exhaustive method:
  - Compute the global alignment for each pair of substrings and then pick the best one
  - **Time-commplexity**: $\mathcal{O}(|S_1|^3 \times |S_2|^3)$
- Dynamic programming:
  - Proposed by Smith & Waterman [SW81]
  - **Time-commplexity**: $\mathcal{O}(|S_1| \times |S_2|)$

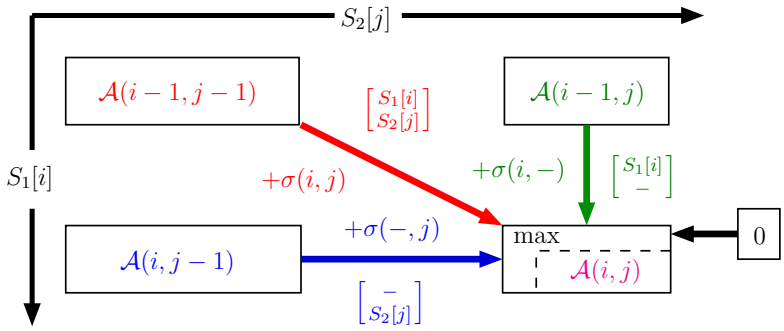
Smith-Waterman Method

- $\mathcal{A}(i,j)$: the highest alignment score of a suffix of $S_1[1,i]$ and a suffix of $S_2[1,j]$

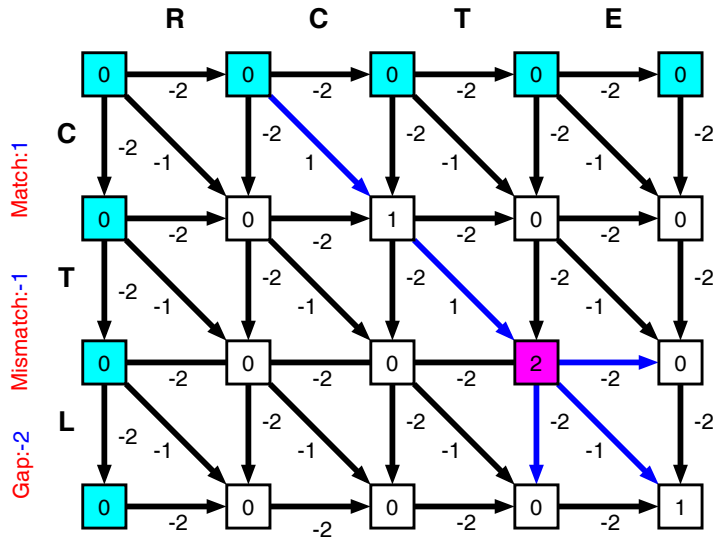


Recursive Function

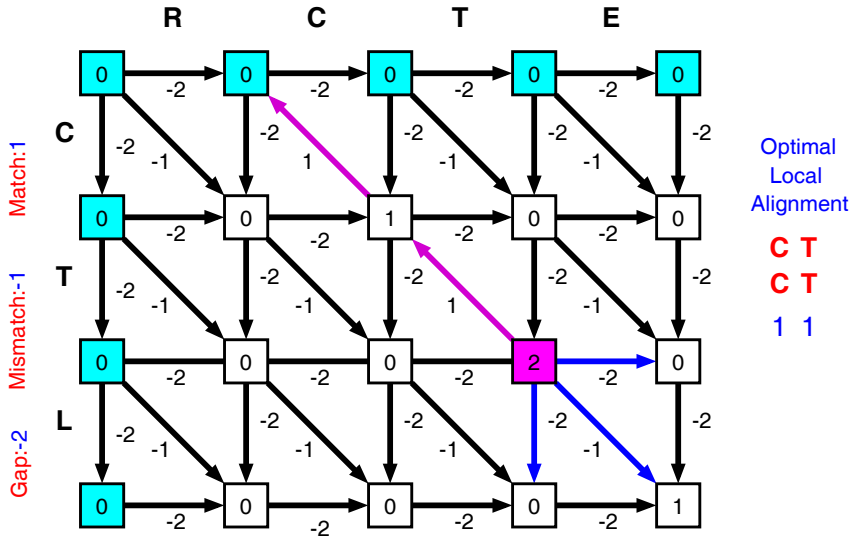
- $\mathcal{A}(i,j) = \max \begin{cases} \mathcal{A}(i-1,j-1) + \sigma(i,j) \\ \mathcal{A}(i-1,j) + \sigma(i,-) \\ \mathcal{A}(i,j-1) + \sigma(-,j) \\ 0 \end{cases}$



