Sequence Alignment

Karl Stratos

1 Warmup

Let $x = (x_1 \dots x_m)$ and $y = (y_1 \dots y_n)$ be two sequences of tokens, aka. **strings**. We assume the lengths m, n are known and a pair of tokens can be compared in constant time. We are interested in establishing a relationship between x and y as efficiently as possible. The simplest relationship is **string equality**, which can be established in $O(m)$ time by comparing each pair of tokens. We must look at every token, so this is the best we can do. However, the situation becomes trickier for even a slightly more complex relationship like if x **is a substring** of y (i.e., x appears in y). Naively, we will compare x against every length-m span in y. This takes $O(mn)$ time, but it does not use the fact that consecutive spans overlap in all but two tokens. To that end, let us hash x to a number $H(x) \in \mathbb{N}$ using $O(m)$ time. Then we move through y one step at a time. At each step i, we hash the corresponding span $y^{(i)} = (y_i \dots y_{i+m-1})$ to $H(y^{(i)}) \in \mathbb{N}$, and compare x and $y^{(i)}$ *only if* $H(x) = H(y^{(i)})$. Hashing a length-m string generally takes $O(m)$, but the stepwise span overlap allows us to compute a "rolling hash" $H(y^{(i)}) = H(y^{(i-1)}) - H(y_{i-1}) + H(y_{i+m-1})$ in $O(1)$ time. This is called the Rabin–Karp algorithm (Karp and Rabin, 1987). It still has the worst-case runtime of $O(mn)$ since we may always get a hash collision, but the expected runtime of $O(m+n)$ assuming a good hash function. One should note, however, that in the typical case in which m is small the overhead of hashing will actually make this algorithm slower than the naive algorithm.

What about non-Boolean relationships like the **longest common substring** between x and y? A straightforward approach of comparing every substring pair for each length takes $O(\min(m,n)mn)$ time. Again, we should not have to treat substrings independently. A standard approach to pairwise sequence analysis is dynamic programming in which we define a table $T \in \mathbb{R}^{mn}$ where the base values $T(i,0)$ and $T(0,j)$ are given and each $T(i,j)$ is determined by $\{T(k,l)\}_{k<i,l<j}$ so that we can incrementally expand the border. In this case, we may define $T(i,j)$ to be the length of the longest common substring ending at x_i and y_j. This has the base case of $T(i,0) = T(0,j) = 0$ and the main body of $T(i,j) = T(i-1,j-1) + 1$ if $x_i = y_j$, otherwise $T(i,j) = 0$. For instance, if $x =$ BCFAB and $y =$ ABCDC, we have

		A	B	C	D	C
	0	0	0	0	0	0
B	0	0	1	0	0	0
C	0	0	0	2	0	1
F	0	0	0	0	0	0
A	0	1	0	0	0	0
B	0	0	2	0	0	0

where the actual longest common substrings (BC and AB) can be extracted easily. The runtime is $O(mn)$. However, even here the table seems wasteful since the cell (i,j) only uses one cell $(i-1,j-1)$. Indeed, there is a $O(m+n)$-time algorithm based on generalized suffix trees. It builds a tree that captures all suffixes in x, y and returns the substring corresponding to the lowest common ancestor node.

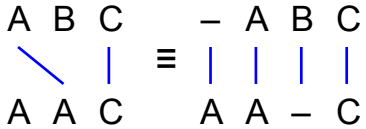
What about the **longest common subsequence (LCS)** between x and y? For clarification, in the algorithmic term, x is a "subsequence" of y if x can be obtained by deleting some (or no) tokens in y. There are exponentially many subsequences (vs quadratically many substrings), so the problem is considerably harder. In fact, the problem is NP-hard in the general case of K sequences, but when $K = 2$ is held constant we can do much better. A straightforward approach of checking every subsequence in x against y takes $O(n2^m)$ time. Again, we turn to dynamic programming. Define $T(i,j)$ to be the length of the LCS *up to* x_i and y_j. Note the subtle definitional difference ("up to" vs "ending at") which is more suitable for tracking subsequences. The base case is still $T(i,0) = T(0,j) = 0$. The main body is $T(i,j) = \max(T(i-1,j-1) + \mu_{\text{LCS}}(x_i, y_j), T(i-1,j), T(i,j-1))$ where $\mu_{\text{LCS}}(a,b) = 1$ if $a = b$, otherwise $\mu_{\text{LCS}}(a,b) = -\infty$. For instance, if $x =$ ABC and $y =$ AAC, we have

