Sequence Alignment — dynamic programming - TopHat

Problem. Given two strings \( X = [x_1 \ x_2 \ldots \ x_m] \) and \( Y = [y_1 \ y_2 \ldots y_n] \) and costs \( \alpha, \delta \) find the minimum-cost alignment \( \text{Align}(X,Y) \).

prefix of string \( X \) with \( i \) characters \( X_i = X[:i] = x_1x_2\ldots x_i \)

**Question.** What are the subproblems in the DP, i.e. \( \text{OPT}(i,j) = \ldots \)?

A. min cost that exactly \( r = \min\{i,j\} \) characters of \( X \) are matched to characters in \( Y \).

B. min cost that for \( r = \min\{i,j\} \) there are at most \( r \) characters in \( X \) matched with \( r \) characters in \( Y \).

C. min cost of matching the prefix \( X_i = x_1x_2\ldots x_i \) to the prefix \( Y_j = y_1y_2\ldots y_j \).

D. min cost of matching the prefix \( X_i = x_1x_2\ldots x_i \) to the prefix \( Y_j = y_1y_2\ldots y_j \), such that \( x_i \) is assigned (matched) to \( y_j \) forcing the matching.
Sequence Alignment — dynamic programming

Problem. Given two strings $X = [x_1 \ x_2 \ldots \ x_m]$ and $Y = [y_1 \ y_2 \ldots \ y_n]$ and costs $\alpha, \delta$ find the minimum-cost alignment $\text{Align}(X,Y)$.

prefix of string $X$ with $i$ characters $X_i = X[: \ i] = x_1x_2\ldots x_i$

Idea: start by aligning short prefix sequences. Increase the length until you get to the full sequences.

subproblems for sequence alignment: for all pairs $i$ and $j$ align the prefix strings $X_i$ and $Y_j$ compute the optimal cost $\text{Align}(X_i,Y_j)$

Optimal solution to the sequence alignment problem: $\text{Align}(X_m,Y_n)$ such that $\text{cost}(\text{Align}(X_m,Y_n))$ is min

- there are $m \times n$ subproblems
- requires two dimensional recursion
- the cost parameters $\alpha, \delta$ are part of the input!
Sequence Alignment — dynamic programming

DP approach:

- make recursive calls to solving subproblems — align prefix strings of X and Y
- in order to avoid exponential many recursive calls, “memoize” (cache) the optimal solutions for subproblems

$A = m \times n$ table

- $A[i,j] = \text{Cost(Align( X_i, Y_j ))}$

Now we need to find the recursive formula.
3 cases for alignment

Align($X_3$, $Y_3$): take min cost of three options. The options are based on what the last character of each sequence is matched to.

- **match “N” to gap**: recurse on $X_2$, $Y_3$
  
- **match “T” to “N”**: recurse on $X_2$, $Y_2$
  
- **match “T” to right gap**: recurse on $X_3$, $Y_2$

We don't know whether “T” is matched to the gap between “A” and “N” or to “A”, that is decided in the recursive call.

We will make the optimal decision involving the last character of each substring.
Prefix sequence alignment

Idea. Start by aligning short prefix sequences, and increase the length until you get to the full sequence.

$$A[i, j] = \text{cost} (\text{Align}(X_i, Y_j)) = \alpha \cdot \#\text{(mismatches)} + \delta \cdot \#\text{(gaps)}$$

Let us decide what to do with characters $X_i$ and $Y_j$.
Prefix sequence alignment

Idea. Start by aligning short prefix sequences, and increase the length until you get to the full sequence.

\[ A[i, j] = \text{cost} (\text{Align}(X_i, Y_j)) = \alpha \cdot \#(\text{mismatches}) + \delta \cdot \#(\text{gaps}) \]

Case 1. the optimal solution for \(i\) and \(j\) matches \(x_i\) with \(y_j\)

\[
\text{cost}_1 = \alpha \cdot I_{x_i \neq y_j} + A[i-1, j-1]
\]

Case 2. the optimal solution for \(i\) and \(j\) leaves \(x_i\) unmatched

Case 3. the optimal solution for \(i\) and \(j\) leaves \(y_j\) unmatched

indicator of mismatch:

\[
I_{x_i \neq y_j} = \begin{cases} 
1 & x_i \neq y_i \\
0 & x_i = y_i 
\end{cases}
\]
Prefix sequence alignment

Idea. Start by aligning short prefix sequences, and increase the length until you get to the full sequence.

