

ALGORITHMS OF BIOINFORMATICS

3

Comparing Sequences

13 November 2025

Prof. Dr. Sebastian Wild

Outline

3 Comparing Sequences

- 3.1 Sequence Alignment
- 3.2 Dynamic Programming
- 3.3 Global – Local – Semilocal
- 3.4 Affine Gap Costs & General Scores
- 3.5 Exhaustive Tabulation
- 3.6 Linear-Space Alignments
- 3.7 Adaptive Alignment
- 3.8 Multiple Sequence Alignment

3.1 Sequence Alignment

Sequence Similarity

Example: two proteins from *human hemoglobin*

Human Hemoglobin α globin subunit <https://www.uniprot.org/uniprotkb/P69905>

Human Hemoglobin β globin subunit <https://www.uniprot.org/uniprotkb/P68871>

→ essentially symmetric copies with same function

Sequences of the subunits (142 resp. 147 amino acids):

MVLSPADKTNVKAANGVKVGAHEYGEAEALERMFLSFPTTKYFPHFDLSHGSAQVKGHGKKVADALTNAVAHVDDMPNALSDLHAKHLRVDPVNFKLLSHCLLVTLAALHPAFTPAVHASLDKFASVSTVLTSKY

MVHL TPEEKSAVTALWGKVNVDEVGGEALGRLLVVPNTORFFESFGDLSTPDAVMGNPKVKAHGKKVLGAFSDGLAHLNDLKGTFTLSELHCDKLHYDENFLLGQNLVLCVLAHFFGKEFTPPVAAYOKVAGVANALAHKYH

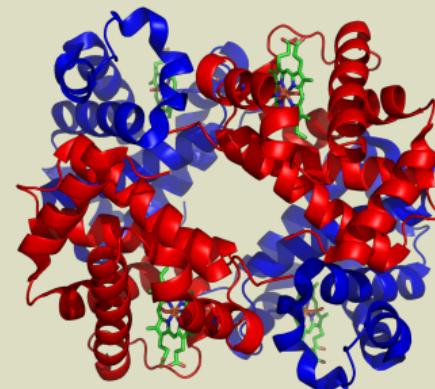
These are supposed to be “similar”!?

Alignment by EMBOSS Needle <https://www.ebi.ac.uk/jdispatcher/psa>

| = same amino acid (65x); : = similar amino acids (25x)

\rightsquigarrow 60% same

3D Structure of hemoglobin



https://commons.wikimedia.org/wiki/File:1GZX_Haemoglobin.png

String Distances

Mutations mean much in bioinformatics needs fuzzy comparisons . . .

How can we formally define these?

- ▶ This unit studies wide class of options
- ▶ Algorithmically, all are similar to deal with
- ▶ Unfortunately, general case again hard . . .
- ▶ Simplest string distance function: **Hamming distance** $d_H = \#\text{mismatches}$
 - ⚡ only defined for strings of same length
 - ▶ How about strings like this:

$A = \text{alongsharedstring}$

$B = \text{longsharedstrings}$



$$d_H(A, B) = |A| = 17$$

These are maximally different!?

\rightsquigarrow Need a more flexible notion . . .

Edit Distance

Natural idea for distances: describe **how** to get from A to B \rightsquigarrow *relative compression!*

$A[0..17] = \text{alongsharedstring}$

$B[0..17] = \text{longsharedstrings}$

“Edit script”:

0. Start with S_1 .
1. Delete $S_1[0]$
2. Insert s at end of S_1 .

\rightsquigarrow 2 character operations needed $\rightsquigarrow d_{\text{edit}}(A, B) = 2$

Edit Distance Problem

- **Given:** String $A[0..m]$ and $B[0..n]$ over alphabet $\Sigma = [0..\sigma)$.
- **Goal:** $d_{\text{edit}}(A, B) =$ minimal # symbol operations to transform A into B
operations can be insertion/deletion/substitution of single character
+ optimal edit script (with this number of operations)

Edit Distance Example

Example: edit distance $d_{\text{edit}}(A, B)$ with $A = \text{algorithm}$, $B = \text{logarithm}$?

