Computational Biology

Approximate Patter Matching (Dyn. Prg, Alignments&Co)

Department of Mathematics
Stockholm University

Dynamic Programming (DP)

Dynamic Programming is . . .

- . . . a general, powerful algorithm design technique for solving optimization problems.
- . . . a type of “very smart” exhaustive search that can be applied when the problem can be “subdivided” into overlapping subproblems.
- . . . solves problems by combining the solutions to subproblems
- . . . computes the value of an optimal solution first. Optionally, the optimal solution can be constructed from computed information (backtracking).

Example: Fibonacci Numbers...

...are recursively defined:

- $f(1) = f(2) = 1$
- $f(n) = f(n-1) + f(n-2)$, $n > 2$.

naive recursive way:

F(positive integer n)

1: **if** $n \leq 2$ **then** $f = 1$
2: **else**
3: $\quad f = F(n-1) + F(n-2)$
4: **return** f

recursive way with memo:

F(positive integer n)

1: **if** `memo`$[n] \neq$NIL **then**
2: $\quad$ return `memo`$[n]$
3: **if** $n \leq 2$ **then** $f = 1$
4: **else**
5: $\quad f = F(n-1) + F(n-2)$
6: `memo`$[n]$ = f
7: **return** f

Which algorithm is more efficient and why? WHITEBOARD

Example: Fibonacci Numbers...

...are recursively defined:

- $f(1) = f(2) = 1$
- $f(n) = f(n-1) + f(n-2)$, $n > 2$.

naive recursive way:

F(positive integer n)

1: **if** $n \leq 2$ **then** $f = 1$
2: **else**
3: $\quad f = F(n-1) + F(n-2)$
4: **return** f

recursive way with memo:

F(positive integer n)

1: **if** memo$[n] \neq$NIL **then**
2: $\quad$ return memo$[n]$
3: **if** $n \leq 2$ **then** $f = 1$
4: **else**
5: $\quad f = F(n-1) + F(n-2)$
6: memo$[n]$ = f
7: **return** f

Which algorithm is more efficient and why? WHITEBOARD

Example: Fibonacci Numbers...

...are recursively defined:

- $f(1) = f(2) = 1$
- $f(n) = f(n-1) + f(n-2)$, $n > 2$.

naive recursive way:

F(positive integer n)

1: **if** $n \leq 2$ **then** $f = 1$
2: **else**
3: $\quad f = F(n-1) + F(n-2)$
4: **return** f

recursive way with memo:

F(positive integer n)

1: **if** memo$[n] \neq$NIL **then**
2: $\quad$ return memo$[n]$
3: **if** $n \leq 2$ **then** $f = 1$
4: **else**
5: $\quad f = F(n-1) + F(n-2)$
6: memo$[n]$ = f
7: **return** f

Which algorithm is more efficient and why? WHITEBOARD

Example: Fibonacci Numbers...

...are recursively defined:

- $f(1) = f(2) = 1$
- $f(n) = f(n-1) + f(n-2)$, $n > 2$.

naive recursive way:

F(positive integer n)

1: **if** $n \leq 2$ **then** $f = 1$
2: **else**
3: $\quad f = F(n-1) + F(n-2)$
4: **return** f

recursive way with memo:

F(positive integer n)

1: **if** `memo`$[n] \neq$NIL **then**
2: $\quad$ return `memo`$[n]$
3: **if** $n \leq 2$ **then** $f = 1$
4: **else**
5: $\quad f = F(n-1) + F(n-2)$
6: `memo`$[n]$ = f
7: **return** f

Which algorithm is more efficient and why? WHITEBOARD

Longest common subsequence (LCS)

Classical problem in bioinformative is to understand how “close” to genes or genomes are. There are several ways to adress this problem. A simple approach is the “Longest common subsequence” problem.

- String $X = x_1 x_2 \dots x_m$
- $Z = z_1 z_2 \dots z_k$ is subsequence of X, if there are indices $i_1, i_2, \dots, i_k \in \{1, \dots, m\}$ such that $i_1 < i_2 < \dots < i_k$ and $z_j = x_{i_j}$

  E.g. $Z = \texttt{BCDB}$ is subsequence of $X = \texttt{ABCBDAB}$

  $Z \neq X[i..j]$ may hold!
- A subsequence Z of X and Y is a common subsequence of X and Y

Aim: Find longest subsequence of of X and Y.

Solution: via Dynamic Programming (WHITEBOARD)

Longest common subsequence (LCS)

Classical problem in bioinformative is to understand how “close” to genes or genomes are. There are several ways to adress this problem. A simple approach is the “Longest common subsequence” problem.

- String $X = x_1 x_2 \dots x_m$
- $Z = z_1 z_2 \dots z_k$ is subsequence of X, if there are indices $i_1, i_2, \dots, i_k \in \{1, \dots, m\}$ such that $i_1 < i_2 < \dots < i_k$ and $z_j = x_{i_j}$

  E.g. $Z = \texttt{BCDB}$ is subsequence of $X = \texttt{ABCBDAB}$

  $Z \neq X[i..j]$ may hold!
- A subsequence Z of X and Y is a common subsequence of X and Y

Aim: Find longest subsequence of of X and Y.

Solution: via Dynamic Programming (WHITEBOARD)

Longest common subsequence (LCS)

Classical problem in bioinformative is to understand how “close” to genes or genomes are. There are several ways to adress this problem. A simple approach is the “Longest common subsequence” problem.