		A	A	C
	0	0	0	0
A	0	1	1	1
B	0	1	1	1
C	0	1	1	2

The table solves the LCS subproblems for all positions (i, j) where optimal subsequences can be extracted by backtracking to $(0, 0)$. The solution is not unique. For instance, the partial LCS of $x_{\leq 1} = \text{A}$ and $y_{\leq 2} = \text{AA}$ can be obtained by matching either A$\underline{\text{A}}$ or $\underline{\text{A}}$A, corresponding to two different paths from $(1, 2)$ to $(0, 0)$.[1] The runtime is $O(mn)$, exponentially better than the naive approach.

LCS can be framed as finding a highest scoring sequence alignment. A (global) **sequence alignment** of $x, y \in \mathcal{V}^+$ is a sequence pair $\bar{x}, \bar{y} \in (\mathcal{V} \cup \{-\})^+$ where $-$ is a special "gap" token such that (1) $|\bar{x}| = |\bar{y}|$, (2) $x = \bar{x}.\text{delete}(-)$, (3) $y = \bar{y}.\text{delete}(-)$, and (4) $\binom{\bar{x}_i}{\bar{y}_i} \neq \binom{-}{-}$ for all i. The number of alignments is the number of ways x and y can be merged, thus $\binom{m+n}{m} = \frac{(m+n)!}{m!n!}$.[2] For instance, the subsequence pair ($\underline{\text{AB}}\underline{\text{C}}$, A$\underline{\text{AC}}$) is associated with the alignment ($-$ABC, AA$-$C):

		A	A	C
	0	0	0	0
A	0	1	1	1
B	0	1	1	1
C	0	1	1	2



where each vertical move in backtracking introduces a gap for a token in x, each horizontal move introduces a gap for a token in y, and each diagonal move consumes a token pair in x, y. The LCS solves

$$J^\star_{\text{LCS}} = \max_{\bar{x},\bar{y} \in \mathcal{A}(x,y)} \sum_{i=1}^{|\bar{x}|} s_{\text{LCS}}(\bar{x}_i, \bar{y}_i)$$

where $\mathcal{A}(x, y)$ is the set of all possible alignments of x, y and $s_{\text{LCS}} : (\mathcal{V} \cup \{-\})^2 \to \mathbb{R}$ is a token-pair alignment score

$$s_{\text{LCS}}(a, b) = \max \begin{cases} 0 & \text{if } a = - \text{ or } b = - \\ \mu_{\text{LCS}}(a, b) & \text{otherwise} \end{cases}$$

(recall $\mu_{\text{LCS}}(a, b) = 1$ if $a = b$ else $-\infty$). Since the score of 0 can be always achieved by not aligning, the LCS will never mismatch tokens. The dynamic programming table can be re-expressed as $T(i, j) =$ highest score of any alignment up to x_i and y_j where the alignment $(\bar{x}, \bar{y})$ may end in either $\binom{x_i}{-}$, $\binom{-}{y_j}$, or $\binom{x_i}{y_j}$, and $T(m, n) = J^\star_{\text{LCS}}$.

2 Sequence Alignment

More generally, we may search for a sequence alignment while allowing for token mismatches. This can be achieved by simply generalizing the token-pair alignment score to

$$s(a, b) = \max \begin{cases} \delta & \text{if } a = - \text{ or } b = - \\ \mu(a, b) & \text{otherwise} \end{cases}$$

where $\delta \leq 0$ is a penalty for introducing a gap and $\mu : \mathcal{V}^2 \to \mathbb{R}$ is any function. In the simple case we may define $\mu(a, b) = \text{sign}\,(a = b)$, but it can be more elaborate (e.g., in biology, there is a whole pairwise score table like

[1] In many LCS formulations, the main body is specified as $T(i, j) = T(i-1, j-1) + 1$ if $x_i = y_j$, otherwise $T(i, j) = \max(T(i-1, j), T(i, j-1))$. While correct, it fails to cover all possible solutions. In this example, it only yields A$\underline{\text{A}}$.