Computation of Optimal Score



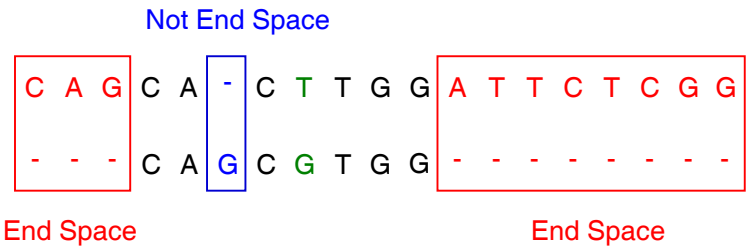
Backtracking: Local Alignment



Semiglobal Alignment Problem slide 25 / 63

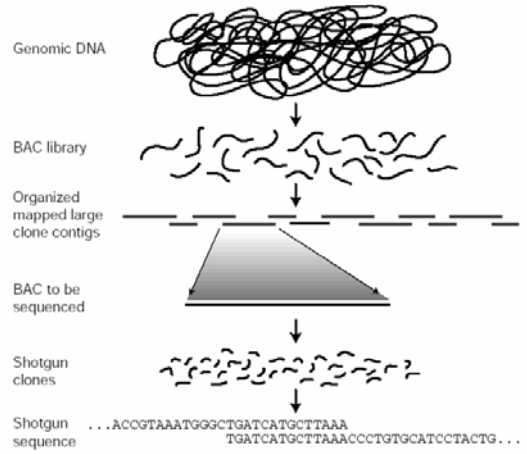
Semiglobal Alignment

- To find the optimal alignment between suffix (prefix) of S_1 with prefix (suffix) of S_2
- End space:



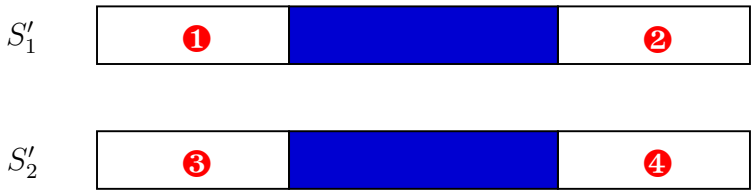
An Applcation of Semiglobal Alignment

- Application to assembly of shotgun seqeunces:



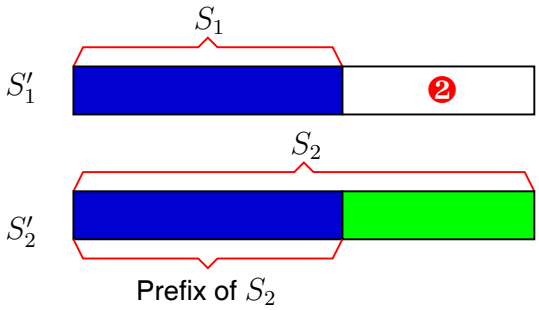
Where Spaces Are Not Charged?

1. Beginning of first sequence S_1'
2. End of the first sequence S_1'
3. Beginning of the second sequence S_2'
4. End of the second sequence S_2'



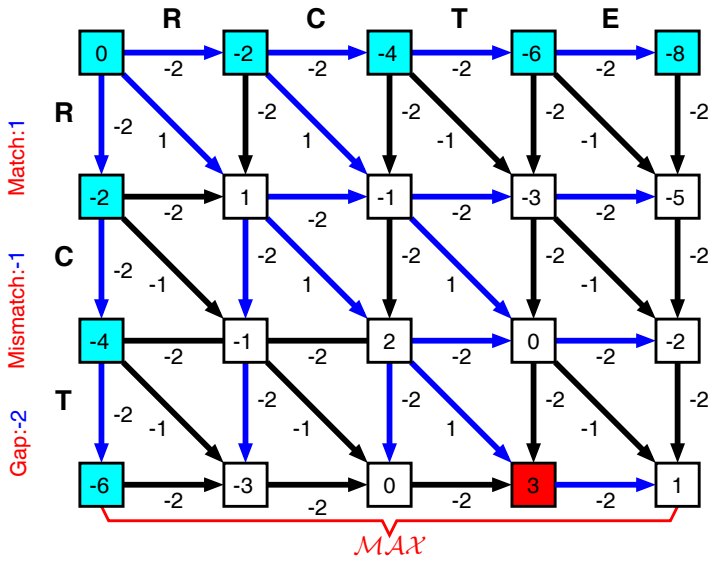
❷ Ignoring Final Spaces in the First Sequence