\[ A[i, j] = \text{cost (Align}(X_i, Y_j)) = \alpha \cdot \#(\text{mismatches}) + \delta \cdot \#(\text{gaps}) \]

Case 1. the optimal solution for \( i \) and \( j \) matches \( x_i \) with \( y_j \)

\[ \text{cost} = \alpha \cdot I_{x_i \neq y_j} + A[i-1, j-1] \]

Case 2. the optimal solution for \( i \) and \( j \) leaves \( x_i \) unmatched

\[ \text{cost} = \delta + A[i-1, j] \]

Case 3. the optimal solution for \( i \) and \( j \) leaves \( y_j \) unmatched

Idea. Start by aligning short prefix sequences, and increase the length until you get to the full sequence.

1. 2  \( \cdots \)  \( i \)  \( \cdots \)  \( m-1 \)  \( m \)

\( x \)

1 2  \( \cdots \)  \( j \)  \( \cdots \)  \( n-1 \)  \( n \)

\( y \)

\( \text{We don't know whether } y_j \text{ is matched to a gap or a char indicator of mismatch:} \)

\[ I_{x_i \neq y_j} = \begin{cases} 1 & x_i \neq y_j \\ 0 & x_i = y_j \end{cases} \]
Prefix sequence alignment

Idea. Start by aligning short prefix sequences, and increase the length until you get to the full sequence.

\[ A[i, j] = \text{cost} (\text{Align}(X_i, Y_j)) = \alpha \cdot \#(\text{mismatches}) + \delta \cdot \#(\text{gaps}) \]

Case 1. the optimal solution for \( i \) and \( j \) matches \( x_i \) with \( y_j \)

\[ \text{cost}_1 = \alpha \cdot I_{x_i \neq y_j} + A[i-1, j-1] \]

Case 2. the optimal solution for \( i \) and \( j \) leaves \( x_i \) unmatched

\[ \text{cost}_2 = \delta + A[i-1, j] \]

Case 3. the optimal solution for \( i \) and \( j \) leaves \( y_j \) unmatched

\[ \text{cost}_3 = \delta + A[i, j-1] \]

Idea. Start by aligning short prefix sequences, and increase the length until you get to the full sequence.

\[ I_{x_i \neq y_j} = \begin{cases} 1 & x_i \neq y_j \\ 0 & x_i = y_j \end{cases} \]
Prefix sequence alignment

Idea. Start by aligning short prefix sequences, and increase the length until you get to the full sequence.

$$A[i, j] = cost(\text{Align}(X_i, Y_j)) = \alpha \cdot \#(\text{mismatches}) + \delta \cdot \#(\text{gaps})$$

Case 1. the optimal solution for \(i\) and \(j\) matches \(x_i\) with \(y_j\)

$$cost_1 = \alpha I_{x_i \neq y_j} + A[i - 1, j - 1]$$

Case 2. the optimal solution for \(i\) and \(j\) leaves \(x_i\) unmatched

$$cost_2 = \delta + A[i - 1, j]$$

Case 3. the optimal solution for \(i\) and \(j\) leaves \(y_j\) unmatched

$$cost_3 = \delta + A[i, j - 1]$$

$${A[i, j]} = \min\{cost_1, cost_2, cost_3\}$$

indicator of mismatch:

$$I_{x_i \neq y_j} = \begin{cases} 1 & x_i \neq y_i \\ 0 & x_i = y_i \end{cases}$$
Dynamic programming recursion

\[
A[i, j] = \begin{cases} 
\delta \cdot i \\
\delta \cdot j \\
\min \left\{ \alpha \cdot I_{x_i \neq y_j} + A[i - 1, j - 1] \right. \\
\delta + A[i - 1, j] \\
\delta + A[i, j - 1] 
\end{cases} \quad \text{otherwise}
\]

Note that the formula above is already written wrt the memoization table \( A \).