- String $X = x_1 x_2 \dots x_m$
- $Z = z_1 z_2 \dots z_k$ is subsequence of X, if there are indices $i_1, i_2, \dots, i_k \in \{1, \dots, m\}$ such that $i_1 < i_2 < \dots < i_k$ and $z_j = x_{i_j}$

  E.g. $Z = \texttt{BCDB}$ is subsequence of $X = \texttt{ABCBDAB}$

  $Z \neq X[i..j]$ may hold!
- A subsequence Z of X and Y is a common subsequence of X and Y

Aim: Find longest subsequence of of X and Y.

Solution: via Dynamic Programming (WHITEBOARD)

Longest common subsequence (LCS)

Classical problem in bioinformative is to understand how “close” to genes or genomes are. There are several ways to adress this problem. A simple approach is the “Longest common subsequence” problem.

- String $X = x_1 x_2 \dots x_m$
- $Z = z_1 z_2 \dots z_k$ is subsequence of X, if there are indices $i_1, i_2, \dots, i_k \in \{1, \dots, m\}$ such that $i_1 < i_2 < \dots < i_k$ and $z_j = x_{i_j}$
  E.g. $Z = \texttt{BCDB}$ is subsequence of $X = \texttt{ABCBDAB}$
  $Z \neq X[i..j]$ may hold!
- A subsequence Z of X and Y is a common subsequence of X and Y

Aim: Find longest subsequence of of X and Y.

Solution: via Dynamic Programming (WHITEBOARD)

Longest common subsequence (LCS)

Classical problem in bioinformative is to understand how “close” to genes or genomes are. There are several ways to adress this problem. A simple approach is the “Longest common subsequence” problem.

- String $X = x_1 x_2 \dots x_m$
- $Z = z_1 z_2 \dots z_k$ is subsequence of X, if there are indices $i_1, i_2, \dots, i_k \in \{1, \dots, m\}$ such that $i_1 < i_2 < \dots < i_k$ and $z_j = x_{i_j}$

  E.g. $Z = \texttt{BCDB}$ is subsequence of $X = \texttt{ABCBDAB}$

  $Z \neq X[i..j]$ may hold!
- A subsequence Z of X and Y is a common subsequence of X and Y

Aim: Find longest subsequence of of X and Y.

Solution: via Dynamic Programming (WHITEBOARD)

Longest common subsequence (LCS)

Classical problem in bioinformative is to understand how “close” to genes or genomes are. There are several ways to adress this problem. A simple approach is the “Longest common subsequence” problem.

- String $X = x_1 x_2 \dots x_m$
- $Z = z_1 z_2 \dots z_k$ is subsequence of X, if there are indices $i_1, i_2, \dots, i_k \in \{1, \dots, m\}$ such that $i_1 < i_2 < \dots < i_k$ and $z_j = x_{i_j}$

  E.g. $Z = \texttt{BCDB}$ is subsequence of $X = \texttt{ABCBDAB}$

  $Z \neq X[i..j]$ may hold!
- A subsequence Z of X and Y is a common subsequence of X and Y

Aim: Find longest subsequence of of X and Y.

Solution: via Dynamic Programming (WHITEBOARD)

Longest common subsequence (LCS)

Classical problem in bioinformative is to understand how “close” to genes or genomes are. There are several ways to adress this problem. A simple approach is the “Longest common subsequence” problem.

- String $X = x_1 x_2 \dots x_m$
- $Z = z_1 z_2 \dots z_k$ is subsequence of X, if there are indices $i_1, i_2, \dots, i_k \in \{1, \dots, m\}$ such that $i_1 < i_2 < \dots < i_k$ and $z_j = x_{i_j}$

  E.g. $Z = \texttt{BCDB}$ is subsequence of $X = \texttt{ABCBDAB}$

  $Z \neq X[i..j]$ may hold!
- A subsequence Z of X and Y is a common subsequence of X and Y

Aim: Find longest subsequence of of X and Y.

Solution: via Dynamic Programming (WHITEBOARD)

Longest common subsequence (LCS) (WHITEBOARD)

LCS(strings X, Y)

1: m= X.length, n= Y.length
2: Let $b[1 \ldots m, 1 \ldots n]$ be new array
3: Let $c[0 \ldots m; 0 \ldots n]$ be new array
4: **for** $i = 1 \ldots m$ **do** $c[i, 0] = 0$
5: **for** $j = 0 \ldots n$ **do** $c[0, j] = 0$
6: **for** $i = 1 \ldots m$ **do**
7: **for** $j = 1 \ldots n$ **do**
8: **if** $x_i = y_j$ **then**
9: $c[i, j] = c[i - 1, j - 1] + 1$
10: $b[i, j] = \text{“}\nwarrow\text{”}$
11: **else if** $c[i - 1, j] \geq c[i, j - 1]$ **then**
12: $c[i, j] = c[i - 1, j]$
13: $b[i, j] = \text{“}\uparrow\text{”}$
14: **else**
15: $c[i, j] = c[i, j - 1]$
16: $b[i, j] = \text{“}\leftarrow\text{”}$
17: **return** c and b

PRINT_LCS(b, X, i, j)

// Initial call PRINT_LCS(b, X, m, n)

1: **if** $i = 0$ or $j = 0$ **then return**
2: **if** $b[i, j] = \text{“}\nwarrow\text{”}$ **then**
3: PRINT_LCS(b, X, $i - 1, j - 1$)
4: print x_i
5: **else if** $b[i, j] = \text{“}\uparrow\text{”}$ **then**
6: PRINT_LCS(b, X, $i - 1, j$)
7: **else**
8: PRINT_LCS(b, X, $i, j - 1$)

Theorem

4.3 LCS( ) *and* PRINT_LCS( ) *correctly returns length and LCS of two strings* $X = x_1 \ldots x_n$ *and* $Y = y_1 \ldots y_m$ *in* $O(mn)$ *time.*

"Non-exact" matching

Aim:
Compare strings to score/evaluate the (dis)similarity between them.