- To find the optimal alignment between S_1 and a prefix of S_2

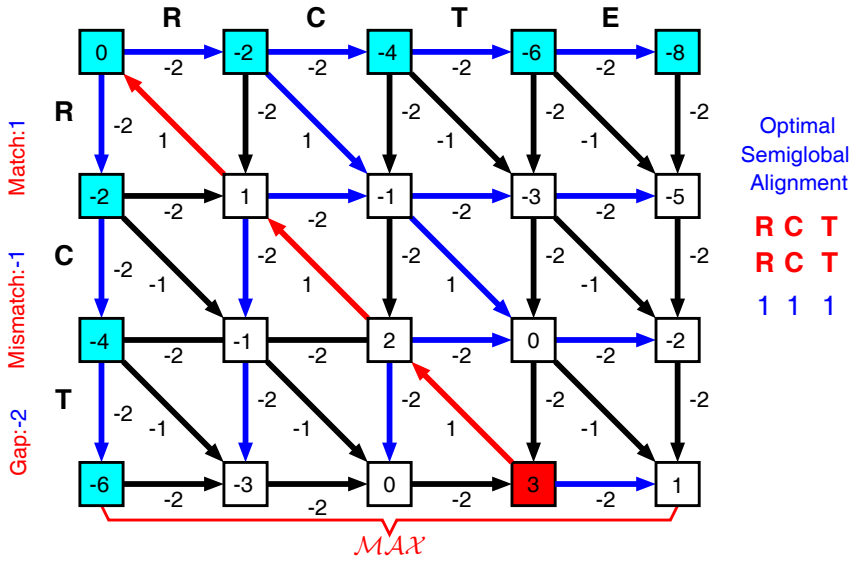


- $Sim(S_1, S_2) = \max\{\mathcal{A}(|S_1|, j) : 1 \leq j \leq |S_2|\}$

❷ Computation of Optimal Score

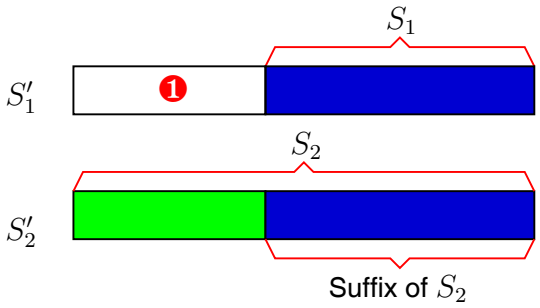


❷ Backtracking: Semiglobal Alignment



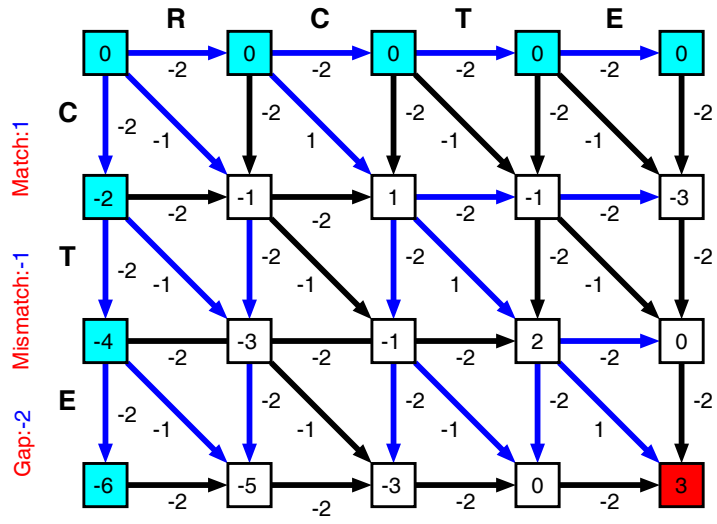
❶ Ignoring Initial Spaces in the First Sequence