What order can we compute the values?

\[
A = m \times n \text{ table} \\
\text{fill left \to right and top-down} \\
\text{for } i = 1, \ldots n : \\
\quad \text{for } j = 1, \ldots n \\
A[i,j] = \min \left\{ \alpha \cdot I_{x_i \neq y_j} + A[i-1,j-1], \right. \\
\delta + A[i-1,j], \\
\delta + A[i,j-1] \}
\]

\[
\text{memo} = \{ \} \\
\text{def align } (x, y, i, j) : \\
\quad \text{if } (i,j) \text{ in memo:} \\
\quad \quad \text{return memo[(i,j)]} \\
\quad \text{else:} \\
\quad \quad \quad \text{...} \\
\quad \quad \quad \text{memo[(i,j)] = min} \left\{ \alpha \cdot I + \text{align}(x,y,i,j), \right. \\
\quad \quad \quad \delta + \text{align}(x,y,i-1,j), \right. \\
\quad \quad \quad \delta + \text{align}(x,y,i,j-1) \}
\]

29
DP table for sequence alignment

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>—</th>
<th>P</th>
<th>A</th>
<th>N</th>
<th>T</th>
<th>H</th>
<th>E</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

mismatch $\alpha = 3$

gap $\delta = 2$

$$A[i,j] = \begin{cases} 2i \quad & j = 0 \\ 2j \quad & i = 0 \\ \min\{3 \cdot I_{x_i \neq y_j} + A[i-1,j-1]; 2 + A[i-1,j]; 2 + A[i,j-1]\} & \text{otherwise} \end{cases}$$
Dynamic Programming

Algorithm 1: SeqAlign(strings $X, Y$, const $\alpha, \beta$)

1. $A \leftarrow n \times m$ empty table;
2. for $i = 0 \ldots n$ do
   /* base cases */
   3. $A[i, 0] \leftarrow \delta i$;
4. for $j = 0 \ldots m$ do
   5. $A[0, j] \leftarrow \delta j$;
6. for $i = 1 \ldots n$ do
   /* recursive part */
   7. for $j = 1 \ldots m$ do
   8. $A[i, j] \leftarrow \min \{ A[i - 1, j - 1] + \alpha I_{x_i \neq y_j}; A[i - 1, j] + \delta; A[i, j - 1] + \delta \}$
9. return $A[n, m]$

Running time?

$\Theta(n \cdot m)$

The above algorithm returns the minimum value. Find an algorithm to backtrack the actual alignment.

worthwhile to store the “predecessor” (which of the three cases in the recursion) in a second table.
Backtracking the alignment

Input: X, Y, A, α, δ

Return: the alignment — a set of pairs of indices (i,j) to be matched

---

**Algorithm 1**: backtrackAlignment(X, Y, A, α, δ)

/* X, Y are strings, A is the computed memoization table, α mismatch cost, δ gap cost */

1. \( m \leftarrow \text{len}(X), n \leftarrow \text{len}(Y) \);
2. \( \text{alignment} \leftarrow \{ \} /* \text{list containing the assignments} */ \);
3. \( i \leftarrow m, j \leftarrow n; \)
4. while \( i + j > 0 \) do
   5.      if \( i == 0 \) then
      6.         \( \text{alignment} = \text{alignment} \cup (\ast, Y_j); \)
      7.         \( j = j - 1; \)
      8.      else if \( j == 0 \) then
      9.         \( \text{alignment} = \text{alignment} \cup (X_i, \ast); \)
     10.        \( i = i - 1; \)
      11. else if \( A[i, j] == \alpha \cdot I_{X_i \neq Y_j} + A[i - 1, j - 1] \) then
      12.            /* \(X_i\) and \(Y_j\) get matched */
      13.            \( \text{alignment} = \text{alignment} \cup (X_i, Y_j); \)
      14.            \( i = i - 1, j = j - 1; \)
       15. else if \( A[i, j] == \delta + A[i - 1, j] \) then
      16.            /* \(X_i\) is matched to a gap */
      17.            \( \text{alignment} = \text{alignment} \cup (X_i, \ast); \)
      18.            \( i = i - 1; \)
      19. else
      20.            /* \(Y_j\) is matched to a gap */
      21.            \( \text{alignment} = \text{alignment} \cup (\ast, Y_j); \)
      22.            \( j = j - 1; \)

20. return \( \text{alignment} \)

---

Exercise: backtrack the solution on the example.

notation: \((x_i, \ast)\) indicates that character i was matched to a gap.
Single-destination shortest paths problem

Let \( G(V,E) \) be a directed graph.

We are given some weight \( w(u,v) \) of each directed edge \( (u,v) \).
- we call it length, weight or cost (depends on the application)
- the weight can be any numerical value

The length (weight, cost) of a path is the sum of the weights along its edges.

