Sequenzanalyse 2

by Jens Stoye

Summer semester 2026

Aufarbeitung des Skripts, Vorlesungsnotizen und ergänzende Lehrmaterialien by Liliana Sanfilippo

Last update 2026-08-03

Contents

1 Sequence Comparison Beyond Edit Distance **5**

1.1 The q-gram Distance . . . 5

1.1.1 Practical Computation . . . 6

1.1.2 Choice of q . . . 8

1.1.3 Efficiency . . . 9

1.1.4 Relation to the Edit Distance . . . 10

1.2 The Maximal Matches Distance . . . 11

1.2.1 Efficient Computation . . . 11

1.2.2 Relation to the Edit Distance . . . 12

1.2.3 Maximal Matches Metric . . . 12

1.3 Filtration for Edit Distance . . . 13

2 Practical Aspects of de Bruijn Graphs **15**

2.1 Definition and Basic Properties . . . 15

2.2 Words with the same ($k+1$)-mer Profile . . . 15

2.3 Variants of de Bruijn Graphs . . . 18

2.4 Implementation of de Bruijn Graphs: Sets of k-mers . . . 21

3 Approximative String Matching **23**

3.1 Sellers' Algorithm . . . 23

3.2 Ukkonen's Cutoff Algorithm . . . 27

3.3 Vergleich von Sellers' und Ukkonen Cutoff . . . 30

3.4 Search Schemes and Bidirectional Indices . . . 30

4 Biologically Inspired Alignment Scores **33**

4.1 Sensible Similarity Scores . . . 34

4.2 Log-Odds Score Matrices . . . 35

4.3 Non-symmetric score matrices . . . 36

4.4 Position-Specific Scores . . . 37

5 RNA Secondary Structure Prediction **39**

5.1 Introduction . . . 39

5.2 The Optimization Problem . . . 40

5.3 Context-Free Grammars . . . 40

5.4 The Nussinov Algorithm . . . 41

6 Pairwise Sequence Alignment in Linear Space **45**

6.1 The Forward-Backward Technique . . . 45

6.2 Hirschberg's Algorithm . . . 47

7 Suboptimal Local Alignments **51**
7.1 Smith-Waterman-Algorithm . . . 52
7.2 Waterman-Eggert-Algorithm . . . 53

8 Exact Algorithms for Sum-of-Pairs Multiple Sequence Alignment **55**
8.1 The Exact Algorithm Revisited . . . 55
8.1.1 The Basic Algorithm . . . 55
8.1.2 Variations of the Basic Algorithm . . . 56
8.2 Carrillo and Lipman's Search Space Reduction . . . 57

9 An Approximation Algorithm for Sum-of-Pairs Multiple Sequence Alignment **61**
9.1 Digression: NP-completeness . . . 61
9.2 Approximation Algorithms . . . 64
9.3 The Center-Star Approximation . . . 64

10 Heuristics for Multiple Sequence Alignment **67**
10.1 Divide-and-Conquer Alignment . . . 67
10.2 Segment-Based Alignment . . . 70
10.2.1 Segment Identification . . . 70
10.2.2 Segment Selection and Assembly . . . 71
10.2.3 DIALIGN . . . 71

11 Examples and longer explanations **73**
11.1 Sellers' algorithm example . . . 73
11.2 Ukkonen's algorithm example . . . 81
11.3 Nussinov example . . . 94
11.4 Hirschberg Example . . . 101
11.4.1 Anders betrachtet . . . 103
11.5 Smith-Waterman example . . . 105
11.6 Waterman-Eggert Example . . . 108

12 Übersichten, Listen und Co. **113**
12.1 List of equations . . . 113
12.2 Algorithms . . . 116
12.2.1 Detailed time complexities . . . 116
12.3 Alignment categories and types . . . 117

Backmatter **i**
Definitionen . . . i
Index . . . vi
References . . . viii

CHAPTER 1

Sequence Comparison Beyond Edit Distance

The edit distance (or alignment distance) is one of the most fundamental ways of quantifying the dissimilarity of two sequences x and y over a finite alphabet Σ. It is based on the cost of an alignment $A = (A_1, A_2, \ldots, A_n)$, , defined as the sum of the costs of A's columns, i.e.,

$$cost(A) = \Sigma_{i=1}^{n} cost(A_i),$$

where the cost of alignment column $A_i = \binom{a_i}{b_i}$, $a_i, b_i \in \Sigma$, in the simplest (unit cost) case is 0 for a match column ($a_i = b_i$), and 1 for a mismatch column ($a_i \neq b_i$) or an indel column ($a_i = -orb_i = -$). Then we have:

Definition 1.1 Alignment distance The alignment distance of two sequences $x, y \in \sum^*$ is defined as $d(x, y) := min\{cost(A) : A \in \mathcal{A}*$ is an alignment of x and $y\}$.

The corresponding computational problem is as follows:

Problem 1.1 Alignment Problem
For two given strings $x, y \in \Sigma^*$ and a given cost function, find the alignment distance of x and y and one or all optimal alignment(s).

From the "Sequence Analysis 1" lecture we know that this problem can be solved in $O(mn)$ time using a simple and elegant dynamic programming algorithm, where $m = |x|$ and $n = |y|$ are the two sequence lengths.

However, there are situations in which edit distance and alignments are not the perfect model to compare sequences. In the following two sections we will study two interesting alternatives.

1.1 | The q-gram Distance

Idea 1.1 Die q-Gramm-Distanz vergleicht zwei Strings nicht zeichenweise, sondern über die Häufigkeiten aller Länge-q-Teilwörter. Sie ist sehr schnell ($O(n)$) und dient als Filter für die (teure) Editdistanz.

Edit distances emphasize the correct order of the characters in a string. If, however, a large block of a text is moved somewhere else (e.g. two paragraphs of a text are exchanged), the edit distance will be high, although from a certain standpoint, the texts are still quite similar. A more appropriate notion of distance can be defined in several ways. Here we introduce the q-gram distance d_q, which is actually a pseudo-metric: There exist strings $x \neq y$ with $d_q(x, y) = 0$.

Remember that for $q \in N$, a q-gram is a string of length q. Wir reden hier über überlappende q-gramme!

Definition 1.2 occurrence count Let $x \in \Sigma^m$ and choose $q \in [1, m]$. The occurrence count of a q-gram $z \in \Sigma^q$ in x is the number

$$Occ(x, z) := |i \in [1, m - q + 1] : x[i...i + q - 1] = z| \tag{1.1}$$

Definition 1.3 Q-gram profile Let $\sigma = |\sum|$. The q-gram profile of x is a σ^q-element vector $p_q(x) := (p_q(x)_z)_{z \in \sum^q}$, indexed by all q-grams, defined by $p_q(x)_z := Occ(x, z)$.

Comment 1.1 Also eine geordnete Liste davon, wie oft ein q-gram in einem Wort vorkam.

Definition 1.4 q-gram distance The q-gram distance of $x \in \Sigma^m$ and $y \in \Sigma^n$ is defined for $1 \leq q \leq min\{m, n\}$ as:

$$d_q(x, y) := \Sigma_{z \in \Sigma^q} |p_q(x)_z - p_q(y)_z| \tag{1.2}$$

Comment 1.2 Einfacher/Umgangssprachlicher:

$$d_q(x, y) := \sum_{\text{every } q-\text{gram } z} |Occ(x, z) - Occ(y, z)|$$

bzw. noch etwas weiter verunstaltet:

$$d_q(x, y) := \sum_{\text{every } q-\text{gram}} |\#x - \#y|$$

Example 1.1 Consider $\Sigma = A, B$, $x = ABAA$, $y = ABAB$, $z = AABA$, and $q = 2$. Using lexicographic order (AA, AB, BA, BB) among the q-grams, we have $p_2(x) = (1, 1, 1, 0)$, $p_2(y) = (0, 2, 1, 0)$ and $p_2(z) = (1, 1, 1, 0)$. Therefore, we obtain $d_2(x, y) = 2$ and $d_2(x, z) = 0$ (although $x \neq z$).

Theorem 1.1 The q-gram distance d_q is a pseudo-metric.

Proof. It fulfills nonnegativity, symmetry and the triangle inequality (exercise).

1.1.1 | Practical Computation

If the q-gram profile of a string $x \in \Sigma^m$ is dense (i.e., if it does not contain mostly zeros), it makes sense to implement it as an array $p\,[0 \cdots \Sigma^q - 1]$, where $p[i]$ counts the number of occurrences of the i-th q-gram in some arbitrary but fixed (e.g. lexicographic) order.

Definition 1.5 Ranking function A bijective function $r : \mathcal{X} \to [0, |\mathcal{X}| - 1]$ for a finite set $\mathcal{X}$.

Comment 1.3 Warum $|X| - 1$?

Definition 1.6 Ranking algorithm An algorithm that implements a ranking function.

Definition 1.7 Unranking function The inverse of a ranking function.

Definition 1.8 Unranking algorithm An algorithm that implements a unranking function.

Of course, we are interested in an efficient (un-)ranking algorithm for the set of all q-grams over Σ. A classical one is to first define a ranking function r_Σ on the characters (alphabet encoding) and then extend it to a ranking function

r on the q-grams by interpreting them as base-σ numbers. There are two possibilities to number the positions of a q-gram: From q to 1 falling or from 1 to q rising, as shown in the Example 1.2.

Comment 1.4 Was sind base-σ numbers?

Example 1.2 Assume the alphabet $\Sigma = A, C, G, T$ and the alphabet encoding $r_\Sigma(A) = 0$, $r_\Sigma(C) = 1$, $r_\Sigma(G) = 2$, and $r_\Sigma(T) = 3$. For $q = 5$, the two different possibilities to rank the q-gram $z = \texttt{ATACG}$ are:

(a) Falling ranking

$$\begin{aligned} r(ATACG) &= \underbrace{r_\Sigma(z[1]) \cdot \sigma^4}_{0 \cdot 4^4} + \underbrace{r_\Sigma(z[2]) \cdot \sigma^3}_{3 \cdot 4^3} + \underbrace{r_\Sigma(z[3]) \cdot \sigma^2}_{0 \cdot 4^2} + \underbrace{r_\Sigma(z[4]) \cdot \sigma^1}_{1 \cdot 4^1} + \underbrace{r_\Sigma(z[5]) \cdot \sigma^0}_{2 \cdot 4^0} \\ &= 0 + 192 + 0 + 4 + 2 = 198 \end{aligned}$$

(b) Rising ranking

$$\begin{aligned} r(ATACG) &= \underbrace{r_\Sigma(z[1]) \cdot \sigma^0}_{0 \cdot 4^0} + \underbrace{r_\Sigma(z[2]) \cdot \sigma^1}_{3 \cdot 4^1} + \underbrace{r_\Sigma(z[3]) \cdot \sigma^2}_{0 \cdot 4^2} + \underbrace{r_\Sigma(z[4]) \cdot \sigma^3}_{1 \cdot 4^3} + \underbrace{r_\Sigma(z[5]) \cdot \sigma^4}_{2 \cdot 4^4} \\ &= 0 + 12 + 0 + 64 + 512 = 588 \end{aligned}$$

The first numbering of positions in (a) corresponds naturally to the ordering of positional number systems; in the decimal number 23, the first 2 has positional value (weight) $10^1 = 10$ whereas the 3 has the lower weight of $10^0 = 1$. For texts, however, the second numbering in (b) is more natural since we expect lower position numbers on the left side of higher position numbers.

In general, we see that $z \in \Sigma^q$ has rank

$$r(z) = \Sigma_{i=1}^{q} r_\Sigma(z[i]) \cdot \sigma^{q-i} \tag{1.3}$$

(falling) or

$$r(z) = \Sigma_{i=1}^{q} r_\Sigma(z[i]) \cdot \Sigma^{i-1} \tag{1.4}$$

(rising).

Whichever numbering system we use, we have to do it consistently.

For each of the two ways to rank a q-gram, there is a corresponding unranking method. Both methods work with integer division and remainder. The one for the falling ranking function uses a dynamic divisor and determines the characters of the q-gram based on the integer results, the other one for the rising ranking function uses a fixed divisor and determines the characters based on the remainder.

Example 1.3 Unranking the rank values 198 and 588 from Example 1.2.

(a) Unranking for falling ranking function

$$\begin{aligned} \frac{198}{4^4} &= 0, \quad \text{R } 198 \to A \\ \frac{198}{4^3} &= 3, \quad \text{R } 6 \to T \\ \frac{6}{4^2} &= 0, \quad \text{R } 6 \to A \\ \frac{6}{4^1} &= 1, \quad \text{R } 2 \to C \\ \frac{2}{4^0} &= 0, \quad \text{R } 0 \to G \end{aligned}$$

(b) Unranking for rising ranking function

$$\begin{aligned}\frac{588}{4} &= 147, \quad \text{R } 0 \to A\\ \frac{147}{4} &= 36, \quad \text{R } 3 \to T\\ \frac{36}{4} &= 9, \quad \text{R } 0 \to A\\ \frac{9}{4} &= 2, \quad \text{R } 1 \to C\\ \frac{2}{4} &= 0, \quad \text{R } 2 \to G\end{aligned}$$

Comment 1.5 To example 1.3.

$$\frac{198}{4^4} = 0, \quad \text{R } 198 \to A$$

Because $\frac{198}{4^4} = \frac{198}{256} \approx \underline{0}, 77$ which means the rest (R) ist 198, since $198 - 0 = 198$. We decode it into A, since we are using the result of the fraction to find the letters: $r_\Sigma(A) = 0$.

$$\frac{588}{4} = 147, \quad \text{R } 0 \to A$$

Because with the rising function we are using the rest to find the letters instead of the result of the fraction.

Both methods need q iterations and reconstruct the q-gram from the first position to the last. Each iteration takes a starting value r and determines the corresponding character of the q-gram while updating r for the next iteration. The starting value for the first iteration is the rank of the q-gram. In iteration i in the unranking method (1.5), for $1 \leq i \leq q$, the starting value r is divided by σ^{q-i}. The integer result c of the division determines the decoded character as described by r_Σ of the used ranking function. The remainder r' of the division serves as starting value for the next iteration.

$$\frac{r}{\sigma^{q-1}} = c, \quad \text{R } r' \tag{1.5}$$

In unranking method (1.6), the starting value r is always divided by σ. Here, the remainder c of the integer division decodes the corresponding character and the integer result r is the starting value for the next iteration.

$$\frac{r}{\sigma} = r', \quad \text{R } c \tag{1.6}$$

1.1.2 | Choice of q

In principle, we could use any $q \in [1, \min\{m, n\}]$ (Any q that is only as ling as the shorter of the two strings being compared) to compare two strings of lengths m and n. In practice, some choices are more reasonable than others.

For example, for $q = 1$, we merely add the differences of the single letter counts. If m and n are large, there are many (even very differently looking) strings with the same letter counts, which would all have distance zero from each other.

If on the other hand $q = m \leq n$, and $m \approx n$, the distance d_q (q-gram distance) between the strings is always close to $n - m$, which does not give us a good resolution either. Also, the q-gram profile of both strings is extremely sparse in these cases. We thus ask for a value of q such that each q-gram occurs about once (or $c \geq 1$ times) on average.

Assuming a uniform random distribution of letters in a given string $x \in \Sigma^m$, the probability that a fixed q-gram z starts at a fixed position $i \in [1, m - q + 1]$ is $1/\sigma^q$, where $\sigma = |\Sigma|$.

Comment 1.6 It is NOT $1/\sigma$ (1 divided though the number of letters in the alphabet). One might think that, since at that "fixed position" (aka. any position) every letter of the alphabet can occur and if the distribution is uniform every letter has the same chance to occur there so it could start there, right? No. We have to account for the

length of the q-gram. The chance that the starting letter of the q-gram is at that position is $1/\sigma$. The chance that the second letter of the q-gram is in the position thereafter is $1/\sigma$. The chance that the third letter of the q-gram ...and so on. Meaning we have

$$\underbrace{1/\sigma \cdot 1/\sigma \cdots 1/\sigma}_{q\ times} = 1/\sigma^q.$$

The expected number of occurrences of z in x is thus $(m-q+1)/\sigma^q$.

Comment 1.7 Because there are $(m-q+1)$ possible starting position for a q-gram in a string of length m.

The condition that this number equals c (The number each q-gram occurs approximately) leads to the condition $(m+1)/c = q/c + \sigma^q$ (see Comment 1.8), or approximately $m/c = \sigma^q$, since $1/c$ and q/c are comparatively small.

Comment 1.8 It leads to the condition $(m+1)/c = q/c + \sigma^q$, because:

$$\begin{aligned} \frac{(m-q+1)}{\sigma^q} &= c && \mid \cdot\sigma^q \\ m-q+1 &= c \cdot \sigma^q && \mid +q \\ m+1 &= c \cdot \sigma^q + q && \mid : c \\ \frac{m+1}{c} &= \sigma^q + \frac{q}{c} \end{aligned}$$

Thus setting

$$q = \left\lfloor \frac{\log(m)\log(c)}{\log(\sigma)} \right\rfloor$$

ensures an expected occurrence count $\geq c$ of each q-gram.

1.1.3 | Efficiency

Obviously, for $z \in \Sigma^q$, the ranking function $r(z)$ can be computed in O(q) time. Both the falling and the rising ranking functions have the advantage that when a window of length q is shifted along a text, the rank of the q-gram can be updated in O(1) time:

Assume that $t = aub$ with $a \in \Sigma$, $u \in \Sigma^{q-1}$ and $b \in \Sigma$. The first q-gram is au, whereas ub is the updated one. Now we can define an update system for each of the ranking systems above. $Letj = r(au)$. For the falling ranking function we compute:

$$\text{(a) } r(ub) = (j \mod \sigma^{q-1}) \cdot \sigma + r_\Sigma(b),$$

Or without using j:

$$r(ub) = (r(au) \mod \sigma^{q-1}) \cdot \sigma + r_\Sigma(b) \tag{1.7}$$

$$\text{(b) } r(zb) = \lfloor \frac{j}{\sigma} \rfloor + f(b), \quad \text{where } f(b) = r_\Sigma(b) \cdot \sigma^{q-1}.$$

Or without using j an f: and for the rising ranking function:

$$r(zb) = \lfloor \frac{r(au)}{\sigma} \rfloor + r_\Sigma(b) \cdot \sigma^{q-1} \tag{1.8}$$

If we take a more detailed look at both systems now, we can see that the second way is possibly more efficient (as it avoids the modulo operation and only uses integer division) if f is implemented as a lookup table $f : \Sigma \to \mathbb{N}_0$. If $\sigma = 2^k$ for some $k \in \mathbb{N}$, multiplications and divisions can be implemented by bit shifting operators, e.g. in the second system (b), $r(ub) = (j \gg k) + f(b)$.

It is allegedly easy to see that the q-gram corresponding to a ranking value can be obtained in O(q) time for both unranking methods.

Now, to compute $d_q(x, y)$ for $x \in \Sigma^m$ and $y \in \Sigma^n$,

1. initialize integer array $p_q[0 \dots \sigma^q - 1]$ with zeroes
2. for each $i \in [1, mq + 1]$, increment $p_q[r(x[i \cdot i + q1])]$
3. for each $i \in [1, nq + 1]$, decrement $p_q[r(y[i \cdot i + q1])]$
4. initialize $d := 0$,
5. for each $j \in [0, \sigma^q - 1]$, increment d by $|p_q[j]|$

Comment 1.9 Explanation of the steps and complexites.

	Step	Complexity	Because...
1.	Initialise empty array	$O(\sigma^q)$	the array has σ^q entries
2.	Count q-grams in x	$O(m)$	compute hash of first q-gram in $O(q)$, then use rolling hash to update in $O(1)$ for each remaining position
3.	Count q-grams in y	$O(n)$	see above
4.	Initialize distance	$O(1)$	trivial
5.	Summarize the absolute difference	$O(\sigma^q)$	Iteration through σ^q items

Leading to:

$$O(\sigma^q)+O(m)+O(n)+O(\sigma^q) = 2 \cdot O(\sigma^q)+O(m)+O(n) = O(2 \cdot \sigma^q)+O(m)+O(n) = O(\sigma^q)+O(m)+O(n) = O(\sigma^q+m+n)$$

This procedure very obviously takes $O(\sigma^q+m+n)$ time. If $m \leq n$ and q is as suggested above $\left(q = \left\lfloor \frac{\log(m)\log(c)}{\log(\sigma)} \right\rfloor = \log(\sigma) \cdot \right.$, this is $O(n)$ time overall. The space complexity is $O(\sigma^q)$ to store the array p_q.

If q is comparatively large, it does not make sense to maintain a full array. Instead, we maintain a data structure that contains only the nonzero entries of p_q. If $q = O(\log(m + n))$, we can still achieve $O(m + n)$ time using hashing.

1.1.4 | Relation to the Edit Distance

While the q-gram distance d_q has its own applications in the comparison of sequences that have undergone block movements, it also has a weak connection to the unit cost edit distance d. Recall that we can have a high edit distance even though $d_q(x, y) = 0$. On the other hand, a single edit operation changes up to q q-grams. Generally, one can show the following relationship.

Theorem 1.2 Let d denote the standard unit cost edit distance. Then

$$\frac{d_q(x, y)}{2q} \leq d(x, y) = d_q(x, y) \leq d(x, y) \cdot 2q \tag{1.9}$$

Proof. Each edit operation locally affects up to q q-grams. Each changed q-gram can cause a change of up to 2 in d_q, as the occurrence count of one q-gram decreases, the other increases. Therefore, each edit operation can lead to an increase of $2q$ of the q-gram distance in the worst case (e.g. consider AAAAAAAAAAA vs. AABAABAABAA with edit distance 3, $q = 3$, and, as can be verified, $d_3 = 18$).

How this relationship can be used productively in order to speed up the computation of the edit distance will be discussed in Section 1.2.

[2]
Notiz

Warnung 1.1 Achtung:

- **Pseudometrik!** $d_q(s,t) = 0 \not\Rightarrow s = t$. Verschiedene Strings können dasselbe q-Gramm-Profil haben (z. B. Anagramme von q-Grammen). Nur eine Richtung der Metrik-Axiome gilt strikt.
- **rising vs. falling** innerhalb einer Aufgabe konsistent halten – sonst passen Rank und Un-Rank nicht zusammen.
- Rand: ein String mit $|s| < q$ hat keine q-Gramme.

1.2 | The Maximal Matches Distance

Another notion of distance that admits large block movements is the maximal matches distance to x from y. It is based on the idea that we can cover a sequence x with substrings of another sequence (and vice versa) that are interspersed with single characters.

Idea 1.2 Wie gut lässt sich s aus langen Stücken von t zusammensetzen? Wenige, lange "Matches" $\Rightarrow$ ähnlich; viele kurze $\Rightarrow$ unähnlich.

Greedy von links, jeweils der längste Teilstring, der in t vorkommt.

Partitioning a sequence. Consider a sequence $x = z_0 b_1 z_1 b_2 \dots z_{l-1} b_l z_l$, with $|x| = m$, such that each z_i is a (possibly empty) string (for $0 \le i \le l$) and each b_i is a single character (for $1 \le i \le l$). The tuple $P = (z_0, \langle b_1 \rangle, z_1, \langle b_2 \rangle, \dots, z_{l-1}, \langle b_l \rangle, z_l)$ is a partition of x with respect to another sequence y if each z_i is also a substring of y. The size of the partition P , denoted by $|P|$, is the number of its separating single characters, that is, $|P| = l$. Note that, if x is a substring of y, there exists the shortest partition $P_0 = (x)$, of size 0. In the other extreme, the longest partition $P_m = (\epsilon, \langle x[1] \rangle, \epsilon, \langle x[2] \rangle, \dots, \langle x[m] \rangle, \epsilon)$ of size m always exists, even if $y = \epsilon$.

Example 1.4 Let $x = CTAATGCT$ and $y = ATCTA$. The following sequences are partitions of x with respect to y: $P = (CTA, \langle A \rangle, T, \langle G \rangle, CT)$ with $P| = 2, P' = (CTA, \langle A \rangle, T, \langle G \rangle, C, \langle T \rangle, \epsilon)$ with $|P'| = 3$ and $P'' = (CT, \langle A \rangle, AT, \langle G \rangle, CT)$ with $|P''| = 2$.

Definition 1.9 maximal matches distance to x from y $\delta(x||y) := min\{|P| : P$ is a partition of x with respect to $y\}$

1.2.1 | Efficient Computation

The next question is how to find a minimum size partition of x with respect to y among all the partitions. Fortunately, this turns out to be easy: A greedy strategy does the job. We start covering x at the first position by a match of maximal length k such that $z_0 := x[1 \dots k]$ is a substring of y, but $z_0 b_1 = x[1 \dots k+1]$ is not a substring of y. We apply the same strategy to the remainder of x, $x[k+2 \dots m]$.

Definition 1.10 left-to-right partition of x with respect to y $Plr(x,y) = (z_0, \langle b_1 \rangle, z_1, \langle b_2 \rangle, ..., z_{\ell 1}, \langle b_\ell \rangle, z_\ell)$ of x with respect to y is the partition defined by the condition that for all $i \in [1, \ell], z_{i1} b_i$ is not a substring of y.

Comment 1.10 Sie wird so gewählt, dass für jedes $i \in [1, \ell]$ die Konkatenation $z_{i-1} b_i$ kein Teilstring von y ist. Also: Die Blöcke werden beim Durchlaufen von x von links nach rechts genau an den Stellen getrennt, an denen das Anhängen des nächsten Zeichens dazu führen würde, dass der aktuelle Teilstring nicht mehr in y vorkommt.

Definition 1.11 right-to-left partition of x with respect to y $Prl(x, y) = (z_0, \langle b_1 \rangle, z_1, ..., \langle b_{\ell 1} \rangle, z_{\ell 1}, \langle b_\ell \rangle, z_\ell)$ of x with respect to y is the partition defined by the condition that for all $i \in [1, \ell], b_i z_i$ is not a substring of y.

Comment 1.11 Die Partition wird so gewählt, dass für jedes $i \in [1, \ell]$ die Konkatenation $b_i z_i$ kein Teilstring von y ist.

Theorem 1.3 The left-to-right partition of x with respect to y is a partition of minimal length, i.e.,

$$|P_{LR}(x, y)| = \min\{|P| : P \text{is a partition of } x \text{ with respect to } y\} = \delta(x||y).$$

The same is true for the right-to-left partition of x with respect to y, which means:

$$|P_{RL}(x, y)| = |P_{LR}(x, y)| = \delta(x||y).$$

Proof. Skript S. 7

It is important to discuss how (and how fast) we can compute the size of a left-to-right partition. In fact, because the longest common substring z that starts at a given position i in x and occurs somewhere in y can easily be found in $O(|z|)$ time using the suffix tree of y, we obtain the following result.

Theorem 1.4 The maximal matches distance $\delta(x||y)$ to a sequence x from another sequence y can be computed in $O(|x| + |y|)$ time.

Proof. Skript S. 8

1.2.2 | Relation to the Edit Distance

Like the q-gram distance, also the maximal matches distance can be used to compute a lower bound of the edit distance.

Theorem 1.5 Let $d(x, y)$ be the unit cost edit distance between sequences x and y. Then $\delta(x||y) \leq d(x, y)$.

Proof. Skript S. 8

Again, this bound permits us to speed up the computation of the edit distance, as will be shown in Section 1.2.3.

1.2.3 | Maximal Matches Metric

A final observation is that δ is obviously not symmetric: $\delta(x||y) = 0$ is equivalent to x being a substring of y. Therefore, the maximal matches distance is not a metric and the name distance is misleading. But it is possible to use δ for designing an appropriate symmetric function that satisfies the triangular inequality and is a metric.

Theorem 1.6 Given sequences x and y from Σ^*, let the function $d_{||}(x, y)$ defined as $d_{||}(x, y) = \log(\delta(x||y) + 1) + \log(\delta(y||x) + 1)$. Then $d_{||}(x, y)$ is a metric on Σ^*.

Proof. Skript S. 8

Warnung 1.2 Achtung: Die Distanz ist **asymmetrisch** definiert (s zerlegt bzgl. t), je nach genauer Skript-Definition wird noch symmetrisiert / mit einem Term normiert. Nach Preprocessing von t läuft die Zerlegung von s in $O(|s|)$. Es ist eine **untere Schranke**-Heuristik für die Editdistanz, keine exakte Distanz.

1.3 | Filtration for Edit Distance

Coming back to the edit distance, while – given the fact that the search space (consisting of all pairwise alignments of x and y) is of exponential size – the quadratic time dynamic programming solution can be considered very efficient, it can also be considered quite slow in practice when the sequences are long. Moreover, the quadratic space requirements (unless a space-saving technique is used that we will discuss in Chapter 6) can cause even larger problems.

Now, remember that in many applications the goal of sequence comparison is not the construction of an accurate alignment. Instead, for example in a database search context, one is mostly interested in identifying sequences similar to a given query. More formally:

Problem 1.2 Given a database $X = x_1, x_2, \ldots, x_k$ of k sequences $x_i \in \Sigma^*$, a query sequence $y \in \Sigma^*$ and a distance threshold $t \geq 0$, find all sequences $x \in X$ such that $d(x, y) \leq t$.

Clearly, a straightforward way to solve this problem is to compare y to each sequence $x \in X$, one after the other. Whenever in such a comparison the distance $d(x, y)$ is smaller or equal to the threshold t, then the sequence x is reported, otherwise it is discarded. This strategy clearly takes $O(k \cdot mn)$ time, where m is an upper bound for the database sequence length and $n = |y|$.

However, since usually t is small and many sequences in x will be quite dissimilar from y,it may be possible to quickly screen the database and separate interesting candidates from irrelevant sequences, faster than computing the edit distance for all of them. If such a screening procedure guarantees that all relevant sequences $x \in X$ with $d(x, y) \leq t$ are among the remaining candidates, it is called a filter. Generally, we thus have a two-step procedure. First, in a fast filtration step, candidates are identified that have a chance to be hits. Then, in a verification step these candidates are checked with the (usually slower) exact method whether they really qualify as hits. Obviously, filtration makes sense only if the procedure employed in the filtration step has a lower time complexity than the procedure used in the verification step. Indeed, if one has multiple filters, one can first apply all of them, and then employ the verification only to those elements $x \in X$ that pass all the filters. In the previous sections we have seen two fast sequence comparison methods that can be used as filters for the unit cost edit distance. Both of them require only time proportional to the sum of the lengths of the two sequences ("linear time") and provide lower bounds for the edit distance.