- To find the optimal alignment between S_1 and a suffix of S_2

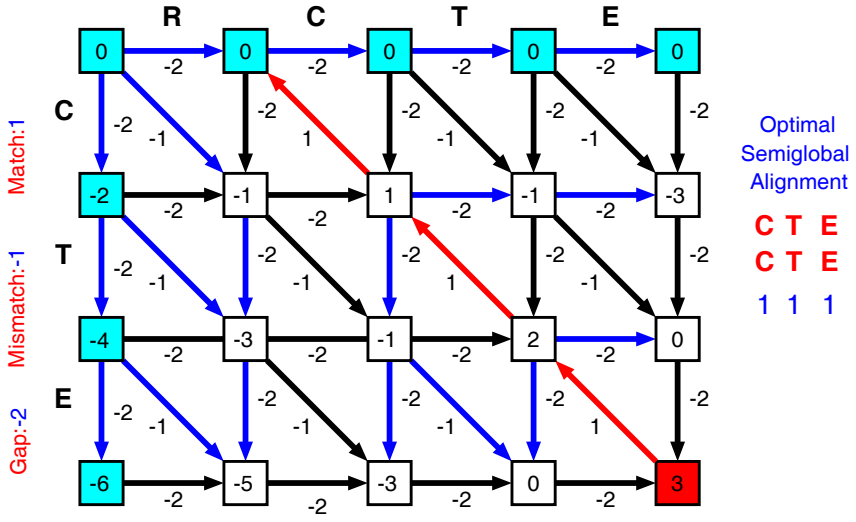


- $Sim(S_1, S_2) = \max\{\mathcal{A}(|S_1|, S_2[j, |S_2|]) : 1 \leq j \leq |S_2|\}$

❷ Computation of Optimal Score



❶ Backtracking: Semiglobal Alignment

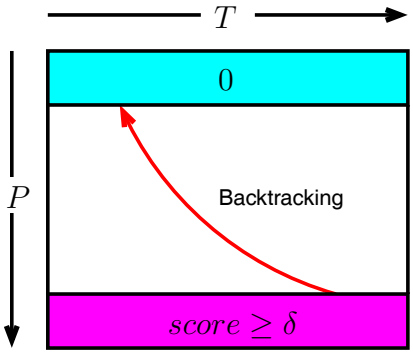


Summary: Semiglobal Alignment

Where spaces are not charged	Action
❶ Beginning of 1st sequence	Initialize first row with zeros
❷ End of 1st sequence	Look for maximum in last row
❸ Beginning of 2nd sequence	Initialize first column with zeros
❹ End of 2nd sequence	Look for maximum in last column

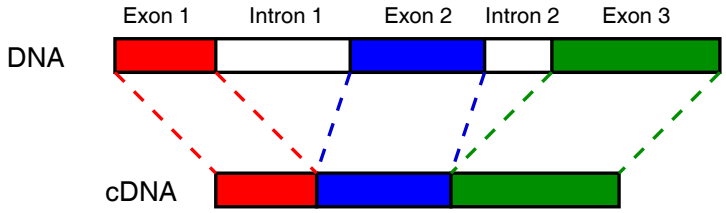
Application: Semiglobal Alignment

- Inexact matching problem: Given a text T, a pattern P and a constant $\delta > 0$, determine if there is a substring T' in T such that the optimal alignment of P to T' has score at least δ?



Gaps

- Gap: a consecutive number of spaces
- Why gaps?
  - Mutation might insert/delete several residues at once.
  - Match cDNA (without intron) to genomic DNA (with exons and introns)



Gap Penalties with k Spaces

1. Linear gap penalty: $P_l(k) = k \times w_s$
   - w_s (the cost of a space) = constant
   - **Time-complexity**: $\mathcal{O}(mn)$
2. Affine gap penalty: $P_a(k) = w_g + k \times w_s$
   - w_g (the cost of a gap) = constant
   - **Time-complexity**: $\mathcal{O}(mn)$
3. Logarithm (convex) gap penalty: $P_c(k) = w_g + \log k$
   - **Time-complexity**: $\mathcal{O}(mn)$
4. General gap penalty: $P_g(k)$: any function
   - **Time-complexity**: $\mathcal{O}(m^2n + mn^2)$

Blocks

(1) Two aligned chars

(2) A maximal series of consecutive chars in S_2 aligned with spaces in S_1

(3) A maximal series of consecutive chars in S_1 aligned with spaces in S_2

S_1	A	A	C	-	-	-	C	G	C	T
S_2	A	C	T	A	C	C	-	-	G	C

- Block of type 2 or 3 cannot follow another block of the same type.

