ALGORITHMS OF BIOINFORMATICS

3

Comparing Sequences

13 November 2025

Prof. Dr. Sebastian Wild

CS594 (Winter 2025-26)
Philipps-Universität Marburg
version `2025-11-13 11:57`

Outline

3 Comparing Sequences

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

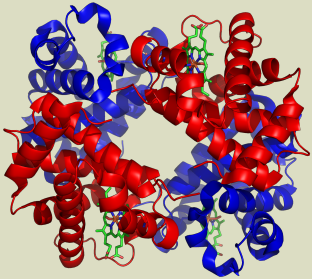
3.1 Sequence Alignment

Sequence Similarity

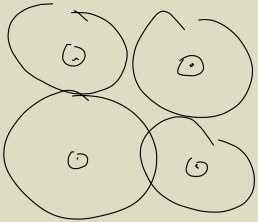
Example: two proteins from *human hemoglobin*
Human Hemoglobin α globin subunit https://www.uniprot.org/uniprotkb/P69905
Human Hemoglobin β globin subunit https://www.uniprot.org/uniprotkb/P68871

$\rightsquigarrow$ *essentially symmetric copies with same function*

3D Structure of hemoglobin



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

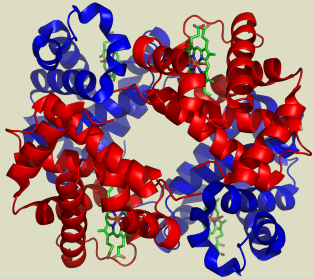


Sequence Similarity

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

$\rightsquigarrow$ *essentially symmetric copies with same function*

3D Structure of hemoglobin



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

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

MVLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHFDLSHGSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKLRVDPVNFKLLSHCLLVTLAAHLPAEFTPAVHASLDKFLASVSTVLTSKYR

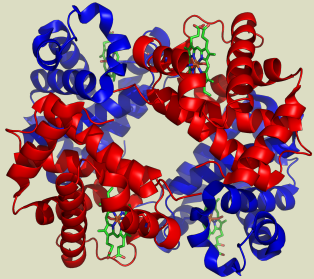
MVHLTPEEKSAVTALWGKVNVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAVMGNPKVKAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKLHVDPENFRLLGNVLVCVLAHHFGKEFTPPVQAAYQKVVAGVANALAHKYH

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

$\rightsquigarrow$ *essentially symmetric copies with same function*

3D Structure of hemoglobin



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

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

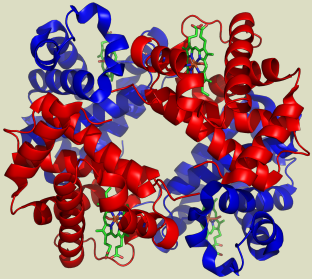
MVLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHFDLSHGSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKLRVDPVNFKLLSHCLLVTLAAHLPAEFTPAVHASLDKFLASVSTVLTSKYR

MVHLTPEEKSAVTALWGKVNVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAVMGNPKVKAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKLHVDPENFRLLGNVLVCVLAHHFGKEFTPPVQAAYQKVVAGVANALAHKYH

These are supposed to be "similar"!?

Sequence Similarity

3D Structure of hemoglobin



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

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):

```
MVLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHFDLSHGSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKLRVDPVNFKLLSHCLLVTLAAHLPAEFTPAVHASLDKFLASVSTVLTSKYR
MVHLTPEEKSAVTALWGKVNVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAVMGNPKVKAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKLHVDPENFRLLGNVLVCVLAHHFGKEFTPPVQAAYQKVVAGVANALAHKYH
```

These are supposed to be "similar"!?

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

```
MV-LSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHF-DLS-----HGSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKLRVDPVNFKLLSHCLLVTLAAHLPAEFTPAVHASLDKFLASVSTVLTSKYR
|| |:| :|: | | |||| :  | | ||| |: : :| |: :|  | |||      |: :|| ||||| | :: :||:|::    : ||:||  || ||| ||:|| : |:  || |  |||| | |:  | :| |:  |  ||
MVHLTPEEKSAVTALWGKV--NVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAVMGNPKVKAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKLHVDPENFRLLGNVLVCVLAHHFGKEFTPPVQAAYQKVVAGVANALAHKYH
```

| = same amino acid (65x); : = similar amino acids (25x) ⇝ 60% same

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 . . .

