Bellman’s principle of optimality

"An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."

How similar are two strings?

$S_1 = \text{tentation}$  $S_2 = \text{temptation}$

Pairwise sequence alignment

**Definition** (Pairwise sequence alignment)

Let $a=a_1…a_m$ and $b=b_1…b_n$ be two sequences over an alphabet $\Sigma$ (i.e. $a, b \in \Sigma^*$). A pairwise alignment is a mapping $f$ of the letters of $a$ to $b$, such that if $f(a_i, b_j)$ and $f(a_k, b_l)$ then $i< k$ & $j<l$ or $k<i$ & $l<j$.

Example:

- $a=\text{ABBCDE}$, $b=\text{BBCDDE}$
- $\text{ABBC-EE, ABB-C-EE, ABBCDE, A-B-C-E-E=}$
- $\text{-BBCCDE=, -BBCCD-E, BBCCDE, B-B-C-C-D-E}$

Note: letters can be mapped to an empty character.

Question: All are valid alignments, but which one is best?

**Vocabulary**

- **Match**: letters are identical
- **Substitution**: letters are different
- **Insertion**: a letter of $b$ is mapped to the empty character
- **Deletion**: a letter of $a$ is mapped to the empty character

Insertions & deletions are also called *indels*.

Example: $a=\text{ABBCEE}$, $b=\text{BBCCDE}$

- $\text{ABB-CEE, } | | : |$
- $\text{-BBCCDE}$

**Notations**

The empty character/letter is noted -

An alignment can be decomposed into column (or vectors).

$$
\begin{array}{l}
\begin{vmatrix}
 a_1 & a_2 & \ldots & a_i & a_{i+1} & a_{i+2} & \ldots \\
 b_1 & b_2 & \ldots & b_i & b_{i+1} & b_{i+2} & \ldots \\
\end{vmatrix}
\end{array}
$$

- $a_i, a_{i+1}$
- $b_i, b_{i+1}$
Bioinformatics

Let \( a \) and \( b \) be two homologous biological sequences (DNA, RNA, Protein).

A sequence alignment measure the similarity between the 2 sequences and can be used to understand evolution.

\[
\text{ACCAGTAGCGGGCGACA} \rightarrow \text{GACCTGCAT}
\]

\[
\text{ACC} \rightarrow \text{TAGGGGGGACATTTGACGACGC} \\
\]

Counting alignments (1)

Let \( a = a_1 \ldots a_m \) and \( b = b_1 \ldots b_n \) be two sequences over an alphabet \( \Sigma \). Let \( c(m,n) \) be the number of alignments that can be formed between them.

First, we note that an alignment of \( a \) and \( b \) must end by:

\[
c(m,n) = c(m-1,n) + c(m-1,n-1) + c(m,n-1)
\]

(i.e. total number of alignments is the sum of (1) the number of alignments ending with a deletion, plus (2) the number of alignment ending with a match/mismatch, plus (3) the number of alignments ending with an insertion)

Counting alignments (2)

We have a recursion:

\[
c(m,n) = c(m-1,n) + c(m-1,n-1) + c(m,n-1)
\]

Initialization?

\( f(0,n) = f(m,0) = f(0,0) = 1 \)

Recursive evaluation (top-down):

\[
\begin{align*}
5 + 3 + 5 &= 13 \\
1 + 1 + 3 &= 5 \\
1 + 1 + 1 &= 3 \\
1 + 1 + 1 &= 3
\end{align*}
\]

Counting alignments (3)

\[
\begin{align*}
f(1,1) &= f(0,1) + f(1,0) = 1 \\
f(1,2) &= f(0,2) + f(1,1) = 2 \\
f(2,1) &= f(1,1) + f(2,0) = 3 \\
f(2,2) &= f(1,2) + f(2,1) = 5
\end{align*}
\]

Note: \( f(1,1) \) appears 3 times, and evaluated 3 times...

How to speed up this calculation?

- Memoization
- Dynamic programming

Counting alignments (4)

\[
c(m,n) = c(m-1,n) + c(m-1,n-1) + c(m,n-1)
\]

- Indices of \( c \) are strictly decreasing during the recursion
- We can compute \( c \) for smaller indices first (bottom-up)
- Define a partial order on the \( c \) such that \( c(i,j) \leq c(i',j') \) if \( i \geq i' \) or \( j \geq j' \)
- Compute \( c \) using this partial order:

\[
\begin{align*}
\text{for } i = 0 \text{ to } m \\
\text{for } j = 0 \text{ to } n \\
c(i,j) &= c(i-1,j) + c(i-1,j-1) + c(i,j-1)
\end{align*}
\]

- Complexity: \( O(mn) \)
Optimal pairwise alignment

\[ d(\text{ABBCEE}, \text{BBCCDE}) \]

\[
\begin{align*}
\text{ABB} & \rightarrow \text{EE} & \text{ABB} & \rightarrow \text{C} \rightarrow \text{EE} & \text{ABBEE} & \rightarrow \text{B} = \text{C} = \text{E} = \text{E} \\
-\text{BBCCDE} & \rightarrow -\text{BBCCD} \rightarrow -\text{E} & \text{BBCCDE} & \rightarrow -\text{B} = -\text{B} = -\text{C} = -\text{D} = -\text{E} \\
0 & = 4 \times 1 + 1 \times 1 + (-1) \times 1 + 1 \times 1 + (-1) = 1 \\
\end{align*}
\]

Among all alignments, which one is the best?