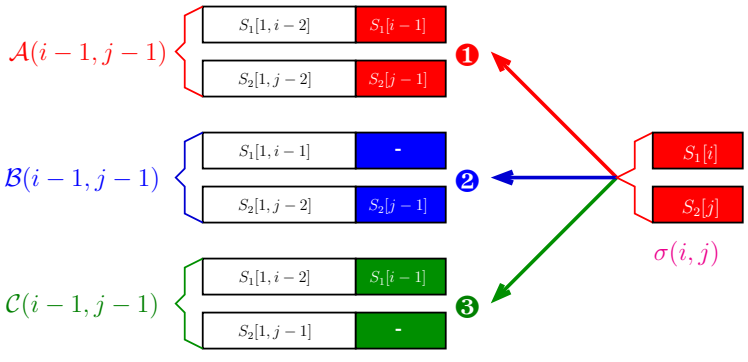
Alignment with General Gap Penalty slide 41 / 63

General Gap Penalty

- $\mathcal{A}(i,j)$: similarity between $S_1[1,i]$ and $S_2[1,j]$ ending with a block of type 1
- $\mathcal{B}(i,j)$: similarity between $S_1[1,i]$ and $S_2[1,j]$ ending with a block of type 2
- $\mathcal{C}(i,j)$: similarity between $S_1[1,i]$ and $S_2[1,j]$ ending with a block of type 3
- $Sim(S_1,S_2) = \max \begin{cases} \mathcal{A}(|S_1|,|S_2|) \\ \mathcal{B}(|S_1|,|S_2|) \\ \mathcal{C}(|S_1|,|S_2|) \end{cases}$

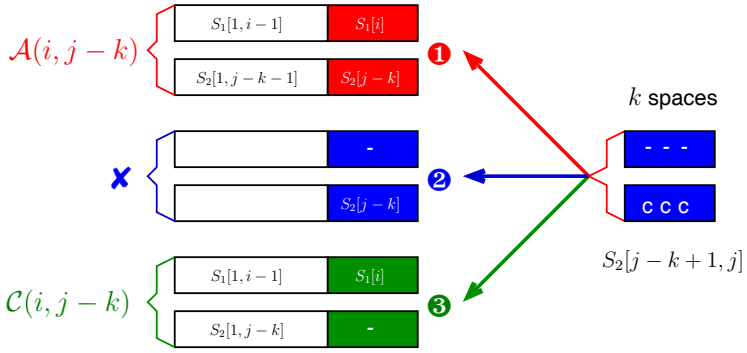
Recursive Function of $\mathcal{A}(i,j)$

- $\mathcal{A}(i,j) = \sigma(i,j) + \max \begin{cases} \mathcal{A}(i-1,j-1) \\ \mathcal{B}(i-1,j-1) \\ \mathcal{C}(i-1,j-1) \end{cases}$



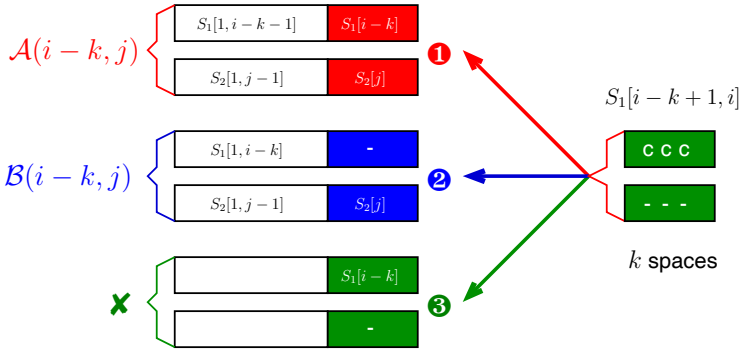
Recursive Function of $\mathcal{B}(i,j)$

- $\mathcal{B}(i,j) = \max \begin{cases} \mathcal{A}(i,j-k) - P_g(k) \\ \mathcal{C}(i,j-k) - P_g(k) \end{cases}$ where $1 \le k \le j$



