



***Molecular Sequence Algorithms, Spring 2004***  
***Lecture 9: Core String Edits and***  
***Alignments***

Pekka Kilpeläinen

University of Kuopio

Department of Computer Science



# ***III: Inexact Matching, Sequence Alignment, and Dynamic Programming***

**(Likimääräinen hahmonsovitus, sekvenssien rinnastus ja  
dynaaminen ohjelmointi)**

# Introduction to Part III

“Today, the most powerful method for inferring the biological function of a gene (or the protein that it encodes) is by sequence similarity searching on protein and DNA sequence databases.” (*W.R. Pearson, 1995*)

*Inexact* or *approximate* matching and sequence comparison are central tools in computational molecular biology. Because of . . .

- ⑥ errors in molecular data
- ⑥ trying to understand mutational evolution of the sequences

exact equality is a too rigid notion of “similarity”

Inexact matching and comparison use **alignments** (*rinnastus*) of strings to recognize their similarities

Computing alignments involves

- ⑥ considering **subsequences** (*alisekvenssi*) (instead of contiguous *substrings*), and
- ⑥ solving optimization problems (to maximize sequence similarity), often applying **dynamic programming**

# The Edit Distance Problem

The most classic inexact matching problem solved by dynamic programming: the **edit distance** problem

Common way to formalize the difference of distance of two strings  $S_1$  and  $S_2$ : Number of single-character edit operations needed to transform  $S_1$  into  $S_2$

An **edit transcript** is a sequence of operations I (insert the next char of  $S_2$ ), D (delete), R (replace by the next char of  $S_2$ ), and M (match) that transform  $S_1$  to  $S_2$

**Example:** A transcript to transform “Vintner” to “writers”:

```
RIMDMDMMI
V intner
wri t ers
```

# Definition of the Edit Distance

The **edit distance** between strings  $S_1$  and  $S_2$  is the *minimum number* of operations I, D and R in any transcript that transforms  $S_1$  into  $S_2$

⦿ also called the *Levenshtein distance*

An edit transcript with the smallest number of operations I, D and R is an **optimal transcript**

**Edit distance problem:** Compute the edit distance and an optimal transcript for the given strings  $S_1$  and  $S_2$

**NB:** The roles of  $S_1$  and  $S_2$  are symmetric, since a deletion in one string corresponds to an insertion in the other

# String Alignment

An **alignment** is an equivalent alternative to an edit transcript for indicating differences and similarities between strings

A **(global) alignment** (*kokonaisrinnastus*) of  $S_1$  and  $S_2$  is obtained by inserting spaces in the strings, and then placing them one above the other s.t. each char or space is opposite a unique char or space of the other string

- ⑥ “*global*”  $\sim$  entire strings participate in the alignment
- ⑥ (*local* alignment  $\sim$  *regions* of high similarity)

**Example:** A global alignment of “Vintner” and “writers”:

```
V _ i n t n e r _  
w r i _ t _ e r s
```

# Computing the Edit Distance

The edit distance btw strings  $S_1[1 \dots n]$  and  $S_2[1 \dots m]$  can be computed applying *dynamic programming*

Define  $D(i, j)$  to be the edit distance of prefixes  $S_1[1 \dots i]$  and  $S_2[1 \dots j]$

$\rightsquigarrow D(n, m)$  is the edit distance of  $S_1$  and  $S_2$

Dynamic programming computes  $D(n, m)$  by computing  $D(i, j)$  for all  $i \leq n$  and  $j \leq m$

# Components of dynamic programming

Dynamic programming (of the edit distance) has three essential components:

- ⑥ **Recurrence relation** (*palautuskaavat*)
  - △ How is  $D(i, j)$  determined from values  $D(i', j')$  with smaller  $i'$  and  $j'$ ?
- ⑥ **Tabular computation** (*taulukointi*)
  - △ How to memorize computed values, to avoid computing them over and over again?
- ⑥ **Traceback** (*jäljitys*)
  - △ How to find an optimal edit transcript?

# The recurrence relation

How to determine the  $D(i, j)$  values?