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 = #mismatches
  - ↯ only defined for strings of same length

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 = #mismatches
  - ↯ only defined for strings of same length
  - How about strings like this:

$A =$ `alongsharedstring`
$B =$ `longsharedstrings`
$\rightsquigarrow$ $d_H(A, B) = |A| = 17$ *These are maximally different!?*

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 = #mismatches
  - ↯ only defined for strings of same length
  - How about strings like this:

A = `alongsharedstring`
B = `longsharedstrings`
$\rightsquigarrow$ $d_H(A, B) = |A| = 17$ *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!*

$A[0..17) =$ `alongsharedstrings`
$B[0..17) =$ `longsharedstrings`

"Edit script":

0. Start with A_1.

1. Delete $A_1[0]$

2. Insert `s` at end of A_1.

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

Edit Distance

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

$A[0..17)$ = `alongsharedstring`
$B[0..17)$ = `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 $A = \texttt{algorithm}$, $B = \texttt{logarithm}$?

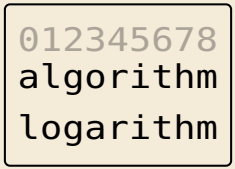
Edit Distance Example

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

```
012345678
algorithm
logarithm
```

Edit Distance Example

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



Edit script:

1. Delete $A[0]$
2. Insert `o` after $A[1] = \texttt{l}$
3. Replace $A[3] = \texttt{o}$ by `a`

Edit Distance Example

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

```
012345678
algorithm
logarithm
```

Edit script:

1. Delete $A[0]$
2. Insert `o` after $A[1] = \texttt{l}$
3. Replace $A[3] = \texttt{o}$ by `a`

Compact representation of edit script: *String alignment*

```
0123456789
al-gorithm
-|+|x|||||
-logarithm
```

Edit Distance Example

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

```
012345678
algorithm
logarithm
```

Edit script:

1. Delete $A[0]$
2. Insert `o` after $A[1] = \texttt{l}$
3. Replace $A[3] = \texttt{o}$ by `a`

Compact representation of edit script: *String alignment*

```
0123456789
al-gorithm
-|+|x|||||
-logarithm
```

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

Edit Distance and Longest Common Subsequence

- Note: close relation to *longest common subsequence*
  Optimal edit script ≈ 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

1. Define **subproblems** (and relate to original problem)
2. **Guess** (part of solution) $\rightsquigarrow$ 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

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

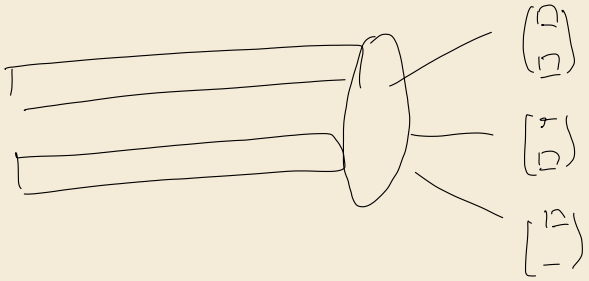
 Correctness proof usually at level of DP recurrence

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

Edit Distance by DP

original $i = m, j = n$

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



Edit Distance by DP

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

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

Edit Distance by DP

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

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

- $\rightsquigarrow$ $O(nm)$ 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)

$\rightsquigarrow$ 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

Edit Distance – Step 4: Memoization

- Write **recursive** function to compute recurrence
- But *memoize* all results! (symbol table: subproblem ↦ 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 := cachedED(i, j − 1) + 1
9      D_d := cachedED(i − 1, j) + 1
10     D_m := cachedED(i − 1, j − 1) + [A[i] ≠ B[j]]
11     best := min{D_d, D_i, D_m}
12     return best
```

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