Recursive Function of $\mathcal{C}(i,j)$

- $\mathcal{C}(i,j) = \max \begin{cases} \mathcal{A}(i-k,j) - P_g(k) \\ \mathcal{B}(i-k,j) - P_g(k) \end{cases}$ where $1 \le k \le i$



Time-Complexity: General Gap Penalty

- Initializations: $k \ge 1$
  - $\mathcal{A}(0,0) = 0, \mathcal{B}(0,k) = \mathcal{C}(k,0) = -P_g(k)$, others = $-\infty$
- Computing $\mathcal{A}(i,j)$, $\mathcal{B}(i,j)$ and $\mathcal{C}(i,j)$ needs to perform $3 + 2j + 2i$ accesses to previous entries.
- Let $|S_1| = m$ and $|S_2| = n$.
  Then time-complexity for general gap penalty is:

$$\sum_{i=1}^{m}\sum_{j=1}^{n}(3 + 2j + 2i) = \mathcal{O}(m^2 n + mn^2)$$

Affine Gap Penalty Function

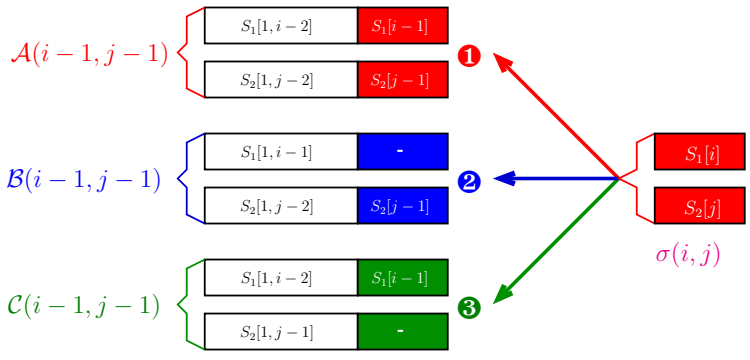
- Affine function $P_a(k)$ for a gap with k spaces:
  - $P_a(k) = \begin{cases} w_g + kw_s & \text{if } k \geq 1 \\ 0 & \text{if } k = 0 \end{cases}$
- The first space costs $w_g + w_s$ (also called gap opening), but the others cost w_s (called gap extension).
- From the evolutionary viewpoint, the occurrence of a gap with k spaces (i.e., a mutation) is more probable than the occurrence of k isolated spaces (i.e., k mutations).

Affine Gap Penalty

- $\mathcal{A}(i,j)$: similarity between $S_1[1,i]$ and $S_2[1,j]$ with "$S_1[i]$ is aligned with $S_2[j]$"
- $\mathcal{B}(i,j)$: similarity between $S_1[1,i]$ and $S_2[1,j]$ with "a space is aligned with $S_2[j]$"
- $\mathcal{C}(i,j)$: similarity between $S_1[1,i]$ and $S_2[1,j]$ with "$S_1[i]$ is aligned with a space"
- $Sim(S_1,S_2) = \max \begin{cases} \mathcal{A}(|S_1|,|S_2|) \\ \mathcal{B}(|S_1|,|S_2|) \\ \mathcal{C}(|S_1|,|S_2|) \end{cases}$

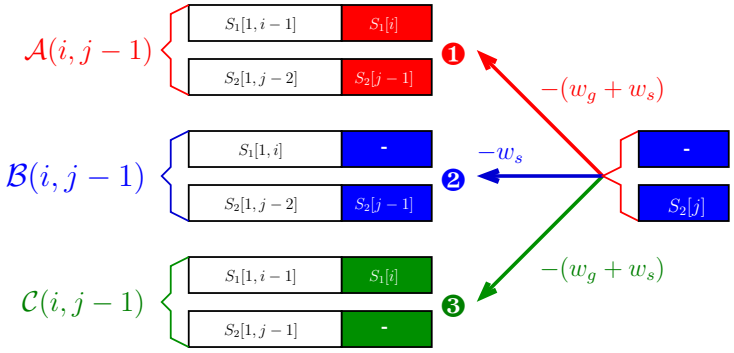
Recursive Function of $\mathcal{A}(i,j)$

- $\mathcal{A}(i,j) = \sigma(i,j) + \max \begin{cases} \mathcal{A}(i-1,j-1) \\ \mathcal{B}(i-1,j-1) \\ \mathcal{C}(i-1,j-1) \end{cases}$