"Non-exact" matching arises in many fields:

- Molecular biology
- Inexact text matching (e.g. spell checkers; web page search)
- Speech recognition

Biology:
In biomolecular sequences (DNA,RNA,Proteins) high sequence similarity often implies significant functional or structural similarity.

Important:
similar function $\not\Rightarrow$ similar structure $\not\Rightarrow$ similar sequences

How to measure "sequence similarity"?

Naive/simple ways:

- Hamming distance (board)
- LCS (board)
- ..

Edit Distance

Edit Operations:

- `I`nsertion of character
- `D`eletion of character
- `R`eplacement of one character by some other one

Edit Distance = Min. Nr. of Edit Operations to transform string u to string v (equivalent transform string v to string u)

D	M	M	R	M	M	I	
w	r	i	t	e	r	-	(M = Match)
-	r	i	d	e	r	s	

Edit Script = string over alphabet $\{I, D, R, M\}$ that describes transformation from u to v.

Edit Distance Problem: For two strings compute edit distance and optimal edit script.

Example

$u =$ TGCATAT $v =$ ATCCGAT

$u =$ TGCATAT $\xrightarrow{\text{del. last T}}$ TGCATA $\xrightarrow{\text{del. last A}}$ TGCAT $\xrightarrow{\text{add A 1.pos}}$ ATGCAT $\xrightarrow{\text{repl. G by C 3.pos}}$ ATCCAT $\xrightarrow{\text{insert G 5.pos}}$ ATCCGAT $= v$ Edit Distance ≤ 5

$u =$ TGCATAT $\xrightarrow{\text{ins. A 1.pos}}$ ATGCATAT $\xrightarrow{\text{del. T 6.pos}}$ ATGCATAT $\xrightarrow{\text{repl. A by G 5.pos}}$ ATGCGTAT $\xrightarrow{\text{repl. G by C 3.pos}}$ ATCCGAT $= v$

Edit Distance ≤ 4

(How to find OPTIMAL one?)

Example

$u =$TGCATAT $v =$ATCCGAT

$u =$ TGCATAT $\xrightarrow{\text{del. last T}}$ TGCATA $\xrightarrow{\text{del. last A}}$ TGCAT $\xrightarrow{\text{add A 1.pos}}$ ATGCAT $\xrightarrow{\text{repl. G by C 3.pos}}$
ATCCAT $\xrightarrow{\text{insert G 5.pos}}$ ATCCGAT$= v$ Edit Distance ≤ 5

$u =$ TGCATAT $\xrightarrow{\text{ins. A 1.pos}}$ ATGCATAT $\xrightarrow{\text{del. T 6.pos}}$ ATGCATAT $\xrightarrow{\text{repl. A by G 5.pos}}$
ATGCGTAT $\xrightarrow{\text{repl. G by C 3.pos}}$ ATCCGAT$= v$

Edit Distance ≤ 4

(How to find OPTIMAL one?)

Example

$u =$TGCATAT $v =$ATCCGAT

$u =$ TGCATAT $\xrightarrow{\text{del. last T}}$ TGCATA $\xrightarrow{\text{del. last A}}$ TGCAT $\xrightarrow{\text{add A 1.pos}}$ ATGCAT $\xrightarrow{\text{repl. G by C 3.pos}}$
ATCCAT $\xrightarrow{\text{insert G 5.pos}}$ ATCCGAT$= v$ Edit Distance ≤ 5

$u =$ TGCATAT $\xrightarrow{\text{ins. A 1.pos}}$ ATGCATAT $\xrightarrow{\text{del. T 6.pos}}$ ATGCATAT $\xrightarrow{\text{repl. A by G 5.pos}}$
ATGCGTAT $\xrightarrow{\text{repl. G by C 3.pos}}$ ATCCGAT$= v$

Edit Distance ≤ 4

(How to find OPTIMAL one?)

Example

$u =$TGCATAT $v =$ATCCGAT

$u =$ TGCATAT $\xrightarrow{\text{del. last T}}$ TGCATA $\xrightarrow{\text{del. last A}}$ TGCAT $\xrightarrow{\text{add A 1.pos}}$ ATGCAT $\xrightarrow{\text{repl. G by C 3.pos}}$
ATCCAT $\xrightarrow{\text{insert G 5.pos}}$ ATCCGAT$= v$ Edit Distance ≤ 5

$u =$ TGCATAT $\xrightarrow{\text{ins. A 1.pos}}$ ATGCATAT $\xrightarrow{\text{del. T 6.pos}}$ ATGCATAT $\xrightarrow{\text{repl. A by G 5.pos}}$
ATGCGTAT $\xrightarrow{\text{repl. G by C 3.pos}}$ ATCCGAT$= v$

Edit Distance ≤ 4

(How to find OPTIMAL one?)

Example

$u =$TGCATAT $v =$ATCCGAT

$u =$ TGCATAT $\xrightarrow{\text{del. last T}}$ TGCATA $\xrightarrow{\text{del. last A}}$ TGCAT $\xrightarrow{\text{add A 1.pos}}$ ATGCAT $\xrightarrow{\text{repl. G by C 3.pos}}$
ATCCAT $\xrightarrow{\text{insert G 5.pos}}$ ATCCGAT$= v$ Edit Distance ≤ 5

$u =$ TGCATAT $\xrightarrow{\text{ins. A 1.pos}}$ ATGCATAT $\xrightarrow{\text{del. T 6.pos}}$ ATGCATAT $\xrightarrow{\text{repl. A by G 5.pos}}$
ATGCGTAT $\xrightarrow{\text{repl. G by C 3.pos}}$ ATCCGAT$= v$

Edit Distance ≤ 4

(How to find OPTIMAL one?)