[2] This is often approximated as $\frac{2^{2n}}{\sqrt{\pi n}}$ for $m = n$ by Stirling's formula $n! \approx \sqrt{2\pi} n^{n+\frac{1}{2}} e^{-n}$. The number of alignments $\binom{m+n}{m}$ is larger than the number of subsequence pairs 2^{m+n} because the latter does not count the multiple ways tokens can be skipped by introducing gaps, corresponding to different paths in backtracking for the skipped tokens E.g., the subsequence pair $\underline{\text{AB}}\underline{\text{C}}$ and A$\underline{\text{AC}}$ is associated with two alignments: $\binom{\text{A–BC}}{\text{AA–C}}$ and $\binom{\text{AB–C}}{\text{A–AC}}$.

BLOSUM). The optimization problem

$$J^\star = \max_{\bar{x},\bar{y}\in\mathcal{A}(x,y)} \sum_{i=1}^{|\bar{x}|} s(\bar{x}_i, \bar{y}_i) \tag{1}$$

can be solved with essentially the same dynamic programming algorithm as LCS, where the base case is $T(i,0) = \delta i$ and $T(0,j) = \delta j$ (since we now have a nonzero penalty for a gap) and the main body is $T(i,j) = \max(T(i-1,j-1) + \mu(x_i, y_i), T(i-1,j) + \delta, T(i,j-1) + \delta)$. This is called the **Needleman-Wunsch algorithm** (Needleman and Wunsch, 1970). Depending on the score specification we will have a different alignment. For example, aligning (AB, BA) with $\mu(a,b) = \text{sign}\,(a=b)$,

$\delta = -1 \Rightarrow$

		B	A
	0	−1	−2
A	−1	−1	0
B	−2	0	−1

$\begin{pmatrix} \text{A} & \text{B} & - \\ - & \text{B} & \text{A} \end{pmatrix}$

$\delta = -2 \Rightarrow$

		B	A
	0	−2	−4
A	−2	−1	−3
B	−4	−1	−2

$\begin{pmatrix} \text{A} & \text{B} \\ \text{B} & \text{A} \end{pmatrix}$

Aside: the algorithm can be used to compute the edit distance between two sequences, in particular Levenshtein distance (i.e., minimal number of edits to transform one to another), by defining the scores appropriately. In this case, substitution corresponds to token-token matching; insertion/deletion corresponds to token-gap matching.

2.1 Segmental Gap Penalty

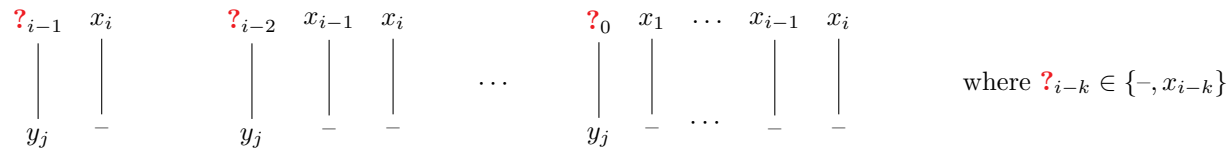
We can consider a more general version of (1)

$$J^\star_\gamma = \max_{\bar{x},\bar{y}\in\mathcal{A}(x,y)} \sum_{i=1:\ (\bar{x}_i\neq -)\wedge(\bar{y}_i\neq -)}^{|\bar{x}|} \mu(\bar{x}_i, \bar{y}_i) + \sum_{g\in\text{Gaps}(\bar{x})\cup\text{Gaps}(\bar{y})} \gamma(|g|) \tag{2}$$