```
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] == NULL
16         D[i][j] := editDist(i, j)
17     return D[i][j]
```

Edit Distance – Step 5: Table Filling

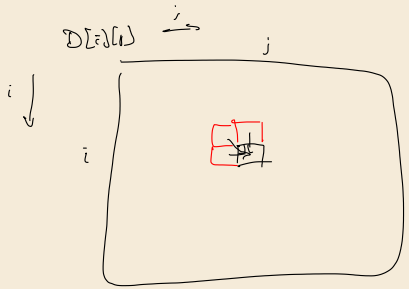
- Recurrence induces a DAG on subproblems (who calls whom)
  - Memoized recurrence traverses this DAG (DFS!)
  - We can slightly improve performance by systematically computing subproblems following a fixed topological order

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

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

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] := ∞ // initialize to ∞
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 { D[i][j − 1] + 1,
                                  { D[i − 1][j] + 1,
                                  { D[i − 1][j − 1] + [A[i − 1] ≠ B[j − 1]]
11     return D[m][n]
```

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

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

```
1  procedure editDist(A[0..m), B[0..n)):
2      D[0..m)[0..n) := ∞ // 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 { D[i][j-1] + 1,
                                  { D[i-1][j] + 1,
                                  { D[i-1][j-1] + [A[i-1] ≠ B[j-1]]
11     return D[m][n]
```

