1 Alignment Dynamic Programming

1.1 Score Function

Let’s remember the problem we want to solve. Given two sequences, we want to find a way to align them such that the distance between the two aligned sequences is minimal. An example of alignment is given in Figure 1.

We first have to define a distance function between 2 strings (sequences) \( \sigma: (\Sigma \cup \{-\})^2 \rightarrow \mathbb{R} \).

In our case we have two DNA sequences: \( S = AGT \) and \( T = AACT \). Our Alphabet \( \Sigma = \{A,C,G,T\} \) and the distance function is:

\[
\sigma(x, y) = \begin{cases} 
+1 & x = y, x, y \in \Sigma \\
-1 & x! = y, x, y \in \Sigma \\
-2 & x =' -' y =' -', x = y
\end{cases}
\]

Let’s define the group of legal assignments between \( S \) and \( T \) as \( A(S, T) \). \( n, m \) are the lengths of \( S \) and \( T \) respectively and \( N \) is length of the alignment. Let’s define

\[
Score(a) := \sum_{i=1}^{N} \sigma(a_{1,i}, a_{2,i})
\]

Figure 1: Alignment of 2 DNA sequences
Denote
\[ a^* := \text{argmax}_{a \in A} \text{score}(a) \]

Thus \( L = \text{score}(a^*) \) is the optimal solution of the problem.

### 1.2 Dynamic Programming

Dynamic programming is a solution we use to solve problems that have specific structure. We can break down the problem into simpler problems and we can use the solutions of this subproblems just once, to solve the original one. The solutions of the subproblems are stored, so when we meet the same subproblem again, we don't recompute its solution.

In our case, we can calculate the best alignment between \( S_{1, i} \) and \( T_{1, j} \), for all \( 1 \leq i \leq n \) and \( 1 \leq j \leq m \).

For example, we know \( a^* \) belongs to one of the following:

\[
\begin{array}{c|c|c}
S_n & S_n & - \\
T_m & - & T_m \\
\end{array}
\]

We can compute:
\[
L_{n,m} = \max \left\{ \begin{array}{l}
L(S_{1..n-1}, T_{1..m-1}) + \sigma(S_{n}, T_{m}) \\
L(S_{1..n-1}, T_{1..m}) + \sigma(S_{n}, -') \\
L(S_{1..n}, T_{1..m-1}) + \sigma(-', T_{m})
\end{array} \right.
\]

Using this formula recursively we can compute the score of the best alignment.

### 1.3 Global Alignment - Needleman-Wunsch

We fill a table \( V \in M_{n+1,m+1} \).

We fill \( V \) such that the cell \( V_{i,j} \) contains the best score for an alignment between \( S_{1..i} \) and \( T_{1..j} \): \( V_{i,j} = L(S_{1..i}, T_{1..j}) \)

We use the next recursion formula:
\[
V_{i,j} = \max \left\{ \begin{array}{l}
L(S_{1..i-1}, T_{1..j-1}) + \sigma(S_{i}, T_{j}) \\
L(S_{1..i-1}, T_{1..j}) + \sigma(S_{i}, -') \\
L(S_{1..i}, T_{1..j-1}) + \sigma(-', T_{j})
\end{array} \right.
\]

We get a recursion formula, allowing us to fill each cell \( V_{i,j} \) in \( O(1) \) if the cells \( V_{i-1,j}, V_{i,j-1}, V_{i-1,j-1} \) are already filled.

Thus, \( L = V_{n,m} \). We fill \( V \) as follows:
Initialization: Fill $V_{0,0} = 0$.
Next, assign values for the first row and column as follows:
$V(0, i) = V(0, i - 1) - 2$ and $V(j, 0) = V(j - 1, 0) - 2$.

<table>
<thead>
<tr>
<th></th>
<th>-</th>
<th>A</th>
<th>A</th>
<th>C</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0</td>
<td>-2</td>
<td>-4</td>
<td>-6</td>
<td>-8</td>
</tr>
<tr>
<td>A</td>
<td>-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>-4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>-6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Initialization of $V$

Fill: Fill $V$ column by column, from top to bottom, left to right, according to the recursion formula.
We want to be able to trace-back the path.
To do so, at each cell, we keep a pointer to the cell that leads to its score. We fill a parallel matrix, indicating the pointer chosen.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>A</th>
<th>A</th>
<th>C</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0</td>
<td>$\rightarrow$-2</td>
<td>$\rightarrow$-4</td>
<td>$\rightarrow$-6</td>
<td>$\rightarrow$-8</td>
</tr>
<tr>
<td>A</td>
<td>-2</td>
<td>↓$\rightarrow$1</td>
<td>$\rightarrow$-1</td>
<td>$\rightarrow$-3</td>
<td>$\rightarrow$-5</td>
</tr>
<tr>
<td>G</td>
<td>-4</td>
<td>↓$\rightarrow$1</td>
<td>$\rightarrow$-0</td>
<td>$\rightarrow$-2</td>
<td>$\rightarrow$-4</td>
</tr>
<tr>
<td>T</td>
<td>-6</td>
<td>↓$\rightarrow$3</td>
<td>$\rightarrow$-2</td>
<td>$\rightarrow$-1</td>
<td>$\rightarrow$-1</td>
</tr>
</tbody>
</table>

Table 2: $V$ filled

Trace-Back The score of the optimal alignment is in the cell $V_{n,m}$.
To backtrace the path, we follow the pointers from $V_{n,m}$, until we reach $V_{0,0}$.
We get that the optimal alignment is:

```
A | G | - | T
A | A | C | T
```

Optimal alignment
1.4 Local Alignment - Smith-Waterman

1.4.1 Aligning a Strand to the Genome

We want to align a strand of RNA, for example a gene (approximately \(10^3\) bases), to the human genome (\(\approx 3 \cdot 10^9\)). We want only the region of the gene to align. We don’t want to penalize gaps at the start and at the end of the sequence.