(global pairwise) Alignment

Alternative way to edit script: Alignment

For two strings $u = u_1 \dots u_m$ and $v = v_1 \dots v_n$ an *alignment* $\mathcal{A}$ is a matrix with two rows and entries $\mathcal{A}[i, j]$ that are characters from Alphabet Σ (e.g.$\Sigma = \{A, C, G, T\}$) or a *gap* "-" s.t.

- 1st row $= u$ after deleting all gaps
- 2st row $= v$ after deleting all gaps
- in no column are two gaps

w	r	i	t	e	r	-		-	T	G	C	A	T	A	T
-	r	i	d	e	r	s		A	T	C	C	G	-	A	T

Cost-Function	$\delta : \Sigma \cup \{-\} \times \Sigma \cup \{-\} \to \mathbb{R}$
Unit-Cost-Function	$\delta(a, b) = 1$ if $a \neq b$
	$\delta(a, b) = 0$ if $a = b$
Alignment Costs	$\delta(\mathcal{A}) := \sum_{i=1} \delta(a_i, b_i)$

(global pairwise) Alignment

Alternative way to edit script: Alignment

For two strings $u = u_1 \dots u_m$ and $v = v_1 \dots v_n$ an *alignment* $\mathcal{A}$ is a matrix with two rows and entries $\mathcal{A}[i, j]$ that are characters from Alphabet Σ (e.g.$\Sigma = \{A, C, G, T\}$) or a *gap* "-" s.t.

- 1st row $= u$ after deleting all gaps
- 2st row $= v$ after deleting all gaps
- in no column are two gaps

w	r	i	t	e	r	-		-	T	G	C	A	T	A	T
-	r	i	d	e	r	s		A	T	C	C	G	-	A	T

Cost-Function $\delta : \Sigma \cup \{-\} \times \Sigma \cup \{-\} \to \mathbb{R}$

Unit-Cost-Function $\delta(a, b) = 1$ if $a \neq b$

$\delta(a, b) = 0$ if $a = b$

Alignment Costs $\delta(\mathcal{A}) := \sum_{i=1} \delta(a_i, b_i)$

(global pairwise) Alignment

Alternative way to edit script: Alignment

For two strings $u = u_1 \dots u_m$ and $v = v_1 \dots v_n$ an *alignment* $\mathcal{A}$ is a matrix with two rows and entries $\mathcal{A}[i, j]$ that are characters from Alphabet Σ (e.g.$\Sigma = \{A, C, G, T\}$) or a *gap* "-" s.t.

- 1st row $= u$ after deleting all gaps
- 2st row $= v$ after deleting all gaps
- in no column are two gaps

w	r	i	t	e	r	-		-	T	G	C	A	T	A	T
-	r	i	d	e	r	s		A	T	C	C	G	-	A	T

Cost-Function $\delta : \Sigma \cup \{-\} \times \Sigma \cup \{-\} \to \mathbb{R}$

Unit-Cost-Function $\delta(a, b) = 1$ if $a \neq b$

$\delta(a, b) = 0$ if $a = b$

Alignment Costs $\delta(\mathcal{A}) := \sum_{i=1} \delta(a_i, b_i)$

(global pairwise) Alignment

Alternative way to edit script: Alignment

For two strings $u = u_1 \dots u_m$ and $v = v_1 \dots v_n$ an *alignment* $\mathcal{A}$ is a matrix with two rows and entries $\mathcal{A}[i, j]$ that are characters from Alphabet Σ (e.g.$\Sigma = \{A, C, G, T\}$) or a *gap* "-" s.t.

- 1st row $= u$ after deleting all gaps
- 2st row $= v$ after deleting all gaps
- in no column are two gaps

w	r	i	t	e	r	-		-	T	G	C	A	T	A	T
-	r	i	d	e	r	s		A	T	C	C	G	-	A	T

Cost-Function	$\delta : \Sigma \cup \{-\} \times \Sigma \cup \{-\} \to \mathbb{R}$
Unit-Cost-Function	$\delta(a, b) = 1$ if $a \neq b$
	$\delta(a, b) = 0$ if $a = b$
Alignment Costs	$\delta(\mathcal{A}) := \sum_{i=1} \delta(a_i, b_i)$

Lemma 1

Edit Distance of two strings u, v equals the min. alignments costs $\delta(\mathcal{A})$ between u and v with unit-cost function.

How to compute Edit Distance? Dynamic Programming!

Recurrence Function D (Needleman-Wunsch Algorithm)

Given the strings $u = u_1 \dots u_m$ and $v = v_1 \dots v_n$

Assume $D[i, j]$ are the costs for an optimal alignment of substrings $u_1 \dots u_i$ and $v_1 \dots v_j$, $1 \leq i \leq m$, $1 \leq j \leq n$

$i = 0$: alignment empty string ϵ and $v_1 \dots v_j$
$j = 0$: alignment $u_1 \dots u_i$ and empty string ϵ

Init: $D[i, 0] = D[i-1, 0] + \delta(s_i, -)$; $D[0, j] = D[0, j-1] + \delta(-, t_j)$, $i, j \geq 0$;

Compute

$$D[i,j] = \min \begin{cases} D[i-1,j] & + & \delta(u_i, -) \\ D[i-1,j-1] & + & \delta(u_i, v_j) \\ D[i,j-1] & + & \delta(-, v_j) \end{cases}$$

(if δ = unit-cost-function, then $D[i, 0] = i$ and $D[0, j] = j$ for $i, j \geq 1$)

Lemma 2

$D[m, n]$=cost of optimal alignment between u and v.