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

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) := 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] → B[j − 1])
```

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

- follow recurrence a second time

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) := 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] → 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 = O(computing M)

$\rightsquigarrow$ computing optimal cost and computing optimal solution have same complexity

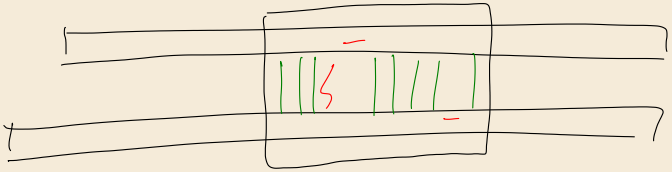
3.3 Global – Local – Semilocal

Local Alignment

So far, we assumed that we know similar regions.
How to detect significantly similar regions hidden in larger strings?

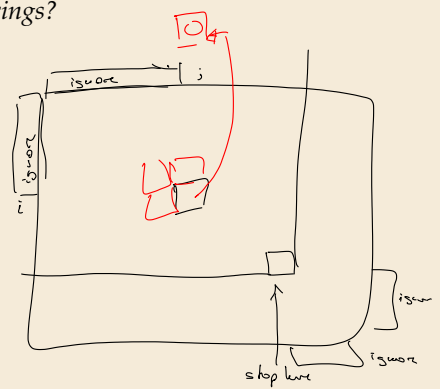
- Allow new edit script operations (all cost 0):
  - IgnorePrefix($A[0..i)$) free deletes at beginning
  - IgnorePrefix($B[0..j)$) free inserts at beginning
  - IgnoreSuffix($A[i..m)$) free deletes at end
  - IgnoreSuffix($B[j..n)$) free inserts at end

- *Local Alignment*



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):

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

⇝ *Local Alignment*

- Easy to incorporate in DP recurrence:
  - ***0.*** switch to **maximizing score** (instead min difference), 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 to finish at any $D[i][j]$ ⇝ free suffix

Local Alignment Recurrence

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

$-\left[A[i-1] \neq B[j-1]\right]$

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

Semilocal Aligment a.k.a. Fitting Alignment

Slight twist: We know conserved region, but need to find best match in larger sequence.
What substring of $B[0..n)$ is the best match for $A[0..m)$? (typically then $m \ll n$)

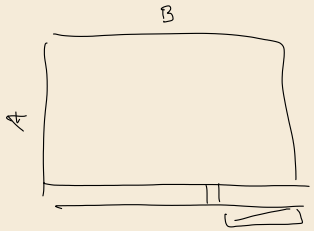
Semilocal Alignment a.k.a. Fitting Alignment

Slight twist: We know conserved region, but need to find best match in larger sequence.
What substring of $B[0..n)$ is the best match for $A[0..m)$? (typically then $m \ll n$)

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

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

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



3.4 General Scores & Affine Gap Costs

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
  $\rightsquigarrow$ contributes small positive score
- replacing amino acid with dissimilar one $\rightsquigarrow$ negative score

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
  $\rightsquigarrow$ contributes small positive score
- replacing amino acid with dissimilar one $\rightsquigarrow$ negative score

Formally, any function giving additive scores for columns $S : (\Sigma \cup \{-\})^2 \setminus \left\{\left[\begin{smallmatrix}-\\-\end{smallmatrix}\right]\right\} \to \mathbb{R}$ works.

General Alignment Score S:

- symmetric matches/substitutions matrix $p : \Sigma \times \Sigma \to \mathbb{R}$ $\qquad (p(a,b) = p(b,a))$
- gap penalty $g \in \mathbb{R}$