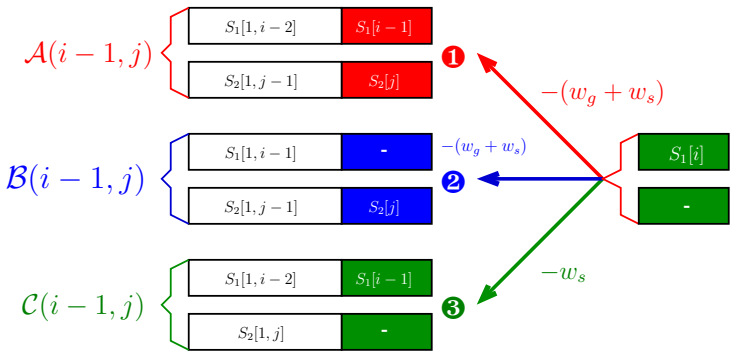
Recursive Function of $\mathcal{B}(i,j)$

- $\mathcal{B}(i,j) = \max \begin{cases} \mathcal{A}(i,j-1) - (w_g + w_s) \\ \mathcal{B}(i,j-1) - w_s \\ \mathcal{C}(i,j-1) - (w_g + w_s) \end{cases}$



Recursive Function of $\mathcal{C}(i,j)$

- $\mathcal{C}(i,j) = \max \begin{cases} \mathcal{A}(i-1,j) - (w_g + w_s) \\ \mathcal{B}(i-1,j) - (w_g + w_s) \\ \mathcal{C}(i-1,j) - w_s \end{cases}$



Complexity: Affine Gap Penalty

- Initializations: $k \geq 1, 0 \leq i \leq m, 0 \leq j \leq n$
  - $\mathcal{A}(0,0) = 0, \mathcal{A}(k,0) = \mathcal{A}(0,k) = -\infty$
  - $\mathcal{B}(i,0) = -\infty, \mathcal{B}(0,k) = -(w_g + kw_s)$
  - $\mathcal{C}(k,0) = -(w_g + kw_s), \mathcal{C}(0,j) = -\infty$
- Computing $\mathcal{A}(i,j), \mathcal{B}(i,j)$ and $\mathcal{C}(i,j)$ needs to perform $3+3+3$ accesses to previous entries.
- The time-complexity for affine gap penalty is:

$$\sum_{i=1}^{m}\sum_{j=1}^{n} 9 = \mathcal{O}(mn)$$

Summary: Gap Penalty Functions

Type	Time	Space
Linear gap penalty $P_l(k) = kw_s$	$\mathcal{O}(mn)$	1 array
Affine gap penalty $P_a(k) = w_g + kw_s$	$\mathcal{O}(mn)$	3 arrays
General gap penalty	$\mathcal{O}(m^2n + mn^2)$	3 arrays

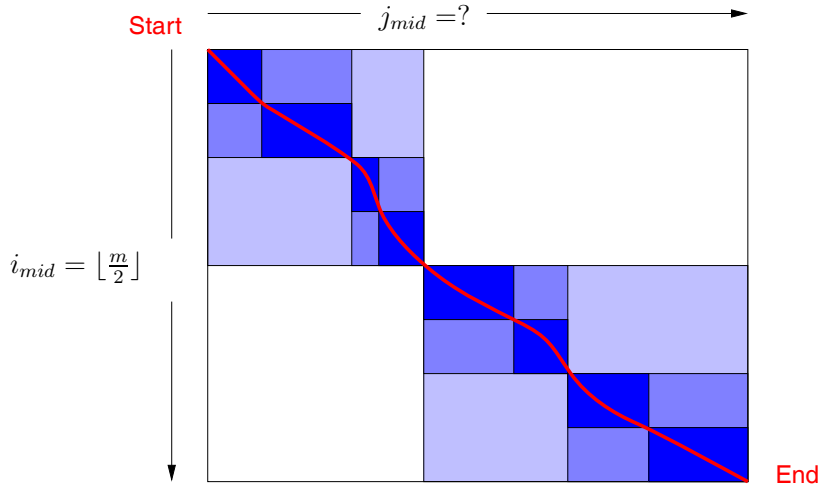
Alignment with Linear Space

slide 55 / 63

Linear-Space Algorithm

- Complexity of dynamic programming for global alignment:
  - Time compelxity: $\mathcal{O}(mn)$ time
  - Space compelxity: $\mathcal{O}(mn)$ space
- How to reduce the memory requirement above?
  1. Computation of the optimal score:
     - Need only constant $\mathcal{O}(n)$ space
  2. Computation of an optimal alignment:
     - Can be done in linear $\mathcal{O}(n)$ space by divide and conquer approach