Recurrence Function D (Needleman-Wunsch Algorithm)

Given the strings $u = u_1 \dots u_m$ and $v = v_1 \dots v_n$

Assume $D[i, j]$ are the costs for an optimal alignment of substrings $u_1 \dots u_i$ and $v_1 \dots v_j$, $1 \leq i \leq m$, $1 \leq j \leq n$

$i = 0$: alignment empty string ϵ and $v_1 \dots v_j$
$j = 0$: alignment $u_1 \dots u_i$ and empty string ϵ

Init: $D[i, 0] = D[i-1, 0] + \delta(s_i, -)$; $D[0, j] = D[0, j-1] + \delta(-, t_j)$, $i, j \geq 0$;

Compute

$$D[i,j] = \min \begin{cases} D[i-1,j] & + & \delta(u_i, -) \\ D[i-1,j-1] & + & \delta(u_i, v_j) \\ D[i,j-1] & + & \delta(-, v_j) \end{cases}$$

(if δ = unit-cost-function, then $D[i, 0] = i$ and $D[0, j] = j$ for $i, j \geq 1$)

Lemma 2

$D[m, n]$=cost of optimal alignment between u and v.

Recurrence Function D (Needleman-Wunsch Algorithm)

Given the strings $u = u_1 \dots u_m$ and $v = v_1 \dots v_n$

Assume $D[i, j]$ are the costs for an optimal alignment of substrings $u_1 \dots u_i$ and $v_1 \dots v_j$, $1 \leq i \leq m$, $1 \leq j \leq n$

$i = 0$: alignment empty string ϵ and $v_1 \dots v_j$
$j = 0$: alignment $u_1 \dots u_i$ and empty string ϵ

Init: $D[i, 0] = D[i-1, 0] + \delta(s_i, -)$; $D[0, j] = D[0, j-1] + \delta(-, t_j)$, $i, j \geq 0$;

Compute

$$D[i,j] = \min \begin{cases} D[i-1,j] & + & \delta(u_i, -) \\ D[i-1,j-1] & + & \delta(u_i, v_j) \\ D[i,j-1] & + & \delta(-, v_j) \end{cases}$$

(if δ = unit-cost-function, then $D[i, 0] = i$ and $D[0, j] = j$ for $i, j \geq 1$)

Lemma 2

$D[m, n]$=cost of optimal alignment between u and v.

Recurrence Function D (Needleman-Wunsch Algorithm)

Given the strings $u = u_1 \dots u_m$ and $v = v_1 \dots v_n$

Assume $D[i, j]$ are the costs for an optimal alignment of substrings $u_1 \dots u_i$ and $v_1 \dots v_j$, $1 \leq i \leq m$, $1 \leq j \leq n$

$i = 0$: alignment empty string ϵ and $v_1 \dots v_j$
$j = 0$: alignment $u_1 \dots u_i$ and empty string ϵ

Init: $D[i, 0] = D[i-1, 0] + \delta(s_i, -)$; $D[0, j] = D[0, j-1] + \delta(-, t_j)$, $i, j \geq 0$;

Compute

$$D[i,j] = \min \begin{cases} D[i-1,j] & + & \delta(u_i, -) \\ D[i-1,j-1] & + & \delta(u_i, v_j) \\ D[i,j-1] & + & \delta(-, v_j) \end{cases}$$

(if δ = unit-cost-function, then $D[i, 0] = i$ and $D[0, j] = j$ for $i, j \geq 1$)

Lemma 2

$D[m, n]$=cost of optimal alignment between u and v.

Recurrence Function D (Needleman-Wunsch Algorithm)

Given the strings $u = u_1 \dots u_m$ and $v = v_1 \dots v_n$

Assume $D[i, j]$ are the costs for an optimal alignment of substrings $u_1 \dots u_i$ and $v_1 \dots v_j$, $1 \leq i \leq m$, $1 \leq j \leq n$

$i = 0$: alignment empty string ϵ and $v_1 \dots v_j$
$j = 0$: alignment $u_1 \dots u_i$ and empty string ϵ

Init: $D[i, 0] = D[i-1, 0] + \delta(s_i, -)$; $D[0, j] = D[0, j-1] + \delta(-, t_j)$, $i, j \geq 0$;

Compute

$$D[i,j] = \min \begin{cases} D[i-1,j] & + & \delta(u_i, -) \\ D[i-1,j-1] & + & \delta(u_i, v_j) \\ D[i,j-1] & + & \delta(-, v_j) \end{cases}$$

(if δ = unit-cost-function, then $D[i, 0] = i$ and $D[0, j] = j$ for $i, j \geq 1$)

Lemma 2

$D[m, n]$=cost of optimal alignment between u and v.

Backtracing

Given the strings $u = u_1 \dots u_m$ and $v = v_1 \dots v_n$

Tracematrix is an $m \times n$ matrix with $T[i,j] \subseteq \{\leftarrow, \nwarrow, \uparrow\}$.