**Base**  $D(i, 0)$ : How to edit  $S_1[1 \dots i]$  to  $S_2[1 \dots 0] = \epsilon$ ?

With  $i$  deletions  $\rightsquigarrow D(i, 0) = i$

Similarly,  $D(0, j) = j$  ( $\leftarrow$  insert  $S_2[1], S_2[2], \dots, S_2[j]$ )

**Inductive case** for  $i, j > 0$ :

$$D(i, j) = \min \begin{cases} D(i-1, j) + 1 & \text{(D)} \\ D(i, j-1) + 1 & \text{(I)} \\ D(i-1, j-1) + t(i, j) & \text{(MR)} \end{cases}$$

where  $t(i, j) = \mathbf{if } S_1[i] = S_2[j] \mathbf{ then } 0 \mathbf{ else } 1$

# Edit distance recurrence

What do the inductive cases mean?

A transcript that transforms  $S_1[1 \dots i]$  to  $S_2[1 \dots j]$  either

- ⑥ transforms  $S_1[1 \dots i - 1]$  to  $S_2[1 \dots j]$  and deletes  $S_1[i]$  (D)
- ⑥ transforms  $S_1[1 \dots i]$  to  $S_2[1 \dots j - 1]$  and inserts  $S_2[j]$ , or (I)
- ⑥ transforms  $S_1[1 \dots i - 1]$  to  $S_2[1 \dots j - 1]$  and then matches or replaces  $S_1[i]$  by  $S_2[j]$  (MR)

An *optimal* transcript is one that minimizes over the three possibilities

**NB:** More than one of the above cases may give the same minimum  $\rightsquigarrow$  there may be many *cooptimal* transcripts

# Tabular computation of edit distance

It is easy to implement recurrences for  $D(i, j)$  as a recursive procedure

⌚ and some really try to use this!

The problem is that a recursive procedure for  $D(i, j)$  computes the same values *exponentially many times*

But there are only  $(n + 1) \times (m + 1)$  different  $D(i, j)$  values  
( $0 \leq i \leq n, 0 \leq j \leq m$ )

↔ compute them in a suitable order (bottom-up), and store them in an array so that each value is computed only once

# Bottom-up computation

Fill a table  $D(i, j)$ , where  $i = 0, \dots, n$  and  $j = 0, \dots, m$ , in an increasing order of pairs  $(i, j)$

First initialize column 0 and row 0 according to the base cases of the recurrence:

```
for  $i := 0$  to  $n$  do  $D(i, 0) = i$ ;  
for  $j := 0$  to  $m$  do  $D(0, j) = j$ ;
```

Table  $D(i, j)$  after initializing row 0 and column 0:

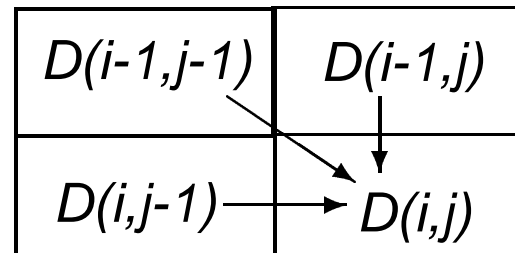
# Initialization of the dynamic programming table

$D(i,j)$	$S_2:$	w	r	i	t	e	r	s	
$S_1$		0	1	2	3	4	5	6	7
V i n t n e r	0	0	1	2	3	4	5	6	7
	1	1							
	2	2							
	3	3							
	4	4							
	5	5							
	6	6							
	7	7							

Then compute the remaining cells of the table

# Computing inner cells

Inner cells  $D(i, j)$  ( $i, j > 0$ ) can be computed in any order s.t. the three values required by the recurrence have been computed:



For example, in a row-first order:

```
for  $i := 1$  to  $n$  do  
  for  $j := 1$  to  $m$  do  
    Compute  $D(i, j)$   
    according to the recurrence;
```

# Example of row-first computation

Table after computing rows 1–3, and cols 1–3 of row 4:

$D(i, j)$	$S_2:$	w	r	i	t	e	r	s	
$S_1$		0	1	2	3	4	5	6	7
V i n t n e r	0	0	1	2	3	4	5	6	7
	1	1	1	2	3	4	5	6	7
	2	2	2	2	3	4	5	6	
	3	3	3	3	3	4	5	6	
	4	4	4	4	4				
	5	5							
	6	6							
	7	7							

# Complexity of the tabulation

Each of the  $\Theta(nm)$  cells is filled in constant time  $\rightsquigarrow$

**Theorem 11.3.2** The edit distance  $D(n, m)$  of strings  $S_1[1 \dots n]$  and  $S_2[1 \dots m]$  can be computed in  $O(nm)$  time