Filtration with the q-gram distance. The property outlined in Theorem 1.2 can be used as a filter criterion for the edit distance: For a given query sequence y and edit distance threshold t, and a given value of q, a database sequence $x \in X$ can be discarded whenever $d_q(x, y)/(2q) > t$. In case of a global similarity of the sequences and the differences spread out evenly, this filter will work quite well. If the difference between the two sequences, however, is by block movements, then the edit distance is usually very high, although the q-gram distance may still be low and produce a false positive candidate when used as a filter.

Filtration with the maximal matches distance. As shown in Theorem 1.5, the maximal matches distance also gives us a bound for the unit cost edit distance. In fact, since the edit distance is symmetric, the filtration can even be used twice, once with $d(x||y)$ and once with $d(y||x)$. This results in the following filter: For a given query sequence y and edit distance threshold t, a database sequence $x \in X$ can be discarded whenever $\delta(x||y) > t$ or $\delta(y||x) > t$.

Note that the distinguishing feature of a filter is that it never discards a hit (has no false negatives). On the other hand, one can also develop heuristics that approximate the edit distance and are efficiently computable. Good heuristics (like BLAST) will be close to the true result with high probability; hence the error made by using them is small most of the time. There is no guarantee, though, that a true hit will not be missed. [2]

CHAPTER 2

Practical Aspects of de Bruijn Graphs

A note on terminology: In Section 1.1 we were speaking of q-grams when referring to short (sub)strings of a certain length q. This notation has originally been introduced in the computer science literature and is there in use until today. An alternative, equivalent notion that one finds in most of the biological[i] and bioinformatics literature, is that of a k-mer. Here k refers to the (sub)string length. To be more consistent with the literature, in this chapter we will adopt that notation.

2.1 | Definition and Basic Properties

Remember the following definition, where $s_{k,i} := s[i \dots i+k-1]$ denotes the k-mer starting at index i, $1 \leq i \leq |s|-k+1$:

Definition 2.1 de Bruijn subgraph $DBG(s,k) = (V,E)$ for a given sequence s of dimension k () The de Bruijn subgraph $DBG(s,k) = (V,E)$ for a given sequence s of dimension k (positive integer $\leq |s|$) is a directed multigraph whose vertices V correspond to the distinct k-mers of $s, s_{k,1}, \cdots, s_{k,|s|-k+1}$, and whose edges are derived from the (k + 1)-mers of s as follows: Let a (k + 1)-mer of s be denoted as $s[i \dots i+k] = aub$, where a and b are symbols and $|u| = k-1$, then a directed edge is contained in E, linking vertex au to vertex ub.

Definition 2.2 Dimension of a de Bruijn subgraph $DBG(s,k) = (V,E)$ for a given sequence s of dimension k () The positive integer $\leq |s|$ in a $DBG(s,k) = (V,E)$.

Note 2.1 **To 2.1** If the same ($k+1$)-mer occurs multiple times in s, say m times, we have m parallel edges occurring in the multigraph $DBG(s,k)$.

Comment 2.1 Für jeden DB Graph gibt es immer einen Eulerpfad

2.2 | Words with the same ($k+1$)-mer Profile

Motivated by the fact that the q-gram distance does not satisfy the identity property of a metric (see Section 1.1) because there can be distinct sequences with the same q-gram profile, we want to determine whether for a given sequence s and integer k there are other sequences that are distinct but have the same ($k+1$)-mer profile as s. And, if there are, how many?

[i] The true origin is probably in biochemistry, where notions like monomer or polymer refer to molecular chains of certain lengths.

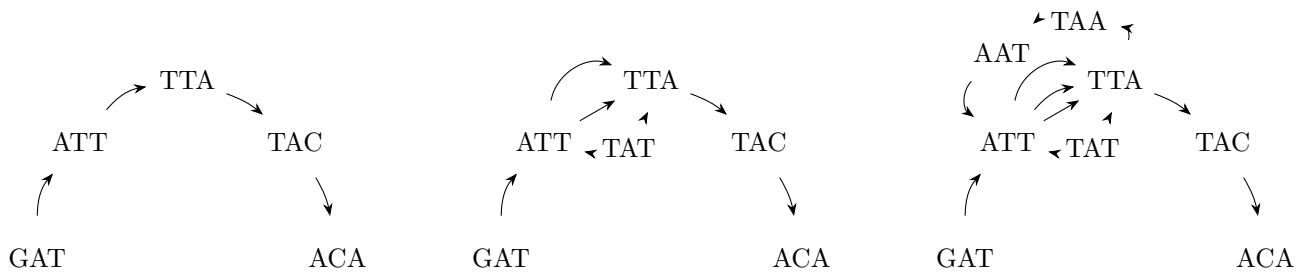
For $k = 0$, the problem is trivial. Indeed, if a sequence t that is distinct from s corresponds to a permutation of the symbols of s, then s and t clearly share the same 1-mer profile. For $k \geq 1$, an elegant answer can be found with the help of the de Bruijn subgraph $DBG(s, k)$.

Clearly, by construction $DBG(s, k)$ forms an Eulerian path p_s, represented by its sequence of vertices

$$s_{k,1}, s_{k,2}, \ldots, s_{k,|s|-k+1}.$$

Consequently, $DBG(s, k)$ is connected and balanced and might contain even more than one Eulerian path. In fact, there is a one-to-one correspondence between Eulerian paths in $DBG(s, k)$ and sequences sharing the same $(k+1)$-mer profile. In other words, a sequence t that is distinct but has the same ($k+1$)- mer profile as s exists if and only if $DBG(s, k)$ has another Eulerian path p_t corresponding to the sequence of ($k+1$)-mers of t. Since p_t uses the same edges as p_s in a different order, the path p_t an only exist if $DBG(s, k)$ contains a cycle. Note, however, that the presence of a cycle is not a sufficient condition. Indeed, it is possible that $DBG(s, k)$ contains one or more cycles, but still admits only one Eulerian path. Some examples are given in Figures 2.1 and

Example 2.1



(a) $x = GATTACA$,k= 3 - Since this graph contains no cycle, it has only one Eulerian path. There can be no other word sharing the same 4-mer profile.

(b) $x = GATTATTACA$,k= 3 - Here the graph contains two cycles, but still only one Eulerian path. There is no other word sharing the same 4-mer profile.

(c) $x = GATTATTAATTACA$,k= 3 - This graph contains several cycles and two distinct Eulerian paths. There is exactly one other word with the same 4-mer profile, `GATTAATTATTACA`.

Figure 2.1: Examples of de Bruijn subgraphs for distinct sequences and k= 3 (Skript).

Example 2.2

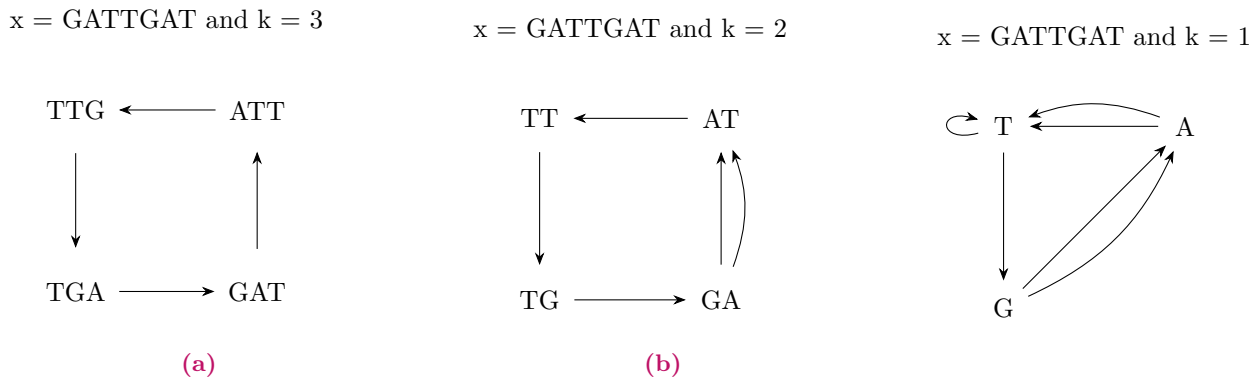


Figure 2.2: De Bruijn subgraphs for the same sequence but with k varying from 3 to 1. (Skript)

Example 2.3 Additional example from lecture

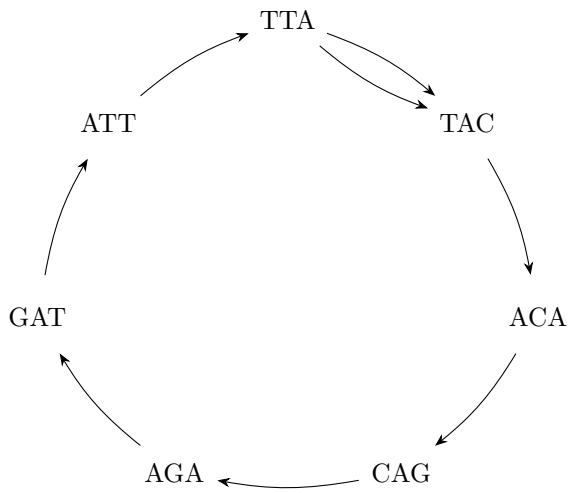


Figure 2.3: It could also be on big circle. (Vorlesung)

Comment 2.2 Reminder that the edges are basically this:

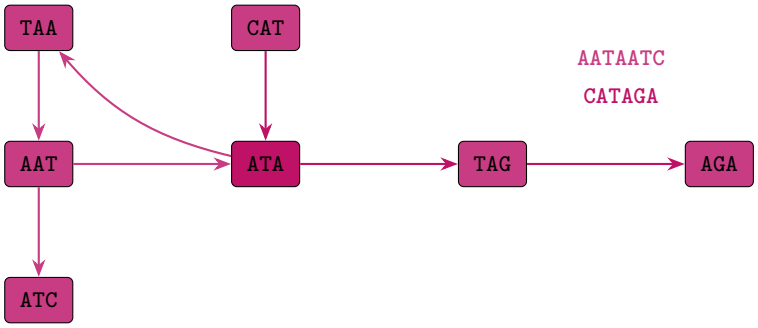
Comment 2.3 **To figure 2.1** man könnte bei fig 2.1a denken hey bei a kann es nur einen EP geben, weil es kein Kreis ist, und bei 2.1b muss es dann mehrere geben, weil da eine Schleife drin ist, aber NEIN der Pfad ist die Reihenfolge der Knoten und nicht der Kanten, also gibt es nur einen. Man kann nicht unterscheiden, ob man zuerst die obere oder untere Kante der Schleife zuerst gegangen ist. Gibt also nur eine sequenz mit em k+1 meer profil. Denn die Sequenz wird ja *aus* den Knoten gemacht, nicht den Kanten ...sozusagen.

We summarize the observations above in the following theorem.

Theorem 2.1 Given a sequence s and an integer $k \geq 1$, the number of distinct sequences that have the same ($k+1$)-mer profile as s equals the number of distinct Eulerian paths in $DBG(s,k)$. If $DBG(s,k)$ is acyclic, it has a single Eulerian path, and no sequence t exists that is distinct but has the same $(k+1)$-mer profile as s.

Comment 2.4 Auf deutsch: # worte mit gleichen $(k+1$)-mer Profil wie s = # Euler-Pfade in DBG(s,k)

Example 2.4 Beispiel von Blatt 8 `AATAATC`: 3-Mere `AAT,ATA,TAA,AAT,ATC` ⇒ Kanten `AAT`→`ATA`→`TAA`→`AAT`→`ATC`. `CATAGA`: 3-Mere `CAT,ATA,TAG,AGA` ⇒ Kanten `CAT`→`ATA`→`TAG`→`AGA`. Gemeinsamer Knoten: `ATA`.



Warnung 2.1 Achtung:

- **Konvention prüfen!** Andere Bücher setzen Knoten = $(k-1)$-Mere, Kanten = k-Mere. Hier gilt: Knoten = k-Mere, Kanten = $(k+1)$-Mere. Immer die Vorlesungs-Konvention nutzen.
- Kanten sind ein **Multiset** (mehrfaches Vorkommen zählt), wenn es um Euler-Pfade/Assembly geht.
- k-Mere ranken/un-ranken = dasselbe Schema wie q-Gramme (Kap. 1.1). Speicherung als Set von k-Meren (Kap. 2.4).

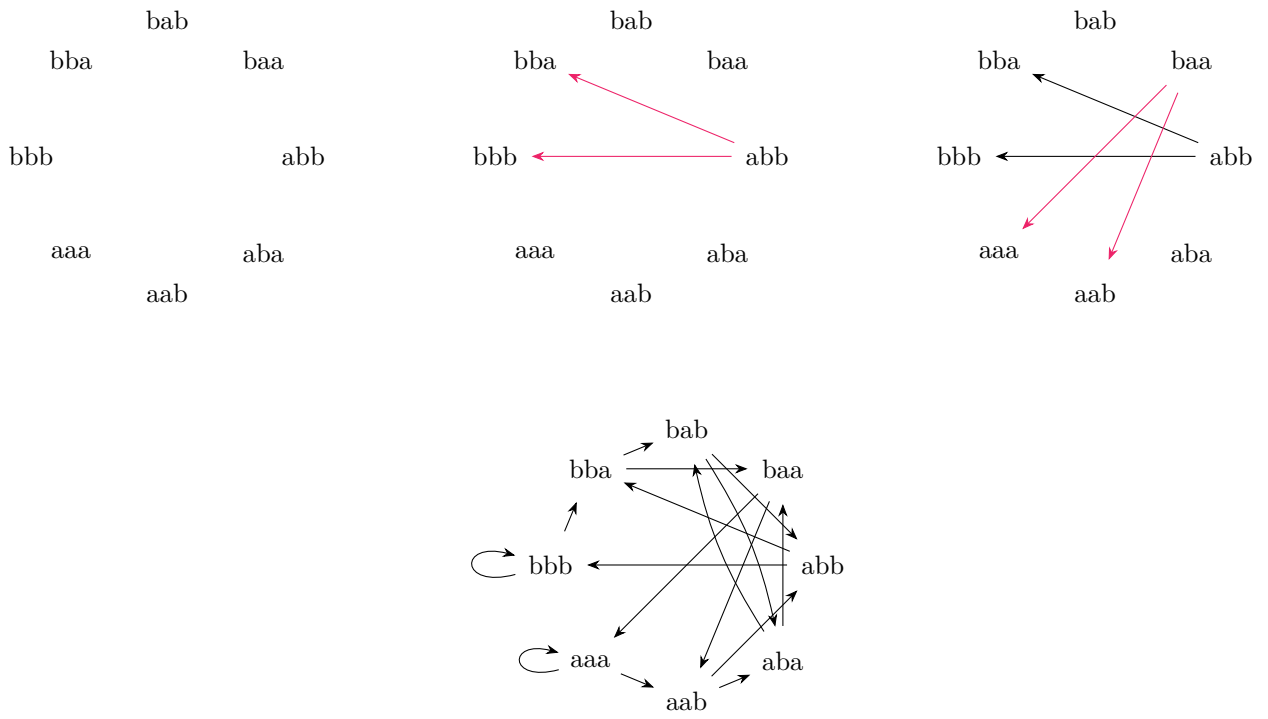
2.3 | Variants of de Bruijn Graphs

Comment 2.5 Der Begriff dbg ist sehr dehnbar, es ist in der Literatur nicht immer das gleiche damit gemeint. Viele sagen dbg, wenn sie eigentlich dbsg meinen.

Although, strictly speaking, we consider most of the time de Bruijn subgraphs (for a given sequence), often they are just called de Bruijn graphs. In fact, there are more variants of de Bruijn graphs, and often they are also not named very carefully. Here we will discuss them and introduce a precise nomenclature.

Definition 2.3 original de Bruijn graph of dimension k over a finite alphabet Σ of size σ The graph that contains one vertex for each k-mer (and therefore σ^k vertices) and a directed edge for each overlap of length k − 1 (and therefore σ^{k+1} edges)

Example 2.5 for original de Bruijn graph (Vorlesung)



The de Bruijn subgraph for a given sequence s of dimension $k$is the one that we introduced in Definition 2.1 It has several interesting properties (see Section 2.1) and is used in various contexts in bioinformatics like genome assembly and pangenomics.

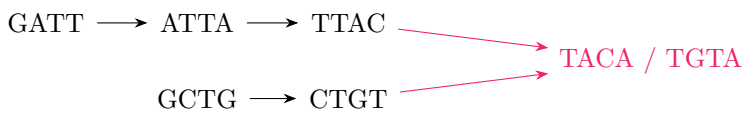
Problem 2.1 In most bioinformatics applications where DNA sequence reads come from a sequencing machine, it is not clear from which strand of the DNA double helix a read originates.

$$\text{GATT} \longrightarrow \text{ATTA} \longrightarrow \text{TTAC} \longrightarrow \text{TACA}$$

$$\text{GCTG} \longrightarrow \text{CTGT} \longrightarrow \text{TGTA}$$

Therefore, one prefers to identify a k-mer z with its reverse complement $\overleftarrow{z^{-1}}$ and represent them by the same vertex in the graph:

Example 2.6 reverse complement (Vorlesung)

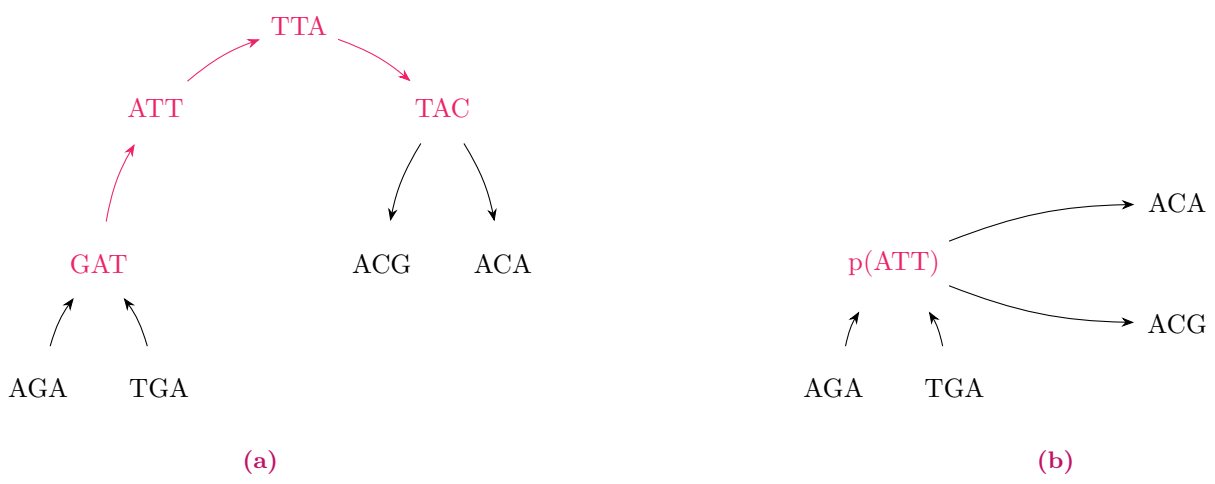


Definition 2.4 Bidirected de Bruijn subgraph Erweitert den klassischen de-Bruijn-Graphen, indem jedes k-Mer mit seinem Reverse-Komplement identifiziert wird. Beide Sequenzen werden somit durch denselben Knoten repräsentiert. Diese Darstellung berücksichtigt, dass bei DNA-Sequenzierungen häufig nicht bekannt ist, von welchem Strang der DNA-Doppelhelix ein Read stammt. Die übrige Struktur des de-Bruijn-Graphen, insbesondere die Definition der Kanten, bleibt unverändert..

Finally, in a (bidirected) de Bruijn subgraph one often finds stretches of vertices that have only one successor, the next k-mer in the originating sequence(s). Such a non-branching path can be collapsed into a single vertex (representing a k'-mer for some $k' \geq k$),called unitig, saving space and often computation time.

Definition 2.5 Compacted de Bruijn subgraph Entsteht durch das Zusammenfassen nicht verzweigender Pfade eines de-Bruijn-Graphen. Besitzt eine Folge aufeinanderfolgender Knoten jeweils genau einen Nachfolger und einen Vorgänger (mit Ausnahme der Endknoten), so wird diese zu einem einzelnen Knoten, einem sogenannten *Unitig*, zusammengefasst. Dieser repräsentiert ein längeres Teilstück der ursprünglichen Sequenz. Durch diese Kompaktierung werden Speicherbedarf und häufig auch die Laufzeit nachfolgender Algorithmen reduziert, ohne dass die wesentliche Struktur des Graphen verloren geht..

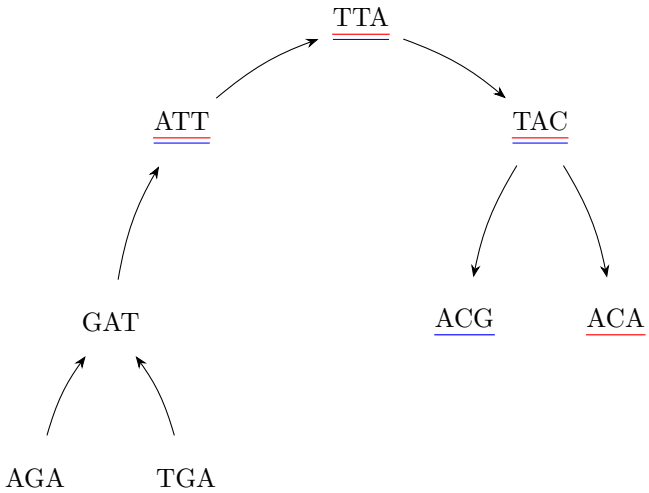
Example 2.7 for compacted de Bruijn subgraph (Vorlesung)



An additional variant comes into play when the input sequences are partitioned into several groups, for example representing different genomes, and these groups shall also be distinguished in the de Bruijn graph. Then one assigns a color (usually a single bit indicating presence or absence) to each group. A k-mer inherits this color, so that each k-mer gets assigned a color set, indicating in which groups it appears and in which not.

Definition 2.6 Colored de Bruijn subgraph Erweitert den de-Bruijn-Graphen um Informationen über die Herkunft der k-Mere. Hierzu werden die Eingabesequenzen in mehrere Gruppen, beispielsweise verschiedene Genome oder Datensätze, unterteilt. Jeder Gruppe wird eine Farbe zugewiesen. Ein Knoten erhält die Menge aller Farben der Gruppen, in denen das entsprechende k-Mer vorkommt. Dadurch lässt sich nachvollziehen, in welchen Sequenzmengen ein k-Mer enthalten ist. Colored de-Bruijn-Graphen eignen sich insbesondere für den Vergleich mehrerer Genome, erfordern jedoch aufgrund der Speicherung der Farbinformationen deutlich mehr Speicher als klassische de-Bruijn-Graphen..

Example 2.8 for colored de Bruijn subgraph (Vorlesung)



Problem 2.2 Colored de Bruijn subgraphs form a very powerful data structure, but their memory-efficient implementation is not trivial. In fact, the main challenge is no longer the storage of the k-mers (whose number is limited by σ^k , see above), but the storage of the colors, which may occupy several thousand bits per k-mer in case of large pangenomes.
log der alphabetgröße zur basis 2 ist wie viele bits man braucht zum codieren k*2 bits für dbg, aber mit Farben muss man ja noch bits dran hängen, das ist groß und noch ungelöst, denn es ist dann k*2 + # datensätze

Example 2.9 bsp zu bits (Vorlesung) acg codiert als 000110 wenn

A	0	0
C	0	1
G	1	0
T	1	1

2.4 | Implementation of de Bruijn Graphs: Sets of k-mers

Here we consider only the basic variant of de Bruijn subgraphs. With a little bit of effort all results can also be generalized to the other graph types.

A simple observation allows to save considerable space: Assume we have a very fast method to test if a certain k-mer is represented in de Bruijn graph $G = DBG(s, k)$ or not. Then one does not need to store the edges explicitly, for the following reason. When traversing from a vertex representing k-mer au to an adjacent vertex representing k-mer ub (a,b$\in \Sigma, u \in \Sigma^{k-1}$), instead of following the edge $au \to ub$ explicitly (if it exists), we can instead just test if ub is represented as a vertex in G or not. If the test is successful, we know that the edge exists and therefore can be traversed. If the test is unsuccessful, then the edge does not exist and can not be traversed. Similarly, if we want to find all possible successors of the vertex representing k-mer au, we perform σ tests, one for each possible last character c of the new, overlapping k-mer uc. Especially for small alphabets like DNA, this is not very much work.

Therefore, in (DNA-) bioinformatics applications, a data structure is very popular that represents only a set S of k-mers and offers a very quick presence/absence test, instead of storing a complete graph data structure: the Bloom filter.

The idea is to use one large bit array B of length m, say, as a hash table with all bits initially set to $false$ (F), and then have a small number of h different hash functions $f_1, f_2, \cdots, f_h$ that map k-mers (respectively, their ranks) to integers in the range [0,m−1]. In order to store a k-mer (respectively its rank r), the hash functions are computed

and the corresponding bits in B are set:

$$B[f_1(r)] = B[f_2(r)] = \cdots = B[f_h(r)] = T.$$

This is repeated for each k-mer.
As always, the hash functions should be independent and uniformly distributed. A very simple one is of the form $f_i(r) = (r^{i+1} \gg k) \mod m$, where $\gg k$ denotes k-fold bitwise right-shift. There exist, however, much more advanced hash functions in the literature.

In order to test whether a k-mer with rank r has been stored in a Bloom filter B, we perform the same procedure: We evaluate all hash functions in parallel, and if all of them point to T entries in the hash table, then it is very likely that the element r has been stored in B:

$$\text{probably-contains}(B, r) = B[f_1(r)] \wedge B[f_2(r)] \wedge \cdots \wedge B[f_h(r)],$$

where $\wedge$ denotes logical AND.

By the combination of several distinct hash functions, a k-mer z (with $rank\ r = rank(z)$) will be reported as a false positive, although it has not been stored in the Bloom filter, only if all hash functions are involved in collisions, i.e., if for each of the h hash values $f_i(r), 1 \leq i \leq h$, some other k-mer $z' \neq z$ produces with some hash function fi the same hash value ($f'_i(rank(z')) = f_i(r)$) and therefore sets B at this position to T. While this is quite unlikely in practice as long as the hash table is only moderately filled, it can not be completely avoided. Therefore the above function is called "probably-contains" and not "contains". There are some techniques to handle false positives in certain applications manually, so that they do not distort the result and the Bloom filter becomes exact. [2]

CHAPTER 3

Approximative String Matching

Comment 3.1 "In computer science, approximate string matching (often colloquially referred to as fuzzy string searching) is the technique of finding strings that match a pattern approximately (rather than exactly). The problem of approximate string matching is typically divided into two sub-problems: finding approximate substring matches inside a given string and finding dictionary strings that match the pattern approximately." - [7]

Idea 3.1 Semi-globales Alignment: Finde alle Endpositionen j im Text y, an denen das Muster x mit Editdistanz $\leq k$ endet. Trick: 0. Zeile $= 0$ (Muster darf überall im Text starten), letzte Zeile $\leq k \Rightarrow$ Treffer (Darf überall im text enden).

"In the "Sequence Analysis 1" class, we have already discussed a variant of the Universal Alignment Algorithm that can be used to find a best (highest score) alignment of a (short) pattern x within a (long) text y: semi-global alignment. Often, we are interested in all matches with at least a certain score. A simple modification makes this possible: We just need to visit all predecessors of the final cell vE and check whether the corresponding solutions fulfill the given criterion. (In the case of semi-global alignment, these cells correspond to the last row of the alignment matrix.)

In this section, we consider the cost-based formulation and therefore alignment matrix D. We assume that a sensible cost function cost: $\mathcal{A} \to \mathbb{R}_0^+$ for alignment columns is given where $cost(((\binom{a}{a}))) = 0$, $cost(((\binom{a}{b}))) > 0$ for $a \neq b$ and all indels have the same cost $\gamma > 0$.
Then we consider the following problem variant: " [2]

Definition 3.1 Approximate string matching problem todo.

"In fact, it suffices to discover the ending positions j of the substrings y' because we can always reconstruct the whole substring and its starting position by backtracing." [2]

Definition 3.2 k-approximate matches of x in y $i^*(C) := \max\{\, i \mid C(i) \leq k \,\}$

3.1 | Sellers' Algorithm

Comment 3.2 Elaborate example on page 73.

"For a change in style, here we present an explicit formulation of an algorithm that solves the problem not as a recurrence, but in pseudo-code notation. Algorithm 1 is known as Sellers' algorithm (Sellers, 1980). Of course, it corresponds to the Universal Alignment Algorithm variant discussed in the "Sequence Analysis 1" class for semi-global alignment, except that we do not look at the final vertex vE to find the best match, but look at all the vertices in the last row to identify all matches with $D(m, j) \leq k \; for \; 0 \leq j \leq n$.

Looking closely at Algorithm 1, we see that each iteration $j = 1, \ldots, n$ transforms the previous column $C_{j-1} := D(\cdot, j-1)$ of the edit matrix into the current column $C_j := D(\cdot, j)$, based on character $y[j]$. Starting with C_0, Sellers' algorithm effectively consists of repeatedly computing C_j from C_{j-1} and the next text character $y[j]$, for each $j = 1, \ldots, n$. " [2]

Algorithm 1 Sellers' Algorithm

Input: pattern $x \in \Sigma^m$, text $y \in \Sigma^n$, threshold $k \in \mathbb{N}_0$, sensible cost function with indel cost $\gamma > 0$, defining an edit distance $d(\cdot, \cdot)$
Output: ending positions j such that $d(x, y') \leq k$ where $y' := y[j'...j]$ for some $j' \leq j$
// initialize the 0-th column of the alignment matrix
for $i \leftarrow 0, ..., m$ **do**
  $C_0(i) \leftarrow i \cdot \gamma$
end for
// proceed column-by-column
for $j \leftarrow 1, ..., n$ **do**
  $C_j(0) \leftarrow 0$
  for $i \leftarrow 1, ..., m$ **do**

$$C_j(i) \leftarrow min \begin{cases} C_{j-1}(i-1) + cost(x[i], y[j]) \\ C_{j-1}(i) + \gamma \\ C_j(i-1) + \gamma \end{cases}$$

  end for
  if $C_j(m) \leq k$ **then**
//report match ending at column j and its cost
report $(j, C_j(m))$
  end if
end for

Comment 3.3 Hier ist: Zeilen = pattern und Spalten = text. Nicht wundern, einige Erklärungen machen das andersherum.

Comment 3.4 Die erste Spalte ("0-th column of the alignment matrix") beinhaltet sozusagen die Kosten den Text des patterns bis zu der Stelle an einen leeren Text anzupassen.