Problem: find the shortest weighted path from each node/source \( s \) to \( t \)

\[
\text{length of path } 0 \rightarrow 6 = 9 + 4 + 13 = 26
\]
Shortest paths and negative edges

edge weights are non-negative
• shortest path always exists

edge weight may be negative
• least cost path may or may not exist

\( n = \# \text{ of nodes in a graph} \)

\[
\begin{align*}
\text{dist}(a, c) &= 3 \\
\text{dist}(a, c) &= -\infty
\end{align*}
\]

\( \text{you can't interpret } -\infty, \text{ hence we can't allow negative cycles} \)
Shortest paths and negative cycles

The definition of the minimum weight path is not always meaningful when negative weight edges are present. Specifically, a path of weight \(-\infty\) cannot be interpreted.

Claim. If some path from \(v\) to \(t\) contains a negative cycle, then there does not exist a lowest weight path from \(v\) to \(t\).

negative cycle = a cycle along which the sum of weights is negative

Proof: Each additional transversal of the cycle would decrease the length even more \(\Rightarrow\) no lower bound.
currency exchange — application with negative edge weights

What is the best way to convert USD to Euros?
• n different currencies
• exchange rates between them
represent this as a least-cost path problem:
currency exchange – application with negative edge weights

There is something funny about this example!

\[ \text{CA$} \rightarrow E \rightarrow ¥ \rightarrow \text{CA$} \]

\[ $10 \rightarrow 3 \rightarrow 168 \rightarrow 10.08 \]
Design the algorithm - recursive computation

subproblems:

OPT(i,v) = minimum cost of path from a node v to t using \textit{at most} i edges
TopHat

subproblems:
OPT(i,v) = minimum cost of path from a node v using at most i edges

Select all correct values:
A. OPT(1,a) = 15
B. OPT(2,e) = ∞
C. OPT(4,b) = 13
D. OPT(4,a) = 15
E. OPT(4,a) = 7
F. OPT(3,b) = 16
Design the algorithm - idea

Suppose the least-cost path from s to t exists.

subproblems:

\[ \text{OPT}(i,v) = \text{minimum cost of path from a node } v \text{ to } t \text{ using at most } i \text{ edges} \]

\[ \uparrow \text{ node id} \]
\[ \# \text{ of subproblems} \]

**fact.** a simple path in a graph contains at most n-1 edges.

\[ \text{OPT}(n-1,v) = \text{length of the shortest (simple) path from } v \text{ to } t \]

Is it ok to limit ourselves to simple paths?

Yes, we don't want negative cycles
Design the algorithm - recursive computation

subproblems:
OPT(i,v) = minimum cost of path from a node v to t using \textit{at most} i edges

solve problem recursively: divide possible solutions (= paths) based on the first edge on the path from v to t.
• the path with i edges from v to t using edge \((v,u)\) is the concatenation of the path from u to t (with i-1 edges) and edge \((v,u)\)
Recursive formula for DP

\[ \text{OPT}(i,v) = \text{shortest path } P \text{ from } v \text{ to } t \text{ using } \text{at most } i \text{ edges} \]

**case 1: \( P \) contains at most \( i-1 \) edges**
- it does not help to have an extra edge
- cost = cost of a path with less than \( i \) edges
- cost = \( \text{OPT}(i-1,v) \)

**case 2: \( P \) uses exactly \( i \) edges**
- what are the possible subproblems to take the min over?

consider the weight: \( c(v,w) + \text{OPT}(i-1, w) \)

\[ \text{OPT}(i,v) \]

\[ w \in G[v] \]

**base case:**
\[ \text{base cases: } \begin{cases} 0, & \text{if } v = t \\ \infty, & \text{if } i = 0, v \neq t \end{cases} \]
What is the recursive formula to compute the least cost from v to t using at most i edges? You may assume that the two base cases are:

$$OPT(0, v) = 0 \text{ if } v = t \quad OPT(0, v) = +\infty \text{ if } v \neq t$$

A. $$OPT(i, v) = \min \{ OPT(i, v); \min_{w \in G[v]} OPT(i - 1, w) \}$$

B. $$OPT(i, v) = \min \{ OPT(i - 1, v); \min_{w \in G[v]} c(v, w) + OPT(i - 1, w) \}$$

C. $$OPT(i, v) = \min \{ OPT(i, v); \min_{w \in G[v]} c(v, w) + OPT(i, w) \}$$

D. $$OPT(i, v) = \min \{ OPT(i - 1, v); \min_{w \in G[v]} OPT(i - 1, w) \}$$