where $\text{Gaps}(\bar{x})$ is the set of all gap segments in $\bar{x}$ (likewise for $\bar{y}$), and $\gamma(k) \leq 0$ is an arbitrary gap penalty for a *segment* of length $k \geq 1$.[3] Choosing the linear gap penalty $\gamma(k) = \delta k$ reduces (2) to (1). It is possible to solve (2) in $O(mn^2 + m^2 n)$ time by more careful dynamic programming, which involves case analysis to additionally track the length of a gap. Define $A(i,j), B(i,j), C(i,j)$ as the highest score of alignment up to x_i and y_j ending in $\binom{x_i}{-}, \binom{-}{y_j}, \binom{x_i}{y_j}$, wherefore $T(i,j) = \max(A(i,j), B(i,j), C(i,j))$. The base case is given by $A(0,0) = B(0,0) = C(0,0) = 0$, $A(i,0) = \gamma(i)$, $B(0,j) = \gamma(j)$, and all other base values filled with $-\infty$ (e.g., $A(0,j) = -\infty$ because we cannot have $\binom{\varnothing}{-}$). We have

$$A(i,j) = \max\left(\max_{k=1}^{i} B(i-k,j) + \gamma(k),\ \max_{k=1}^{i} C(i-k,j) + \gamma(k)\right)$$

To see why, $A(i,j)$ is the highest score of alignment up to x_i and y_j ending in $\binom{x_i}{-}$. Such an alignment has one of the following $2i$ suffixes:



where $?_{i-k} \in \{-, x_{i-k}\}$

(we define $x_0 = \varnothing$). The final gap in $\binom{x_i}{-}$ must be the k-th gap in some existing gap situation for which we have already computed the optimal alignment, and this is where we apply the length-k gap penalty $\gamma(k)$. Symmetrically, $B(i,j) = \max(\max_{k=1}^{j} A(i,j-k) + \gamma(k),\ \max_{k=1}^{j} C(i,j-k) + \gamma(k))$. The case for C is easy because there are only three possible situations when aligning $x_i \leftrightarrow y_j$: $C(i,j) = \max(A(i-1,j-1), B(i-1,j-1), C(i-1,j-1)) + \mu(x_i, y_j)$. An optimal alignment can be extracted by cross-table backtracking. This generalization using an arbitrary gap penalty is referred to as the **Smith-Waterman algorithm** (Smith *et al.*, 1981).

[3] This particular formulation is motivated to model the lengths of the introduced gaps. For instance, we can prefer a long single gap over multiple short gaps by choosing a gap penalty satisfying $\gamma(k+l) \geq \gamma(k) + \gamma(l)$.

2.1.1 Affine gap penalty

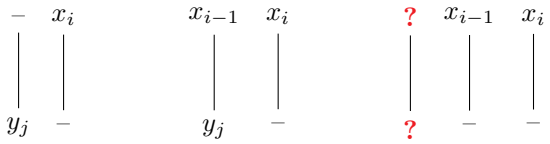
Typically the only reason we consider the segmental gap penalty formulation (2) is we want to avoid alignments with segmented gaps. This can be achieved by a tiny extension of the linear gap penalty, namely **affine gap penalty**:

$$\gamma(k) = \alpha + \delta(k-1) \qquad\qquad \alpha \leq \delta$$

(note $\gamma(k+l) \geq \gamma(k)+\gamma(l)$ and $\alpha = \delta$ reduces it to linear). In this case, we do not need to compute the gap penalty for each possible length; we just need to check if the current gap is the first one. We have

$$A(i,j) = \max\left(B(i-1,j)+\alpha,\ C(i-1,j)+\alpha,\ A(i-1,j)+\delta\right)$$

corresponding to the three relevant situations:



where we don't care what the past ? is as long as up to y_j is aligned. Symmetrically, $B(i,j)$ is the max of $A(i,j-1)+\alpha$, $C(i,j-1)+\alpha$, and $B(i,j-1)+\delta$. The case for C is the same: $C(i,j) = \max(A(i-1,j-1), B(i-1,j-1), C(i-1,j-1)) + \mu(x_i, y_j)$. This is called the **Gotoh algorithm** (Gotoh, 1982), which has the favorable runtime of $O(mn)$ compared to the general case $O(mn^2 + m^2n)$.

References

Gotoh, O. (1982). An improved algorithm for matching biological sequences. *Journal of molecular biology*, **162**(3), 705–708.

Karp, R. M. and Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM journal of research and development*, **31**(2), 249–260.

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, **48**(3), 443–453.

Smith, T. F., Waterman, M. S., *et al.* (1981). Identification of common molecular subsequences. *Journal of molecular biology*, **147**(1), 195–197.