Init: $T[0,0] = \emptyset$, $T[i,0] = \uparrow$, $T[0,j] = \leftarrow$ for $1 \leq i \leq m$, $1 \leq j \leq n$

Set:
$$\begin{array}{lll} \uparrow \in T[i,j] & \text{if} & D[i-1,j] + \delta(u_i, -) \\ \nwarrow \in T[i,j] & \text{if} & D[i-1,j-1] + \delta(u_i, v_j) \\ \leftarrow \in T[i,j] & \text{if} & D[i,j-1] + \delta(-, v_j) \end{array}$$

Runtime: $O(mn)$

Alignment with variable Gap-Costs

```
ACCGTCTGCT   ACCGTCTGCT
A-C--C-G-T   ACCGT----
```
$\delta(\mathcal{A}) = 5$

This contradicts “biological intuition”:
Insertion of gap of length k is “evolutionary simpler to realize” then insertion of k gaps of length 1.

gap penalty function $g : \mathbb{N} \to \mathbb{R}$
$g(k)$ is penalty for inserting a gap of length k.
we need:

$$g(k + l) \leq g(k) + g(l),$$

as otherwise it might be better to insert 2 gaps of length k and l then one gap of length $k + l$.

Alignment with variable Gap-Costs (Smith-Waterman-Alg.)

Init: $D[0,0] = 0$; $D[0,k] = D[k,0] = g(k)$, $k \geq 1$;

$$D[i,j] = \min \begin{cases} D[i-1, j-1] & + & \delta(u_i, v_j) \\ \min_{1 \leq k \leq i} D[i-k, j] & + & g(k) \\ \min_{1 \leq k \leq j} D[i, j-k] & + & g(k) \end{cases}$$

Tracematrix is an $m \times n$ matrix with $T[i,j] \subseteq \{\leftarrow_k, \nwarrow, \uparrow_k, k \in \mathbb{N}\}$

Distance VS Scoring Function

Note: Instead of using a distance matrix D we can use a Similarity/Scoring Matrix S and maximize.

Init: $S[i,0] = -i * gap - cost$; $S[0,j] = -j * gap - cost$; for $i,j \geq 0$;

Compute

$$S[i,j] = \max \begin{cases} S[i-1,j] & + & \delta(u_i,-) \\ S[i-1,j-1] & + & \delta(u_i,v_j) \\ S[i,j-1] & + & \delta(-,v_j) \end{cases}$$

with e.g.

$$\delta(a,b) = \begin{cases} 1 & \text{if } a = b \\ -1 & \text{if } a \neq b \text{ and } a,b \neq - \\ -3 & \text{else (gap-costs)} \end{cases}$$

Local vs Global Alignment

Needleman-Wunsch computes a *global optimal Alignment*

NW reasonable if sequences have almost same length

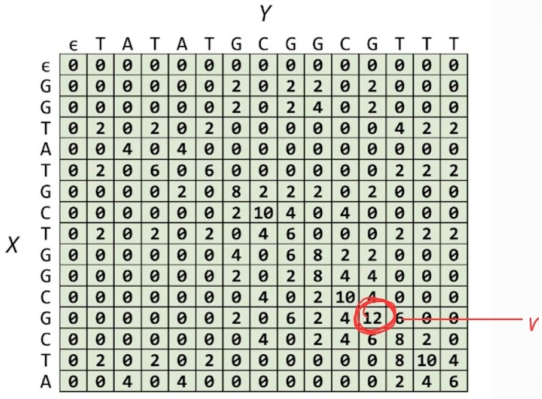
If sequences have quite different length, then the sequences are “shredded”:

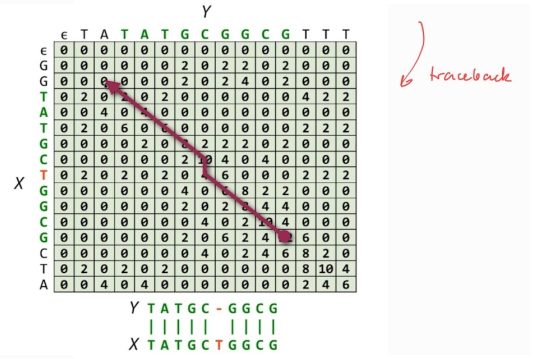
```
R------LCPMNLCGCSQ-----------------KY
RCGEQGSNMECPNNLC-CSQYGYCGMGGDYCGKGCQNGACWTSKR
```

Reason: gaps are penalized equally on each position
Reasonable: less penalization of gaps at end and beginning

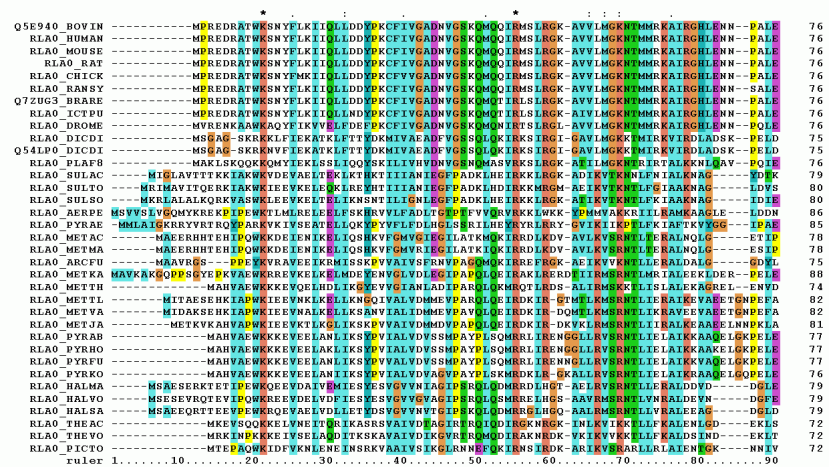
Local Alignment: find best alignment of two substrings of two sequences
(Smith-Waterman-Algorithm)

Local vs Global Alignment





Example Multiple Alignment



First 90 positions of a protein multiple sequence alignment of instances of the acidic ribosomal protein P0 (L10E) from several organisms. (wikipedia)

Now a short overview of three classical (collections) of algorithm that are based on or concerned with Alignments.