See an example of the desired result below.

We fill \(V\) as follows:

**Initialization:** Fill \(V_{0,0} = 0\).

Next, we fill the first column as follows:
\[
V(j, 0) = V(j - 1, 0) - 2
\]

We assign 0 to the first row. Thus, we don’t penalize gaps for the beginning of the strand.

![Initialization of V](image)

**Fill:** Fill \(V\) column by column, from top to bottom, left to right, according to the recursion formula, as we did in the the previous algorithm.

**Trace-Back** The score of the optimal alignment is in \(V_{n,i^*}\), with
\[
i^* = \max_i V_{n,i}
\]

To backtrace the path, we follow the pointers from \(V_{n,i^*}\), until we reach \(V_{0,0}\).

Choosing \(V_{i^*,m}\), we don’t penalize gaps at the end of the strand.
1.5 Complexity

1.5.1 Time Complexity (Needleman-Wunsch):

Running time: The table is of dimension \( n + 1 \) rows and \( m + 1 \) columns, and we fill each one of the cells in \( O(1) \). We get

\[
O(n \cdot m) \cdot O(1) = O(n^2)
\]

1.5.2 Space complexity (Needleman-Wunsch):

We maintain the whole table along the algorithm. This gives

\[
O(n \cdot m) = O(n^2)
\]

2 Divide and Conquer

We saw the running time and space complexity of Needleman-Wunsch is \( O(n \cdot m) \). Say we want to align a gene to the human genome, it may require too much memory. So we want to reduce the space used by the algorithm. Note that in Needleman-Wunsch, when computing the best score, to fill column \( j \), the only values we need are the values of the \( j-1 \)th column. Therefore it’s enough to maintain only 2 columns throughout the whole process. Thereby we can compute the best score in linear space: \( O(n) \)

2.1 Backtracking

The space reduction presented above will not work for path backtracking since we need to backtrack after filling the whole table back to \( V_{0,0} \).

To solve this problem, we use the Divide and Conquer technique. In this technique, we divide the problem into two sub-problems recursively. Afterwards, we combine the solutions of the sub-problems into a solution of the original problem.

Note that each alignment from \((0,0)\) to \((n,m)\) must pass through \( V(\frac{n}{2}, j) \) for \( 0 \leq j \leq m \), when we don’t know \( j \).

Remark: each sub-path of the optimal path is also optimal.

We want to find \( j \), without knowing the whole path in the graph.

This can be done in linear space by calculating the scores \( V_{\frac{n}{2}, k} \), for each \( 0 \leq k \leq m \).

For each \( 0 \leq k \leq m \), we define \( prefix(k) \) as the score of the best path from \((0,0)\) to \((\frac{n}{2}, k)\) in \( V \) and \( suffix(k) \) as the score of the best path from \((\frac{n}{2}, k)\) to \((n,m)\). This last step can be done by going in “reverse” from \((n,m)\) to \((\frac{n}{2}, k)\).
The value $V_{\frac{m}{2},k} + V_{\frac{m}{2},i}$ is the score of the optimal path from $(0,0)$ to $(n,m)$ that passes through the vertex $(\frac{n}{2},k)$. Therefore, $\max_i(V_{\frac{m}{2},i} + V_{\frac{m}{2},i})$ computes the score of the optimal path and finds the corresponding $i$.

After we have found the optimal $i$, we can divide the problem into two sub-problems: find the optimal path from $(0,0)$ to $(\frac{n}{2},i)$ and the optimal path from $(\frac{n}{2},i)$ to $(n,m)$. For the former, we do it by computing the optimal paths from $(0,0)$ to $(\frac{n}{4},k)$ and from $(\frac{n}{4},k)$ to $(\frac{n}{2},i)$ with $0 \leq k \leq i$. For the latter, we do it by computing the optimal paths from $(\frac{n}{2},i)$ to $(\frac{3n}{4},h)$ and from $(\frac{3n}{4},h)$ to $(n,m)$ with $i \leq h \leq m$. We repeat the procedure until we get a matrix of size $1 \cdot 1$.

2.2 Running time

Denote by $M$ the running time of the first iteration of the algorithm. We have $M = O(m \cdot n)$.

In the second iteration, we divide by two the area of the relevant table (see the figure above), we get that the running time of the second iteration is $\frac{M}{2}$.

We do it again and again, until we get a matrix of size $1 \cdot 1$.

That is, we get:

$$M + \frac{M}{2} + \frac{M}{4} + \frac{M}{8} + ... + 1$$
We know that

\[\sum_{k=0}^{\infty} \frac{1}{2^k} \to 2\]

From this, we get that the running time of the whole algorithm is

\[O(2M) = O(n \cdot m).\]

We’ve got linear space for a quadratic running time.