012345678
algorithm
logarithm

Edit script:

1. Delete $A[0]$
2. Insert o after $A[1] = l$
3. Replace $A[3] = o$ by a

Compact representation of edit script: *String alignment*

0123456789
al-gorithm
- + x
- logarit hm

Formally: string over pairs of letters or *gap symbols*

$$\left\{ \begin{bmatrix} c \\ c \end{bmatrix} : c \in \Sigma \right\} \cup \left\{ \begin{bmatrix} c \\ - \end{bmatrix}, \begin{bmatrix} - \\ c \end{bmatrix} : c \in \Sigma \right\} \cup \left\{ \begin{bmatrix} c \\ c' \end{bmatrix} : c, c' \in \Sigma, c \neq c' \right\}$$

\rightsquigarrow Edit distance = # $\begin{bmatrix} c \\ - \end{bmatrix}, \begin{bmatrix} - \\ c \end{bmatrix}, \begin{bmatrix} c \\ c' \end{bmatrix}$ with $c \neq c'$

Edit Distance and Longest Common Subsequence

- ▶ Note: close relation to *longest common subsequence*
Optimal edit script \approx maximal number of matches = longest common subsequence
- ▶ But: Optimal alignment may not contain any longest common subsequence

axxa axxa axxa

| | | | | |

a ayya ayya ayy

axxaaxxaaxxa

| || ||

aayyaayyaayy

- ▶ LCS and edit distance are equivalent if we only allow insert and delete operations

3.2 Dynamic Programming

Recap: The 6 Steps of Dynamic Programming

~~> see *Efficient Algorithms*

1. Define **subproblems** (and relate to original problem)
2. **Guess** (part of solution) ~~ local brute force
3. Set up **DP recurrence** (for quality of solution)
4. Recursive implementation with **Memoization**
5. Bottom-up **table filling** (topological sort of subproblem dependency graph)
6. **Backtracing** to reconstruct optimal solution

► Steps 1–3 require insight / creativity / intuition;
Steps 4–6 are mostly automatic / same each time

~~> Correctness proof usually at level of DP recurrence

 running time too! worst case time = #subproblems · time to find single best guess

Edit Distance by DP

1. **Subproblems:** (i, j) for $0 \leq i \leq m, 0 \leq j \leq m$ compute $d_{\text{edit}}(A[0..i], B[0..j])$
2. **Guess:** What to do with last positions? (insert/delete/(mis)match)
3. **Recurrence:** $D(i, j) = d_{\text{edit}}(A[0..i], B[0..j])$

$$D(i, j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} D(i - 1, j) + 1, \\ D(i, j - 1) + 1, \\ D(i - 1, j - 1) + [A[i - 1] \neq B[j - 1]] \end{cases} & \text{otherwise} \end{cases}$$

$\rightsquigarrow O(nm)$ subproblems

- ▶ $O(1)$ time to check all guesses (per subproblem)
 - $\rightsquigarrow O(nm)$ overall time and space
- ▶ An optimal *edit script* can be constructed by a *backtrace* (see below)

Edit Distance – Step 4: Memoization

- ▶ Write **recursive** function to compute recurrence
- ▶ But *memoize* all results! (symbol table: subproblem \mapsto optimal cost)
- ~~ First action of function: check if subproblem known
 - ▶ If so, return cached optimal cost
 - ▶ Otherwise, compute optimal cost and remember it!

1. Subproblems
2. Guess!
3. DP Recurrence
4. Memoization
5. Table Filling
6. Backtrace

```
1 procedure editDist(i, j):
2     if i == 0
3         return j
4     else if j == 0
5         return i
6     end if
7     best := +∞
8      $D_i := \text{cachedED}(i, j - 1) + 1$ 
9      $D_d := \text{cachedED}(i - 1, j) + 1$ 
10     $D_m := \text{cachedED}(i - 1, j - 1) + [A[i] \neq B[j]]$ 
11    best := min{ $D_d, D_i, D_m$ }
12    return best
```

$$D(i, j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} D(i, j - 1) + 1, \\ D(i - 1, j) + 1, \\ D(i - 1, j - 1) + [A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

```
13 procedure cachedED(r[i..j], c[i..j]):
14     //  $D[0..m][0..n]$  initialized to NULL at start
15     if  $D[i][j] == \text{NULL}$ 
16          $D[i][j] := \text{editDist}(i, j)$ 
17     return  $D[i][j]$ 
```

Edit Distance – Step 5: Table Filling

- ▶ Recurrence induces a DAG on subproblems (who calls whom)
 - ▶ Memoized recurrence traverses this DAG (DFS!)
 - ▶ We can slightly improve performance by systematically computing subproblems following a fixed topological order
- ▶ Topological order here: lexicographic by (i, j)

```
1 procedure editDist(A[0..m], B[0..n]):  
2     D[0..m][0..n) :=  $\infty$  // initialize to 0  
3     for i = 0, 1, ..., m // iterate over subproblems ...  
4         for j = 0, 1, ..., n // ... in topological order  
5             if i == 0  
6                 D[i][j] := j  
7             else if j == 0  
8                 D[i][j] := i  
9             else  
10                D[i][j] := min  $\begin{cases} D[i][j - 1] + 1, \\ D[i - 1][j] + 1, \\ D[i - 1][j - 1] + [A[i - 1] \neq B[j - 1]] \end{cases}$   
11    return D[m][n]
```