- BLAST
- Clustal
- MUSCLE

BLAST = Basic Local Alignment Search Tool

Umbrella term for a collection of the world's most widely used programs for analyzing biological sequence data

- BLAST is used to compare experimentally determined DNA or protein sequences with sequences already existing in a database.
- Basic idea: BLAST divides query sequences into short strings and initially only looks for (exact) matches of those strings in database strings.
  This is afterwards extended to get the entire alignment.
- very fast local alignment heuristic, but no optimality guarantee
- output: series of local alignments, i.e. comparisons of pieces of the searched sequence with similar pieces from the database. In addition, BLAST indicates how significant of the hits that have been found.

Databases e.g. for nucleotide sequences (Genbank of NCBI, EMBL, ...) or protein databases (SwissProt, RefSeq, Pfam, ...).

BLAST homepage: `blast.ncbi.nlm.nih.gov`
Tutorial: `digitalworldbiology.com/BLAST`

BLAST = Basic Local Alignment Search Tool

Umbrella term for a collection of the world's most widely used programs for analyzing biological sequence data

- BLAST is used to compare experimentally determined DNA or protein sequences with sequences already existing in a database.
- Basic idea: BLAST divides query sequences into short strings and initially only looks for (exact) matches of those strings in database strings.
  This is afterwards extended to get the entire alignment.
- very fast local alignment heuristic, but no optimality guarantee
- output: series of local alignments, i.e. comparisons of pieces of the searched sequence with similar pieces from the database. In addition, BLAST indicates how significant of the hits that have been found.

Databases e.g. for nucleotide sequences (Genbank of NCBI, EMBL, ...) or protein databases (SwissProt, RefSeq, Pfam, ...).

BLAST homepage: `blast.ncbi.nlm.nih.gov`
Tutorial: `digitalworldbiology.com/BLAST`

BLAST = Basic Local Alignment Search Tool

Umbrella term for a collection of the world's most widely used programs for analyzing biological sequence data

- BLAST is used to compare experimentally determined DNA or protein sequences with sequences already existing in a database.
- Basic idea: BLAST divides query sequences into short strings and initially only looks for (exact) matches of those strings in database strings.
  This is afterwards extended to get the entire alignment.
- very fast local alignment heuristic, but no optimality guarantee
- output: series of local alignments, i.e. comparisons of pieces of the searched sequence with similar pieces from the database. In addition, BLAST indicates how significant of the hits that have been found.

Databases e.g. for nucleotide sequences (Genbank of NCBI, EMBL, ...) or protein databases (SwissProt, RefSeq, Pfam, ...).

BLAST homepage: `blast.ncbi.nlm.nih.gov`
Tutorial: `digitalworldbiology.com/BLAST`

BLAST = Basic Local Alignment Search Tool

Umbrella term for a collection of the world's most widely used programs for analyzing biological sequence data

- BLAST is used to compare experimentally determined DNA or protein sequences with sequences already existing in a database.
- Basic idea: BLAST divides query sequences into short strings and initially only looks for (exact) matches of those strings in database strings.
  This is afterwards extended to get the entire alignment.
- very fast local alignment heuristic, but no optimality guarantee
- output: series of local alignments, i.e. comparisons of pieces of the searched sequence with similar pieces from the database. In addition, BLAST indicates how significant of the hits that have been found.

Databases e.g. for nucleotide sequences (Genbank of NCBI, EMBL, . . .) or protein databases (SwissProt, RefSeq, Pfam, . . .).

BLAST homepage: `blast.ncbi.nlm.nih.gov`
Tutorial: `digitalworldbiology.com/BLAST`

BLAST = Basic Local Alignment Search Tool

Umbrella term for a collection of the world's most widely used programs for analyzing biological sequence data

- BLAST is used to compare experimentally determined DNA or protein sequences with sequences already existing in a database.
- Basic idea: BLAST divides query sequences into short strings and initially only looks for (exact) matches of those strings in database strings.
  This is afterwards extended to get the entire alignment.
- very fast local alignment heuristic, but no optimality guarantee
- output: series of local alignments, i.e. comparisons of pieces of the searched sequence with similar pieces from the database. In addition, BLAST indicates how significant of the hits that have been found.

Databases e.g. for nucleotide sequences (Genbank of NCBI, EMBL, . . .) or protein databases (SwissProt, RefSeq, Pfam, . . .).

BLAST homepage: `blast.ncbi.nlm.nih.gov`
Tutorial: `digitalworldbiology.com/BLAST`

BLAST = Basic Local Alignment Search Tool

Umbrella term for a collection of the world's most widely used programs for analyzing biological sequence data

- BLAST is used to compare experimentally determined DNA or protein sequences with sequences already existing in a database.
- Basic idea: BLAST divides query sequences into short strings and initially only looks for (exact) matches of those strings in database strings.
  This is afterwards extended to get the entire alignment.
- very fast local alignment heuristic, but no optimality guarantee
- output: series of local alignments, i.e. comparisons of pieces of the searched sequence with similar pieces from the database. In addition, BLAST indicates how significant of the hits that have been found.

Databases e.g. for nucleotide sequences (Genbank of NCBI, EMBL, . . .) or protein databases (SwissProt, RefSeq, Pfam, . . .).