An optimal edit transcript can be computed within the same time bound, too

For that, store at each cell  $(i, j)$  three pointers, and set them to point to the cell(s) that gave the value for  $D(i, j)$

- ⦿ pointers are not necessary, but they are useful for explanation
- ⦿ pointers on row 0 point to the cell on the left, and pointers on column 0 to the cell above

# Example of traceback pointers

	$S_2:$	w	r	i	t	e	r	s	
$S_1$		0	1	2	3	4	5	6	7
V i n t n e r	0	0	←1	←2	←3	←4	←5	←6	←7
	1	↑1	↖1	↖←2	↖←3	↖←4	↖←5	↖←6	↖←7
	2	↑2	↖↑2	↖2	↖2	←3	←4	←5	←6
	3	↑3	↖↑3	↖↑3	↖↑3	↖3	↖←4	↖←5	↖←6
	4	↑4	↖↑4	↖↑4	↖↑4	↖3	↖←4	↖←5	↖←6
	5	↑5	↖↑5	↖↑5	↖↑5	↑4	↖4	↖←5	↖←6
	6	↑6	↖↑6	↖↑6	↖↑6	↑5	↖4	↖←5	↖←6
	7	↑7	↖↑7	↖6	↖←↑7	↑6	↑5	↖4	←5

# Finding optimal transcripts and alignments

An optimal transcript can be found (as a reversal) by following the traceback pointers from cell  $(n, m)$  to  $(0, 0)$

1. pointer ' $\leftarrow$ ' from  $(i, j)$  to  $(i, j - 1) \sim$  insertion of  $S_2[j]$
2. pointer ' $\uparrow$ ' from  $(i, j)$  to  $(i - 1, j) \sim$  deletion of  $S_1[i]$
3. pointer ' $\swarrow$ ' from  $(i, j)$  to  $(i - 1, j - 1) \sim$  match/replace, depending on whether  $S_1[i]$  and  $S_2[j]$  match or not

Alternatively, the path specifies an **optimal alignment**, where

1. pointer ' $\leftarrow$ '  $\sim$  space in  $S_1$  opposite to  $S_2[j]$ , and
2. pointer ' $\uparrow$ '  $\sim$  space in  $S_2$  opposite to  $S_1[i]$
3. pointer ' $\swarrow$ '  $\sim$   $S_1[i]$  and  $S_2[j]$  are aligned

# Finding a optimal alignments

**Example:** Optimal alignments specified by pointers of the preceding table:

1.  $S'_1$  : \_ V i n t n e r \_  
 $S'_2$  : w r i \_ t \_ e r s

2.  $S'_1$  : V \_ i n t n e r \_  
 $S'_2$  : w r i \_ t \_ e r s

3.  $S'_1$  : V i n t n e r \_  
 $S'_2$  : w r i t \_ e r s

# Complexity of tracing an optimal transcript

**Theorem 11.3.3** After computing the dynamic programming table with pointers, an optimal transcript can be found in time  $O(n + m)$

**Proof.** Starting from cell  $(n, m)$ , a path to  $(0, 0)$  can be followed by choosing *any* of the pointers: Each cell  $(i, j) \neq (0, 0)$  has at least one pointer to one of  $(i - 1, j)$ ,  $(i, j - 1)$  and  $(i - 1, j - 1)$   
 $\rightsquigarrow$  at most  $n + m$  pointers need to be followed □

The  $\Theta(nm)$  time and space requirements of the basic dynamic programming solution can be improved with more advanced techniques

# Generalizations

We can also consider edit operations with *weights* (or *costs* or *scores*):  $d$  for deletion/insertion,  $r$  for substitution, and  $e$  for match

**Operation-weight edit distance** problem is to find a transcript that transforms  $S_1$  to  $S_2$  with the *minimum total weight*

Edit distance is a special case with  $d = r = 1$  and  $e = 0$

**Example:** An alignment with  $r = 2$ ,  $d = 4$  and  $e = 1$ :

```
V i n t n e r _  
w r i t _ e r s  
weight = 2+2+2+1+4+1+1+4 = 17
```