"pattern"	*Kosten*
ϵ	0
A	1
AC	2
ACG	3
$ACGT$	4

Deswegen ja auch $C_0(i) \leftarrow i \cdot \gamma$, also $C_0(i) = i \cdot \gamma$. Denn γ ist die Kost für ein Indel und wenn man i viele Zeichen löschen will, muss man eben i-oft mal γ bezahlen.

Comment 3.5 Wichtiger Unterschied zu anderen Algorithmen: Das Pattern darf an jeder beliebigen Position im Text beginnen! Deswegen ist die erste Zeile eine Nullzeile...sozusagen. Dann kann man mit Null Kosten jederzeit einsteigen und muss nicht für das Überspringen etwas bezahlen.
Betrachtet man $TTACG$ und ACG, dann würde man bspw. bei Wagner-Fischer berechnen

$$d(ACG, TTACG) = 2$$

und bei Sellers würde man berechnen

$$d(ACG, ACG) = 0,$$

weil man das "TT" einfach überspringen kann.
Das liegt auch daran, dass Sellers eine andere Fragestellung als Wagner-Fischer verfolgt. Sellers will kein komplettes

Alignment finden, nur ein ungefähres.
Deswegen darf es auch irgendwo im Text enden und man muss die ganze letzte Zeile betrachten (und schauen wo der höchste Wert ist) anstatt nur ganz rechts unten.

Since we only report end positions and costs, "there is no need to store back pointers and the entire alignment matrix D in Algorithm 1. The current and the previous column suffice. This decreases the space requirement to O(m), where m is the (short) pattern length. We still need to store the text, which is O(n), however this need not necessarily be in memory." [2]

If we have a very good match "with cost $< k$ at position j, the neighboring positions will also match with cost $\leq k$. To avoid this redundancy, it makes sense to post-process the output and only look for runs of local minima. Without formally defining it, if $k = 3$ and the respective costs at positions $j-2, \ldots, j+4$ are $(3, 2, 1, 1, 2, 2, 3)$, the original algorithm will report all of these positions. In most applications, however, we are only interested in the locally minimal value 1, which occurs as a run of length 2. So it would make sense to report the run interval and the minimum cost as $([j, j+1], 1)$." [2]

Note 3.1 "Note that at no time in the algorithm we need to know the exact value of $C_j(m)$ as long as we can be sure that it exceeds k and therefore will not be reported." [2]

Example 3.1 Beispielimplementierung Sellers Von Franzili [1]

```
/**
 * This function computes a distance matrix with the Seller's Algorithm.
 *
 * Returns a Pair of
 * 1. The distance-matrix as a result of the pairwise Alignment
 * 2. The columns where a match ends and the corresponding costs
 */
fun sellers(pattern: String, text: String, costs: Costs, emptyStringSymbol: Char): Pair<Array<IntArray>, ColAndCosts> {

    // Pairs of columns where a match ends and the corresponding costs
    val outputColumns: MutableList<Pair<Int, Int>> = ArrayList()

    // Add a symbol to the beginning of the given words, that are not part of the alphabet
    val patternE: String = addEmptyString(pattern, emptyStringSymbol)
    val textE: String    = addEmptyString(text, emptyStringSymbol)

    // Matrix -> Array of IntArrays
    val dMatrix: Array<IntArray> = Array(patternE.length) { IntArray(textE.length) { Int.MAX_VALUE } }

    // Initialize the first column
    for (i in dMatrix.indices) {
        dMatrix[i][0] = i * costs.gap
    }

    // Proceed column-wise
    val n = dMatrix[0].size     // text
    val m = dMatrix.size        // pattern
    for (j in (1 until n)) {
        dMatrix[0][j] = 0
        for (i in (1 until m)) {
            val diagonal  = dMatrix[i-1][j-1] + cost(patternE[i], textE[j],
                            Costs(costs.threshold, costs.match, costs.mismatch, costs.gap))
            val left      = dMatrix[i][j-1] + costs.gap
            val above     = dMatrix[i-1][j] + costs.gap
            dMatrix[i][j] = min(min(diagonal, left), above)
        }
        // If we are in the last row and the costs are under the threshold, return this column
        // and the costs
        if (dMatrix[m-1][j] <= costs.threshold) {
            outputColumns.add(Pair(j, dMatrix[m - 1][j]))
        }
    }
```

```
    val toAlign = Pair(patternE, textE)
    printDictanceMatrix(DistanceMatrix(dMatrix, toAlign))
    return Pair(dMatrix, outputColumns)
}
```

Ergänzung 3.1 Vergleich der Implementierung mit dem Pseudocode

Beschreibung	Pseudocode	Implementierung
Input Muster	pattern $x \in \Sigma^m$	pattern: String
Input Text	text $y \in \Sigma^n$	text: String
Input cost Threshold	threshold $k \in \mathbb{N}_0$	in Kosten enthalten
Input cost function	sensible cost function with indel cost $\gamma > 0$	costs: Costs
Input edit distance	$d(\cdot,\cdot)$	
Length of pattern	m	val m = dMatrix.size
Length of text	n	val n = dMatrix[0].size
Epsilon/empty word column and row	assumed	"Add a symbol to the beginning of the given words, that are not part of the alphabet"
		val patternE: String = addEmptyString(pattern, emptyStringSymbol
		val textE: String = addEmptyString(text, emptyStringSymbol)
(Empty) matrix	assumed	val dMatrix: Array<IntArray> = Array(patternE.length) { IntArray(textE.length) { Int.MAX_VALUE } }
Output	ending positions j such that $d(x, y') \leq k$ where $y' := y[j'...j]$ for some $j' \leq j$	val outputColumns
initialize the 0-th column of the alignment matrix	For $i \leftarrow 0, ..., m$ do $C_0(i) \leftarrow i \cdot \gamma$	for (i in dMatrix.indices) { dMatrix[i][0] = i * costs.gap }
proceed column-by-column	For $j \leftarrow 1, ..., n$	for (j in (1 until n))
And initialize first row	$C_j(0) \leftarrow 0$	dMatrix[0][j] = 0
For each row in each column	For $i \leftarrow 1, ..., m$	for (i in (1 until m))
calculate the cost/recurrence	$C_j(i) = min$ ...	min(min(diagonal, left), above)
Wert links oben + score/Kostenfunktion	$C_{j-1}(i-1) + cost(x[i], y[j])$	val diagonal = dMatrix[i-1][j-1] + cost(patternE[i], textE[j], Costs(costs.threshold, costs.match, costs.mismatch, costs.gap))
Wert links + Indel Kosten	$C_{j-1}(i) + \gamma$	val left = dMatrix[i][j-1] + costs.gap
Wert oben + Indel Kosten	$C_j(i-1) + \gamma$	val above = dMatrix[i-1][j] + costs.gap
In case the threshold is reached	If $C_j(m) \leq k$	if (dMatrix[m-1][j] <= costs.threshold
report match ending at column j and its cost	report $(j, C_j(m))$	outputColumns.add(Pair(j, dMatrix[m - 1][j]))
Ausgabe	assumed	return Pair(dMatrix, outputColumns)

Comment 3.6 Zum Threshold k Wenn $k = 0$, werden nur exakte Matches erlaubt. Bei $k = 1$ dementsprechend nur maximal ein fehler, etc.

Zusammenfassung 3.1 $N(i,j) = \max$. Basenpaare in $s_i \dots s_j$; Basis: $N(i,i) = N(i,i-1) = 0$.

$$N(i,j) = \max \begin{cases} N(i+1,\, j) & (i \text{ ungepaart}) \\ N(i,\, j-1) & (j \text{ ungepaart}) \\ N(i+1,\, j-1) + \text{score}(s_i, s_j) & (i, j \text{ paaren, falls } j - i > \delta) \\ \max\limits_{i<k<j} \; N(i,k) + N(k+1,j) & (\text{Bifurkation}) \end{cases}$$

$\delta = \min$. Schleifenlänge (hier $\delta = 1$): ein Paar (i,j) nur, wenn $j - i > \delta$, also $j - i \geq 2$.
2-Regel-Form (äquivalent): $N(i,j) = \max\big(N(i+1,j-1) + \text{score}(s_i,s_j),\ \max_{i \leq k < j} N(i,k) + N(k+1,j)\big)$, denn "'$i/j$ ungepaart"' steckt in der Bifurkation mit $k = i$ bzw. $N(i,i) = 0$.

Warnung 3.1 Achtung:

- **0. Zeile** $= 0$ (nicht $j \cdot \gamma$!). Das ist der Unterschied zum globalen Alignment und macht die Suche semi-global.
- Es werden **alle** Endpositionen gemeldet, nicht nur das beste Match.

- **Runs of local minima:** sehr gute Matches erzeugen mehrere benachbarte Treffer $\leq k$. Praxis: nur **lokale Minima** (Kostenketten $\ldots, k, k-1, k, \ldots$) melden, um Redundanz zu vermeiden.

3.2 | Ukkonen's Cutoff Algorithm

Idea 3.2 Beschleunigung von Sellers: Zellen mit Wert $> k$ (weit unten in der Spalte) sind für das Ergebnis irrelevant. Man führt einen Last essential index ℓ mit und rechnet jede Spalte nur bis ℓ $(+1)$. Erwartete Laufzeit $O(kn)$ statt $O(mn)$.

Comment 3.7 Elaborate example on page 81.

"The last of the above three observations leads to a considerable improvement of Sellers' algorithm: If we know that in column j all values after a certain index ${i^*}_j$ exceed k, we do not need to compute them (e.g. we could assume that they are all equal to ∞ or $k+1$). The following definition captures this formally." [2]

Definition 3.3 last essential index i^* (for threshold k) of a column C todo

Example 3.2 Extrabeispiel - Visualisierung was man sich bei Ukkonen Cutoff spart Last essential indices for $k = 2$ highlighted green-ish und die Werte, wo man sich durch Ukkonen Cutoff bei der Berechnung etwas spart sind rötlich

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	0
A	1	1	1	0	1	1
C	2	2	2	1	0	1
G	3	3	3	2	1	0

Comment 3.8 Weiteres Beispiel/Visualisierung von last esential indoices unter Variation von k auf Seite 80.

"The last essential index ${i^*}_j$ of column C_j is ${i^*}_j := i^*(C_j) = \max\{i | C_j(i) \leq k\}$. From the definition it is clear that we cannot miss any k-approximate matches if we do not compute the cells below the last essential index.

To set this in operation, the last essential index ${i^*}_0$ of the 0-th column is easily found, because the scores increase monotonically from 0 to $m \cdot \gamma$. The other columns, however, are not necessarily monotone! Therefore, as we move column-by-column through the matrix, we have to make sure that we can efficiently find the last essential index of the next column, given the previous column. Consider the effects of the C_{j-1}-entries below the last essential index on the next column C_j. Since $C_{j-1}(i) > k$ for all $i > {i^*}_j - 1$ and costs are non-negative, these cells provide candidate values $> k for C_j(i)$ for $i > {i^*}_{j-1} + 1$ which therefore do not need to be considered during the minimization. The only chance that such $C_j(i)$ remain bounded by k is vertically, i.e., if $C_j(i) = C_j(i-1) + \gamma$. As soon as $C_j(i) > k$ for some $i > {i^*}_{j-1} + 1$, we know that the last essential index ${i^*}_j$ of C_j occurs before that i. Algorithm 3 shows the details." [2]

Algorithm 3 Ukkonens cutoff algorithm

Input: Pattern $x \in \Sigma^m$, Text $y \in \Sigma^n$, Schwellenwert $k \in \mathbb{N}_0$, Kostenfunktion mit Indel-Kosten $\gamma > 0$
Output: Endpositionen j mit $d(x, y') \leq k$, wobei $y' = y[j' \dots j]$ für ein $j' \leq j$

▷ Initialize the 0-th column of the alignment matrix

$i_0^* \leftarrow \min\{m, \lfloor k/\gamma \rfloor\}$
for $i \leftarrow 0$ to i_0^* **do**
  $C_0(i) \leftarrow i \cdot \gamma$
end for

▷ Process the matrix column by column

for $j \leftarrow 1$ to n **do**
  $C_j(0) \leftarrow 0$
  $i_j^* \leftarrow 0$
  $i \leftarrow 1$
  while $i \leq i_{j-1}^*$ **do**

$$C_j(i) \leftarrow \min \begin{cases} C_{j-1}(i-1) + \text{cost}(x[i], y[j]), \\ C_{j-1}(i) + \gamma, \\ C_j(i-1) + \gamma \end{cases}$$

    if $C_j(i) \leq k$ **then**
      $i_j^* \leftarrow i$
    end if
    $i \leftarrow i + 1$
  end while
  if $i \leq m$ **then**

$$C_j(i) \leftarrow \min \begin{cases} C_{j-1}(i-1) + \text{cost}(x[i], y[j]), \\ C_j(i-1) + \gamma \end{cases}$$

    if $C_j(i) \leq k$ **then**
      $i_j^* \leftarrow i$
    end if
    $i \leftarrow i + 1$
    while $i \leq m$ **and** $C_j(i-1) + \gamma \leq k$ **do**
      $C_j(i) \leftarrow C_j(i-1) + \gamma$
      if $C_j(i) \leq k$ **then**
        $i_j^* \leftarrow i$
      end if
      $i \leftarrow i + 1$
    end while
  end if
  if $i_j^* = m$ **and** $C_j(m) \leq k$ **then** ▷ report match ending at column j and its cost
    return $(j, C_j(m))$
  end if
end for

Example 3.3 Adapted from script Let $x =$ `AABB`, $y =$ `BABAABAABBABAA` and $k = 1$. The following table shows which values are computed by Algorithm 3 considering unit cost. The last essential index in each column is highlighted green-ish . The k-approximate matches are underlined in the last row.

	$x \backslash y$	ϵ	B	A	B	A	A	B	A	A	B	B	A	B	A	A
$i = 0$	ϵ	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$i = 1$	A	1	1	0	1	0	0	1	0	0	1	1	0	1	0	0
$i = 2$	A		2	1	1	1	0	1	1	0	1	2	1	1	1	0
$i = 3$	B				1	2	1	0	1	1	0	1	2	1	2	1
$i = 4$	B					2		$\underline{1}$	$\underline{1}$	2	$\underline{1}$	$\underline{0}$	$\underline{1}$	2	2	
								I	II		II	IV	V			

Five exemplary alignments are shown below, one for each of the underlined positions.

	I	II	III	IV	V
x :	AABB	AABB	AABB	AABB	AABB-
y :	AAB-	AABA	AAB-	AABB	AABBA

Example 3.4 Erweitert Direkter Vergleich zu einer Sequenzen analog zu Bsp. 3.3

Sellers – volle Matrix

ε		C	A	C	A	A	C	A	A	C	C	A	C	A	A
ε	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	1	1	0	1	0	0	1	0	0	1	1	0	1	0	0
A	2	2	1	1	1	0	1	1	0	1	2	1	1	1	0
C	3	2•	2•	1	2	1	0	1	1	0	1	2	1	2	1
C	4	3•	3•	2•	2	2•	1	1	2	1	0	1	2	2	2•

Ukkonen – Cutoff

ε		C	A	C	A	A	C	A	A	C	C	A	C	A	A
ε	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	1	1	0	1	0	0	1	0	0	1	1	0	1	0	0
A		2	1	1	1	0	1	1	0	1	2	1	1	1	0
C				1	2	1	0	1	1	0	1	2	1	2	1
C					2		1	1	2	1	0	1	2	2	

Treffer (≤k) · aktiv ≤k · berechnet >k · von Ukkonen übersprungen · Cutoff-Linie

Zelle (i=4, j=14) · Cutoff in Spalte 14: lact=3 · Ukkonen spart bisher 7 Zellen · Treffer bisher: 5

Unterschied: Sellers berechnet (4,14) = 2 – Ukkonen überspringt diese Zelle (unter dem Cutoff).

"Sometimes, because it was first published by Ukkonen (1985), this algorithm is referred to as Ukkonen's cutoff algorithm, although it is more common to use this name if the cost function is standard unit cost. In that case, even further optimizations are possible, but we do not discuss the details here." [2]

3.3 | Vergleich von Sellers' und Ukkonen Cutoff

Aspekt	**Sellers**	**Ukkonen**	
Input			
Muster	pattern $x \in \Sigma^m$		
Text	text $y \in \Sigma^n$		
Threshold	threshold $k \in N_0$		
Kosten	sensible cost function with indel cost $\gamma > 0$, defining an edit distance $d(\cdot,\cdot)$		
Initialisierung 0-ter Spalte			
last essential index	-	$i_0^* \leftarrow \min\{m, \lfloor k/\gamma \rfloor\}$	
Range	For $i \leftarrow 0, ..., m$	For $i \leftarrow 0$ to i_0^*	
Zuweisung	$C_0(i) \leftarrow i \cdot \gamma$		
Spaltenweise Berechnung des Rests			
Range Spalten	For $j \leftarrow 1, ..., n$		
	Initialisierung		
Initialisiere Zelle 0	$C_j(0) \leftarrow 0$		
Initialisiere LEI 0	-	$i_j^* \leftarrow 0$	
	Iteration Zeilen		
	Bis Ende	**1. Bis last essential index**	**2. Falls nicht in letzter Zeile angekommen (If $i \leq m$)**
LEI final	-	-	$C_j(i) \leftarrow max\{C_{j-1}(i-1) + \text{cost}(x[i], y[j]), C_j(i-1) + \gamma\}$
Range Zeilen	**For** $i \leftarrow 1, ..., m$	**While** $i \leq i_{j-1}^*$ mit $i \leftarrow 1$; $i \leftarrow i+1$	**While** $i \leq m$ **and** $C_j(i-1)+\gamma \leq k$ mit $i \leftarrow 1$; $i \leftarrow i+1$
Zellenwert	$C_j(i) \leftarrow min\{C_{j-1}(i-1+cost(x[i], y[j]), \quad C_{j-1}(i) + \gamma, \quad C_j(i-1)+\gamma\}$		$C_j(i) \leftarrow C_j(i-1) + \gamma$
Check/Update LEI	-	If $C_j(i) \leq k$ $i_j^* \leftarrow i$	
Return			
Rückgabebedingung	If $C_j(m) \leq k$	$i_j^* = m$ **and** $C_j(m) \leq k$	
Rückgabewert	$(j, C_j(m))$		
Output			
	ending positions j such that $d(x, y') \leq k$ where $y' := y[j'...j]$ for some$j' \leq j$		

Comment 3.9 "Bis last essential index" und nicht "Bis VOR last essential index", weil der lei zwar immer neu gesetzt wird ("geschoben" wird) bis der errechnete Zellwert über dem Threshold liegt und man mit dem nächsten $i = i + 1$ aus der while Schleife herausfliegt. der LEI ist dann der Index, vor dem man rausgeflogen ist.(?)
Dann würde einmal der LEI berechnet mit $C_j(i) \leftarrow max\{C_{j-1}(i-1) + \text{cost}(x[i], y[j]), C_j(i-1) + \gamma\}$ udn dann geht es an die weiteren Indizes, wo nur noch die Indelkosten auf den Wert von oben drauf gerechnet werden.

3.4 | Search Schemes and Bidirectional Indices

Idea 3.3 Approximate Matching über einem Index (FM-Index) statt über dem Text. Muster in p Teile splitten; jedem Teil eine untere/obere Fehlerschranke zuweisen; die Teile in einer festgelegten Reihenfolge suchen und dabei mit einem bidirektionalen Index nach links und rechts erweitern. So werden viele erfolglose Pfade früh abgeschnitten.

"Assume we want to do approximate string matching with a text index, as a generalization of exact string matching that can be very efficiently solved with suffix trees, suffix arrays or the Burrows-Wheeler Transform (BWT), as we have seen. In principle one can use these data structures also for approximate matching, if the solution space of pattern alignments with all candidate substrings of the text is enumerated by backtracking. However, if this is done in a straight left-to-right (as with suffix trees) or right-to-left (as with the BWT) organization, it will be very inefficient already for small error threshold k. A better way is the use of *search schemes* in combination with more flexible *bidirectional indices* that we will study in this section.

The general idea is that the pattern x is divided into several non-overlapping parts $x_1, x_2, \ldots, x_p$. The search scheme (Kucherov et al., 2016) then formally consists of multiple searches, where a search is represented by the triple $S = (\pi, L, U)$. The permutation π of $\{1, 2, \ldots, p\}$ specifies the order in which the parts of the pattern are matched. L and U are arrays of size p such that $L[i]$ and $U[i]$ denote lower and upper bounds of the number of errors after the first i parts are matched (in the order specified by π). " [2]

Definition 3.4 Pigeonhole principle search scheme Zerlegt das Muster x bei Fehlerschwelle k in $p = k + 1$ Teile $x_1, \ldots, x_{k+1}$. Da bei $\leq k$ Fehlern mindestens ein Teil exakt vorkommen muss, besteht das Schema aus $k + 1$ Suchen S_i: Teil x_i exakt matchen, dann Backtracking nach links und rechts bis insgesamt k Fehler erreicht sind.

Pigeonhole search scheme. "A simple and very natural search scheme is the following: For error threshold k, split the pattern x in $p = k + 1$ parts $x_1, x_2, \ldots, x_{k+1}$, usually of roughly the same length. Clearly, if the whole pattern occurs in the text with at most k errors, then at least one of the parts must match exactly in the corresponding text region. Therefore, the search scheme consists of $k + 1$ searches $S_1, S_2, \ldots, S_{k+1}$ where search S_i first matches pattern part x_i exactly, followed by a backtracking search to the left $(x_{i-1}, x_{i-2}, \ldots, x_1)$ with not more than k errors, and finally a backtracking search to the right $(x_{i+1}, x_{i+2}, \ldots, x_{k+1})$, with additional errors, until a total of k is reached (or the pattern is exhausted). " [2]

Example 3.5 From script Formally, for $k = 4$ the pigeonhole principle search scheme $S = (S_1, \ldots, S_5)$ looks as follows:

$$S_1 = (12345,\ 00000,\ 04444)$$
$$S_2 = (21345,\ 00000,\ 04444)$$
$$S_3 = (32145,\ 00000,\ 04444)$$
$$S_4 = (43215,\ 00000,\ 04444)$$
$$S_5 = (54321,\ 00000,\ 04444)$$

Note that there may be redundant results that need to be filtered in a postprocessing step, for example if more than one pattern part occurs in one match with no errors.

Definition 3.5 MinU search scheme Search Scheme (Renders et al., 2024), dessen Fehlerbereiche $(L[i], U[i])$ langsamer wachsen als beim pigeonhole-Schema und das daher effizienter ist. Es ist *lossless*, d. h. kein Match mit bis zu k Fehlern wird verpasst (alle Fehlerkonfigurationen $(e_1, \ldots, e_p)$ mit $\sum e_i \leq k$ sind abgedeckt)

MinU search scheme. "Another example is the MinU search scheme (Renders et al., 2024). Its definition for $k = 4$ is as follows:

$$S_1 = (12345,\ 00222,\ 02244)$$
$$S_2 = (23145,\ 00000,\ 01244)$$
$$S_3 = (32145,\ 01111,\ 01244)$$
$$S_4 = (45321,\ 00003,\ 01444)$$
$$S_5 = (54321,\ 01114,\ 01444)$$

It is easy to see that MinU is more efficient, since the ranges $(L[i], U[i])$ grow slower than for the pigeonhole scheme. The main question is, though, whether it is *lossless*, i.e., no approximate match with up to k errors will be missed. This can be proven by showing that all possible error configurations $(e_1, e_2, \ldots, e_p)$ are covered, where e_i is the number of errors in part x_i and $e_1 + e_2 + \ldots + e_p \leq k$. This has been shown for the MinU scheme, although we will not go into details here. " [2]

Definition 3.6 bidirectional text indices Textindizes, die einen partiellen Match flexibel *sowohl nach links als auch nach rechts* verlängern können, wie es Search Schemes erfordern. Beispiele: Affix-Bäume/-Arrays, bidirektionale BWT, bidirektionaler Wavelet-Index, br-index, bidirektionaler Move-Index

Bidirectional indices. "In order to combine search schemes with a text index, it is necessary that the search can be organized more flexibly than just left-to-right or just right-to-left. To this end, bidirectional text indices are very useful. They allow for extending a partial match either to the left or to the right, just as the search scheme requires. Several bidirectional indices have been developed over time, starting with affix trees (Stoye, 1995, 2000; Maaß, 2003) and affix arrays (Strothmann, 2007), followed by the bidirectional BWT (Lam et al., 2009), the bidirectional wavelet index (Schnattinger et al., 2012), the br-index (Arakawa et al., 2022) and finally the bidirectional move index (Depuydt et al., 2025). Details go beyond the scope of these lecture notes.

" [2]

CHAPTER 4

Biologically Inspired Alignment Scores

"The unit cost edit distance simply counts certain differences between strings. There is no inherent biological meaning behind this scheme. If we want to compare DNA or protein sequences, for example, biologically more meaningful distances should be used." [2]

Definition 4.1 Transversion/transition cost Biologisch motivierte Kostenfunktion auf dem DNA-Alphabet: Purin/Purin- und Pyrimidin/Pyrimidin-Ersetzungen (Transitionen) kosten 1, Purin/Pyrimidin-Ersetzungen (Transversionen) kosten 2; oft mit Indel-Kosten 3. Reflektiert, dass Transitionen wahrscheinlicher sind als Transversionen.

"The following is called the *transversion/transition cost* on the DNA alphabet:

	A	G	C	T
A	0	1	2	2
G	1	0	2	2
C	2	2	0	1
T	2	2	1	0

Bases A and G are called *purines*, and bases C and T are called *pyrimidines*. The transversion/transition cost function reflects the biological fact that a purine/purine and a pyrimidine/pyrimidine replacement is more likely to occur than a purine/pyrimidine replacement. Often, an insertion/deletion cost of 3 is used with the above costs to take into account that a deletion or an insertion of a base is even more seldom. These costs define the transition/transversion distance between two DNA sequences.

To compare protein sequences and define a cost function on the amino acid alphabet, we may count how many DNA bases must change minimally in a codon to transform one amino acid into another.

A more sophisticated cost function for replacement of amino acids was calculated by W. Taylor according to the method described in Taylor and Jones (1993) and is shown in Table 4.1. (To completely define a distance function, we would also have to define the costs for insertions and deletions.)" [2]

Note 4.1 "Note, however, that in a distance model a match has always zero cost, $\text{cost}\binom{a}{a} = 0$ for all $a \in \Sigma$. This restricts flexibility a little bit because any match is scored in the same way." [2]

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	0	14	7	9	20	9	8	7	12	13	17	11	14	24	7	6	4	32	23	11
R	14	0	11	15	26	11	14	17	11	18	19	7	16	25	13	12	13	25	24	18
N	7	11	0	6	23	5	6	10	7	16	19	7	16	25	10	7	7	30	23	15
D	9	15	6	0	25	6	3	9	11	19	22	10	19	29	11	11	10	34	27	17
C	20	26	23	25	0	26	25	21	25	22	26	25	26	26	22	20	21	34	22	21
Q	9	11	5	6	26	0	5	12	7	17	20	8	16	27	10	11	10	32	26	16
E	8	14	6	3	25	5	0	9	10	17	21	10	17	28	11	10	9	34	26	16
G	7	17	10	9	21	12	9	0	15	17	21	13	18	27	10	9	9	33	26	15
H	12	11	7	11	25	7	10	15	0	17	19	10	17	24	13	11	11	30	21	16
I	13	18	16	19	22	17	17	17	17	0	9	17	8	17	16	14	12	31	18	4
L	17	19	19	22	26	20	21	21	19	9	0	19	7	14	20	19	17	27	18	10
K	11	7	7	10	25	8	10	13	10	17	19	0	15	26	11	10	10	29	25	16
M	14	16	16	19	26	16	17	18	17	8	7	15	0	18	17	16	13	29	20	8
F	24	25	25	29	26	27	28	27	24	17	14	26	18	0	27	24	23	24	8	19
P	7	13	10	11	22	10	11	10	13	16	20	11	17	27	0	9	8	32	26	14
S	6	12	7	11	20	11	10	9	11	14	19	10	16	24	9	0	5	29	22	13
T	4	13	7	10	21	10	9	9	11	12	17	10	13	23	8	5	0	31	22	10
W	32	25	30	34	34	32	34	33	30	31	27	29	29	24	32	29	31	0	25	32
Y	23	24	23	27	22	26	26	26	21	18	18	25	20	8	26	22	22	25	0	20
V	11	18	15	17	21	16	16	15	16	4	10	16	8	19	14	13	10	32	20	0

Table 4.1: Amino acid replacement costs as suggested by W. Taylor.

4.1 | Sensible Similarity Scores

"The concept of a metric is useful because it embodies the essential properties of our intuition about distances. For example, if the triangle inequality is violated, we could find a shorter way from x to y via a detour over z, which is not intuitive. However, distances also have disadvantages, as we have seen in the discussion of local alignment, where distance functions are not useful." [2]

Definition 4.2 General similarity score Allgemeine Ähnlichkeitsfunktion, bei der Matches positiv, Mismatches und Indels negativ bewertet werden – im Gegensatz zu einem Distanzmodell, in dem ein Match stets Kosten 0 hat. Nützlich insbesondere fürs lokale Alignment, wo Distanzfunktionen ungeeignet sind.

"Similarity scores, instead, where matches are scored positively, while mismatches and indels receive negative scores, provide a useful alternative. Here we consider a general similarity score that should satisfy certain intuitive properties." [2]

Definition 4.3 sensible score function for columns in a pairwise alignment a function score : $\mathcal{A}(\Sigma) \to \mathbb{R}$ such that

$$\begin{aligned}
&\text{score}\left(\binom{a}{a}\right) \geq 0 \geq \text{score}\left(\binom{a}{-}\right) = \text{score}\left(\binom{-}{a}\right) && \text{for all } a \in \Sigma \\
&\text{score}\left(\binom{a}{a}\right) \geq \text{score}\left(\binom{a}{c}\right) = \text{score}\left(\binom{c}{a}\right) && \text{for all } a, c \in \Sigma \\
&\text{score}\left(\binom{a}{c}\right) \geq \text{score}\left(\binom{a}{-}\right) + \text{score}\left(\binom{-}{c}\right) && \text{for all } a, c \in \Sigma
\end{aligned} \tag{4.1}$$

Comment 4.1 Also

1. Ein Match bekommt einen positiven Score und eine Lücke einen negativen
2. Mismatch-Scores sind kleiner als Match-Scores
3. Scores sind immer Symmetrisch
4. Substitution bekommt besseren Score als erst Deletion und dann Insertion

"The first and second conditions impose symmetry and state that the similarity between a and itself is always higher than between a and anything else. In particular, insertions and deletions never score positively. The third condition states that a direct substitution is preferable to a deletion followed by an insertion.