Levenshtein distance

\[ d_{\text{Levenshtein}}(x,y) \]

**Definition (Levenshtein Distance)**

The Levenshtein Distance between two words/sequences is the minimal number of substitutions, insertions and deletions to transform one into the other.

Example:

- \( \text{ABBCEE} \)
- \( -\text{BBCCDE} \)

1 deletion + 1 insertion + 1 substitution \( \Rightarrow d = 3 \)

Edit cost/distance

**Definition (edit cost)**

Let \( \delta(x,y) \) be a cost function for each edit operation (match, substitution, insertion, deletion). The edit cost of two words/sequences is the sum of the cost of each edit operation used to transform one into the other.

**Definition (edit distance)**

The edit distance between two words/sequences is the maximal (or minimal) cost to transform one into the other.

**Example:**

\[
\delta(x,y) =
\begin{cases}
1 & \text{if } x = y \\
-1 & \text{otherwise}
\end{cases}
\]

4 match + 1 deletion + 1 insertion + 1 substitution

\( \Rightarrow d = 4 \times 1 + 1 \times 1 + (-1) \times 1 + 1 \times 1 + (-1) = 1 \)

Edit distance

If

- Every edit operation has positive cost
- for every operation, there is an inverse operation with equal cost

Then, the edit distance is a metric:

- \( d(x,y) \geq 0 \) (separate axiom)
- \( d(x,y) = 0 \) iff \( x = y \) (coincidence axiom)
- \( d(x,y) = d(y,x) \) (symmetry)
- \( d(x,y) \leq d(x,z) + d(z,y) \) (triangle inequality)

Optimal sub-structure

"A sub-alignment of an optimal alignment w.r.t. the edit cost is also optimal"

**Proof:** cut-and-paste argument & contradiction

- Let A be an optimal alignment
- Let \( \Lambda = \Lambda_1 \Lambda_2 \Lambda_3 \) be a decomposition of A such that \( \Lambda_2 \) is not optimal.
- Let \( \Lambda' \) be an optimal alignment of the substrings in \( \Lambda_3 \)
- Substitute \( \Lambda_2 \) by \( \Lambda' \) to build a new alignment \( \Lambda' \)

\[
\delta(\Lambda) = \delta(\Lambda_1 \Lambda_2 \Lambda_3) = \delta(\Lambda_1) + \delta(\Lambda_2) + \delta(\Lambda_3)
\]

> \( \delta(\Lambda_1) + \delta(\Lambda_2) + \delta(\Lambda_3) = \delta(\Lambda_1 \Lambda_3) = \delta(\Lambda) \)

- contradiction with \( \Lambda \) optimal

Problem Structure

**Definition (dynamic array):**

\( d_{ij} \): maximal cost of aligning prefix strings \( a_1 \ldots a_i \) and \( b_1 \ldots b_j \)

**Case 1** (\( a_i \) matches \( b_j \))

Pay cost of (mis)match of \( a_i \) with \( b_j \) + max cost of aligning \( a_1 \ldots a_{i-1} \) and \( b_1 \ldots b_{j-1} \).

**Case 2a** (\( a_i \) deleted)

Pay cost of deletion of \( a_i \) + max cost of aligning \( a_1 \ldots a_{i-1} \) and \( b_1 \ldots b_j \).

**Case 2b** (\( b_j \) inserted)

Pay cost of insertion of \( b_j \) + max cost of aligning \( a_1 \ldots a_i \) and \( b_1 \ldots b_{j-1} \).
Recursion

\[
d(i, j) = \begin{cases} 
  j \cdot \delta(-, -) & \text{if } i = 0 \\
  \delta(a_i, b_j) + d(i-1, j-1) & \text{otherwise} \\
  \delta(-, b_j) + d(i, j-1) \\
  i \cdot \delta(-, -) & \text{if } j = 0
\end{cases}
\]