Schmatic Diagram of Divide and Conquer Method

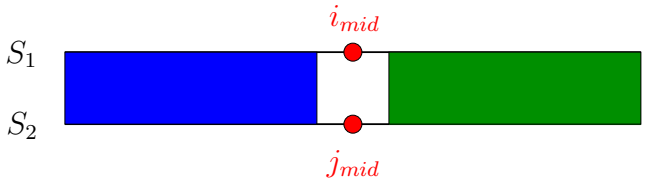


Divide and Conquer Method ❶

- Fix the position of i_{mid} by letting $i_{mid} = \lfloor \frac{m}{2} \rfloor$.
- Find an entry (i_{mid}, j_{mid}) such that it is on a path corresponding to an optimal solution.
- How to compute j_{mid}?
- Two possibilities occur in the optimal alignment:
  1. $S_1[i_{mid}]$ is aligned with $S_2[j_{mid}]$
  2. $S_1[i_{mid}]$ is aligned with a space between $S_2[j_{mid}]$ and $S_2[j_{mid}+1]$

Divide and Conquer Method ❷

Case 1: $S_1[i_{mid}]$ is aligned with $S_2[j_{mid}]$

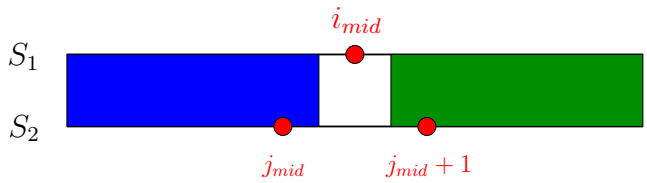


$$Optimal(S_1, S_2) = Optimal\begin{pmatrix} S_1[1, i_{mid}-1] \\ S_2[1, j_{mid}-1] \end{pmatrix}$$

$$+\sigma(a_{i_{mid}}, b_{j_{mid}}) + Optimal\begin{pmatrix} S_1[i_{mid}+1, m] \\ S_2[j_{mid}+1, n] \end{pmatrix}$$

Divide and Conquer Method ❸

Case 2: $S_1[i_{mid}]$ is aligned with a space between $S_2[j_{mid}]$ and $S_2[j_{mid}+1]$



$$Optimal(S_1, S_2) = Optimal\begin{pmatrix} S_1[1, i_{mid}-1] \\ S_2[1, j_{mid}] \end{pmatrix}$$

$$+\sigma(a_{i_{mid}}, -) + Optimal\begin{pmatrix} S_1[i_{mid}+1, m] \\ S_2[j_{mid}+1, n] \end{pmatrix}$$

Divide and Conquer Method ❹

- In summary , j_{mid} is the j such that the following maximum value is obtained.

  $\max_{0 \le j \le n+1}\{case1, case2\}$,

  where

  $$case1 = Opt\begin{pmatrix} S_1[1, i_{mid}-1] \\ S_2[1, j-1] \end{pmatrix} + \sigma(a_{i_{mid}}, b_j) + Opt\begin{pmatrix} S_1[i_{mid}+1, m] \\ S_2[j+1, n] \end{pmatrix},$$

  $$case2 = Opt\begin{pmatrix} S_1[1, i_{mid}-1] \\ S_2[1, j] \end{pmatrix} + \sigma(a_{i_{mid}}, -) + Opt\begin{pmatrix} S_1[i_{mid}+1, m] \\ S_2[j+1, n] \end{pmatrix}$$

Complexity of Divide and Conquer Method

- Space complexity: $\mathcal{O}(n)$ (why?)
- Time complexity: $\mathcal{O}(2mn)$
  - Regardless of where the optimal path crosses the middle row, the total of the computed entries of the two subproblems is just half the entries of the original problem.
  - Let T be the computed entries at the first recursion.
  - The total computed entries of all recursions are at most $T + \frac{1}{2}T + \frac{1}{4}T + \cdots = 2T$.

References

S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Evolution*, 48:443–453, 1970

T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981