In general, it is a good idea to verify that a score function is sensible, like in the case of log-odds score matrices introduced in the next section. In other cases, one might deviate from this property for good reason, see Section 4.3." [2]

4.2 | Log-Odds Score Matrices

Idea 4.1 Scores sind keine Willkür, sondern Log-Odds: Wie viel wahrscheinlicher ist ein Alignment-Paar (a, b) unter dem *Homologie-Modell* M als unter dem Zufalls-Modell R? Positiver Score = eher homolog.

"Now it is time to discuss how we can define similarity scores for biological sequences from an evolutionary point of view; we focus on proteins and define a similarity function on the alphabet Σ of 20 amino acids.

The basic observation is that over evolutionary time-scales, in functional proteins two similar amino acids are replaced more frequently by each other than dissimilar amino acids. The twist is now to define similarity by observing how often one amino acid is replaced by another one in a given amount of time." [2]

Definition 4.4 frequency vector Vektor π der natürlich vorkommenden Aminosäuren mit $\pi_a \geq 0$ für alle $a \in \Sigma$ und $\sum_{a\in\Sigma} \pi_a = 1$. Für zwei zufällige Positionen ist die Wahrscheinlichkeit des geordneten Paares (a, b) gleich $\pi_a \cdot \pi_b$:

$$\pi = (\pi_a)_{a\in\Sigma} \tag{4.2}$$

"Let $\pi = (\pi_a)_{a\in\Sigma}$ be the frequency vector, which means that $\pi_a \geq 0$ for all $a \in \Sigma$ and $\sum_{a\in\Sigma} \pi_a = 1$, of the naturally occurring amino acids. If we look at two random positions in two randomly selected proteins, the probability that we see the ordered pair (a, b) is then $\pi_a \cdot \pi_b$. The entries of π are also called *background frequencies* of the amino acids." [2]

Definition 4.5 Background frequencies Die Einträge des frequency vector π; gewinnbar als Randverteilungen (Marginals) von M_t: $\pi_a = \sum_{b\in\Sigma} M_t(a, b)$ und $\pi_b = \sum_{a\in\Sigma} M_t(a, b)$ (wegen Symmetrie).

Definition 4.6 Fixed evolutionary time period Zeitraum t, über den die Ersetzung einzelner Aminosäuren verfolgt wird. t ist so zu wählen, dass die Score-Matrix zum Problem passt: Für Sequenzen mit gemeinsamem Vorfahren vor $\approx t/2$ sollte eine aus Paaren mit Divergenz t konstruierte Matrix genutzt werden.

"Now assume that we can track the fate of individual amino acids over a fixed evolutionary time period t.

Let $M_t(a, b)$ be the observed frequency table of homologous amino acid pairs where we observed a at the beginning of the period and b at the end of the period, such that $\sum_{a,b} M_t(a, b) = 1$. Usually, we count substitutions and identities twice, i.e., once in each direction. This ensures the symmetry of M_t: $M_t(a, b) = M_t(b, a)$. The reason is that we can in fact not track amino acids during evolution; we can only observe the state of two present-day sequences and do not know their most recent common ancestor. The fields of molecular evolution and phylogenetics examine more of the resulting implications.

We can find the background frequencies as the marginals of M_t: $\pi_a = \sum_{b\in\Sigma} M_t(a, b)$ and $\pi_b = \sum_{a\in\Sigma} M_t(a, b)$ because of symmetry.

There are two reasons why an entry $M_t(a, b)$ can be relatively large. The first reason is the one we are interested in: because a and b are similar. The second reason is that simply a or b could be frequent amino acids. To remove the second effect, we consider the so-called *likelihood ratio* $M_t(a, b)/(\pi_a \cdot \pi_b)$, which relates the probability of the pair (a, b) in an evolutionary model to its probability in a random model." [2]

Definition 4.7 Likelihood ratio Verhältnis $M_t(a,b)/(\pi_a \cdot \pi_b)$, das die Wahrscheinlichkeit des Paares (a,b) im Evolutionsmodell zu der im Zufallsmodell in Beziehung setzt. Werte > 1 bedeuten Ähnlichkeit, Werte < 1 Unähnlichkeit von a und b.

"We declare that a and b are similar if this ratio exceeds 1 and that they are dissimilar if this ratio is below 1." [2]

"To obtain an additive function, we take the logarithm and define the *log-odds score matrix* with respect to time period t by

$$\text{score}_t(a,b) := \log\left(\frac{M_t(a,b)}{\pi_a \cdot \pi_b}\right).$$

The time parameter t should be chosen in such a way that the score matrix is optimal for the problem at hand: If we want to compare two sequences whose most recent common ancestor existed, say, 530 million years ago, then we should ideally use a score matrix constructed from sequence pairs with distance $t \approx 1060$ million years." [2]

Definition 4.8 log-odds score matrix Additive Score-Matrix bzgl. Zeitraum t, definiert als Logarithmus der likelihood ratio: (Wichtige Familien: PAM(t) und BLOSUM(s))

$$\text{score}_t(a,b) := \log\left(\frac{M_t(a,b)}{\pi_a \cdot \pi_b}\right) \tag{4.3}$$

Bemerkung 4.1 For convenience, scores are often scaled by a constant factor and then rounded to integers. The score unit is called a *bit* if the score is obtained by a $\log_2(\cdot)$ operation. When multiplied by a factor of three, say, we get scores in *third-bits*.

Definition 4.9 Third-bits Score-Einheit: Ein Score ist in *bits*, wenn er per $\log_2(\cdot)$ gebildet wird; wird er zusätzlich mit dem Faktor 3 multipliziert (und gerundet), erhält man Scores in third-bits.

Bemerkung 4.2 Since we cannot easily observe how DNA or proteins change over millions of years, constructing a score matrix is somewhat difficult. Since the pioneering work of Margaret Dayhoff and colleagues in the 1970s, Markov processes are used to model molecular evolution. Methods that allow to integrate information from sequences of different divergence times in a consistent fashion have been developed more recently.

Bemerkung 4.3 Important score matrix families for proteins are the PAM(t) (Dayhoff et al., 1978) and the BLOSUM(s) (Henikoff and Henikoff, 1992) matrices. Here t is a divergence time parameter and s is a clustering similarity parameter inversely correlated to t. (The inventors of BLOSUM have chosen to index their matrices in a different way.) The BLOSUM matrices are the most widely used ones for protein sequence comparison.

4.3 | Non-symmetric score matrices

"We usually demand that similarity functions and hence score matrices are symmetric. In some cases, however, there are good reasons to choose a non-symmetric similarity function. Often then, the unsymmetry does not stem from an unsymmetry of M_t (for reasons explained above), but from different background frequencies. We mention an example.

Assume that we have a particular protein fragment (the "query") that forms a transmembrane helix. The amino acid composition in membrane regions differs strongly from that of the "average" protein: It is hydrophobically biased. Assume that we want to look for similar fragments in a large database of peptide sequences (of unknown or "average" type). It follows that the matrix M_t of pair frequencies should be one derived from an evolutionary process acting on transmembrane helices and that two different types of background frequencies should be used: The query background frequencies τ are those of the transmembrane model, different from the background frequencies π of the database. It

follows that we should use

$$\text{score}_t(a,b) := \log\left(\frac{M_t(a,b)}{\tau_a \cdot \pi_b}\right),$$

or a rounded multiple thereof, so that now $\text{score}(a,b) \neq \text{score}(b,a)$ in general. Table 4.2 shows the SLIM161 matrix for transmembrane helices (Müller et al., 2001). The 161 is the time parameter t referring to the expected number of evolutionary events per 100 positions." [2]

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	5	-8	-5	-10	3	-7	-11	0	-5	1	1	-10	1	1	-5	2	1	-2	-3	2
R	-3	10	-4	-10	-2	-3	-11	-4	-4	-2	-1	-3	-1	-2	-7	-4	-3	-2	-3	-3
N	-1	-5	8	-2	0	-2	-7	-2	2	-1	-1	-6	0	1	-5	2	0	-2	2	-2
D	-3	-9	1	9	-4	-2	1	-2	-2	-2	-2	-6	-2	-1	-5	-3	-3	-3	-2	-2
C	2	-8	-5	-11	11	-7	-12	-3	-8	0	1	-12	1	3	-11	2	0	-1	0	1
Q	-1	-2	0	-3	0	7	-4	-2	0	0	0	-4	2	2	-4	0	-1	4	1	0
E	-3	-7	-2	3	-2	-1	7	-3	-1	-2	-2	-8	-1	0	-4	-1	-3	-1	-1	-2
G	1	-7	-4	-7	0	-5	-10	7	-7	-1	0	-8	0	1	-5	1	-1	-3	-2	-1
H	-1	-5	3	-5	-2	-1	-5	-4	10	-1	-1	-8	-1	2	-7	-1	-2	1	5	-2
I	-1	-9	-7	-11	-1	-7	-12	-4	-7	6	3	-11	4	2	-6	-3	-1	-2	-3	4
L	-1	-8	-6	-10	1	-7	-12	-4	-7	4	5	-11	4	3	-7	-3	-1	-1	-2	2
K	-1	2	-1	-4	-2	0	-7	-1	-3	0	-1	6	0	0	-1	-1	-1	-1	1	-2
M	-1	-7	-6	-10	1	-5	-11	-3	-7	4	4	-11	7	3	-7	-2	0	-1	-2	2
F	-1	-9	-5	-10	2	-6	-11	-3	-4	1	2	-11	2	8	-7	-2	-2	3	4	0
P	-3	-9	-7	-9	-7	-8	-10	-4	-9	-2	-3	-8	-3	-2	11	-3	-3	-3	-4	-2
S	2	-8	-1	-8	4	-5	-9	0	-4	0	0	-9	0	1	-5	6	1	-2	-1	0
T	2	-7	-3	-8	2	-6	-10	-2	-5	2	2	-9	3	2	-4	2	4	-4	-2	2
W	-3	-7	-6	-10	0	-2	-11	-5	-3	-1	0	-10	0	4	-6	-4	-5	15	2	-2
Y	-4	-8	-2	-9	1	-5	-10	-5	0	-2	-1	-8	-1	6	-7	-3	-3	2	11	-2
V	1	-9	-7	-10	1	-7	-11	-4	-7	5	3	-12	3	2	-6	-2	0	-2	-3	5

Table 4.2: The non-symmetric SLIM 161 score matrix. Scores are given in third-bits, i.e., as $\text{score}(a,b) = \text{round}\big(3 \cdot \log_2(M(a,b)/(\tau_a \pi_b))\big)$.

4.4 | Position-Specific Scores

"The match/mismatch score in all our alignment algorithms depends on a (sensible) similarity function score as defined in Definition 4.1, often given by a log-odds score matrix. However, this does not need to be the case. The algorithm does not change in any way if indel and match/mismatch scores depend also on the position in the sequences. Therefore we can replace $\text{score}(x[i], y[j])$ by $\text{score}(i,j)$ and worry later about where to obtain reasonable score values.

A useful application of this observation is as follows: Local alignment is often used in large-scale database searches to find remote homologs of a given protein. Transmembrane (TM) proteins have an unusual sequence composition (which has a large influence on the score matrix entries, as pointed out in Section 4.2). According to Müller et al. (2001), one can increase the sensitivity of the homology search by using a bipartite scoring scheme with a (non-symmetric) transmembrane helix specific scoring matrix (such as SLIM) for the TM helices and a general-purpose scoring matrix (such as BLOSUM) for the remaining regions of the query protein; see Figure 4.1. As a consequence, the score of aligning $x[i]$ with $y[j]$ does not only depend on the amino acids $x[i]$ and $y[j]$ themselves, but also on the position i within the query sequence x." [2]

CHAPTER 5

RNA Secondary Structure Prediction

5.1 | Introduction

"RNA is in many ways similar to DNA, with a few important differences:

- Thymine (T) is replaced by uracil (U), so the RNA alphabet is $\Sigma_{\text{RNA}} = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{U}\}$.
- The additional base pair $\{\texttt{G}, \texttt{U}\}$ can be formed.
- RNA is less stable than DNA and more easily degraded.
- RNA mostly occurs as a single-stranded molecule that forms secondary structures by base pairing.
- While DNA is mainly the carrier of genetic information, RNA has a wide variety of tasks in the cell, e.g.
  - it acts as messenger (mRNA), transporting the transcribed genetic information,
  - it has regulatory functions,
  - it has structural tasks, e.g. in forming a part of the ribosome (rRNA),
  - it takes an active role in cellular processes, e.g. in the translation of mRNA to proteins, by transferring amino acids (tRNA).

As with proteins, the three-dimensional structure of an RNA molecule is crucial in determining its function. Since 3D-structure is hard to determine, an intermediate structural level, the *secondary structure*, is frequently considered. The secondary structure of an RNA molecule simply determines the intra-molecular basepairs. As with DNA, the Watson-Crick pairs $\{\texttt{A}, \texttt{U}\}$ and $\{\texttt{C}, \texttt{G}\}$ stabilize the molecule, but also the "wobble pair" $\{\texttt{G}, \texttt{U}\}$ adds structural stability. We now define this formally.

Let $s \in \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{U}\}^n$ be an RNA sequence of length n." [2]

Definition 5.1 s-Basepair a set of two different numbers $\{i, j\} \subset \{1, \ldots, n\}$ such that $\{s[i], s[j]\} \in \{\{\texttt{A}, \texttt{U}\}, \{\texttt{C}, \texttt{G}\}, \{\texttt{G}, \texttt{U}\}\}$. Basepairs must bridge at least $\delta \geq 0$ positions (e.g. $\delta = 3$); to account for this, we have the additional constraint $|i - j| > \delta$ for a basepair $\{i, j\}$..

Definition 5.2 simple structure on s A set $S = \{b_1, b_2, \ldots, b_N\}$ of s-basepairs with $N \geq 0$ and the following properties:

- Either $N \leq 1$, or
- if $N \geq 2$ and $\{i, j\} \in S$ and $\{k, l\} \in S$ are two different basepairs such that $i < j$ and $k < l$, then $i < k < l < j$ or $k < i < j < l$ (nested basepairs) or $i < j < k < l$ or $k < l < i < j$ (separated basepairs).

"In other words, basepairs in a simple structure may not cross.

The restriction that basepairs may not cross is actually unrealistic, but it helps to keep several computational problems regarding RNA analysis of manageable complexity, as we shall see shortly.

With the above definition of a simple structure, we cover the following structural elements of RNA molecules: *single-stranded RNA*, *stem* or *stacking region*, *hairpin loop*, *bulge loop*, *interior loop*, *junction* or *multi-loop*;

Several more complex RNA interactions are *not* covered, e.g. *pseudoknot*, *kissing hairpins*, *hairpin-bulge contact*. This is not such a severe restriction, because these elements seem to be comparatively rare and can be considered separately at a later stage.

We point out that all of the above structural elements can be rigorously defined in terms of sets of basepairs. For our purposes, an intuitive visual understanding of these elements is sufficient. There exist different ways to represent a simple structure in the computer. A popular way is the *dot-bracket* notation, where dots represent unpaired bases and corresponding parentheses represent paired bases:

```
5' AACGCUCCCAUAGAUUACCGUACAAAGUAGGGCGC 3'
    ..(((.((..((...)).(.(...).)..)))))).
```

Another popular way to visualize a structure is by means of a *mountain plot*: Starting with the dot-bracket representation, replace dots by `-` and brackets `()` by `/\`, and use a second dimension. Another alternative is a *circle plot*;" [2]

5.2 | The Optimization Problem

"Over time, an RNA molecule may assume several distinct structures. The more energetically stable a structure is, the more likely the molecule will assume this structure. To make this precise, we need to define an energy model for all secondary structure elements. This can become quite complicated (the problem of predicting RNA structures could fill an entire course), so here we only consider a drastically simplified version of the problem: We assign a score (that can be interpreted as "stability" measure or negative free energy) to each basepair, depending on its type. We can simply count basepairs by assigning $+1$ to each pair, or we may take their relative stability into account by assigning $\text{score}(\{\texttt{A},\texttt{U}\}) := 2$, $\text{score}(\{\texttt{C},\texttt{G}\}) := 3$, $\text{score}(\{\texttt{G},\texttt{U}\}) := 1$, for example. On a sequence s of length n, the score of a basepair $b = \{i, j\}$ is then defined as $\text{score}(b) := \text{score}(\{s[i], s[j]\})$." [2]

Problem 5.1 Simple RNA Secondary Structure Prediction Problem "Let an RNA sequence s of length n be given, determine a simple structure S^* on s such that

$$\text{score}(S^*) = \max\{\text{score}(S) \mid S \text{ is a simple structure on } s\} \tag{5.1}$$

where $\text{score}(S) := \sum_{b \in S} \text{score}(b)$ is the sum of the base pair scores in the structure S." [2]

"Generally, the number of valid simple structures for a molecule of length n is exponential in n, so the most stable structure cannot be found in reasonable time by enumerating all possible structures. After a short digression to context free grammars, we present an efficient algorithm to solve the problem." [2]

5.3 | Context-Free Grammars

"From the above discussion of the dot-bracket notation, we may say that a (simple secondary) structure is

0. either the empty structure,

1. or an unpaired base, followed by a structure,

2. or a structure, followed by an unpaired base,

3. or an outer basepair (with a sequence constraint), enclosing a structure (with a minimum length constraint),
4. or two consecutive structures.

We recall the notion of a *context-free grammar* here to show that simple secondary structures allow a straightforward recursive description." [2]

Definition 5.3 context-free grammar (CFG) A quadruple $G = (N, T, P, S)$, where

- N is a finite set of *non-terminal symbols*,
- T is a finite set of *terminal symbols*, $N \cap T = \emptyset$,
- P is a finite subset of $N \times (N \cup T)^*$, the *productions*, which are rules for transforming a non-terminal symbol into a (possibly empty) sequence of terminal and non-terminal symbols, and
- $S \in N$ is the *start symbol*.

"Ignoring basepair sequence and distance constraints, in the RNA structure setting, the terminal symbols are $T = \{\texttt{(}, \texttt{.}, \texttt{)}\}$ and $N = \{S\}$, where S is also the start symbol and represents a secondary structure. The productions are formalized versions of the above rules, i.e.,

0. $S \to \varepsilon$, where ε denotes the empty sequence,
1. $S \to \texttt{.}S$
2. $S \to S\texttt{.}$
3. $S \to \texttt{(}S\texttt{)}$
4. $S \to SS$

Note that this context-free recursive description is only possible because we imposed the constraint that basepairs do not cross.

Also note that this grammar is ambiguous: There are generally many ways to produce a given dot-bracket string from the start symbol S. For the Nussinov Algorithm in the next section, this ambiguity is not a problem. However, for other tasks, e.g. counting the number of possible structures, we need to consider a grammar that produces each structure in a unique way. We can argue as follows: Either the structure is empty, or it contains at least one base. In each nonempty RNA structure, look at the last base. Either it is unpaired and preceded by a (shorter) prefix structure, or it is paired with some base at a previous position. We obtain the following non-ambiguous grammar:

$$S \to \varepsilon \mid S\texttt{.} \mid S\texttt{(}S\texttt{)}$$

where the vertical bar (|) denotes "or", i.e., it separates alternative right-hand sides of the production." [2]

5.4 | The Nussinov Algorithm

Comment 5.1 Example calculation for GGUCCA on page 94.

"To solve Problem 5.1, we now give a dynamic programming algorithm based on the structural description of simple secondary structures from the previous section. This algorithm is due to Nussinov and Jacobson (1980). We also take into account the distance and sequence constraints for basepairs.

In the previous section, we have seen how to build longer secondary structures from shorter ones that can be arbitrary substrings of the longer ones. As before, let $s \in \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{U}\}^n$ be an RNA sequence of length n. For $i \leq j$, we define $M(i, j)$ as the solution to Problem 5.1 (i.e., maximally attainable score) for the string $s[i \ldots j]$.

Since it is easy to find an optimal secondary structure of short sequences, let us start with those.

- If $j - i \leq \delta$ (in particular if $i = j$, and to avoid complications also if $i > j$), then $M(i, j) = 0$, since no basepair is possible due to the length or distance constraints (case 0).
- If $j - i > \delta$, then we can decompose the optimal structure according to one of the four cases. In each case, the resulting substructures must also be the optimal ones. Thus

$$M(i,j) = \max \begin{cases} M(i+1,j) & \text{(case 1: } S \to \texttt{.}S\text{)} \\ M(i,j-1) & \text{(case 2: } S \to S\texttt{.}\text{)} \\ M(i+1,j-1) + \mu(i,j) & \text{(case 3: } S \to \texttt{(}S\texttt{)}\text{)} \\ \max_{i+1<k<j}\{M(i,k-1) + M(k,j)\} & \text{(all cases 4: } S \to SS\text{)} \end{cases} \tag{5.2}$$

  where $\mu(i, j) := \text{score}(\{s[i], s[j]\}) > 0$ if $\{s[i], s[j]\}$ is a valid basepair, and $\mu(i, j) := -\infty$ otherwise, excluding the possibility of any forbidden basepair $\{i, j\}$.

This recurrence states: To find an optimal structure for $s[i \ldots j]$, consider all possible decompositions according to the grammar and their respective scores, and evaluate which score for $s[i \ldots j]$ would result from each decomposition. Then pick the best one.

This is possible because the composing elements of the optimal structure are themselves optimal. The proof is by contradiction: If they were not optimal and there existed better sub-structures, we could build a better overall structure from them, which would contradict its optimality.

The above recurrence has still to be developed into an algorithm. In order to evaluate $M(i, j)$, we need access to all entries $M(x, y)$ for which $[x, y]$ is a (strictly shorter) subinterval of $[i, j]$. This suggests to compute matrix M in order of increasing $j - i$ values. The initialization for $j - i \leq \delta$ is easily done. Then we proceed for increasing $d = j - i$, as in Algorithm 4." [2]

Algorithm 4 Nussinov Algorithm

Input: RNA sequence $s = s[1 \ldots n]$, scoring function for basepairs, minimum number δ of unpaired bases in a hairpin loop
Output: Maximal score $M(i, j)$ for a simple secondary structure of each substring $s[i \ldots j]$ and traceback indicators $T(i, j)$ indicating which case leads to the optimum

```
for i ← 2 to n do
    M(i, i − 1) ← 0
    T(i, i − 1) ← "init"                                  ▷ ϵ (unpaired)
end for
for d ← 0 to δ do
    for i ← 1 to n − d do
        M(i, i + d) ← 0
        T(i, i + d) ← "unpaired"                          ▷ .S and/or S. (case 1 or 2)
    end for
end for
for d ← δ + 1 to n − 1 do
    for i ← 1 to n − d do
        Compute M(i, i + d) according to the recurrence
        Store the maximizing case in T(i, i + d)          ▷ cases 1–3; or case 4 with maximizing k
    end for
end for
Report score M(1, n)
Start traceback with T(1, n)
```

"To find an optimal secondary structure (in addition to the optimal score), we also store traceback indicators in a second matrix T that shows which of the production rules (and which value of k for the $S \to SS$ rule) was used at each step. By following the traceback pointers backwards, we can build the dot-bracket representation of the secondary structure. For this, we start in cell $(1, n)$ and look at the recursion of the algorithm: When we follow a $S \to SS$ production traceback, the process branches recursively into the two substructures, reconstructing each substructure

independently. Diagonals mean $S \to (S)$ and horizontal or vertical movement means $S \to S. \mid .S$. Note that when reaching the δ-diagonals only horizontal and vertical movement is allowed. Finally, when we end in the lowest diagonal (below the $i = j$ diagonal) we have $S \to \varepsilon$." [2]

Analysis. "The memory requirement for the whole procedure is obviously $O(n^2)$ and its time complexity is $O(n^3)$.

There exists a faster algorithm using the Four-Russians speedup yielding an $O\left(\frac{n^3}{\log n}\right)$ time algorithm (Frid and Gusfield, 2009), but this goes far beyond the scope of these lecture notes." [2]

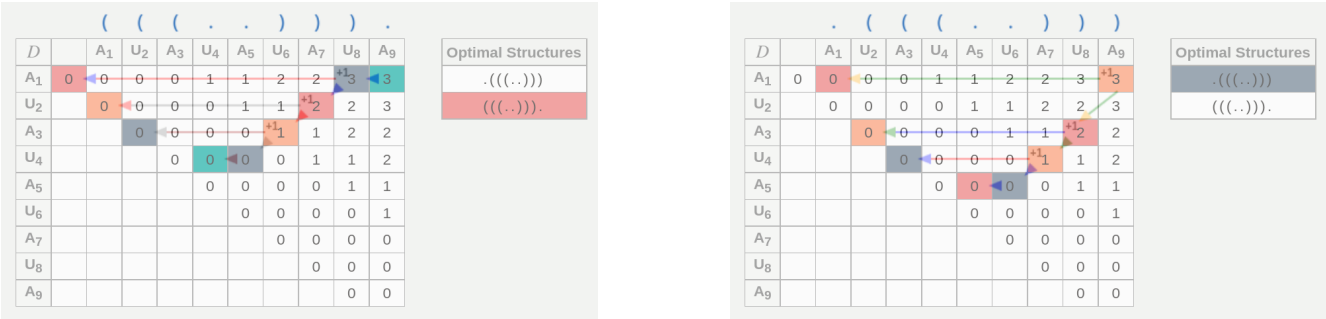
Example 5.1 Aus Skript We use $\delta = 1$ and the scoring scheme: score(G, C) = 3, score(A, U) = 2 and score(G, U) = 1.

There exist two distinct variants (indicated in solid and dashed traces) in which the given RNA sequence can optimally fold under this basepair scoring scheme. The resulting dot-bracket notations are `((.))((.))` and `((((..))))` for the solid and dashed variant, respectively.

"When reaching the δ-diagonals we could move vertically or horizontally which is possible in the ambiguous grammar. However, since in a hairpin loop a sequence of $S \to .S \mid S.$ will eventually end with $S \to \varepsilon$ no matter whether the . is put left or right, this would lead to identical dot-bracket notations. Hence, we could also choose to allow *only* horizontal or *only* vertical movement within the initialized 0-diagonals as to simplify the backtracing of all optimal dot-bracket notations (only 1 backtrace instead of 4 for the dashed variant)." [2]

Example 5.2 AUAUAUAUA Das kann zwei Paarungen hervorrufen, je nach Backtracing.

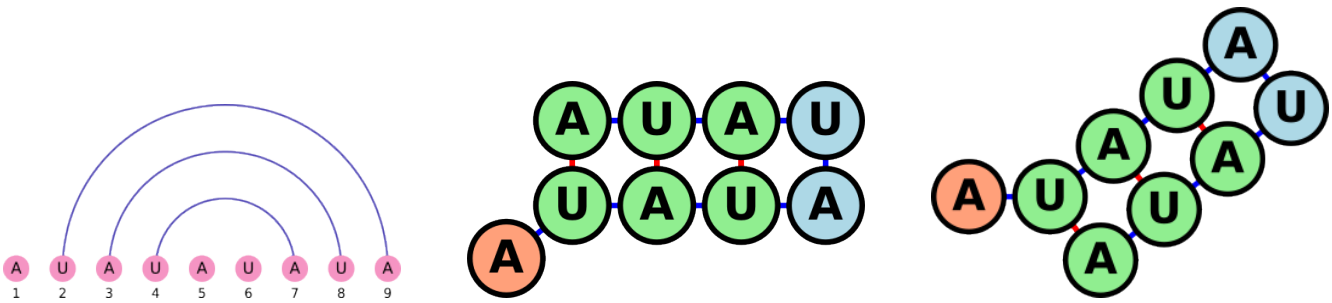
- (((..))).
- .(((..)))



(a) (((..)))-

(b) .(((..)))

Figure 5.1: Berechnung der beiden Möglichkeiten.



(a) .(((..)))

(b) (((..))).

(c) .(((..)))

Figure 5.2: Paarung für Sequenz AUAUAUAUA, verschiedene Darstellungsweisen.

Example 5.3 GGUCCA Hier gibt es nur eine Möglichkeit: ((.)).

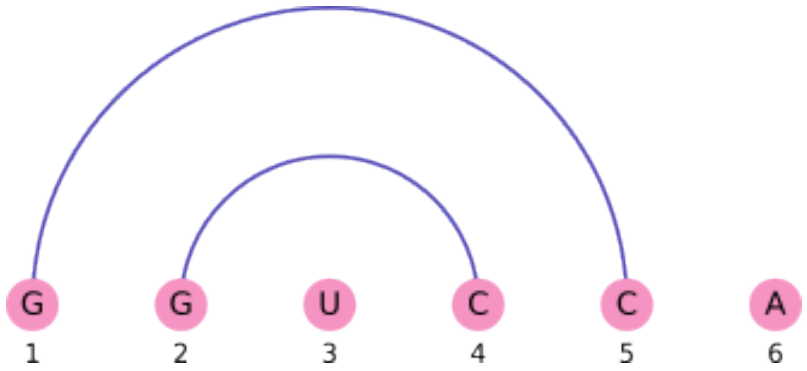


Figure 5.3: Tracing

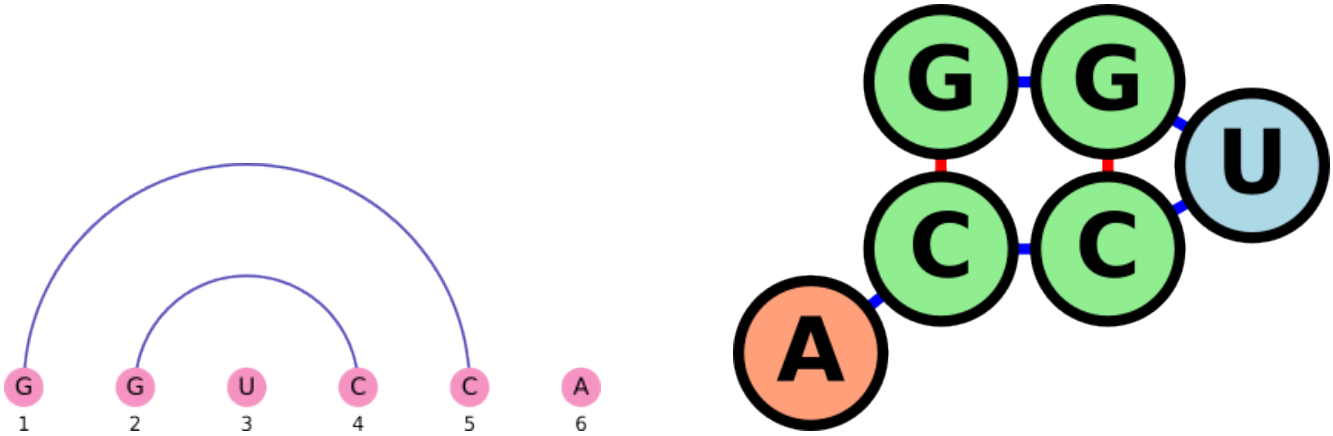


Figure 5.4: Paarung für Sequenz GGUCCA

CHAPTER 6

Pairwise Sequence Alignment in Linear Space

6.1 | The Forward-Backward Technique

"The following problem frequently arises as a sub-problem of more advanced alignment methods, that is why we discuss it beforehand in detail:" [2]

Problem 6.1 Advanced Alignment Problem "Given two sequences x and y of lengths m and n, respectively, find their optimal global alignment under the condition that the alignment path passes through a given cell (i_0, j_0) of the alignment matrix." [2]

"This is not a new problem at all! In fact, we can decompose it into two standard global alignment problems: Aligning the prefixes $x[1 \dots i_0]$ and $y[1 \dots j_0]$, and aligning the suffixes $x[i_0 + 1 \dots m]$ and $y[j_0 + 1 \dots n]$. By combining the results of these two sub-problems in an appropriate way, we obtain the minimally attainable total cost of all alignment paths that go through (i_0, j_0).