Needleman-Wunch Algorithm

\[
\text{for } i=0 \text{ to } m \text{ do} \\
\text{d}(i, 0) = i \cdot \delta(-, -) \\
\text{for } j=0 \text{ to } n \text{ do} \\
\text{d}(0, j) = j \cdot \delta(-, -) \\
\text{for } i=1 \text{ to } m \text{ do} \\
\text{for } j=1 \text{ to } n \text{ do} \\
d(i, j) = \max(d(i-1, j)+\delta(a_i, -), d(i-1, j-1)+\delta(a_i, b_j), d(i, j-1)+\delta(-, b_j)) \\
\text{return } d(m, n)
\]

Example

a=ATTG b=CT

\[\delta(x, y) = \begin{cases} 
  1 & \text{if } x = y \\
-1 & \text{otherwise}
\end{cases}\]

<table>
<thead>
<tr>
<th></th>
<th>-</th>
<th>A</th>
<th>T</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0</td>
<td>-1</td>
<td>-2</td>
<td>-3</td>
<td>-4</td>
</tr>
<tr>
<td>C</td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[\text{match/substitution: } d(0,0) + \delta(A, C) = 0 + (-1) = -1\]

Example

a=ATTG b=CT

\[\delta(x, y) = \begin{cases} 
  1 & \text{if } x = y \\
-1 & \text{otherwise}
\end{cases}\]

<table>
<thead>
<tr>
<th></th>
<th>-</th>
<th>A</th>
<th>T</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0</td>
<td>-1</td>
<td>-2</td>
<td>-3</td>
<td>-4</td>
</tr>
<tr>
<td>C</td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[\text{match/substitution: } d(0,0) + \delta(A, C) = 0 + (-1) = -1\]

\[\text{insertion: } d(1, 0) + \delta(-, C) = -1 + (-1) = -2\]

\[\text{deletion: } d(0, 1) + \delta(A, -) = -1 + (-1) = -2\]
Example

\[ \delta(x,y) = \begin{cases} 1 & \text{if } x = y \\ -1 & \text{otherwise} \end{cases} \]

<table>
<thead>
<tr>
<th></th>
<th>-</th>
<th>A</th>
<th>T</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0</td>
<td>-1</td>
<td>-2</td>
<td>-3</td>
<td>-4</td>
</tr>
<tr>
<td>C</td>
<td>-1</td>
<td>-1</td>
<td>-2</td>
<td>-3</td>
<td>-4</td>
</tr>
<tr>
<td>T</td>
<td>-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- match/substitution: \( d(0,0) + \delta(A,C) = 0 + (-1) = -1 \)
- insertion: \( d(1,0) + \delta(-,C) = -1 + (-1) = -2 \)
- deletion: \( d(0,1) + \delta(A,-) = -1 + (-1) = -2 \)
Example

\(a=\text{ATTG} \quad b=\text{CT}\)

\[\delta(x,y) = \begin{cases} 1 & \text{if } x = y \\ -1 & \text{otherwise} \end{cases}\]

<table>
<thead>
<tr>
<th></th>
<th>-</th>
<th>A</th>
<th>T</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0</td>
<td>-1</td>
<td>-2</td>
<td>-3</td>
<td>-4</td>
</tr>
<tr>
<td>C</td>
<td>-1</td>
<td>-1</td>
<td>-2</td>
<td>-3</td>
<td>-4</td>
</tr>
<tr>
<td>T</td>
<td>-2</td>
<td>-2</td>
<td>0</td>
<td>-1</td>
<td>-2</td>
</tr>
</tbody>
</table>

Backtracking

How to retrieve the optimal alignment?

- Each move is associated to one edit operation
  - Vertical = insertion
  - Diagonal = match/substitution
  - Horizontal = deletion
- We use one of these 3 move to fill a cell of the array
  - From the bottom-right corner (i.e. \(d(m,n)\)), find the move that has been used to determine the value of this cell.
  - Apply this principle recursively.
Example

\[
\alpha = \text{ATTG} \quad \beta = \text{CT}
\]

\[\delta(x, y) = \begin{cases} 
1 & \text{if } x = y \\
-1 & \text{otherwise}
\end{cases}
\]

\[
\begin{array}{c|cccc}
   & A & T & T & G \\
- & - & 0 & -1 & -2 -3 -4 \\
C & -1 & -1 & -2 & -3 -4 \\
T & -2 & -2 & 0 & -1 -2 \\
\end{array}
\]

\[
\text{TTG} \quad \text{C-T-}
\]

Example

\[
\alpha = \text{ATTG} \quad \beta = \text{CT}
\]

\[\delta(x, y) = \begin{cases} 
1 & \text{if } x = y \\
-1 & \text{otherwise}
\end{cases}
\]

\[
\begin{array}{c|cccc}
   & A & T & T & G \\
- & - & 0 & -1 & -2 -3 -4 \\
C & -1 & -1 & -2 & -3 -4 \\
T & -2 & -2 & 0 & -1 -2 \\
\end{array}
\]

\[
\text{ATTG} \quad \text{ATTG} \quad \text{ATTG} \quad \text{C-T-} \quad \text{CT--} \quad \text{CT--} \quad \text{CT--}
\]

Analysis

Theorem. The dynamic programming algorithm computes the edit distance (and optimal alignment) of two strings of length \(m\) and \(n\) in \(O(mn)\) time and \(O(mn)\) space.