BLAST homepage: `blast.ncbi.nlm.nih.gov`
Tutorial: `digitalworldbiology.com/BLAST`

BLAST = Basic Local Alignment Search Tool

Umbrella term for a collection of the world's most widely used programs for analyzing biological sequence data

- BLAST is used to compare experimentally determined DNA or protein sequences with sequences already existing in a database.
- Basic idea: BLAST divides query sequences into short strings and initially only looks for (exact) matches of those strings in database strings.
  This is afterwards extended to get the entire alignment.
- very fast local alignment heuristic, but no optimality guarantee
- output: series of local alignments, i.e. comparisons of pieces of the searched sequence with similar pieces from the database. In addition, BLAST indicates how significant of the hits that have been found.

Databases e.g. for nucleotide sequences (Genbank of NCBI, EMBL, . . .) or protein databases (SwissProt, RefSeq, Pfam, . . .).

BLAST homepage: `blast.ncbi.nlm.nih.gov`
Tutorial: `digitalworldbiology.com/BLAST`

BLAST“Types”

type	query	target
blastn	nucleotide	nucleotide
blastp	protein	protein
blastx	nucleotide (transl)	protein
tblastn	protein	nucleotide (transl)
tblastx	nucleotide (transl)	nucleotide (transl)

Online Example

`https://blast.ncbi.nlm.nih.gov/Blast.cgi` → nucleotide blast → copy&paste → press button BLAST

>Sequence_experimental

```
GACATTACGGCGACCCAGTCTCCCCCGGTGTTGTCAGTGGGACTGGGCC
AGACCGCAACCATCACTTGTACGGCCAGTCAAAGCATCTACAGTAACCT
TGCTTGGTACCAGCAGAGAGAAGGACAGAAGCCCTCTCTCCTGATCTAT
GCTGCGACAACGCGATACGAAGGAGTCTCCGAGCGATTCAGCGGCAGTG
GATCAGGGACCAGTTTCACCCTGACAATCAGCAACGTTCAGAATGAGGA
TGTCGCTGACTATTACTGTCAGATCGCATATTCGATCTACTCCGGTTCC
GTTGTTTTCGGTGAAGGAACCAAGCTCAGACTGAGCCGT
```

specific mRNA of a nurse shark.

Clustal

Clustal is a series of computer programs used in bioinformatics for multiple sequence alignment.

Brief History:

- Clustal (1981, first version)
- CLustalW (1994, great improvements)
- ClustalX (1997, first time with GUI)
- ClustalΩ (latest standard version, 2011)

Clustal

Basic idea explained on ClustalW (3 steps for input ζ= set $\{S_1, \ldots, S_k\}$ of sequences):

W1 Compute for all pairs $S_i, S_j \in \zeta$ a pairwise alignment $\Longrightarrow$ pairwise distances $D(S_i, S_j)$

W2 Use distance matrix D to compute phylogenetic tree T (via NeighborJoining-method)

W3 Use T to carry out a multiple alignment

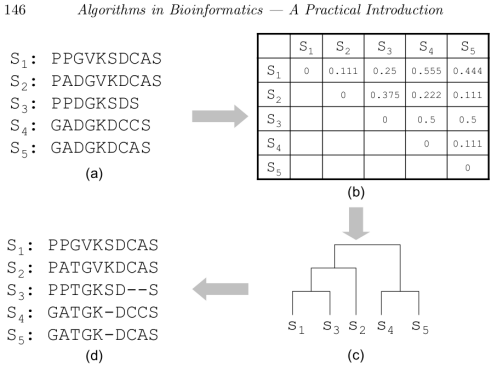


FIGURE 6.6: The three steps of ClustalW (a progressive alignment methods). Five input sequences are given in (a). Step 1 computes the pairwise distance scores for these five sequences (see (b)). Then, Step 2 generates the guide tree such that similar sequences are grouped together first (see (c)). Step 3 aligns the sequences one by one according to the branching order of the guide tree, yielding the multiple alignment of all input sequences (see (d)).

MUSCLE

MUltiple Sequence Comparison by Log-Expectation (2004)

computer software used in bioinformatics for multiple sequence alignment.

Online available via `https://www.ebi.ac.uk/Tools/msa/muscle/`

MUSCLE

MUltiple Sequence Comparison by Log-Expectation (2004)

computer software used in bioinformatics for multiple sequence alignment.

Online available via `https://www.ebi.ac.uk/Tools/msa/muscle/`

Basic idea for input ζ= set $\{S_1, \ldots, S_k\}$ of sequences (2nd and 3rd steps similar to ClustalW):

1 Compute k-mer distances
=(dis)similarities $D(S_i, S_j)$ between the sets of k-mers for all pairs $S_i, S_j \in \zeta$
Much(!) faster than [W1] in Clustal

W2 Use distance matrix D to compute phylogenetic tree T (via UPGMA-method)

W3 Use T to carry out a multiple alignment

4 Several re-iteration and refinement steps follow

MUSCLE

MUltiple Sequence Comparison by Log-Expectation (2004)

computer software used in bioinformatics for multiple sequence alignment.

Online available via `https://www.ebi.ac.uk/Tools/msa/muscle/`

Online Example:

>Sequence_1
GTTTATTAGTGATCATGGCTAAGTTTGCGTCCATCATCGCACTTCTTTTT

>Sequence_2
CTCGAGACAGTGATCATGGCTTCTCTCTCTCGTGCCGCATCTCACACC

>Sequence_3
TCTTGGTGAGGATCCGTTGAGAGTGATCATGGCTCGCCCCATCGCCCTNGTTAGA

>Sequence_4
GACATTACGGCGACCCAGTCTCCCAGTGATCATGGCTTCAGTGGGACTGGGCC