Total cost matrix. In fact, it is even possible to solve this problem for every point (i, j) simultaneously. The solution is called the *forward-backward technique*. In addition to the usual alignment matrix with alignment costs $D(i, j)$ (which record the minimally attainable cost for a prefix alignment), we additionally define the *reverse alignment matrix*: We exchange initial and final cell and reverse the direction of all computations. Effectively, we are thus globally aligning the reversed sequences. Hence, the associated alignment matrix $D^{\text{rev}} = D^{\text{rev}}(i, j)$ records the best possible costs for the suffix alignments." [2]

Definition 6.1 total cost matrix for two sequences x and y of lengths m and n "++" denotes concatenation of alignments in the total cost matrix defined as:

$$T(i,j) = \min \left\{ \text{cost}(A + +B) \middle| \begin{array}{l} A \text{ is an alignment of the prefixes } x[1 \dots i] \text{ and } y[1 \dots j] \\ B \text{ is an alignment of the suffixes } x[i+1 \dots m] \text{ and } y[j+1 \dots n] \end{array} \right\} \tag{6.1}$$

Note 6.1 "Note that $T(0, 0) = T(m, n)$ is the optimal global alignment cost $d(x, y)$ (since all paths, in particular the optimal ones, pass through $(0, 0)$ and (m, n)), and so $T(i, j) \geq d(x, y)$ for all (i, j)." [2]

" The computation of the total cost matrix is particularly simple for additive alignment costs. Here

$$T(i,j) = D(i,j) + D^{\text{rev}}(i,j).$$

Clearly matrices D, D^{rev}, and thus T can be computed in $O(mn)$ time.

For affine gap costs the computation of the total cost matrix is slightly more difficult: Here, the cost of a concatenated alignment is not always the sum of the costs of the individual alignments because gaps might be merged, so that the gap initiation penalty that is imposed twice in the two separate alignments must be counted only once in the concatenated alignment. However, since the efficient computation of affine gap costs with Gotoh's algorithm uses the two history matrices V and H, and we know that by definition the costs stored in these matrices refer to those alignments that end with gaps, the total cost matrix for affine gap costs can also be computed in quadratic time if the history matrices V and H and their reverse counterparts V^{rev} and H^{rev} are given:

$$T(i,j) = \min \begin{cases} D(i,j) + D^{\text{rev}}(i,j) \\ V(i,j) + V^{\text{rev}}(i,j) - d + e \\ H(i,j) + H^{\text{rev}}(i,j) - d + e \end{cases}$$

where d is the gap open cost and e is the gap extension cost.

Additional cost matrix. For every (i,j), we can now obtain the *additional cost* $C(i,j)$ for passing through (i,j), compared to staying on an optimal alignment path." [2]

Definition 6.2 additional cost matrix for two sequences x and y of lengths m and n $d(x,y)$ is the edit distance of sequences x and y for the additional cost matrix defined as:

$$C(i,j) = T(i,j) - d(x,y) \tag{6.2}$$

"By definition, if (i,j) is on the path of an optimal alignment, this value is zero, otherwise it is positive.

Again, for any additive alignment cost, where $\text{cost}(A++B) = \text{cost}(A) + \text{cost}(B)$, we have that $C(i,j) = T(i,j) - d(x,y) = D(i,j) + D^{\text{rev}}(i,j) - d(x,y)$. Since T and $d(x,y)$ can be computed in $O(mn)$ time, the additional cost matrix C can be computed in $O(mn)$ time, too. " [2]

Example 6.1 Aus Skript

Consider the sequences $x = \texttt{CT}$ and $y = \texttt{AGT}$. For unit cost, the edit matrix D and reverse edit matrix D^{rev}, which is drawn in reverse direction, are:

D :

	ϵ	A	G	T
ϵ	0	1	2	3
C	1	1	2	3
T	2	2	2	2

D^{rev} :

	A	G	T	ϵ
C	2	1	1	2
T	2	1	0	1
ϵ	3	2	1	0

This results in the following additional cost matrix:

C :

	ϵ	A	G	T
ϵ	0	0	1	3
C	1	0	0	2
T	3	2	1	0

We see that the paths of the two optimal alignments $A_1^{\text{opt}} = \left(\begin{smallmatrix} \texttt{C} & - & \texttt{T} \\ \texttt{A} & \texttt{G} & \texttt{T} \end{smallmatrix}\right)$ and $A_2^{\text{opt}} = \left(\begin{smallmatrix} - & \texttt{C} & \texttt{T} \\ \texttt{A} & \texttt{G} & \texttt{T} \end{smallmatrix}\right)$ correspond to the two paths of 0-entries in C.

"

Applications. The forward-backward technique is used, for example, in the following problems:

- linear-space alignment (Hirschberg technique, see Section 6.2),

- discovering regions of slightly sub-optimal alignments (those cells where the score gap, resp. additional cost, remains below a small tolerance parameter),

- computing regions for the Carrillo-Lipman heuristic for multiple sequence alignment (see Section 8.2),

- finding cut-points for divide-and-conquer multiple alignment (see Section 10.1).

" [2]

6.2 | Hirschberg's Algorithm

Comment 6.1 Detailliertes Beispiel auf Seite 101.

" We have seen that the *optimal alignment cost* of two sequences x and y of lengths m and n, respectively, as well as an *optimal alignment* can be computed in $O(mn)$ time and $O(mn)$ space.

Even though the $O(mn)$ time required to compute an optimal global or local alignment is sometimes frowned upon, it is not the main bottleneck in sequence analysis. The main bottleneck so far is the $O(mn)$ space requirement. Think of a matrix of traceback-pointers for two bacterial genomes of 5 million nucleotides each. That matrix contains $25 \cdot 10^{12}$ entries and would thus need 25 Terabytes of memory. Fortunately, there is a solution!

We have already seen that the score by itself can easily be computed in $O(m + n)$ space, since we only need to store the sequences and two rows or two columns of the edit matrix at a time. However, if we also want to output an optimal alignment, the whole alignment matrix (or the backtracing matrix) is required for the backtracing phase.

In this section we present an algorithm that allows to compute an optimal global alignment in *linear* space. It was first presented by Hirschberg (1975). Our presentation is for the simple case of homogeneous linear gap costs $g(\ell) = \ell \cdot \gamma$, where γ is the cost for a single gap character. However, it can be generalized to affine gap costs with some additional complications.

The algorithm works in a *divide-and-conquer* manner, where in each recursion the alignment matrix $D = (D(i, j))_{1 \leq i \leq m, 1 \leq j \leq n}$ is divided into an upper half and a lower half at its middle row $m' = \lceil m/2 \rceil$. Hereby, the cut positions are chosen so that an optimal alignment of the two obtained prefixes, concatenated with an optimal alignment of the two obtained suffixes, yields an optimal alignment of the original sequences.

To find the cut positions for the division of the sequences, the "forward-backward" technique is used. The upper half is computed in a forward manner, as usual, and the lower one in a backward manner. At index m' both halves intersect, which makes it possible to compute the additional costs $C(m', j)$ for this row m'. An optimal alignment passes row m' at some column n' if and only if $C(m', n') = 0$.

The procedure is called recursively, once for the upper left "quarter", and once for the lower right "quarter" of the alignment matrix. The recursion is continued until only one row remains, in which case the problem can easily be solved directly." [2] See Figure 6.1 for an illustration.

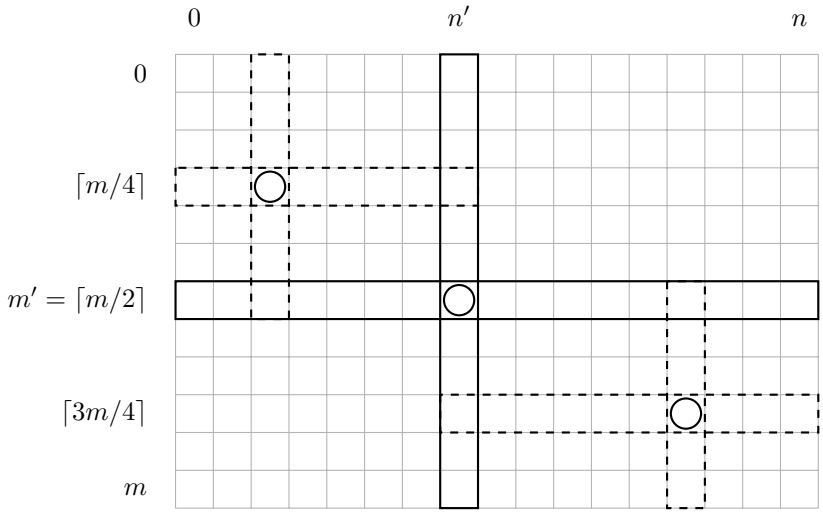


Figure 6.1: Illustration of linear-space alignment.

Example 6.2

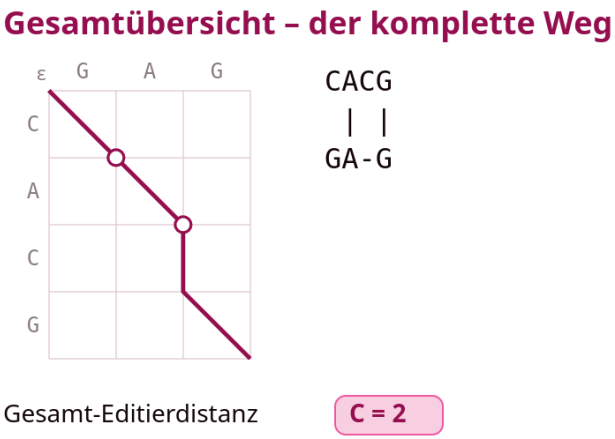


Figure 6.2: Tracing

Complexity analysis. "The space complexity is in $O(m+n)$ since all matrix computations can be performed row-wise using memory for two adjacent rows at most, and the final alignment has at most $m+n$ columns.

A central question is, how much do we have to "pay" in terms of running time to obtain the linear space complexity? Fortunately, the answer is: Only approximately a constant factor of 2. In the original (quadratic-space) algorithm, each cell of the edit matrix S is computed once. In the linear-space version, some cells are computed several times during the recursions.

The amount of cells computed in the first pass is mn, in the second pass it is only half of the matrix, in the third pass it is a quarter of the matrix and so on, for a total of $mn \cdot (1 + 1/2 + 1/4 + \ldots) \leq 2mn$ cells (geometric series). Thus, the asymptotic time complexity is still $O(mn)$." [2]

Example 6.3 Aus Skript Given strings $x =$ `CACG` and $y =$ `GAG`, Figure 6.3 shows how alignment in linear space works. Note that just the black numbers need to be calculated. The grey ones are only for comparison to Section 6.1 to provide an easier comprehension.

The recursion ends in four base cases resulting in the overall alignment $\left(\begin{smallmatrix}\texttt{CACG}\\ \texttt{GA-G}\end{smallmatrix}\right)$. Note that if a cut position n' is found (indicated by a circle in additional cost matrix C), the sequences are cut in two after this position, such that the left side of t' contains the circled character and the right side does not.

Affine and general gap costs. "Combining the Hirschberg technique with affine gap costs is conceptually straightforward, while the devil is in the details. This was first presented by Myers and Miller (1988).

General gap costs cannot be handled in linear space. We cannot even compute just the alignment score in linear space since we always need to refer back to all cells to the left and above of the current one." [2]

Local alignment. "We have described how to compute an optimal global alignment in linear space. What about local alignments? Clearly, we have to use a scoring scheme instead of a cost function, but that modification is an easy exercise. For the main adaptation, it is important to note that a local alignment is in fact a global alignment of the two best-aligning substrings of x and y. So, once we know the start- and endpoints of these substrings, we can use the above global alignment procedure.

It is easy to find the endpoints: They are given by the entry with the highest score in the (forward) local alignment matrix.

To find the start points, we can use the reversal technique: They are just the endpoints of the optimal local alignment of the reversed strings. This technique is simple, but may become problematic if there are several equally high-scoring local alignments, when the start- and end-points have to be matched correctly. Details are left to the reader." [2]

Suboptimal alignments. "In practice, we are often interested in several, say K good alignments instead of a single optimal one. Can this also be done in linear space? Essentially, we need to remember the K highest entries in the edit matrix, and once we have found an optimal alignment we store a list or hash table to remember its path and subsequently take care that the diagonal edges of this path cannot be used anymore when we re-compute the next (now optimal) alignment. This requires additional space proportional to the total lengths of all discovered alignments, which is linear for a constant number of alignments.

A more satisfying way to define and compute suboptimal local alignments is presented in the next chapter." [2]

CHAPTER 7

Suboptimal Local Alignments

"Often it is desirable not only to find one or all *optimal* alignments of two sequences, but also to find *suboptimal alignments*, i.e., alignments that score (slightly) below the optimal alignment score. This is especially relevant in a local alignment context where several similar regions might exist and be of biological interest, while the Smith-Waterman algorithm reports only a single (optimal) one.

A simple way to find suboptimal local alignments could be to report first the alignment(s) (obtained by backtracing) corresponding to the highest score in the local alignment graph, then the alignment(s) corresponding to the second highest score, and so on. This approach has two disadvantages, though:

1. Redundancy: Many of the local alignments reported by this procedure will be largely overlapping, differing only by the precise placement of single gaps, single mismatches or the exact end characters of the alignments. An illustration is given in Figure 7.1.

2. Shadowing effect: Some high scoring alignments will be invisible in this procedure if an even higher scoring alignment is very close by, for example if the two alignments cross each other in the edit graph, where always the higher scoring prefix will be chosen. An illustration is given in Figure 7.2.

" [2]

"In order to circumvent these disadvantages, Waterman and Eggert (1987) gave a more sensible definition of suboptimal local alignments." [2]

Definition 7.1 Non-overlapping suboptimal local alignment An alignment that has no match or mismatch in common with any higher-scoring (sub)optimal local alignment. (Two alignments A and A' have a match or mismatch in common (they overlap) if there is at least one pair of positions (i, j) such that $x[i]$ is aligned with $y[j]$ both in A and in A'.).

"Algorithmically, non-overlapping alignments can be computed as follows: First the Smith-Waterman algorithm is run as usual, returning an optimal local alignment. Then, the matrix S is computed again, but any match or mismatch from the reported optimal alignment is forbidden, i.e., in the alignment matrix the corresponding diagonal steps are ignored in the maximization. In terms of computing the alignment matrix, in the maximizations for these cells only two cases, namely insertion and deletion, are allowed.

This could be done by re-computing the whole matrix, but it is easy to see that it suffices to compute only that part of the matrix from the starting point of the first alignment to the bottom-right of the matrix. Moreover, whenever in a particular row or column the entries in the re-computation do not change compared to the entries in the original matrix, the computation along this row or column can be stopped. Waterman and Eggert suggest to alternate the computation of one row and one column at a time, each one until the new value coincides with the old value.

This procedure can be repeated for the third-best non-overlapping local alignment, and so on.

If we assume the usual type of similarity function with expected negative score of a random symbol pair so that large parts of the alignment matrix are filled with zeros, then the part of the matrix that needs to be re-computed is not much larger than the alignment itself.

Then, to compute the K best non-overlapping suboptimal local alignments, the algorithm takes an expected time of $O(mn + \sum_{k=1}^{K} L_k^2)$, where L_k is the length of the k-th reported alignment. In the worst case this is $O(mnK)$, but typically much less in practice.

Huang and Miller (1991) showed that the Waterman-Eggert algorithm can also be implemented for affine gap costs and in linear space." [2]

7.1 | Smith-Waterman-Algorithm

Example calculation on page 105.

Algorithm 5 Smith-Waterman Darstellung nach Skript

Sequences to be aligned: $A = a_1 a_2 \ldots a_n$ and $B = b_1 b_2 \cdots_m$
score(x,y): Similarity score of x and y
Gap cost: γ
$S(0,0) \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  $S(i,0) \leftarrow 0$
end for
for $j \leftarrow 1$ to m **do**
  $S(0,j) \leftarrow 0$
end for
for $j \leftarrow 1$ to n **do**
  for $i \leftarrow 1$ to n **do**

$$S(i,j) \leftarrow max \begin{cases} S(0,0) & = 0(\text{leeres Suffix}) \\ S(i-1,j-1) + score(x[i],y[j]) & = \text{links oben + sim score((Mis-)Match)} \\ S(i-1,j) - \gamma & = \text{oben(Deletion)} \\ S(i,j-1) - \gamma & = \text{links(Insertion)} \end{cases}$$

  end for
end for
Traceback

Algorithm 6 Smith-Waterman Darstellung angepasst

Sequences to be aligned: $A = a_1 a_2 \ldots a_n$ and $B = b_1 b_2 \cdots_m$
s(x,y): Similarity score of x and y
$S(0,0) \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  $S(i,0) \leftarrow 0$
end for
for $j \leftarrow 1$ to m **do**
  $S(0,j) \leftarrow 0$
end for
for $j \leftarrow 1$ to n **do**
  for $i \leftarrow 1$ to n **do**

$$S(i,j) \leftarrow max \begin{cases} 0 \\ S(i-1,j-1) + s(x[i],y[j]) \\ S(i-1,j) - s(x[i],--) \\ S(i,j-1) - s(--,y[j]) \end{cases}$$

  end for
end for
Traceback

7.2 | Waterman-Eggert-Algorithm

Example calculation on page 108.

Algorithm 7 ONE(!) suboptimal Alignment through Waterman-Eggert

Input S**:** matrix calculated by Smith-Waterman
Take the calculated matrix from Smith-Waterman and set all cells that contribute to the optimal alignmnent to 0.
$i_{start} \leftarrow$ start row
for $i \leftarrow i_{start}$ to n **do**
  for $j \leftarrow 1$ to m **do**
    if (i, j) belongs to optimal aligmnent **then**
      $S(i,j) = 0$
    else

$$S(i,j) \leftarrow max \begin{cases} 0 \\ S(i-1,j-1) + s(x[i],y[j]) \\ S(i-1,j) - s(x[i],--) \\ S(i,j-1) - s(--,y[j]) \end{cases}$$

    end if
  end for
end for

To calculate the k best non-overlapping alignments, repeat the above k times while inputing the result matrix of the previous run.

Wichtige Formel dabei:

$$S(i,j) = max \begin{cases} 0 \\ S(i-1,j-1) + s(x[i],y[j]) \\ S(i-1,j) - s(x[i],--) \\ S(i,j-1) - s(--,y[j]) \end{cases} \tag{7.1}$$

CHAPTER 8

Exact Algorithms for Sum-of-Pairs Multiple Sequence Alignment

"Although sum-of-pairs (SP) multiple sequence alignment is NP-complete, it is sometimes desirable to compute optimal solutions for a moderate number of sequences; either as a sub-procedure of other methods, or for comparison and benchmarking of heuristic alignment algorithms.

Therefore here we will come back to the original model of sum-of-pairs multiple alignment and look at it a bit more closely. We first formally describe the multi-dimensional generalization of the pairwise dynamic programming algorithm. Then we show how to reduce the search space considerably according to an idea of Carrillo and Lipman.

In this section we give the formulation for distance minimization. Naturally, equivalent algorithms exist for score maximization." [2]

8.1 | The Exact Algorithm Revisited

"The problem to be solved is the following. Given k sequences $s_1, s_2, \ldots, s_k$ of lengths $n_1, n_2, \ldots, n_k$, respectively, the goal is to find a multiple alignment A such that

$$\mathrm{cost}_{SP}(A) := \sum_{1 \le p \le q \le k} \mathrm{cost}_2(\pi_{p,q}(A))$$

is minimum among all multiple alignments of $s_1, s_2, \ldots, s_k$." [2]

8.1.1 | The Basic Algorithm

"We have already seen that the SP multiple alignment problem can be solved by applying the Universal Alignment Algorithm from Chapter 5 of the "Sequence Analysis 1" lecture to a k-dimensional alignment matrix, one dimension for each of the input sequences:

For each cell v of the alignment matrix, compute in an appropriate order the dissimilarity value

$$D(v) = \min_{u \in \mathrm{pred}(v)} \big\{ D(u) + \mathrm{cost}(u \to v) \big\}. \tag{8.1}$$

More explicitly, this can be written as follows:

$$D(0, 0, \ldots, 0) = 0$$

$$D(\underbrace{i_1, i_2, \ldots, i_k}_{v}) = \min_{\substack{\Delta_1,\ldots,\Delta_k \in \{0,1\} \\ \Delta_1+\ldots+\Delta_k \neq 0}} \left\{ D(\underbrace{i_1 - \Delta_1, i_2 - \Delta_2, \ldots, i_k - \Delta_k}_{\text{predecessor } u}) + D_{SP} \underbrace{\begin{pmatrix} \Delta_1 s_1[i_1] \\ \Delta_2 s_2[i_2] \\ \vdots \\ \Delta_k s_k[i_k] \end{pmatrix}}_{\text{alignm. col.}} \right\}$$

where for a character $c \in \Sigma$ the notation Δc denotes $\Delta c = c$ if $\Delta = 1$ and $\Delta c = -$ if $\Delta = 0$." [2]

Space and time complexity. "The space complexity $O(n^k)$ is clearly given by the size of the multidimensional alignment matrix, where $n \geq n_1, n_2, \ldots, n_k$.

The time complexity $O(n^k \cdot 2^k \cdot k^2)$ is also easy to see in this notation: For each of the $O(n^k)$ cells, all $2^k - 1$ predecessors have to be evaluated, given by the 2^k possible settings of the vector $(\Delta_1, \ldots, \Delta_k)$ of binary variables Δ_i, without the all-zero vector where $\Delta_1 + \ldots + \Delta_k = 0$. In addition, the sum-of-pairs cost of the resulting alignment column has to be computed, which requires additional $O(k^2)$ time per predecessor." [2]

8.1.2 | Variations of the Basic Algorithm

Memory reduction. "From the pairwise alignment, we remember that it is possible to reduce the space for optimal alignment computation from quadratic to linear using Hirschberg's divide-and-conquer approach (Section 6.2).

In principle, the same idea can also be followed for multiple alignment, but the effect is much less dramatic. The space reduction is by one dimension, from $O(n^k)$ to $O(n^{k-1})$. While for $k = 3$ this has quite some effect, the space savings become less and less impressive as k grows. For example, for $k = 12$ sequences the space consumption using this technique is still $O(n^{11})$. Figure 8.1 sketches on the left side the first division step for the pairwise case, and on the right side the corresponding step in the multiple (here 3-dimensional) case." [2]

Free end gaps. "In the same spirit as in semi-global pairwise alignment, it is sometimes desired to penalize gaps at the beginning and at the end of a multiple alignment less than internal gaps (or not at all, known as *free end-gap* alignment), in order to avoid that shorter sequences are "spread" over longer ones, just to get a slightly higher score that is not counter-penalized by gap costs, because these have to be imposed anyway to account for the difference in sequence length.

This can be done in multiple alignment as in the pairwise case. It is just a different initialization of the base cases, i.e. the outer edges of the k-dimensional alignment matrix." [2]

Affine gap costs. "Affine gap costs can be defined in a straightforward manner as a simple generalization of the pairwise case, called *natural gap costs*: The cost of a multiple alignment is the sum of all costs of the pairwise projections, where the cost of a pairwise projection is computed with affine gap costs. In principle it is possible to perform the computation of such alignments similarly as it is done for pairwise alignments. However, the number of "history matrices" (V and H in the pairwise case) grows exponentially with the number of sequences, so that very much space is needed.

That is why Altschul (1989) suggested to use an approximation of the exact case, so-called *quasi-natural gap costs*. The idea is to look back by only one position in the edit graph and decide from the previous column in the alignment if a new gap is started in the column to be computed or not. The choices, compared to the "correct" choices in natural gap costs, are given in the following table whose first line depicts the different possible scenarios of pairwise alignment projections where an asterisk (`*`) refers to a character, a dash (`-`) refers to a blank, and a question mark (`?`) refers to either a character or a blank. The entries correspond to the cost (d is gap open cost, e is gap extension cost), where "no" means no new gap is opened (substitution cost):

	$\begin{smallmatrix}?*\\?*\end{smallmatrix}$	$\begin{smallmatrix}?-\\?-\end{smallmatrix}$	$\begin{smallmatrix}***-\end{smallmatrix}$	$\begin{smallmatrix}**\\--\end{smallmatrix}$	$\begin{smallmatrix}-**-\end{smallmatrix}$	$\begin{smallmatrix}-*\\--\end{smallmatrix}$
natural	no	0	d	e	d	d/e
quasi-natural	no	0	d	e	d	d

The last case in natural gap costs cannot be decided if only one previous column is given. It would be d for $\begin{smallmatrix}*----**----\end{smallmatrix}$ and e for $\begin{smallmatrix}***--**---\end{smallmatrix}$.

With this heuristic, the overall computation becomes slower by a factor of 2^k, yielding a time complexity for the complete multiple alignment computation of $O(n^k 2^{2k} k^2)$.

The space requirements do not change in the worst case, they remain $O(n^k)$." [2]

8.2 | Carrillo and Lipman's Search Space Reduction

"Carrillo and Lipman (1988) suggested an effective way to prune the search space for multiple alignments. Their algorithm is still exponential in the worst case, but in many cases the time of alignment computation is considerably reduced compared to the full dynamic programming algorithm from Section 8.1, making it applicable in practice for data sets of moderate size (7–10 protein sequences of length 300–400).

Let sequences $s_1, s_2, \ldots, s_k$ be given. The algorithm of Carrillo and Lipman is a *branch-and-bound* running time heuristic, based on a simple observation." [2]

Idea 8.1 "We have seen that the pairwise projections of an optimal multiple alignment are not necessarily the optimal pairwise alignments (making the problem hard). However, intuitively, they can also not be particularly bad alignments: If all pairwise projections were bad, the whole multiple alignment would also have a bad sum-of-pairs score.

If we can quantify this idea, we can restrict the search space to those vertices that are part of close-to-optimal pairwise alignments for each pairwise projection." [2]

Bounding pairwise alignment costs. "We assume that we already know some (suboptimal) multiple alignment A^{heur} of the given sequences with cost $\text{cost}_{SP}(A^{\text{heur}}) \geq \text{cost}_{SP}(A^{\text{opt}})$, for example computed by a progressive alignment method or by the center-star heuristic (Section 9.3).

Remember the definition of the sum-of-pairs multiple alignment cost:

$$\text{cost}_{SP}(A) = \sum_{p<q} \text{cost}_2(\pi_{\{p,q\}}(A)).$$

We now pick a particular index pair (x, y), $x < y$, and remove it from the sum. In the remaining pairs, we use the fact that the pairwise projections of the optimal multiple alignment cannot have a lower cost than the optimal corresponding pairwise alignments: $\text{cost}_2(\pi_{\{p,q\}}(A^{\text{opt}})) \geq d(s_p, s_q)$. Thus

$$\begin{aligned}
\text{cost}_{SP}(A^{\text{heur}}) &\geq \text{cost}_{SP}(A^{\text{opt}}) \\
&= \sum_{p<q} \text{cost}_2(\pi_{\{p,q\}}(A^{\text{opt}})) \\
&= \text{cost}_2(\pi_{\{x,y\}}(A^{\text{opt}})) + \sum_{\substack{p<q \\ (p,q)\neq(x,y)}} \text{cost}_2(\pi_{\{p,q\}}(A^{\text{opt}})) \\
&\geq \text{cost}_2(\pi_{\{x,y\}}(A^{\text{opt}})) + \sum_{\substack{p<q \\ (p,q)\neq(x,y)}} d(s_p, s_q)
\end{aligned}$$

for any pair (x, y), $1 \leq x < y \leq k$.

This implies the following upper bound for the cost of the projection of an optimal multiple alignment on rows x

and y:

$$\text{cost}_2(\pi_{\{x,y\}}(A^{\text{opt}})) \leq \text{cost}_{SP}(A^{\text{heur}}) - \sum_{\substack{p<q \\ (p,q)\neq(x,y)}} d(s_p, s_q) =: U_{(x,y)}.$$

" [2]

Note 8.1 "Note that in order to compute the upper bound $U_{(x,y)}$, only pairwise optimal alignments and a heuristic multiple alignment need to be calculated. This can be done efficiently." [2]

Pruning the alignment matrix. "We now keep only those cells $(i_1, \ldots, i_k)$ of the alignment matrix that satisfy the following condition for all pairs (x, y), $1 \leq x < y \leq k$:

The best pairwise alignment of s_x and s_y through cell (i_x, i_y) has cost $\leq U_{(x,y)}$.

How do we know the cost of the best pairwise alignment through any given cell (i_x, i_y)? This question was answered in Section 6.1: We use the forward-backward technique. In other words, for every pair (x, y), $x < y$, we compute the total cost matrix $T_{(x,y)}$ such that $T_{(x,y)}(i_x, i_y)$ contains the cost of the best alignment of s_x and s_y that passes through the cell (i_x, i_y).