- 1. Subproblems
- 2. Guess!
- 3. DP Recurrence
- 4. Memoization
- 5. Table Filling
- 6. Backtrace

- ▶ Same Θ -class as memoized recursive function
- ▶ In practice usually substantially faster
 - ▶ lower overhead
 - ▶ predictable memory accesses

Edit Distance – Step 6: Backtracing

- ▶ So far, only determine the **cost** of an optimal solution
 - ▶ But we also want the solution itself
- ▶ By *retracing* our steps, we can construct optimal edit script

```
1 procedure editScript( $A[0..m]$ ,  $B[0..n]$ ):  
2      $D[0..m][0..n] := \text{editDist}(A[0..m], B[0..n])$   
3     return traceback( $m, n$ )  
4  
5 procedure traceback( $i, j$ ):  
6     if  $i == 0$   
7         return Insert( $B[0]$ ), ..., Insert( $B[j - 1]$ )  
8     else if  $j == 0$   
9         return Delete( $A[0]$ ), ..., Delete( $A[i - 1]$ )  
10    else if  $D[i][j] == D[i][j - 1] + 1$   
11        return traceback( $i, j - 1$ ), Insert( $B[j - 1]$ )  
12    else if  $D[i][j] == D[i - 1][j] + 1$   
13        return traceback( $i - 1, j$ ), Delete( $B[i - 1]$ )  
14    else if  $A[i - 1] == B[j - 1]$   
15        return traceback( $i - 1, j - 1$ )  
16    else return traceback( $i - 1, j - 1$ ), Replace( $A[i - 1] \rightarrow B[j - 1]$ )
```

- 1. Subproblems
- 2. Guess!
- 3. DP Recurrence
- 4. Memoization
- 5. Table Filling
- 6. Backtrace

- ▶ follow recurrence a second time
- ▶ always have for running time:
backtracing = $\mathcal{O}(\text{computing } M)$
- ~~ computing optimal cost and
computing optimal solution have
same complexity

3.3 Global – Local – Semilocal

Local Alignment

So far, we assumed that we know similar regions.

How to detect significantly similar regions hidden in larger strings?

~~ Allow new edit script operations (all cost 0):

- ▶ $\text{IgnorePrefix}(A[0..i])$ free deletes at beginning
- ▶ $\text{IgnorePrefix}(B[0..j])$ free inserts at beginning
- ▶ $\text{IgnoreSuffix}(A[i..m])$ free deletes at end
- ▶ $\text{IgnoreSuffix}(B[j..n])$ free inserts at end

~~ *Local Alignment*

▶ Easy to incorporate in DP recurrence:

0. switch to maximizing score (instead min error), otherwise empty substring is best
 - ~~ Matches contribute +1 reward, rest penalty (negative score)
1. Always allow 4th option: start a new local alignment from here (at score 0)
2. Allow finish at any $D[i][j]$ ~~ free suffix

Local Alignment Recurrence

$$D(i, j) = \begin{cases} \mathbf{0} & \text{if } j = 0 \\ \mathbf{0} & \text{if } i = 0 \\ \min \begin{cases} \mathbf{0}, \\ D(i - 1, j) - 1, \\ D(i, j - 1) - 1, \\ D(i - 1, j - 1) + [A[i - 1] = B[j - 1]] \end{cases} & \text{otherwise} \end{cases}$$

Optimal local alignment score: $\max_{i \in [0..m], j \in [0..n]} D[i][j]$

Semilocal Alignment a.k.a. Fitting Alignment

Slight twist: We know conserved region, but need to find best match in larger sequence.

Say, $|A| \ll |B|$; where in B is a significant match with A ?

~ only allow IgnorePrefix($B[0..j]$) and IgnoreSuffix($B[j..n]$)

$$\rightsquigarrow D(i, j) = \begin{cases} -i & \text{if } j = 0 \\ \mathbf{0} & \text{if } i = 0 \\ \min \begin{cases} D(i - 1, j) - 1, \\ D(i, j - 1) - 1, \\ D(i - 1, j - 1) + [A[i - 1] = B[j - 1]] \end{cases} & \text{otherwise} \end{cases}$$

Optimal local alignment score: $\max_{j \in [0..n]} D[m][j]$

3.4 Affine Gap Costs & General Scores

General Scores

DP algorithm remains unchanged if we let contribution of (mis)match $A[i - 1]$ vs $B[j - 1]$ depend on used letters.

- ▶ For example, replacing amino acid with chemically similar one might not affect function
 - ~~ contributes small positive score
- ▶ replacing amino acid with dissimilar one ~~ negative score

3.5 Exhaustive Tabulation

3.6 Linear-Space Alignments

3.7 Adaptive Alignment

3.8 Multiple Sequence Alignment