# Computing operation-weight edit distance

Operation weights introduce straight-forward modifications to the recurrences

Base cases with operation weights:

$$D(i, 0) = i \times d$$

$$D(0, j) = j \times d$$

(←how many chars are deleted or inserted)

# Computing operation-weight edit distance (2)

Inductive case, for  $i, j > 0$ , with operation weights:

$$D(i, j) = \min \begin{cases} D(i-1, j) + d \\ D(i, j-1) + d \\ D(i-1, j-1) + t(i, j) \end{cases},$$

with  $t(i, j) = \text{if } S_1[i] = S_2[j] \text{ then } e \text{ else } r$

With these modifications, edit distance with operation weights and the corresponding transcript/alignment can be computed in time  $O(nm)$ , exactly as before

## Another generalization

The cost of edit operations can also be allowed to depend on which *characters of the alphabet* are involved  
( $\rightsquigarrow$  **alphabet-weight edit distance**)

Which are used, alphabet or operation weights?

- ⑥ Proteins are most often compared using alphabet-weights over the amino-acid alphabet
  - △ no universal agreement over the scores exist;  
PAM and BLOSUM are two dominant scoring matrices
- ⑥ DNA strings more often compared applying operation-weight (or unweighted) costs
  - △ DB search program BLAST uses  $e = +5$  and  $r = -4$

**NB:** Above applications *maximize* the score for **similarity**

# String Similarity

An alternative formalization for the relatedness of strings is their **similarity** (rather than distance), which is applied in most bio-applications

Let  $\Sigma'$  be the alphabet extended with the space '\_'

Denote the **score** of aligning chars  $x$  and  $y$  of  $\Sigma'$  by  $s(x, y)$

Let  $S'_1[1 \dots l]$  and  $S'_2[1 \dots l]$  be the versions of strings  $S_1$  and  $S_2$  that are padded with spaces for an alignment  $\mathcal{A}$

The **value (or total score) of alignment  $\mathcal{A}$**  is then

$$\sum_{i=1}^l s(S'_1[i], S'_2[i])$$

# Value on an alignment

Consider the following score matrix for  $\Sigma' = \{a, b, c, d, \_ \}$ :

<i>s</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>_</i>
<i>a</i>	1	-1	-2	0	-1
<i>b</i>		3	-2	-1	0
<i>c</i>			0	-4	-2
<i>d</i>				3	-1
<i>_</i>					0

Often  $s(x, y) \geq 0$  iff  $x = y$

Value of an alignment btw  $S_1 = cacdbd$  and  $S_2 = cabbdb$ :

$S'_1$  : c a c \_ d b d

$S'_2$  : c a b b d b \_

$$0+1-2+0+3+3-1 = 4$$

# Defining string similarity

The **similarity** of strings  $S_1$  and  $S_2$  (determined by a scoring matrix), is the value of an alignment of  $S_1$  and  $S_2$  that *maximizes* the total score, giving the **optimal alignment value**

String similarity and alphabet-weight edit distance are related but not interchangeable; We'll see that similarity is more suitable for computing *local alignments*

Similarity of strings  $S_1$  and  $S_2$  can be computed as a straight-forward modification of edit distance

For this, define  $V(i, j)$  to be the optimal alignment value of prefixes  $S_1[1 \dots i]$  and  $S_2[1 \dots j]$

# Recurrences for String Similarity

Base conditions are rather obvious:

$$V(0, j) = \sum_{k=1}^j s(-, S_2[k]), \text{ and}$$

$$V(i, 0) = \sum_{k=1}^i s(S_1[k], -)$$

(← total score of aligning chars with spaces)

The inductive case for  $i, j > 0$  similarly takes the character-specific scores into account:

# Computing String Similarity

$$V(i, j) = \max \begin{cases} V(i-1, j) + s(S_1[i], \_) \\ V(i, j-1) + s(\_, S_2[j]) \\ V(i-1, j-1) + s(S_1[i], S_2[j]) \end{cases},$$

Applying these recurrences, similarity can be computed likewise to the edit distance, obtaining  $V(n, m)$  at the lower right corner of the table

↪ The similarity and an optimal alignment of  $S_1[1 \dots n]$  and  $S_2[1 \dots m]$  can be computed in time  $O(nm)$

A variation of recurrences for  $V(i, j)$  applies to locating **approximate occurrences** of a pattern  $P$  in text  $T$