We can define the set of cells in the multi-dimensional alignment matrix that satisfy the constraint imposed by the pair (x, y) as follows:

$$B_{(x,y)} := \{(i_1, \ldots, i_k) : T_{(x,y)}(i_x, i_y) \leq U_{(x,y)}\}$$

From the above considerations it is clear that the projection of an SP-optimal multiple alignment on sequences s_x and s_y can not have a cost higher than $U_{(x,y)}$, implying that regions (i, j) with values $T_{(x,y)}(i, j) > U_{(x,y)}$ cannot contain such a projection.

Thus an optimal multiple alignment passes only through cells that are in $B_{(x,y)}$ for all pairs (x, y), i.e., in

$$B := \bigcap_{x<y} B_{(x,y)}.$$

This means: Instead of computing the distances in all $O(n^k)$ cells in the alignment matrix, we need to compute the distances of only $|B|$ cells. The whole procedure is conceptually illustrated in Figure 8.2.

The size of B depends on many things, e.g. on

- the quality of the heuristic alignment with cost $\text{cost}_{SP}(A^{\text{heur}})$ that serves as an upper bound of the optimal alignment cost. The closer $\text{cost}_{SP}(A^{\text{heur}})$ is to the true optimal value, the smaller is B;
- the difference between the optimal pairwise alignment costs and the costs of the pairwise projections of the optimal multiple alignment.

" [2]

"Generally, if all k sequences are similar and the gap costs are small, then a good heuristic alignment (maybe even an optimal one) is easy to find, so B may contain almost no additional vertices besides the true optimal multiple alignment path. Thus, for a medium number of similar sequences, the simple Carrillo-Lipman heuristic may already perform very well (because the relevant regions become very small) while for even fewer but less closely related sequences even the best known heuristics will not finish within weeks of computation. Sequence similarity plays a much higher role than sequence length or the number of sequences.

Some implementations of the Carrillo-Lipman bound attempt to reduce the size of B further by "cheating": Remember that we need a heuristic alignment to obtain an upper bound $\text{cost}_{SP}(A^{\text{heur}}) =: \delta$ on the optimal cost. If we have reason to suspect that we have found a bad alignment, and believe that a better one exists, we may replace δ by a smaller value, which could even be chosen differently for each pair (x, y): $\delta - \epsilon_{(x,y)}$. Of course, since we cannot be sure that such a better alignment exists, we may reduce the size of B too much and in fact exclude the optimal multiple alignment from B. In practice, however, this kind of cheating works quite well if done carefully, and further reduces the running time." [2]

Algorithm. "We give a few remarks about implementing the above idea. We emphasize that the set of relevant matrix cells B is conceptual and does not need to be constructed explicitly. Instead, it is sufficient to precompute all $T_{(x,y)}$ matrices, the optimal pairwise alignment scores $d(s_p, s_q)$ for all (p, q), $1 \leq p < q \leq k$, and a heuristic alignment A^{heur} to obtain δ.

It is advantageous not to implement the "pull-type" recurrence (8.1), but instead to use a "push-type" algorithm as explained in the following.

Imagine that every cell has a value of ∞ at the beginning of the algorithm, although this information is not stored explicitly. Instead, a list (e.g. a queue or priority queue) of "active" cells is maintained, that initially contains only $(0, 0, \ldots, 0)$.

The algorithm then consists of repeatedly "visiting" a cell from the queue until the final cell $(n_1, n_2, \ldots, n_k)$ is reached. When a cell is visited, it already contains the correct alignment distance.

Visiting a cell v consists of the following steps.

1. We look at the $2^k - 1$ successors w of v in the alignment matrix and check for each of them if it belongs to the set B. This is the case if w is already in the queue, or if the current distance value plus edge cost $v \to w$ satisfies the Carrillo-Lipman bound for every index pair (x, y). In the former case, the distance (and backpointer) information of w is updated if the new distance value coming from v is lower than the current one. In the latter case w is "activated" with the appropriate distance and backpointer information and added to the queue.

2. We remove v from the queue. If we want to create an alignment in the end, we need to store v's information somewhere else to reconstruct the traceback information. If we only want the distance value, v can now be completely discarded.

3. We pick the next cell x to visit. The algorithm works correctly if x is

   - a vertex of minimum distance; then the algorithm is essentially Dijkstra's algorithm for single-source shortest paths in the (reduced) alignment graph, or
   - any vertex that has no active predecessors, so we can be sure that we do not need to update x's distance information after visiting x.

   Since often more than one vertex is available for visiting, a good target-driven choice may speed up the algorithm.

" [2]

Implementations. "Implementations of the Carrillo-Lipman heuristic are the programs MSA (Gupta et al., 1995) and QALIGN (Sammeth et al., 2003b)." [2]

CHAPTER 9

An Approximation Algorithm for Sum-of-Pairs Multiple Sequence Alignment

9.1 | Digression: NP-completeness

"It was stated before that sum-of-pairs multiple alignment is NP-complete with respect to the number of sequences k, without further explaining what that means. This section shall now give a brief introduction to the field of NP-completeness, about which every good bioinformatician should know. The contents in this section are based on the book of Cormen et al. (2001)." [2]

Definition 9.1 Problem Q Binary relation on a set I of problem instances and a set S of problem solutions.

"A problem instance can point to no, one, or several problem solutions, and a problem solution can be one of no, one, or several problem instances." [2]

Example 9.1 Aus Skript "A simple example of a problem is integer addition. Then the problem instances could be "$5+3$", "$4+4$" or "$1+4$" pointing to the problem solutions "8", "8" and "5", respectively." [2]

Problem types

Definition 9.2 Optimization problem Addresses the minimization of a cost or maximization of a score function.

Definition 9.3 Search problem Addresses the minimization of a cost or maximization of a score function.

Example 9.2 "For example, finding the minimum-cost alignment of two or more sequences is an optimization problem." [2]

Definition 9.4 Decision problem A "yes/no question" with just two problem solutions, $S = \{\mathtt{T}, \mathtt{F}\}$ (*true*/*false*)..

Example 9.3 "The question whether for two given sequences there exists an alignment with a cost lower than a certain threshold t is a decision problem." [2]

"The discussion about NP-completeness mainly considers decision problems, but the results influence also optimization problems: Searching for a bound, every optimization problem can be stated as a series of decision problems. If a decision problem is difficult, the corresponding optimization problem is usually difficult as well." [2]

Complexity classes. "The set of all decision problems for which algorithms exist that have the same asymptotic behavior (e.g. running time) are contained in one *complexity class*. The class **P** contains all problems for which a deterministic algorithm exists that can *solve* in polynomial time a certain problem instance, i.e., it can find, for a given problem instance, in polynomial time whether there exists a configuration of input variables such that the problem has solution `T` or not. The class **NP** contains all problems for which a deterministic algorithm exists that can *verify* in polynomial time whether a given certificate (configuration of input variables) is able to solve an actual problem instance, i.e., it can tell whether the certificate indeed yields solution `T` or not. In short **P** = efficiently solvable and **NP** = efficiently verifiable. It follows directly that $\mathrm{P} \subseteq \mathrm{NP}$. The question is still open whether $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$." [2]

Bemerkung 9.1 "The given definition of NP above is not the historically original one. The original definition describes NP as the class that contains all problems for which a non-deterministic algorithm exists that can solve in polynomial time a certain problem instance. However, as this definition is less intuitive, the verification-based definition is usually preferred." [2]

Reducibility. "A problem Q is *reducible* to a problem Q' if every instance of Q can be formulated as an instance of Q' so that the solution S of problem Q follows from the solution S' of problem Q'. If Q can be reduced to Q' in a simple way, it is not harder to solve Q than to solve Q'. Q is *polynomial-time reducible* to Q' (shortly: $Q \leq_p Q'$) if the reduction takes only polynomial time. In particular, $Q \leq_p Q'$ implies $Q' \in \mathrm{P} \Rightarrow Q \in \mathrm{P}$." [2]

Example 9.4 "As an example, consider the *Consecutive Ones Problem* (C1P) that asks whether the columns of an $n \times m$ binary matrix M can be permuted, such that the resulting matrix M' has at most one run of ones in each row. In order to solve C1P, one can resort to a well-known problem in graph theory, the *Traveling Salesperson Problem* (TSP), that asks for a shortest Hamiltonian cycle in a weighted graph.

The reduction C1P $\leq_p$ TSP works as follows (see Figure 9.3): Given the binary matrix M, assume without loss of generality that it has one column of zeros only, and no two columns of M are identical. Now, define the weighted complete graph $G(M)$ that has one vertex per column of M, where the weight of an edge (v, w) is defined as the Hamming distance between the two columns represented by the two vertices v and w. Then it is easy to see that the C1P is fulfilled for M if and only if $G(M)$ contains a TSP tour of length $2n$, and otherwise not. Moreover, the construction of $G(M)$ can be done in polynomial time and a solution of the TSP can easily be backtransformed into a solution of the C1P. Therefore, given an algorithm that solves TSP efficiently, we would also have an algorithm solving C1P efficiently." [2]

Definition 9.5 NP-hard .

Definition 9.6 NP-complete .

Lemma 9.1 " The following property holds for the class NPC: If some NP-complete problem is solvable in polynomial time, then P = NP. If some problem in NP is *not* solvable in polynomial time, then no single NP-complete problem is solvable in polynomial time." [2]

"For solving the question whether $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$, NP-complete problems are intensely studied. The most common assumption is that $\mathrm{P} \neq \mathrm{NP}$. An alternative possibility is that $\mathrm{P} = \mathrm{NP}$ (see Figure 9.4 for schematic views of these two alternatives)." [2]

"It is not intuitively clear that any NP-complete problems exist because the conditions are quite strong and the implications enormous. Nevertheless, they do exist, and here we summarize a bit of that story.

The first known example of an NP-complete problem was the *satisfiability problem* (SAT)." [2]

Problem 9.1 The satisfiability problem SAT "Given a boolean formula of length n, is it satisfiable, i.e., is there a configuration of variables so that the evaluation of the formula yields T?" [2]

Example 9.5 "Given the formula $(a \vee b) \wedge (\neg a \vee b) \wedge (\neg b \vee c)$, is there a configuration of variables so that the formula is true? Indeed, there are two certificates that satisfy the given formula, first: $a = b = c = \mathtt{T}$ and second: $a = \mathtt{F}$ and $b = c = \mathtt{T}$. (The symbols denote $\vee$ = OR, $\wedge$ = AND, $\neg$ = NOT.)" [2]

"To show that SAT is NP-complete, we prove two properties:

1. We show that SAT belongs to the class NP. This is the case, because given a certificate (assignment of truth values to the variables in a boolean formula), it is easy to decide in polynomial time whether the evaluation of the formula with these variable values indeed yields T or not.

2. We show that any problem Q belonging to the class NP can be reduced to SAT ($Q \leq_p$ SAT) in polynomial time. This can be seen as follows:

   - $Q \in$ NP implies that a verification algorithm $A(x, y)$ exists for Q that tells in polynomial time for any problem instance $x \in I$ and certificate $y \in S$ whether y is a valid solution of x (i.e., $A(x, y) = \mathtt{T}$), or not (i.e., $A(x, y) = \mathtt{F}$).
   - Let $f(A, x)$ be a reduction function that partially evaluates the verification algorithm A according to its first argument, yielding the function $C_x(y) = f(A(x, y), x)$, which only depends on $y \in S$. Then we have:
     - Encoded as a boolean function, C_x is satisfiable ($C_x(y) = \mathtt{T}$) if and only if y is a solution to x, otherwise $C_x(y) = \mathtt{F}$.
     - It can be shown that for any $Q \in$ NP the corresponding boolean function $C_x(y)$ can be constructed in polynomial time (via CIRCUIT-SAT, see Cormen et al. (2001, Chapter 36.6)), therefore if SAT can be solved in polynomial time, also Q can be solved in polynomial time, i.e. $Q \leq_p$ SAT.

Therefore SAT is an NP-complete problem." [2]

Extension of the class NPC. "For showing that some problem Q is in NPC, it is sufficient to show that $Q \in$ NP and to reduce a known NP-complete problem Q' (e.g. Q' = SAT) to Q:

1. Show that $Q \in$ NP.

2. Choose a known NP-complete problem Q'.

3. Describe an algorithm that calculates in polynomial time a function f that projects any instance of Q' to an instance of Q.

4. Show that for f holds: $x \in Q'$ if and only if $f(x) \in Q$ for all $x \in \{\mathtt{F}, \mathtt{T}\}^*$.

" [2]

Consequences of NP completeness. "While the proof of NP-completeness of a problem implies that it is quite unlikely to find a deterministic polynomial-time algorithm to solve the problem to optimality in general, there are various ways to respond to such a result if one wants to solve the problem in feasible time:

1. Small problem instances might be solvable anyway. See Section 8.1.

2. Using adequate techniques to efficiently prune the search space, even larger instances may be solvable in reasonable time (running time heuristics). See Section 8.2.

3. The problem may not be hard for certain subclasses, so polynomial-time algorithms may exist for them, called FPT (Fixed Parameter Tractability) algorithms.

4. In an optimization problem, it may be sufficient to approximate the optimal solution to a certain degree. If closeness to the optimal solution can be guaranteed, one speaks of an approximation algorithm. See Section 9.2.

5. By no longer confining oneself to any optimality guarantee, one may get very fast heuristic algorithms that produce good but not always optimal or guaranteed close-to-optimal solutions (correctness heuristics). See Sections 10.1 and 10.2

" [2]

9.2 | Approximation Algorithms

"As mentioned in Item 4 above, in some situations it may be sufficient to approximate the optimal solution of a problem to a certain degree. The following definition states more precisely, what that means." [2]

Definition 9.7 c-Approximation An algorithm for a cost minimization problem for $c \geq 1$ where it is guaranteed that its solution has cost at most c times the optimal solution. For a maximization problem, an algorithm is called a c-approximation if it is guaranteed that its solution has score at least $1/c$ times the optimal solution..

9.3 | The Center-Star Approximation

" There exists a series of results on the approximability of the sum-of-pairs multiple sequence alignment problem.

A simple algorithm, the so-called *center star method* (Gusfield, 1991, 1993), is a 2-approximation for the sum-of-pairs multiple alignment problem if the underlying weighted edit distance satisfies the triangle inequality.

The algorithm is the following. First, for each sequence s_p, $1 \leq p \leq k$, its overall distance d_p to the other sequences is computed, i.e. the sum of the pairwise optimal alignment costs:

$$d_p = \sum_{1 \leq q \leq k} d(s_p, s_q).$$

Let c, $1 \leq c \leq k$, be an index such that d_c minimizes this overall distance. The sequence s_c is then called *center sequence*.

Secondly, a multiple alignment A_c is constructed from all pairwise optimal alignments $A_{\{c,q\}}$ where the center sequence is involved, i.e., all the optimal alignments of s_c and the other s_p, $p \neq c$, are combined into one multiple alignment A_c, such that $\pi_{\{c,q\}}(A_c) = A_{\{c,q\}}$.

The claim is that this alignment is a 2-approximation for the optimal sum-of-pairs multiple alignment cost of the sequences $s_1, s_2, \ldots, s_k$, i.e.,

$$\text{cost}_{SP}(A_c) \leq 2 \cdot \text{cost}_{SP}(A^{\text{opt}}).$$

This can be seen as follows:

$$
\begin{aligned}
\mathrm{cost}_{SP}(A_c) &= \sum_{1\le p<q\le k} \mathrm{cost}_2(\pi_{\{p,q\}}(A_c)) && \text{//definition}\\
&= \tfrac{1}{2} \sum_{1\le p\le k,\, 1\le q\le k} \mathrm{cost}_2(\pi_{\{p,q\}}(A_c)) && \text{//since } \mathrm{cost}_2(\pi_{\{p,p\}}) = 0\\
&\le \tfrac{1}{2} \sum_{1\le p\le k,\, 1\le q\le k} \big(\mathrm{cost}_2(\pi_{\{p,c\}}(A_c)) + \mathrm{cost}_2(\pi_{\{c,q\}}(A_c))\big) && \text{//triangle inequality}\\
&= \tfrac{1}{2} \sum_{1\le p\le k,\, 1\le q\le k} \big(d(s_p, s_c) + d(s_c, s_q)\big) && \text{//alignments to } s_c \text{ are optimal}\\
&= k \cdot \sum_{1\le q\le k} d(s_c, s_q) && \text{//star property}\\
&\le \sum_{1\le p\le k,\, 1\le q\le k} d(s_p, s_q) && \text{//minimal choice of } s_c\\
&\le \sum_{1\le p\le k,\, 1\le q\le k} \mathrm{cost}_2(\pi_{\{p,q\}}(A^{\mathrm{opt}}))\\
&= 2 \cdot \sum_{1\le p<q\le k} \mathrm{cost}_2(\pi_{\{p,q\}}(A^{\mathrm{opt}}))\\
&= 2 \cdot \mathrm{cost}_{SP}(A^{\mathrm{opt}})
\end{aligned}
$$

Hence A_c is a 2-approximation of the optimal alignment A^{opt}." [2]

Algorithm 8 Center-Star-Alignment

Input: Sequenzen $s_1, \ldots, s_k$, paarweise Distanz d
Output: Multiples Alignment $\mathcal{A}_c$

```
for p ← 1 to k do
    d_p ← Σ_{1≤q≤k} d(s_p, s_q)        ▷ d_p: Gesamtdistanz von s_p zu allen anderen Sequenzen
end for
c ← arg min_{1≤p≤k} d_p                ▷ Bestimme die Center-Sequenz s_c
A_c ← s_c                              ▷ Initialisiere Alignment mit der Center-Sequenz
for p ← 1 to k do
    if p ≠ c then
        Berechne ein optimales paarweises Alignment A_{c,p} von s_c und s_p
        Füge s_p entsprechend A_{c,p} in A_c ein
        Füge neu eingeführte Gaps in s_c in allen bereits eingefügten Sequenzen an denselben Positionen ein
    end if
end for
return A_c
```

Time and space complexity. "The running time of the algorithm can be estimated as follows, assuming all k sequences have the same length n: In the first step, $\binom{k}{2}$ pairwise alignments are computed, requiring overall $O(k^2n^2)$ time. In the second step, the $k-1$ alignments are combined into one multiple alignment which can be done in time proportional to the size of the constructed alignment, i.e. $O(k^2n)$. This yields an overall running time of $O(k^2n^2)$.

The space complexity is $O(n+k)$ for the linear-space distance computation and to store k computed values d_p. Additionally, $O(k^2n)$ space is used to store k pairwise alignments and the intermediate/final multiple alignment. Hence, the overall space complexity is $O(k^2n)$.

This idea can be generalized to a $2-\frac{\kappa}{k}$-approximation if instead of pairwise alignments, optimal multiple alignments of κ sequences are computed, see (Bafna et al., 1997)." [2]

CHAPTER 10

Heuristics for Multiple Sequence Alignment

10.1 | Divide-and-Conquer Alignment

"In this section we present a heuristic for sum-of-pairs multiple sequence alignment that no longer gives any guarantee about the resulting alignment cost, although in practice it is often very good. Therefore, the method is a *correctness heuristic*. While in the worst case the running time is still exponential in the number of sequences, the advantages are a polynomial space usage and that on average the practical running time is much faster than the exhaustive search, while the result is very often still optimal or close to optimal." [2]

Idea 10.1 "The basic idea of the heuristic is to cut all sequences at suitable cut points into left and right parts, and then to solve the two such generated alignment problems for the left parts and the right parts separately. This is done either by recursion or, if the sequences are short enough, by an exact algorithm. For a graphical sketch of the procedure." [2]

Schneide alle Sequenzen an je einer geeigneten Stelle in zwei Teile, sodass ein optimales MSA garantiert (fast) durch diese Schnitte läuft; löse die kleineren Blöcke (rekursiv oder exakt) und konkateniere. Die Schnittstellen ergeben sich aus Zusatzkosten-(C-)Matrizen, die den minimalen Preis eines Durchgangs durch eine Schnittposition messen.

Finding cut positions. "The main question about this procedure is how to choose good cut positions, since this choice is obviously critical for the quality of the final alignment. The ideal case is easy to formulate:" [2]

Definition 10.1 Optimal cut A family of cut positions $(c_1, c_2, \ldots, c_k)$ of the sequences $s_1, s_2, \ldots, s_k$ of lengths $n_1, n_2, \ldots, n_k$ for which exists an alignment A of the prefixes $(s_1[1 \ldots c_1], s_2[1 \ldots c_2], \ldots, s_k[1 \ldots c_k])$ and an alignment B of the suffixes $(s_1[c_1 + 1 \ldots n_1], s_2[c_2 + 1 \ldots n_2], \ldots, s_k[c_k + 1 \ldots n_k])$ such that the concatenation $A + +B$ is an optimal alignment of $s_1, s_2, \ldots, s_k$..

"There are many optimal cut positions, e.g. the trivial ones $(0, 0, \ldots, 0)$ and $(n_1, n_2, \ldots, n_k)$. In fact:" [2]

Lemma 10.1 "For any cut position $\hat{c}_1 \in \{0, \ldots, n_1\}$ of the first sequence s_1, there exist cut positions $c_2, \ldots, c_k$ of the remaining sequences $s_2, \ldots, s_k$ such that $(\hat{c}_1, c_2, \ldots, c_k)$ is an optimal cut of the sequences $s_1, s_2, \ldots, s_k$.

Proof Assume that A^{opt} is an optimal alignment. Choose one of the points between the characters in the first row of A^{opt} that correspond to the characters $s_1[\hat{c}_1]$ and $s_1[\hat{c}_1 + 1]$. The vertical "cut" at this point through the optimal alignment defines the cut positions $c_2, \ldots, c_k$ that, together with $\hat{c}_1$, are optimal by definition. See Figure 10.2 for an illustration. " [2]

" The lemma tells us that a first cut position $\hat{c}_1$ of s_1 can be chosen arbitrarily (for symmetry reasons and to arrive

at an efficient divide-and-conquer procedure, we will use $\hat{c}_1 = \lceil n_1/2 \rceil$), and then there always exist cut positions of the other sequences that produce an optimal cut.

The NP-completeness of the SP-alignment problem implies, however, that it is unlikely that there exists a polynomial time algorithm for finding optimal cuts of this type. Hence a heuristic is used to find good, though not always optimal cut positions.

This heuristic is based on pairwise sequence comparisons whose results are stored in an additional cost matrix that contains for each pair of cut positions (c_p, c_q) the penalty (additional cost) imposed by cutting the two sequences at these points, instead of cutting them at points compatible with an optimal pairwise alignment (see Definition 6.2). The efficient computation of additional cost matrices was discussed in Section 6.1.

The idea of using additional cost matrices to quantify the quality of a family of cut positions is that the closer a potential cut point is to an optimal pairwise alignment path, the smaller will be the contribution of this pair to the overall sum-of-pairs multiple alignment cost. Since we are dealing with sum-of-pairs alignment, we define the additional cost of a family of cuts along the same rationale." [2]

Definition 10.2 multiple additional cost of a cut $(c_1, c_2, \ldots, c_k)$ The additional cost matrix $C_{(p,q)}$ for sequence pair (s_p, s_q), $p < q$: $C(c_1, c_2, \ldots, c_k) := \sum_{1 \le p < q \le k} C_{(p,q)}(c_p, c_q)$

"Although maybe unintuitive, even a family of cut positions with zero multiple additional cost, i.e. one whose implied pairwise cuts are all compatible with the corresponding pairwise optimal alignments, is not necessarily optimal (see Example 10.1). The converse is also not true: Neither has an optimal cut always multiple additional cost zero, nor is a cut with smallest possible additional cost necessarily optimal." [2]

Example 10.1 " Let $s_1 = \texttt{CT}$, $s_2 = \texttt{AGT}$, and $s_3 = \texttt{G}$. Using unit cost edit distance, the three additional cost matrices are the following:

$$C_{(1,2)}: \begin{array}{c|cccc} & \epsilon & \texttt{A} & \texttt{G} & \texttt{T} \\ \hline \epsilon & 0 & 0 & 1 & 3 \\ \texttt{C} & 1 & 0 & 0 & 2 \\ \texttt{T} & 3 & 2 & 1 & 0 \end{array} \quad C_{(1,3)}: \begin{array}{c|cc} & \epsilon & \texttt{G} \\ \hline \epsilon & 0 & 1 \\ \texttt{C} & 0 & 0 \\ \texttt{T} & 1 & 0 \end{array} \quad C_{(2,3)}: \begin{array}{c|cc} & \epsilon & \texttt{G} \\ \hline \epsilon & 0 & 2 \\ \texttt{A} & 0 & 1 \\ \texttt{G} & 1 & 0 \\ \texttt{T} & 2 & 0 \end{array}$$

There are two cuts $(1, 1, 0)$ and $(1, 2, 1)$, both of which yield a multiple additional cost $C(1, 1, 0) = C(1, 2, 1) = 0$. Nevertheless, the implied multiple alignments are not necessarily optimal as can be seen for the cut $(1, 1, 0)$:

$$\begin{array}{c|c} \texttt{C} & \texttt{T} \\ \texttt{A} & \texttt{GT} \\ \varepsilon & \texttt{G} \end{array} \rightarrow \begin{pmatrix} \texttt{C} \\ \texttt{A} \\ - \end{pmatrix} ++ \begin{pmatrix} - & \texttt{T} \\ \texttt{G} & \texttt{T} \\ \texttt{G} & - \end{pmatrix} = \begin{pmatrix} \texttt{C} & - & \texttt{T} \\ \texttt{A} & \texttt{G} & \texttt{T} \\ - & \texttt{G} & - \end{pmatrix} \text{ (SP-cost 7, not optimal).}$$

But they might be optimal, as for the cut $(1, 2, 1)$:

$$\begin{array}{c|c} \texttt{C} & \texttt{T} \\ \texttt{AG} & \texttt{T} \\ \texttt{G} & \varepsilon \end{array} \rightarrow \begin{pmatrix} - & \texttt{C} \\ \texttt{A} & \texttt{G} \\ - & \texttt{G} \end{pmatrix} ++ \begin{pmatrix} \texttt{T} \\ \texttt{T} \\ - \end{pmatrix} = \begin{pmatrix} - & \texttt{C} & \texttt{T} \\ \texttt{A} & \texttt{G} & \texttt{T} \\ - & \texttt{G} & - \end{pmatrix} \text{ (SP-cost 6, optimal).}$$

" [2]

"Nevertheless, it is a good heuristic to choose cuts minimizing the multiple additional cost." [2]

DCA Algorithm. "The divide-and-conquer alignment algorithm first chooses an arbitrary cut position $\hat{c}_1$ for the first sequence (usually the sequence is cut in half for symmetry reasons) and then searches for the remaining cut positions $c_2, \ldots, c_k$ such that the cut $(\hat{c}_1, c_2, \ldots, c_k)$ minimizes $C(\hat{c}_1, c_2, \ldots, c_k)$.

Then the sequences are cut in this way, and the procedure is repeated recursively until the maximal sequence length drops below a threshold value L, when an exact multiple sequence alignment algorithm MSA (e.g. using Carrillo-Lipman bounds) is applied.

The pseudocode for this procedure is given in Algorithm 4 where $+\!\!+$ is the operator concatenating two alignments." [2]

Algorithm 9 DCA$(s_1, s_2, \ldots, s_k; L)$

```
for p ← 1 to k do
    n_p ← |s_p|
end for
if max{n_1, n_2, ..., n_k} ≤ L then
    return MSA(s_1, s_2, ..., s_k)
else
    ĉ_1 ← ⌈n_1/2⌉
    Compute (c_2, ..., c_k) such that C(ĉ_1, c_2, ..., c_k) is minimum
    return DCA(s_1[1 ... ĉ_1], s_2[1 ... c_2], ..., s_k[1 ... c_k]; L)
        ++ DCA(s_1[ĉ_1 + 1 ... n_1], s_2[c_2 + 1 ... n_2], ..., s_k[c_k + 1 ... n_k]; L)
end if
```

"The search space in the computation of cut positions is sketched in Figure 10.3. Since the cut position of the first sequence s_1 is fixed to $\hat{c}_1$ but the others can vary over their whole sequence length, the search space has size $O(n_2 \cdot n_3 \cdots n_k)$. This implies that the time for computing a cut (i.e., finding the minimum in this search space) by exhaustive search takes $O(k^2n^2 + n^{k-1})$ time and uses $O(k^2n^2)$ space." [2]

Overall complexity. " Let L be the length bound below which the exact MSA algorithm is applied. Assuming that the maximum sequence length n is of the form $n = L \cdot 2^D$ and all cuts are symmetric (i.e., $n_{(d)} = n/2^d$ for all $d = 0, \ldots, D-1$), the whole divide-and-conquer alignment procedure has the overall time complexity

$$
\begin{aligned}
&\left(\sum_{d=0}^{D-1} 2^d \Big(k^2 n_{(d)}^2 + n_{(d)}^{k-1}\Big)\right) + \frac{n}{L}\big(2^k L^{k-1} k^2\big) \\
&= \left(\sum_{d=0}^{D-1} 2^d \Big(k^2 \frac{n^2}{2^{2d}} + \frac{n^{k-1}}{2^{d(k-1)}}\Big)\right) + 2^k n L^{k-1} k^2 \\
&= \left(\sum_{d=0}^{D-1} \Big(k^2 \frac{n^2}{2^{d}} + \frac{n^{k-1}}{2^{d(k-2)}}\Big)\right) + 2^k n L^{k-1} k^2 \\
&\leq \left(\sum_{d=0}^{D-1} \Big(k^2 \frac{n^2}{2^{d}} + \frac{n^{k-1}}{2^{d}}\Big)\right) + 2^k n L^{k-1} k^2 && \text{(since } k \geq 3\text{)} \\
&= \big(k^2 n^2 + n^{k-1}\big) \left(\sum_{d=0}^{D-1} \frac{1}{2^d}\right) + 2^k n L^{k-1} k^2 \\
&\leq \big(k^2 n^2 + n^{k-1}\big) \cdot 2 + 2^k n L^{k-1} k^2 && \text{(geometric series)} \\
&\in O\big(k^2 n^2 + n^{k-1} + 2^k n L^{k-1} k^2\big)
\end{aligned}
$$

and space complexity

$$\max_{0 \leq d < D} \left(k n_{(d)} + \binom{k-1}{2} n_{(d)}^2\right) + L^k \in O\big(k^2 n^2 + L^k\big).$$

Compared to the standard dynamic programming for multiple alignment, the time is reduced only by one dimension, so it is not clear if the whole method is really an advantage. However:

- The space usage for the whole divide-and-conquer alignment procedure can be seen as polynomial since the space is mainly required for the $\binom{k}{2}$ additional cost matrices. The exponential term in the space complexity to compute the exact multiple alignment for sequences shorter than L can be neglected because L is small and can be chosen freely.