$\rightsquigarrow$ $S\left(\left[\begin{smallmatrix}c\\c'\end{smallmatrix}\right]\right) = p(a,b)$, $S\left(\left[\begin{smallmatrix}c\\-\end{smallmatrix}\right]\right) = S\left(\left[\begin{smallmatrix}-\\c\end{smallmatrix}\right]\right) = g$

$\rightsquigarrow$ score of alignment sum of scores of columns

BLOSOM Matrices

	C	S	T	A	G	P	D	E	Q	N	H	R	K	M	I	L	V	W	Y	F	
C	9																				C
S	-1	4																			S
T	-1	1	5																		T
A	0	1	0	4																	A
G	-3	0	-2	0	6																G
P	-3	-1	-1	-1	-2	7															P
D	-3	0	-1	-2	-1	-1	6														D
E	-4	0	-1	-1	-2	-1	2	5													E
Q	-3	0	-1	-1	-2	-1	0	2	5												Q
N	-3	1	0	-2	0	-2	1	0	0	6											N
H	-3	-1	-2	-2	-2	-2	-1	0	0	1	8										H
R	-3	-1	-1	-1	-2	-2	-2	0	1	0	0	5									R
K	-3	0	-1	-1	-2	-1	-1	1	1	0	-1	2	5								K
M	-1	-1	-1	-1	-3	-2	-3	-2	0	-2	-2	-1	-1	5							M
I	-1	-2	-1	-1	-4	-3	-3	-3	-3	-3	-3	-3	-3	1	4						I
L	-1	-2	-1	-1	-4	-3	-4	-3	-2	-3	-3	-2	-2	2	2	4					L
V	-1	-2	0	0	-3	-2	-3	-2	-2	-3	-3	-3	-2	1	3	1	4				V
W	-2	-3	-2	-3	-2	-4	-4	-3	-2	-4	-2	-3	-3	-1	-3	-2	-3	11			W
Y	-2	-2	-2	-2	-3	-3	-3	-2	-1	-2	2	-2	-2	-1	-1	-1	-1	2	7		Y
F	-2	-2	-2	-2	-3	-4	-3	-3	-3	-3	-1	-3	-3	0	0	0	-1	1	3	6	F
	C	S	T	A	G	P	D	E	Q	N	H	R	K	M	I	L	V	W	Y	F	

https://en.wikipedia.org/wiki/BLOSUM#/media/File:Blosum62-dayhoff-ordering.svg

Affine Gap costs

In sequence evolution, insertions of single stretch of k characters much more likely than k isolated (single-character) insertions
So far, we score these the same.

Affine Gap costs

In sequence evolution, insertions of single stretch of k characters much more likely than k isolated (single-character) insertions
So far, we score these the same.

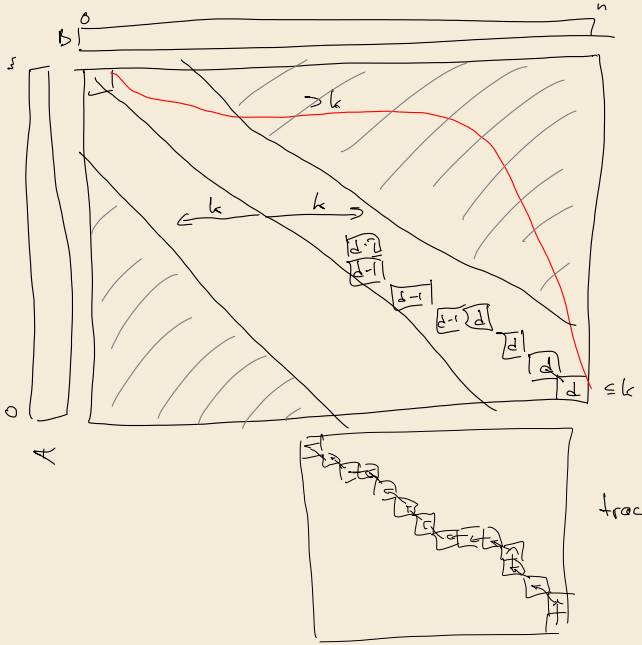
- $\rightsquigarrow$ *affine gap costs:*
  score k contiguous insertions (or k contiguous deletions) instead as $g_0 + k \cdot g$
  (usually then $g_0 \gg g$)

- If we represent contiguous insertions as $\begin{bmatrix} \vdash \\ c_1 \end{bmatrix} \begin{bmatrix} - \\ c_2 \end{bmatrix} \cdots \begin{bmatrix} - \\ c_k \end{bmatrix}$
  can assign $S(\begin{bmatrix} \vdash \\ c \end{bmatrix}) = g_0 + g$ and $S\begin{bmatrix} - \\ c \end{bmatrix} = g$.

- DP algorithm can be extended to handle these refined scores

3.5 Bounded-Distance Alignments

Good Alignment or Abort



Given: $A[0..m)$

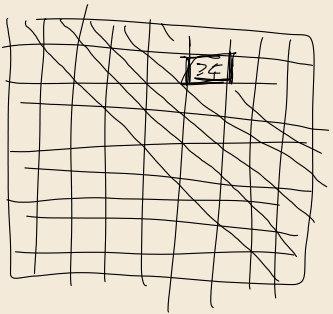
$B(0..n)$

ED score / global alignment

+ $k \in \mathbb{N}$

Goal: If $d_{edit}(A,B) \leq k$ output alignment

otherwise "large dist"



- off diagonal by d steps $\Rightarrow D \geq d$

$\Rightarrow$ only need to consider band matrix of size $n \cdot 2k$ instead of full $m \cdot n$ matrix

$\Rightarrow$ in DP recurrence, skip any option where $|j - i| > k$

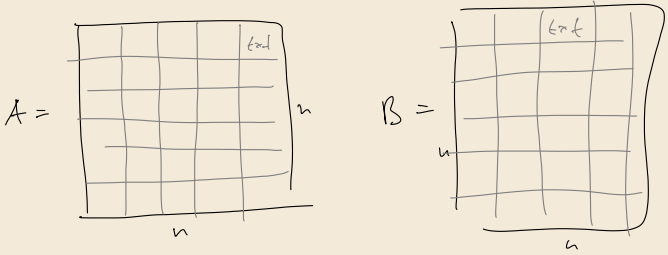
if $D[m][n] > k$ don't get correct distance!

if $\leq k$ yes!

3.6 Exhaustive Tabulation

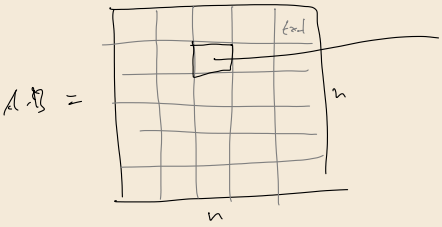
Boolean matrix multiplication

$\frac{n^3}{\log^2 n}$



$A \cdot B$ $O(n^3)$

(ignore D&C)



$\sum$ of product of $t \times t$ matrices from A & B

2^{t^2} different $\square$ ⇒ compute ALL products up front!

+ use indirect addressing