Proof.
* Algorithm computes edit distance.
* Can trace back to extract optimal alignment itself.

Q. Can we avoid using quadratic space?
A. Easy to compute optimal value in \(O(mn)\) time and \(O(m+n)\) space.
* Compute \(OPT[1, 1]\) from \(OPT[1, 1, 1, 1]\).
* But, no longer easy to recover optimal alignment itself.

Knapsack problem

- Given \(n\) objects and a "knapsack."
- Item \(i\) weight \(w_i > 0\) and has value \(v_i > 0\).
- Knapsack has capacity \(W > 0\).
- Goal: fill knapsack so as to maximize total value.

Ex. (1,2,3) has value 35.
Ex. (3,4) has value 40.
Ex. (5,5) has value 46 (but exceeds weight limit).

Knapsack instance

\[
\begin{array}{cccc|c}
   i & w_i & v_i & \text{knapsack instance weight limit } W = 10 \\
1 & 1 & 1 & \\
2 & 2 & 4 & \\
3 & 1 & 2 & \\
4 & 2 & 3 & \\
5 & 1 & 2 & \\
\end{array}
\]

Greedy by value. Repeatedly add item with maximum \(v_i/w_i\).
Greedy by weight. Repeatedly add item with minimum \(w_i\).
Greedy by ratio. Repeatedly add item with maximum ratio \(v_i/w_i\).

Observation. None of greedy algorithms is optimal.

False start...

Def. \(OPT[i] = \text{max profit subset of items } 1, \ldots, i\).

Case 1. \(OPT[i] = 0\) does not select item \(i\).
* \(OPT[i] = \text{best of } \{OPT[j], j = 1, \ldots, i-1\}\).

Case 2. \(OPT[i] = \text{selects item } i\).
* Selecting item \(i\) does not immediately imply that we will have to reject other items.
* Without knowing what other items were selected before \(i\), we don’t even know if we have enough room for \(i\).

Conclusion. Need more subproblems!
New variable

Def. \( OPT(i, w) = \text{max profit subset of items } 1, \ldots, i \text{ with weight limit } w. \)

Case 1. \( OPT \) does not select item \( i \).
- \( OPT \) selects best of \( 1, 2, \ldots, i-1 \) using weight limit \( w \).

Case 2. \( OPT \) selects item \( i \).
- New weight limit is \( w - w_i \).
- \( OPT \) selects best of \( 1, 2, \ldots, i-1 \) using this new weight limit.

\[
OPT(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
OPT(i-1, w) & \text{if } w > w_i \\
\max \{ OPT(i-1, w), v_i + OPT(i-1, w - w_i) \} & \text{otherwise}
\end{cases}
\]

Dynamic programming algorithm

\textbf{KnapSack} \((n, W, w_1, \ldots, w_n)\)

\begin{align*}
\text{FOR } w = 0 \text{ TO } W \\
M[0, w] & = 0.
\end{align*}

\begin{align*}
\text{FOR } i = 1 \text{ TO } n \\
\text{FOR } w = 1 \text{ TO } W \\
\text{IF } (w > w_i) \\
M[i, w] & = M[i-1, w]. \\
\text{ELSE} \\
M[i, w] & = \max \{ M[i-1, w], v_i + M[i-1, w - w_i] \}.
\end{align*}

\text{RETURN } M[n, W].

Example

<table>
<thead>
<tr>
<th>( n )</th>
<th>( w )</th>
<th>( OPT(i, w) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

\text{OPT}(i, w) = \text{max profit subset of items } 1, \ldots, i \text{ with weight limit } w.

\text{Analysis}

\textbf{Theorem.} There exists an algorithm to solve the knapsack problem with \( n \) items and maximum weight \( W \) in \( \Theta(nW) \) time and \( \Theta(nW) \) space.

\textbf{Proof.}
- Takes \( \Theta(i) \) time per table entry.
- There are \( \Theta(nW) \) table entries.
- After computing optimal values, can trace back to find solution:
  - take item \( i \) in \( OPT(i, w) \) if \( M[i, w] < M[i-1, w] \).

\textbf{Remarks.}
- Not polynomial in input size.
- Decision version of knapsack problem is \textit{NP}-Complete. [\textit{Chapter B}]
- There exists a poly-time algorithm that produces a feasible solution that has value within 1\% of optimum. [\textit{Section 11.8}]

[Image of table]

[Image of diagram]