- Additional cost matrices have a very nice structure, since the low values that are of interest here are often close to the main diagonal, and the values increase rapidly when one leaves this diagonal. Based on this observation, several branch-and-bound heuristics have been developed to speed up the search for good cut positions, for more details see (Stoye, 1998).

An implementation of the divide-and-conquer alignment algorithm, plus further documentation, can be found at https://bibiserv.cebitec.uni-bielefeld.de/dca." [2]

10.2 | Segment-Based Alignment

"The global multiple alignment methods mentioned so far, like the progressive methods or the simultaneous method DCA, have properties that are not necessarily desirable in all applications.

In particular, the alignment depends on many parameters, like the cost function or substitution matrix, gap penalties, and a scoring function (e.g. sum of pairs) for the multiple alignment. Apart from the problem that these parameters need to be chosen in advance, the global methods have the well-known weakness that local similarities may be aligned in a wrong way if they are not significant in a global context." [2]

Example 10.2 Potential misalignment that may be caused by the choice of the affine gap penalties (Skript)

$$A = \begin{pmatrix} \mathtt{A} & \mathtt{F} & \mathtt{A} & \mathtt{T} & \mathtt{C} & \mathtt{A} & \mathtt{T} & \mathtt{C} & \mathtt{A} \\ \mathtt{A} & \mathtt{C} & \mathtt{A} & \mathtt{T} & - & - & - & - & \mathtt{A} \end{pmatrix}.$$

If instead of individual positions, whole units of local similarities are compared and used to build a global multiple alignment, the result will depend much less on the particular alignment parameters, and instead reflect more the local similarities in the data, like in the following example:

$$A' = \begin{pmatrix} \mathtt{A} & \mathtt{F} & \mathtt{A} & \mathtt{T} & \mathtt{C} & \mathtt{A} & \mathtt{T} & \mathtt{C} & \mathtt{A} \\ \mathtt{A} & - & - & - & \mathtt{C} & \mathtt{A} & \mathtt{T} & - & \mathtt{A} \end{pmatrix}.$$

This is the idea of segment-based sequence alignment.

10.2.1 | Segment Identification

"The first step in a segment-based alignment approach is to identify the segments. There are different possibilities to obtain segments.

The possibly simplest strategy, followed by the program TWOALIGN (Abdeddaïm, 1997) is to use (suboptimal) local alignments like they are computed by the Smith-Waterman or Waterman-Eggert algorithms. A disadvantage of this approach is that the local alignment computation also requires the usual alignment parameters like score function and gap penalties.

An alternative approach is to use *gap-free* local alignments. In this case, the segments correspond to diagonal segments of the alignment matrix. Since gap-free alignments can be computed faster than gapped alignments, more time can be spent on the computation of their score. The approach followed in the DIALIGN method (Section 10.2.3) is to use a statistical objective function that assigns the score $P_D(l_D, s_D)$ to a diagonal D of length l_D with s_D matches." [2]

Definition 10.3 The probability that a random segment of length l_D has at least s_D matches (p-value of the diagonal)

$$P_D(l_D, s_D) = \sum_{i=s_D}^{l_D} \binom{l_D}{i} p^i (1-p)^{l_D - i} \tag{10.1}$$

where $p = 0.25$ for nucleotides and $p = 0.05$ for amino acids.

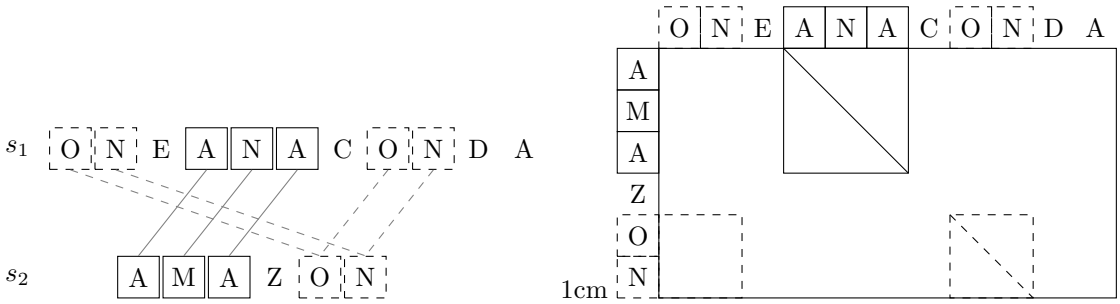


Figure 10.1: Inconsistent pairwise alignment segments.

Definition 10.4 weight of a diagonal (Wobei P_D der p-value of the diagonal ist)

$$w_D = -\ln(P_D) \tag{10.2}$$

"The time to compute all diagonals is $O(n^3)$, but this can be reduced to $O(n^2)$ if the maximal length of a diagonal segment is bounded by a constant." [2]

10.2.2 | Segment Selection and Assembly

" After a set of segments is identified, it is not necessarily possible to assemble them into a global alignment, since they might be *inconsistent*, i.e., the inclusion of some of the segments does not allow the inclusion of other segments into the alignment. In the case of two sequences, this is the case if and only if two segments "cross" as the two segments "ON" and "ANA"." [2]

"More generally, a set of segments of multiple sequences $s_1, s_2, \ldots, s_k$ is *inconsistent* if there exists no multiple alignment of these sequences that induces all these segments. Several methods exist to select from a set of segments a subset of consistent segments.

For example, the methods TWOALIGN and DIALIGN employ a *greedy* algorithm that considers one maximum scoring pairwise local alignment after the other (in order of decreasing alignment weight) and, if it is consistent with the previously assembled segments, adds it to the solution. Alternative methods are based on *chaining*, that we have seen in the context of whole genome alignment.

Given a set of consistent segments, in general there will still remain unaligned characters that are not contained in any of the consistent segments. In order to arrive at a global multiple alignment, it is desirable to also add these characters to the alignment. Different strategies can be followed.

The DIALIGN method simply fills the remaining regions with gaps, letting those characters unaligned that are not contained in any segment. Another possibility is to re-align the regions between the segments. This is not so easy, though, because the region "between the segments" is not clearly defined.

In any case, a global alignment is to be computed that respects the segments as fixed anchors, while for the remaining characters freedom is still given, as long as the resulting alignment is consistent with the segments. It has been suggested to align the sequences progressively (Myers, 1996) or simultaneously (Sammeth et al., 2003a)." [2]

10.2.3 | DIALIGN

"DIALIGN (Morgenstern et al., 1996; Morgenstern, 1999) is a practical implementation of segment-based alignment, enhanced in several ways.

By taking into account the information from various sequences during the selection of consistent segments, DIALIGN uses an optimized objective function in order to enhance the score of diagonals with a weak, but consistent signal over many sequences." [2]

Definition 10.5 overlap weight of a diagonal D Weight with the set of all diagonals $\mathcal{D}$ and where $\tilde{w}(D_1, D_2) := w_{D_3}$ is the weight of the (implied) overlap D_3 of the two diagonals D_1 and D_2:

$$\mathrm{olw}(D) = w_D + \sum_{E \in \mathcal{D}} \tilde{w}(D, E) \tag{10.3}$$

"For coding DNA sequences it is possible that the program automatically translates the codons into amino acids and then scores the diagonal with the amino acid scoring scheme.

In its second version, DIALIGN 2.0, the statistical score for diagonals was changed in order to upweight longer diagonals. Also, the algorithm for consistency checking and greedy addition of diagonals into the growing multiple alignment has been improved several times." [2]

Warnung 10.1 Achtung: DCA und DIALIGN sind Heuristiken, keine Optimalitätsgarantie. DIALIGN eignet sich für lokale Ähnlichkeiten (konservierte Blöcke in sonst unähnlichen Sequenzen); die Greedy-Auswahl kann suboptimal sein. Bei DCA hängt die Qualität von den Schnittpositionen ab.

CHAPTER 11

Examples and longer explanations

11.1 | Sellers' algorithm example

Input

Was	**Bezeichnung Pseudocode**	**Wert im Beispiel**
Pattern	$x \in \Sigma^m$	ACG
Text	$y \in \Sigma^n$	TTACG
threshold	$k \in N_0$	2
cost function with indel cost	$\gamma > 0$ und $cost(x[i], y[j])$	$\gamma = 1$ und $cost(x[i], y[j]) = \begin{cases} 0, & falls\ a = b \\ 1, & falls\ a \neq b \end{cases}$
defining an edit distance $d(\cdot, \cdot)$	$min \begin{cases} C_{j-1}(i-1+cost(x[i], y[j]) \\ C_{j-1}(i) + \gamma \\ C_j(i-1) + \gamma \end{cases}$	ergibt sich beim Berechnen

Leere Matrix

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ						
A						
C						
G						

initialize the 0-th column of the alignment matrix

for $i \leftarrow 0, ..., m$ **do**
  $C_0(i) \leftarrow i \cdot \gamma$
end for

Wird mit Einsetzen zu

for $i \leftarrow 0, ..., 3$ **do**
  $C_0(i) \leftarrow i$
end for

Da $i \cdot \gamma = i \cdot 1 = i$ und $m = 3$.

With i = 0, **Set** $C_0(0) = 0$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0					
A						
C						
G						

With i = 1, **Set** $C_0(1) = 1$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0					
A	1					
C						
G						

With i = 2, **Set** $C_0(2) = 2$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0					
A	1					
C	2					
G						

With i = 3, **Set** $C_0(3) = 3$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0					
A	1					
C	2					
G	3					

proceed column-by-column

Der Pseudocode:

for $j \leftarrow 1, ..., n$ **do**
  $C_j(0) \leftarrow 0$
  for $i \leftarrow 1, ..., m$ **do**

$$C_j(i) \leftarrow min \begin{cases} C_{j-1}(i-1) + cost(x[i], y[j]) \\ C_{j-1}(i) + \gamma \\ C_j(i-1) + \gamma \end{cases}$$

  end for
  if $C_j(m) \leq k$ **then**
report $(j, C_j(m))$
  end if
end for

Wird mit Einsetzen zu:

for $j \leftarrow 1, ..., 5$ **do**
  $C_j(0) \leftarrow 0$
  for $i \leftarrow 1, ..., 3$ **do**

$$C_j(i) \leftarrow min \begin{cases} C_{j-1}(i-1) + cost(x[i], y[j]) \\ C_{j-1}(i) + 1 \\ C_j(i-1) + 1 \end{cases}$$

  end for
  if $C_j(3) \leq 2$ **then**
report $(j, C_j(3))$
  end if
end for

Column $j = 1$

Set $C_1(0) \leftarrow 0$

j =	0	1	2	3	4	5
T / P	ϵ	T	T	A	C	G
ϵ	0	0				
A	1					
C	2					
G	3					

Then iterate through second loop (fill out rest of the column).

$$C_1(i) = min \begin{cases} C_{1-1}(i-1) + cost(x[i], y[1]) \\ C_{1-1}(i) + 1 \\ C_1(i-1) + 1 \end{cases} = min \begin{cases} C_0(i-1) + cost(x[i], y[1]) \\ C_0(i) + 1 \\ C_1(i-1) + 1 \end{cases}$$

With $i = 1$**:**

$$C_1(1) = min \begin{cases} C_0(0) + cost(x[1], y[1]) = 0 + cost(A, T) & = 1 \\ C_0(1) + 1 = 1 + 1 & = 2 \\ C_1(0) + 1 = 0 + 1 & = 1 \end{cases}$$

$= 1$

j =	0	1	2	3	4	5
T / P	ϵ	T	T	A	C	G
ϵ	0	0				
A	1	1				
C	2					
G	3					

With $i = 2$**:**

$$C_1(2) = min \begin{cases} C_0(1) + cost(x[2], y[1]) = 1 + cost(C, T) & = 2 \\ C_0(2) + 1 = 2 + 1 & = 3 \\ C_1(1) + 1 = 1 + 1 & = 2 \end{cases}$$

$= 2$

j =	0	1	2	3	4	5
T / P	ϵ	T	T	A	C	G
ϵ	0	0				
A	1	1				
C	2	2				
G	3					

With $i = 3$**:**

$$C_1(3) = min \begin{cases} C_0(2) + cost(x[3], y[1]) = 2 + cost(G, T) & = 3 \\ C_0(3) + 1 = 3 + 1 & = 4 \\ C_1(2) + 1 = 2 + 1 & = 3 \end{cases}$$

$= 3$

j =	0	1	2	3	4	5
T / P	ϵ	T	T	A	C	G
ϵ	0	0				
A	1	1				
C	2	2				
G	3	3				

$C_1(3) \leq 2$? $C_1(3) = 3$, also Nein. Kein Report.

Column $j = 2$

Set $C_2(0) \leftarrow 0$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0			
A	1	1				
C	2	2				
G	3	3				

Then iterate through second loop (fill out rest of the column).

$$C_2(i) = min \begin{cases} C_{2-1}(i-1) + cost(x[i], y[2]) \\ C_{2-1}(i) + 1 \\ C_2(i-1) + 1 \end{cases} = min \begin{cases} C_1(i-1) + cost(x[i], y[2]) \\ C_1(i) + 1 \\ C_2(i-1) + 1 \end{cases}$$

With $i = 1$**:**

$$C_2(1) = min \begin{cases} C_1(0) + cost(x[1], y[2]) = 0 + cost(A, T) & = 1 \\ C_1(1) + 1 = 1 + 1 & = 2 \\ C_2(0) + 1 = 0 + 1 & = 1 \end{cases}$$

$= 1$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0			
A	1	1	1			
C	2	2				
G	3	3				

With $i = 2$**:**

$$C_2(2) = min \begin{cases} C_1(1) + cost(x[2], y[2]) = 1 + cost(C, T) & = 2 \\ C_1(2) + 1 = 2 + 1 & = 3 \\ C_2(1) + 1 = 1 + 1 & = 2 \end{cases}$$

$= 2$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0			
A	1	1	1			
C	2	2	2			
G	3	3				

With $i = 3$**:**

$$C_2(3) = min \begin{cases} C_1(2) + cost(x[3], y[2]) = 2 + cost(G, T) & = 3 \\ C_1(3) + 1 = 3 + 1 & = 4 \\ C_2(2) + 1 = 2 + 1 & = 4 \end{cases}$$

$= 3$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0			
A	1	1	1			
C	2	2	2			
G	3	3	3			

$C_2(3) \leq 2$? $C_2(3) = 3$, also Nein. Kein Report.

Column $j = 3$

Set $C_3(0) \leftarrow 0$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0		
A	1	1	1			
C	2	2	2			
G	3	3	3			

Then iterate through second loop (fill out rest of the column).

$$C_3(i) = min \begin{cases} C_{3-1}(i-1) + cost(x[i], y[3]) \\ C_{3-1}(i) + 1 \\ C_3(i-1) + 1 \end{cases} = min \begin{cases} C_2(i-1) + cost(x[i], y[3]) \\ C_2(i) + 1 \\ C_3(i-1) + 1 \end{cases}$$

With $i = 1$**:**

$$C_3(1) = min \begin{cases} C_2(0) + cost(x[1], y[3]) = 0 + cost(A, A) & = 0 \\ C_2(1) + 1 = 1 + 1 & = 2 \\ C_3(0) + 1 = 0 + 1 & = 1 \end{cases}$$

$= 0$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0		
A	1	1	1	0		
C	2	2	2			
G	3	3	3			

With $i = 2$**:**

$$C_3(2) = min \begin{cases} C_2(1) + cost(x[2], y[3]) = 1 + cost(C, A) & = 2 \\ C_2(2) + 1 = 2 + 1 & = 3 \\ C_3(1) + 1 = 0 + 1 & = 1 \end{cases}$$

$= 1$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0		
A	1	1	1	0		
C	2	2	2	1		
G	3	3	3			

With $i = 3$**:**

$$C_3(3) = min \begin{cases} C_2(2) + cost(x[3], y[3]) = 2 + cost(G, A) & = 3 \\ C_2(3) + 1 = 3 + 1 & = 4 \\ C_3(2) + 1 = 1 + 1 & = 2 \end{cases}$$

$= 2$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0		
A	1	1	1	0		
C	2	2	2	1		
G	3	3	3	2		

$C_3(3) \leq 2$? $C_3(3) = 2$, also Ja. **Report** $(j, C_j(m)) = (3, 2)$

Column $j = 4$

Set $C_4(0) \leftarrow 0$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	
A	1	1	1	0		
C	2	2	2	1		
G	3	3	3	2		

Then iterate through second loop (fill out rest of the column).

$$C_4(i) = min \begin{cases} C_{4-1}(i-1) + cost(x[i], y[4]) \\ C_{4-1}(i) + 1 \\ C_4(i-1) + 1 \end{cases} = min \begin{cases} C_3(i-1) + cost(x[i], y[4]) \\ C_3(i) + 1 \\ C_4(i-1) + 1 \end{cases}$$

With $i = 1$**:**

$$C_4(1) = min \begin{cases} C_3(0) + cost(x[1], y[4]) = 0 + cost(A, C) & = 1 \\ C_3(1) + 1 = 0 + 1 & = 1 \\ C_4(0) + 1 = 0 + 1 & = 1 \end{cases}$$

$= 1$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	
A	1	1	1	0	1	
C	2	2	2	1		
G	3	3	3	2		

With $i = 2$**:**

$$C_4(2) = min \begin{cases} C_3(1) + cost(x[2], y[4]) = 0 + cost(C, C) & = 0 \\ C_3(2) + 1 = 1 + 1 & = 2 \\ C_4(1) + 1 = 1 + 1 & = 2 \end{cases}$$

$= 0$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	
A	1	1	1	0	1	
C	2	2	2	1	0	
G	3	3	3	2		

With $i = 3$**:**

$$C_4(3) = min \begin{cases} C_3(2) + cost(x[3], y[4]) = 1 + cost(G, C) & = 2 \\ C_3(3) + 1 = 2 + 1 & = 3 \\ C_4(2) + 1 = 0 + 1 & = 1 \end{cases}$$

$= 1$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	
A	1	1	1	0	1	
C	2	2	2	1	0	
G	3	3	3	2	1	

$C_4(3) \leq 2$? $C_4(3) = 1$, also Ja. **Report** $(j, C_j(m)) = (4, 1)$

Column $j = 5$

Set $C_5(0) \leftarrow 0$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	0
A	1	1	1	0	1	
C	2	2	2	1	0	
G	3	3	3	2	1	

Then iterate through second loop (fill out rest of the column).

$$C_5(i) = min \begin{cases} C_{5-1}(i-1) + cost(x[i], y[5]) \\ C_{5-1}(i) + 1 \\ C_5(i-1) + 1 \end{cases} = min \begin{cases} C_4(i-1) + cost(x[i], y[5]) \\ C_4(i) + 1 \\ C_5(i-1) + 1 \end{cases}$$

With $i = 1$**:**

$$C_5(1) = min \begin{cases} C_4(0) + cost(x[1], y[5]) = 0 + cost(A, G) & = 1 \\ C_4(1) + 1 = 1 + 1 & = 2 \\ C_5(0) + 1 = 0 + 1 & = 1 \end{cases}$$

$= 1$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	0
A	1	1	1	0	1	1
C	2	2	2	1	0	
G	3	3	3	2	1	

With $i = 2$**:**

$$C_5(2) = min \begin{cases} C_4(1) + cost(x[2], y[5]) = 1 + cost(C, G) & = 2 \\ C_4(2) + 1 = 0 + 1 & = 1 \\ C_5(1) + 1 = 1 + 1 & = 2 \end{cases}$$

$= 1$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	0
A	1	1	1	0	1	1
C	2	2	2	1	0	1
G	3	3	3	2	1	

With $i = 3$**:**

$$C_5(3) = min \begin{cases} C_4(2) + cost(x[3], y[5]) = 0 + cost(G, G) & = 0 \\ C_4(3) + 1 = 1 + 1 & = 2 \\ C_5(2) + 1 = 1 + 1 & = 2 \end{cases}$$

$= 0$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	0
A	1	1	1	0	1	1
C	2	2	2	1	0	1
G	3	3	3	2	1	0

$C_5(3) \leq 2$? $C_5(3) = 0$, also Ja. **Report** $(j, C_j(m)) = (5, 0)$

Ergebnis:

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	0
A	1	1	1	0	1	1
C	2	2	2	1	0	1
G	3	3	3	2	1	0

Last essential indices for $k = 2$ highlighted:

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	0
A	1	1	1	0	1	1
C	2	2	2	1	0	1
G	3	3	3	2	1	0

Wir hätten k aber auch anders setzen können

Last essential indices for $k = 1$ highlighted:

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	0
A	1	1	1	0	1	1
C	2	2	2	1	0	1
G	3	3	3	2	1	0

Last essential indices for $k = 0$ highlighted: (exaktes match)

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	0
A	1	1	1	0	1	1
C	2	2	2	1	0	1
G	3	3	3	2	1	0

11.2 | Ukkonen's algorithm example

Input

Was	Bezeichnung Pseudocode	Wert im Beispiel
Pattern	$x \in \Sigma^m$	ACG
Text	$y \in \Sigma^n$	TTACG
threshold	$k \in N_0$	2
cost function with indel cost	$\gamma > 0$ und $cost(x[i], y[j])$	$\gamma = 1$ und $cost(x[i], y[j]) = \begin{cases} 0, & falls\ a = b \\ 1, & falls\ a \neq b \end{cases}$
defining an edit distance $d(\cdot, \cdot)$	$min \begin{cases} C_{j-1}(i-1+cost(x[i], y[j]) \\ C_{j-1}(i) + \gamma \\ C_j(i-1) + \gamma \end{cases}$	ergibt sich beim Berechnen

Leere Matrix und ungesetzte Indizes

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ						
A						
C						
G						

- $i_0^* =$
- $i_1^* =$
- $i_2^* =$
- $i_3^* =$
- $i_4^* =$
- $i_5^* =$

Initialisierung

initialize the last essential index of column 0

$i_0^* = \min\{m, \lfloor k/\gamma \rfloor\} = \min\{3, \lfloor 2/1 \rfloor\} = 2$

- $i_0^* = 2$
- $i_1^* =$
- $i_2^* =$
- $i_3^* =$
- $i_4^* =$
- $i_5^* =$

initialize the 0-th column of the alignment matrix

for $i \leftarrow 0$ to i_0^* **do**
  $C_0(i) \leftarrow i \cdot \gamma$
end for

Wird mit Einsetzen zu

for $i \leftarrow 0$ to 2 **do**
  $C_0(i) \leftarrow i$
end for

With i = 0, **Set** $C_0(0) = 0$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0					
A						
C						
G						

With i = 1, **Set** $C_0(1) = 1$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0					
A	1					
C						
G						

With i = 2, **Set** $C_0(2) = 2$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0					
A	1					
C	2					
G						

Fertig mit der Initialisierung.

Spaltenweise Berechnung des Rests

Der Pseudocode:

for $j \leftarrow 1$ to n **do**
  $C_j(0) \leftarrow 0$
  $i_j^* \leftarrow 0$
  $i \leftarrow 1$
  while $i \leq i_{j-1}^*$ **do**

$$C_j(i) \leftarrow \min \begin{cases} C_{j-1}(i-1) + \text{cost}(x[i], y[j]), \\ C_{j-1}(i) + \gamma, \\ C_j(i-1) + \gamma \end{cases}$$

    if $C_j(i) \leq k$ **then**
      $i_j^* \leftarrow i$
    end if
    $i \leftarrow i + 1$
  end while
  if $i \leq m$ **then**

$$C_j(i) \leftarrow \min \begin{cases} C_{j-1}(i-1) + \text{cost}(x[i], y[j]), \\ C_j(i-1) + \gamma \end{cases}$$

    if $C_j(i) \leq k$ **then**
      $i_j^* \leftarrow i$
    end if
    $i \leftarrow i + 1$
    while $i \leq m$ **and** $C_j(i-1) + \gamma \leq k$ **do**
      $C_j(i) \leftarrow C_j(i-1) + \gamma$
      if $C_j(i) \leq k$ **then**
        $i_j^* \leftarrow i$
      end if
      $i \leftarrow i + 1$
    end while
  end if
  if $i_j^* = m$ **and** $C_j(m) \leq k$ **then** ▷ report match ending at column j and its cost
    return $(j, C_j(m))$
  end if
end for

Wird mit Einsetzen zu:

for $j \leftarrow 1$ to 5 **do**
  $C_j(0) \leftarrow 0$
  $i_j^* \leftarrow 0$
  $i \leftarrow 1$
  while $i \leq i_{j-1}^*$ **do**

$$C_j(i) \leftarrow \min \begin{cases} C_{j-1}(i-1) + \text{cost}(x[i], y[j]), \\ C_{j-1}(i) + 1, \\ C_j(i-1) + 1 \end{cases}$$

    if $C_j(i) \leq 2$ **then**
      $i_j^* \leftarrow i$
    end if
    $i \leftarrow i + 1$
  end while
  if $i \leq 3$ **then**

$$C_j(i) \leftarrow \min \begin{cases} C_{j-1}(i-1) + \text{cost}(x[i], y[j]), \\ C_j(i-1) + 1 \end{cases}$$

    if $C_j(i) \leq 2$ **then**
      $i_j^* \leftarrow i$
    end if
    $i \leftarrow i + 1$
    while $i \leq 3$ **and** $C_j(i-1) + 1 \leq 2$ **do**
      $C_j(i) \leftarrow C_j(i-1) + 1$
      if $C_j(i) \leq 2$ **then**
        $i_j^* \leftarrow i$
      end if
      $i \leftarrow i + 1$
    end while
  end if
  if $i_j^* = 3$ **and** $C_j(3) \leq 2$ **then**
    return $(j, C_j(3))$
  end if
end for

Column $j = 1$

- **Set** $C_1(0) = 0$
- **Set** $i_1^* = 0$
- **Set** $i = 1$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0				
A	1					
C	2					
G						

- $i_0^* = 2$
- $i_1^* = 0$
- $i_2^* =$
- $i_3^* =$
- $i_4^* =$
- $i_5^* =$

While-Schleife $i \leq i_0^*$

Greift "While $i \leq i_0^*$"? Ja, $1 \leq 2$.

$$C_1(1) = \min \begin{cases} C_0(0) + \text{cost}(x[1], y[1]) & = 0 + cost(A, T) & = 1 \\ C_0(1) + 1 & = 1 + 1 & = 2 \\ C_1(0) + 1 & = 0 + 1 & = 1 \end{cases}$$

$$= 1$$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0				
A	1	1				
C	2					
G						

Greift "$C_1(1) \leq k$"? Ja, $1 \leq 2$.

- $i_0^* = 2$

Also: $i_1^* \leftarrow 1$

- $i_1^* = 1$

i hochzählen: $i = i + 1 = 2$

Greift "While $i \leq i_0^*$"? Ja, $2 \leq 2$.

$$C_1(2) = \min \begin{cases} C_0(1) + \text{cost}(x[2], y[1]) & = 1 + cost(C, T) & = 2 \\ C_0(2) + 1 & = 2 + 1 & = 3 \\ C_1(1) + 1 & = 1 + 1 & = 2 \end{cases}$$

$$= 2$$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0				
A	1	1				
C	2	2				
G						

Greift "$C_1(2) \leq k$"? Ja, $2 \leq 2$.

- $i_0^* = 2$

Also: $i_1^* \leftarrow 2$

- $i_1^* = 2$

i hochzählen: $i = i + 1 = 3$

Greift "While $i \leq i_0^*$"? Nein, $3 \not\leq 2$.

END WHILE

If-Bedingung $i \leq m$

Greift "$i \leq m$"? Ja, $3 \leq 3$.

$$C_1(3) = \min \begin{cases} C_0(2) + \text{cost}(x[i], y[j]) & = 2 + cost(G, T) & = 3 \\ C_1(2) + 1 & = 2 + 2 & = 3 \end{cases}$$
$$= 3$$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0				
A	1	1				
C	2	2				
G		3				

Greift "$C_1(3) \leq 2$"? Nein, $3 \not\leq 2$

i hochzählen: $i = i + 1 = 4$

Greift "While $i \leq m$ and $C_1(3) + \gamma \leq k$"? Nein, $4 \not\leq 3$

Report If-Bedingung

Greift "$i_1^* = m = 3$ and $C_j(m) \leq k$"? Nein, $i_1^* = 2 \neq 3$

Column $j = 2$

- **Set** $C_2(0) = 0$
- **Set** $i_2^* = 0$
- **Set** $i = 1$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0			
A	1	1				
C	2	2				
G		3				

- $i_0^* = 2$
- $i_1^* = 2$
- $i_2^* = 0$
- $i_3^* =$
- $i_4^* =$
- $i_5^* =$

While-Schleife $i \leq i_0^*$

Greift "While $i \leq i_1^*$"? Ja, $1 \leq 2$.

$$C_2(1) = \min \begin{cases} C_1(0) + \text{cost}(x[1], y[2]) & = 0 + cost(A, T) & = 1 \\ C_1(1) + 1 & = 1 + 1 & = 2 \\ C_2(0) + 1 & = 0 + 1 & = 1 \end{cases}$$

$$= 1$$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0			
A	1	1	1			
C	2	2				
G		3				

Greift "$C_2(1) \leq k$"? Ja, $1 \leq 2$.

- $i_0^* = 2$
- $i_1^* = 2$
- $i_2^* = 1$

Also: $i_2^* \leftarrow 1$

i hochzählen: $i = i + 1 = 2$

Greift "While $i \leq i_1^*$"? Ja, $2 \leq 2$.

$$C_2(2) = \min \begin{cases} C_1(1) + \text{cost}(x[2], y[2]) & = 1 + cost(C, T) & = 2 \\ C_1(2) + 1 & = 2 + 1 & = 3 \\ C_2(1) + 1 & = 1 + 1 & = 2 \end{cases}$$

$$= 2$$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0			
A	1	1	1			
C	2	2	2			
G		3				

Greift "$C_2(2) \leq k$"? Ja, $2 \leq 2$.

- $i_0^* = 2$
- $i_1^* = 2$
- $i_2^* = 2$

Also: $i_2^* \leftarrow 2$

i hochzählen: $i = i + 1 = 3$

Greift "While $i \leq i_1^*$"? Nein, $3 \not\leq 2$.

END WHILE

If-Bedingung $i \leq m$

Greift "$i \leq m$"? Ja, $3 \leq 3$.

$$C_2(3) = \min \begin{cases} C_1(2) + \text{cost}(x[3], y[2]) & = 2 + cost(G, T) & = 3 \\ C_2(2) + 1 & = 2 + 2 & = 3 \end{cases}$$
$$= 3$$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0			
A	1	1	1			
C	2	2	2			
G		3	3			

Greift "$C_2(3) \leq 2$"? Nein, $3 \not\leq 2$

i hochzählen: $i = i + 1 = 4$

Greift "While $i \leq m$ and $C_2(3) + \gamma \leq k$"? Nein, $4 \not\leq 3$

Report If-Bedingung

Greift "$i_2^* = m = 3$ and $C_j(m) \leq k$"? Nein, $i_2^* = 2 \neq 3$

Column $j = 3$

- **Set** $C_3(0) = 0$
- **Set** $i_3^* = 0$
- **Set** $i = 1$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0		
A	1	1	1			
C	2	2	2			
G		3	3			

- $i_0^* = 2$
- $i_1^* = 2$
- $i_2^* = 2$
- $i_3^* = 0$
- $i_4^* =$
- $i_5^* =$

While-Schleife $i \leq i_0^*$

Greift "While $i \leq i_2^*$"? Ja, $1 \leq 2$.

$$C_3(1) = \min \begin{cases} C_2(0) + \text{cost}(x[1], y[3]) & = 0 + cost(A, A) & = 0 \\ C_2(1) + 1 & = 1 + 1 & = 2 \\ C_3(0) + 1 & = 0 + 1 & = 1 \end{cases}$$
$$= 0$$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0		
A	1	1	1	0		
C	2	2	2			
G		3	3			

Greift "$C_3(1) \leq k$"? Ja, $0 \leq 2$.

- $i_0^* = 2$
- $i_1^* = 2$
- $i_2^* = 2$
- $i_3^* = 1$

Also: $i_3^* \leftarrow 1$

i hochzählen: $i = i + 1 = 2$

Greift "While $i \leq i_2^*$"? Ja, $2 \leq 2$.

$$C_3(2) = \min \begin{cases} C_2(1) + \text{cost}(x[2], y[3]) & = 1 + cost(C, A) & = 2 \\ C_2(2) + 1 & = 2 + 1 & = 3 \\ C_3(1) + 1 & = 0 + 1 & = 1 \end{cases}$$
$$= 1$$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0		
A	1	1	1	0		
C	2	2	2	1		
G		3	3			

Greift "$C_3(2) \leq k$"? Ja, $1 \leq 2$.

- $i_0^* = 2$
- $i_1^* = 2$
- $i_2^* = 2$
- $i_3^* = 2$

Also: $i_3^* \leftarrow 2$

i hochzählen: $i = i + 1 = 3$

Greift "While $i \leq i_2^*$"? Nein, $3 \not\leq 2$.

END WHILE

If-Bedingung $i \leq m$

Greift "$i \leq m$"? Ja, $3 \leq 3$.

$$C_3(3) = \min \begin{cases} C_2(2) + \text{cost}(x[3], y[2]) & = 2 + cost(G, A) & = 3 \\ C_3(2) + 1 & = 1 + 1 & = 2 \end{cases}$$
$$= 2$$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0		
A	1	1	1	0		
C	2	2	2	1		
G		3	3	2		

Greift "$C_3(3) \leq 2$"? Ja, $2 \leq 2$

- $i_0^* = 2$
- $i_1^* = 2$
- $i_2^* = 2$
- $i_3^* = 3$

Also: $i_3^* = i = 3$

i hochzählen: $i = i + 1 = 4$

Greift "While $i \leq m$ and $C_3(3) + \gamma \leq k$"? Nein, $4 \not\leq 3$

Report If-Bedingung

Greift "$i_3^* = m = 3$ and $C_3(m) \leq k$"? Ja, $i_3^* = 3$ und $C_3(3) = 2 \leq k = 2$.

REPORT: $(3, 2)$

Column $j = 4$

- **Set** $C_4(0) = 0$
- **Set** $i_4^* = 0$
- **Set** $i = 1$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	
A	1	1	1	0		
C	2	2	2	1		
G		3	3	2		

- $i_0^* = 2$
- $i_1^* = 2$
- $i_2^* = 2$
- $i_3^* = 3$
- $i_4^* = 0$
- $i_5^* =$

While-Schleife $i \leq i_0^*$

Greift "While $i \leq i_3^*$**"?** Ja, $1 \leq 3$.

$$C_4(1) = \min \begin{cases} C_3(0) + \text{cost}(x[1], y[4]) & = 0 + cost(A, C) & = 1 \\ C_3(1) + 1 & = 0 + 1 & = 1 \\ C_4(0) + 1 & = 0 + 1 & = 1 \end{cases}$$
$$= 1$$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	
A	1	1	1	0	1	
C	2	2	2	1		
G		3	3	2		

Greift "$C_4(1) \leq k$**"?** Ja, $1 \leq 2$.

- $i_0^* = 2$
- $i_1^* = 2$
- $i_2^* = 2$
- $i_3^* = 3$
- $i_4^* = 1$

Also: $i_4^* \leftarrow 1$

i hochzählen: $i = i + 1 = 2$

Greift "While $i \leq i_3^*$**"?** Ja, $2 \leq 3$.

$$C_4(2) = \min \begin{cases} C_3(1) + \text{cost}(x[2], y[4]) & = 0 + cost(C, C) & = 0 \\ C_3(2) + 1 & = 1 + 1 & = 2 \\ C_4(1) + 1 & = 1 + 1 & = 2 \end{cases}$$
$$= 0$$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	
A	1	1	1	0	1	
C	2	2	2	1	0	
G		3	3	2		

Greift "$C_4(2) \leq k$**"?** Ja, $0 \leq 2$.

- $i_0^* = 2$
- $i_1^* = 2$
- $i_2^* = 2$
- $i_3^* = 3$
- $i_4^* = 2$

Also: $i_4^* \leftarrow 2$

i hochzählen: $i = i + 1 = 3$

Greift "While $i \leq i_3^*$"? Ja, $3 \leq 3$.

$$C_4(3) = \min \begin{cases} C_3(2) + \text{cost}(x[3], y[4]) & = 1 + cost(G, C) & = 2 \\ C_3(3) + 1 & = 2 + 1 & = 3 \\ C_4(2) + 1 & = 0 + 1 & = 1 \end{cases}$$

$$= 1$$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	
A	1	1	1	0	1	
C	2	2	2	1	0	
G		3	3	2	1	

Greift "$C_4(3) \leq k$"? Ja, $1 \leq 2$.

- $i_0^* = 2$
- $i_1^* = 2$
- $i_2^* = 2$
- $i_3^* = 3$
- $i_4^* = 3$

Also: $i_4^* \leftarrow 3$

i hochzählen: $i = i + 1 = 4$

Greift "While $i \leq i_3^*$"? Nein, $4 \not\leq 3$.

END WHILE

If-Bedingung $i \leq m$

Greift "$i \leq m$"? Nein, $4 \not\leq 3$.

Report If-Bedingung

Greift "$i_4^* = m = 3$ and $C_4(m) \leq k$"? Ja, $i_4^* = 3$ und $C_4(3) = 1 \leq k = 2$.

REPORT: $(4, 1)$

Column $j = 5$

- **Set** $C_5(0) = 0$
- **Set** $i_5^* = 0$
- **Set** $i = 1$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	0
A	1	1	1	0	1	
C	2	2	2	1	0	
G		3	3	2	1	

- $i_0^* = 2$
- $i_1^* = 2$
- $i_2^* = 2$
- $i_3^* = 3$
- $i_4^* = 3$
- $i_5^* = 0$

While-Schleife $i \leq i_0^*$

Greift "While $i \leq i_4^*$"? Ja, $1 \leq 3$.

$$C_5(1) = \min \begin{cases} C_4(0) + \text{cost}(x[1], y[5]) & = 0 + cost(A, G) & = 1 \\ C_4(1) + 1 & = 1 + 1 & = 2 \\ C_5(0) + 1 & = 0 + 1 & = 1 \end{cases}$$
$$= 1$$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	0
A	1	1	1	0	1	1
C	2	2	2	1	0	
G		3	3	2	1	

Greift "$C_5(1) \leq k$"? Ja, $1 \leq 2$.

- $i_0^* = 2$
- $i_1^* = 2$
- $i_2^* = 2$
- $i_3^* = 3$
- $i_4^* = 3$

Also: $i_5^* \leftarrow 1$

- $i_5^* = 1$

i hochzählen: $i = i + 1 = 2$

Greift "While $i \leq i_4^*$"? Ja, $2 \leq 3$.

$$C_5(2) = \min \begin{cases} C_4(1) + \text{cost}(x[2], y[5]) & = 1 + cost(C, G) & = 2 \\ C_4(2) + 1 & = 0 + 1 & = 1 \\ C_5(1) + 1 & = 1 + 1 & = 2 \end{cases}$$
$$= 1$$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	0
A	1	1	1	0	1	1
C	2	2	2	1	0	1
G		3	3	2	1	

Greift "$C_5(2) \leq k$"? Ja, $1 \leq 2$.

- $i_0^* = 2$
- $i_1^* = 2$

Also: $i_5^* \leftarrow 2$

- $i_2^* = 2$

- $i_3^* = 3$
- $i_4^* = 3$
- $i_5^* = 2$

i hochzählen: $i = i + 1 = 3$

Greift "While $i \leq i_4^*$"? Ja, $3 \leq 3$.

$$C_5(3) = \min \begin{cases} C_4(2) + \text{cost}(x[3], y[5]) & = 0 + cost(C, C) & = 0 \\ C_4(3) + 1 & = 1 + 1 & = 2 \\ C_5(2) + 1 & = 1 + 1 & = 2 \end{cases}$$
$$= 1$$

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	0
A	1	1	1	0	1	1
C	2	2	2	1	0	1
G		3	3	2	1	0

Greift "$C_5(3) \leq k$"? Ja, $0 \leq 2$.

- $i_0^* = 2$
- $i_1^* = 2$
- $i_2^* = 2$
- $i_3^* = 3$
- $i_4^* = 3$
- $i_5^* = 3$

Also: $i_5^* \leftarrow 3$

i hochzählen: $i = i + 1 = 4$

Greift "While $i \leq i_4^*$"? Nein, $4 \not\leq 3$.

END WHILE

If-Bedingung $i \leq m$

Greift "$i \leq m$"? Nein, $4 \not\leq 3$.

Report If-Bedingung

Greift "$i_5^* = m = 3$ and $C_5(m) \leq k$"? Ja, $i_5^* = 3$ und $C_5(3) = 0 \leq k = 2$.

REPORT: $(5, 0)$

Ergebnis

j =	0	1	2	3	4	5
T P	ϵ	T	T	A	C	G
ϵ	0	0	0	0	0	0
A	1	1	1	0	1	1
C	2	2	2	1	0	1
G		3	3	2	1	0

- $i_0^* = 2$
- $i_1^* = 2$
- $i_2^* = 2$
- $i_3^* = 3$
- $i_4^* = 3$
- $i_5^* = 3$

11.3 | Nussinov example

Input: RNA sequence $s = GGUCCA$ (meaning $n = 6$), scoring function for base pairs, minimum number $\delta = 1$ of unpaired bases in a hairpin loop

First loop

for $i \leftarrow 2$ to 6 **do**
  $M(i, i-1) \leftarrow 0$
  $T(i, i-1) \leftarrow$ "init" ▷ ϵ (case 0)
end for

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1		0				
2						
3						
4						
5						
6						

(a) $M(2, 1) \leftarrow 0$

$T =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1		init				
2						
3						
4						
5						
6						

(b) $T(2, 1) \leftarrow$ "init"

Figure 11.1: **i = 2** (first iteration)

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1		0				
2			0			
3						
4						
5						
6						

(a) $M(3, 2) \leftarrow 0$

$T =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1		init				
2			init			
3						
4						
5						
6						

(b) $T(3, 2) \leftarrow$ "init"

Figure 11.2: **i = 3** (second iteration)

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1		0				
2			0			
3				0		
4						
5						
6						

(a) $M(4, 3) \leftarrow 0$

$T =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1		init				
2			init			
3				init		
4						
5						
6						

(b) $T(4, 3) \leftarrow$ "init"

Figure 11.3: **i = 4** (third iteration)

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1		0				
2			0			
3				0		
4					0	
5						
6						

(a) $M(5, 4) \leftarrow 0$

$T =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1		init				
2			init			
3				init		
4					init	
5						
6						

(b) $T(5, 4) \leftarrow$ "init"

Figure 11.4: **i = 5** (fourth iteration)

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1		0				
2			0			
3				0		
4					0	
5						0
6						

(a) $M(6,5) \leftarrow 0$

$T =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1		init				
2			init			
3				init		
4					init	
5						init
6						

(b) $T(6,5) \leftarrow$ "init"

Figure 11.5: $\mathbf{i = 6}$ (fifth iteration)

Second loop

for $d \leftarrow 0$ to 1 **do**
  for $i \leftarrow 1$ to 5 **do**
    $M(i, i+d) \leftarrow 0$
    $T(i, i+d) \leftarrow$ "unpaired" ▷ $.S$ and/or $S.$ (case 1 or 2)
  end for
end for

First iteration of sub-loop

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	0	0				
2			0			
3				0		
4					0	
5						0
6						

(a) $M(1,1) \leftarrow 0$

$T =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	unpaired	init				
2			init			
3				init		
4					init	
5						init
6						

(b) $T(1,1) \leftarrow$ "unpaired"

Figure 11.6: $\mathbf{d = 0}$ und $\mathbf{i = 1}$ (first iteration)

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	0	0				
2		0	0			
3				0		
4					0	
5						0
6						

(a) $M(2,2) \leftarrow 0$

$T =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	unpaired	init				
2		unpaired	init			
3				init		
4					init	
5						init
6						

(b) $T(2,2) \leftarrow$ "unpaired"

Figure 11.7: $\mathbf{d = 0}$ und $\mathbf{i = 2}$ (second iteration)

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	0	0				
2		0	0			
3			0	0		
4					0	
5						0
6						

(a) $M(3,3) \leftarrow 0$

$T =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	unpaired	init				
2		unpaired	init			
3			unpaired	init		
4					init	
5						init
6						

(b) $T(3,3) \leftarrow$ "unpaired"

Figure 11.8: $\mathbf{d = 0}$ und $\mathbf{i = 3}$ (third iteration)

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	0	0				
2		0	0			
3			0	0		
4				0	0	
5						0
6						

(a) $M(4,4) \leftarrow 0$

$T =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	unpaired	init				
2		unpaired	init			
3			unpaired	init		
4				unpaired	init	
5						init
6						

(b) $T(4,4) \leftarrow$ "unpaired"

Figure 11.9: $\mathbf{d = 0}$ und $\mathbf{i = 4}$ (fourth iteration)

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	0	0				
2		0	0			
3			0	0		
4				0	0	
5					0	0
6						

(a) $M(5,5) \leftarrow 0$

$T =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	unpaired	init				
2		unpaired	init			
3			unpaired	init		
4				unpaired	init	
5					unpaired	init
6						

(b) $T(5,5) \leftarrow$ "unpaired"

Figure 11.10: $\mathbf{d = 0}$ und $\mathbf{i = 5}$ (fifth iteration)

Second iteration of sub-loop

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	0	0				
2	0	0	0			
3			0	0		
4				0	0	
5					0	0
6						

(a) $M(1,2) \leftarrow 0$

$T =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	unpaired	init				
2	unpaired	unpaired	init			
3			unpaired	init		
4				unpaired	init	
5					unpaired	init
6						

(b) $T(1,2) \leftarrow$ "unpaired"

Figure 11.11: $\mathbf{d = 1}$ und $\mathbf{i = 1}$ (first iteration)

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	0	0				
2	0	0	0			
3		1	0	0		
4				0	0	
5					0	0
6						

(a) $M(2,3) \leftarrow 0$

$T =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	unpaired	init				
2	unpaired	unpaired	init			
3		case 3	unpaired	init		
4				unpaired	init	
5					unpaired	init
6						

(b) $T(2,3) \leftarrow$ case

Figure 11.12: $\mathbf{d = 1}$ und $\mathbf{i = 2}$ (second iteration)

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	0	0				
2	0	0	0			
3		0	0	0		
4			0	0	0	
5					0	0
6						

(a) $M(3,4) \leftarrow 0$

$T =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	unpaired	init				
2	unpaired	unpaired	init			
3		unpaired	unpaired	init		
4			unpaired	unpaired	init	
5					unpaired	init
6						

(b) $T(3,4) \leftarrow$ "unpaired"

Figure 11.13: $\mathbf{d = 1}$ und $\mathbf{i = 3}$ (third iteration)

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	0	0				
2	0	0	0			
3		0	0	0		
4			0	0	0	
5				0	0	0
6						

(a) $M(4,5) \leftarrow 0$

$T =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	unpaired	init				
2	unpaired	unpaired	init			
3		unpaired	unpaired	init		
4			unpaired	unpaired	init	
5				unpaired	unpaired	init
6						

(b) $T(4,5) \leftarrow$ "unpaired"

Figure 11.14: $\mathbf{d = 1}$ und $\mathbf{i = 4}$ (fourth iteration)

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	0	0				
2	0	0	0			
3		0	0	0		
4			0	0	0	
5				0	0	0
6					0	

(a) $M(5,6) \leftarrow 0$

$T =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	unpaired	init				
2	unpaired	unpaired	init			
3		unpaired	unpaired	init		
4			unpaired	unpaired	init	
5				unpaired	unpaired	init
6					unpaired	

(b) $T(5,6) \leftarrow$ "unpaired"

Figure 11.15: $\mathbf{d = 1}$ und $\mathbf{i = 5}$ (fifth iteration)

Third loop

for $d \leftarrow 2$ to 5 **do**
  for $i \leftarrow 1$ to 5 **do**
    Compute $M(i, i+d)$ according to the recurrence
    Store the maximizing case in $T(i, i+d)$ ▷ cases 1–3; or case 4 with maximizing k
  end for
end for

First iteration of sub-loop

We are loking at $M(1,3)$

Is $j - i \leq \delta$?

→ Nein $3 - 1 = 2 > 1$

Is $\{s[1], s[3]\} = \{G, U\}$ a valid base pair?

→ Wobble, yes

$$M(1,3) = max \begin{cases} M(2,3) & = 0 \\ M(1,2) & = 0 \\ M(2,2) + \mu(2,4) = M(2,2) + score(\{G,U\}) = 0 + 1 & = 1 \\ max_{2<k<3}\{M(1,???) + M(???,3)\} & = NaN \end{cases} \quad = 1$$

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	0	0				
2	0	0	0			
3	1	0	0	0		
4			0	0	0	
5				0	0	0
6					0	

(a) Compute $M(1,3)$ according to the recurrence

$T =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	unpaired	init				
2	unpaired	unpaired	init			
3	unpaired	unpaired	unpaired	init		
4			unpaired	unpaired	init	
5				unpaired	unpaired	init
6					unpaired	

(b) $T(1,3) \leftarrow$ case

Figure 11.16: $\mathbf{d = 2}$ und $\mathbf{i = 1}$ (first iteration)

We are loking at $M(2,4)$

Is $j - i \leq \delta$?

$\rightarrow$ No $4 - 2 = 2 > 1$

Is $\{s[2], s[4]\} = \{G, C\}$ a valid base pair?

$\rightarrow$ Yes

$$M(2,4) = max \begin{cases} M(3,4) & = 0 \\ M(2,3) & = 0 \\ M(3,3) + \mu(2,4) = M(3,3) + score(\{G,C\}) = 0 + 1 & = 1 \\ max_{3<k<4}\{M(2,???) + M(???,4)\} & = NaN \end{cases} = 1$$

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	0	0				
2	0	0	0			
3	1	0	0	0		
4		1	0	0	0	
5				0	0	0
6					0	

(a) Compute $M(2,4)$ according to the recurrence

$T =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	unpaired	init				
2	unpaired	unpaired	init			
3	case 3	unpaired	unpaired	init		
4		case 3	unpaired	unpaired	init	
5				unpaired	unpaired	init
6					unpaired	

(b) $T(2,4) \leftarrow$ case

Figure 11.17: $\mathbf{d = 2}$ und $\mathbf{i = 2}$ (second iteration)

We are loking at $M(3,5)$

Is $j - i \leq \delta$?

$\rightarrow$ No $5 - 2 = 2 > 1$

Is $\{s[3], s[5]\} = \{U, C\}$ a valid base pair?

$\rightarrow$ No

$$M(3,5) = max \begin{cases} M(4,5) & = 0 \\ M(3,4) & = 0 \\ M(4,4) + \mu(3,5) = M(4,4) + score(\{U,C\}) = 0 + 0 & = 0 \\ max_{4<k<5}\{M(3,???) + M(???,5)\} & = NaN \end{cases} = 0$$

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	0	0				
2	0	0	0			
3	1	0	0	0		
4		1	0	0	0	
5			0	0	0	0
6					0	

(a) Compute $M(3,5)$ according to the recurrence

$T =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	unpaired	init				
2	unpaired	unpaired	init			
3	case 3	unpaired	unpaired	init		
4		case 3	unpaired	unpaired	init	
5			case 1-3	unpaired	unpaired	init
6					unpaired	

(b) $T(3,5) \leftarrow$ case

Figure 11.18: $\mathbf{d = 2}$ und $\mathbf{i = 3}$ (third iteration)

We are loking at $M(4,6)$

Is $j - i \leq \delta$?

$\rightarrow$ No $6 - 4 = 2 > 1$

Is $\{s[4], s[6]\} = \{C, A\}$ a valid base pair?

$\rightarrow$ No

$$M(4,6) = max \begin{cases} M(5,6) & = 0 \\ M(4,5) & = 0 \\ M(5,5) + \mu(4,6) = M(5,5) + score(\{C,A\}) = 0 + 0 & = 0 \\ max_{5<k<6}\{M(4,???) + M(???,6)\} & = NaN \end{cases} = 0$$

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	0	0				
2	0	0	0			
3	1	0	0	0		
4		1	0	0	0	
5			0	0	0	0
6				0	0	

(a) Compute $M(4,6)$ according to the recurrence

$T =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	unpaired	init				
2	unpaired	unpaired	init			
3	case 3	unpaired	unpaired	init		
4		case 3	unpaired	unpaired	init	
5			case 1-3	unpaired	unpaired	init
6				case 1-3	unpaired	

(b) $T(4,6) \leftarrow$ case

Figure 11.19: $\mathbf{d = 2}$ und $\mathbf{i = 4}$ (fourth iteration)

$M(5,7)$ **does not exist**

Iteration 2 and 3

were taken from [5] (with Minimal loop length l = 1 and Delta #bp to maximum = 0). Resulting in:

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	0	0				
2	0	0	0			
3	1	0	0	0		
4	1	1	0	0	0	
5	2	1	0	0	0	0
6		1	1	0	0	

(a) Compute $M(4,6)$ according to the recurrence

$T =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	unpaired	init				
2	unpaired	unpaired	init			
3	case 3	unpaired	unpaired	init		
4	–	case 3	unpaired	unpaired	init	
5	–	–	case 1-3	unpaired	unpaired	init
6		–	–	case 1-3	unpaired	

(b) $T(4,6) \leftarrow$ case

Figure 11.20: $\mathbf{d = 2..3}$ und $\mathbf{i = 1..5}$ (whole iterations)

Fourth iteration of sub-loop

We are loking at $M(1,6)$

Is $j - i \leq \delta$?

$\rightarrow$ No $6 - 1 = 5 > 1$

Is $\{s[1], s[6]\} = \{G, A\}$ a valid base pair?

$\rightarrow$ No

$$M(1,6) = max \begin{cases} M(2,6) & = 1 \\ M(1,5) & = 2 \\ M(2,5) + \mu(1,6) = M(2,5) + score(\{G,A\}) = 1 + 0 & = 0 \\ max_{2<k<6}\{M(1,k-1) + M(k,6)\} = * & = 1 \end{cases} = 2$$

$$\begin{aligned} *max_{2<k<6}\{M(1,k-1) + M(k,6)\} &= max\{M(1,3-1) + M(3,6), M(1,4-1) + M(4,6), M(1,5-1) + M(5,6)\} \\ &= max\{M(1,2) + M(3,6), M(1,3) + M(4,6), M(1,4) + M(5,6)\} \\ &= max\{0 + 1, 1 + 0, 1 + 0\} \\ &= max\{1, 1, 1\} \\ &= 1 \end{aligned}$$

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	0	0				
2	0	0	0			
3	1	0	0	0		
4	1	1	0	0	0	
5	2	1	0	0	0	0
6	2	1	1	0	0	

(a) Compute $M(4,6)$ according to the recurrence

$T =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	unpaired	init				
2	unpaired	unpaired	init			
3	case 3	unpaired	unpaired	init		
4	–	case 3	unpaired	unpaired	init	
5	–	–	case 1-3	unpaired	unpaired	init
6	case 3	–	–	case 1-3	unpaired	

(b) $T(4,6) \leftarrow$ case

Figure 11.21: $\mathbf{d = 5}$ und $\mathbf{i = 1}$ (first iteration)

11.4 | Hirschberg Example

Gegeben die Sequenzen GAG und CACG.

D	ε	G	A	G
ε	0	1	2	3
C	1	1	2	3
A	2	2	1	2
C	3	3	2	2
G	4	3	3	2

D^{rev}	G	A	G	ε
C	2	2	3	4
A	2	1	2	3
C	2	1	1	2
G	2	1	0	1
ε	3	2	1	0

Mit

$$T(i,j) = D(i,j) + D^{\text{rev}}(i,j).$$

Berechnen wir T:

T	ε	G	A	G
ε	$0+2$	$1+2$	$2+3$	$3+4$
C	$1+2$	$1+1$	$2+2$	$3+3$
A	$2+2$	$2+1$	$1+1$	$2+2$
C	$3+2$	$3+1$	$2+0$	$2+1$
G	$4-3$	$3+2$	$3+1$	$2+0$

$=$

T	ε	G	A	G
ε	2	3	5	7
C	3	2	4	6
A	4	3	2	4
C	5	4	2	3
G	7	5	4	2

Mit

$$C(i,j) = T(i,j) - d(x,y) = T(i,j) - d(GAC, CACG) = T(i,j) - 2$$

Berechnen wir:

C	ε	G	A	G
ε	$2-2$	$3-2$	$5-2$	$7-2$
C	$3-2$	$2-2$	$4-2$	$6-2$
A	$4-2$	$3-2$	$2-2$	$4-2$
C	$5-2$	$4-2$	$2-2$	$3-2$
G	$7-2$	$5-2$	$4-2$	$2-2$

$=$

C	ε	G	A	G
ε	0	1	3	5
C	1	0	2	4
A	2	1	0	2
C	3	2	0	1
G	5	3	2	0

C	ε	G	A	G
ε	0	1	3	5
C	1	0	2	4
A	2	1	0	2
C	3	2	0	1
G	5	3	2	0

I.

II.

I. align CA und GA.

D	ε	G	A
ε	0	1	2
C	1	1	2
A	2	2	1

D^{rev}	G	A	ε
C	1	1	2
A	1	0	1
ε	2	1	0

Mit

$$T(i,j) = D(i,j) + D^{\text{rev}}(i,j).$$

Berechnen wir T:

T	ε	G	A
ε	$0+1$	$1+1$	$2+2$
C	$1+1$	$1+0$	$2+1$
A	$2+2$	$2+1$	$1+0$

$=$

T	ε	G	A
ε	1	2	4
C	2	1	3
A	4	3	1

Mit

$$C(i,j) = T(i,j) - d(x,y) = T(i,j) - d(GA, CA) = T(i,j) - 1$$

Berechnen wir:

C	ε	G	A
ε	$1-1$	$2-1$	$4-1$
C	$2-1$	$1-1$	$3-1$
A	$4-1$	$3-1$	$1-1$

$=$

C	ε	G	A
ε	0	1	3
C	1	0	2
A	3	2	0

C	ε	G	A
ε	0	1	3
C	1	0	2
A	3	2	0

i. ii.

i: Align C und G

$$\begin{pmatrix} C \\ G \end{pmatrix}$$

ii: Align A und A

$$\begin{pmatrix} A \\ A \end{pmatrix}$$

I. align CG und G.

D	ε	G
ε	0	1
C	1	1
G	2	1

D^{rev}	G	ε
C	1	2
G	0	1
ε	1	0

Mit

$$T(i,j) = D(i,j) + D^{\text{rev}}(i,j).$$

Berechnen wir T:

T	ε	G
ε	$0+1$	$1+2$
C	$1+0$	$1+1$
G	$2+1$	$1+0$

$=$

T	ε	G
ε	1	3
C	1	2
G	3	1

Mit

$$C(i,j) = T(i,j) - d(x,y) = T(i,j) - d(CG, G) = T(i,j) - 1$$

Berechnen wir:

C	ε	G
ε	$1-1$	$3-1$
C	$1-1$	$2-1$
G	$3-1$	$1-1$

$=$

C	ε	G
ε	0	2
C	0	1
G	2	0

i.

C	ε	G
ε	0	2
C	0	1
G	2	0

ii.

i: Align C und ϵ

$$\begin{pmatrix} C \\ \epsilon \end{pmatrix}$$

ii: Align G und G

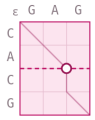
$$\begin{pmatrix} G \\ G \end{pmatrix}$$

11.4.1 | Anders betrachtet

Example 11.1

Schritt 1 · Ebene 0 · Teilung

Rechteck X[0..4] × Y[0..3]



D(i,j)

	ε	G	A	G
ε	0	1 ←	2 ←	3 ←
C	1 ↑	1 ↖	2 ↖←	3 ↖←
A	2 ↑	2 ↖↑	1 ↖	2 ←
C	3 ↑	3 ↖↑	2 ↑	2 ↖
G	4 ↑	3 ↖	3 ↑	2 ↖

D^rev(i,j)

	G	A	G	ε
C	2 ↘	2 ↓	3 ↓	4 ↓
A	2 ↘→	1 ↘	2 ↓	3 ↓
C	2 ↘→	1 ↘	1 ↓	2 ↓
G	2 ↘→	1 →	0 ↘	1 ↓
ε	3 →	2 →	1 →	0

T(i,j) = D(i,j) + D^rev(i,j)

	ε	G	A	G
ε	2	3	5	7
C	3	2	4	6
A	4	3	2	4
C	5	4	2	3
G	7	5	4	2

T (Summe Mittelzeile)

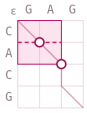
	0	1	2	3
T	4	3	2	4

Minimum von T in Spalte p=2 (Y-Position 2) → Schnittpunkt. Beide Hälften werden anschließend weiter berechnet.

d(x,y) (Editierdistanz der Sequenzen) = 2

Schritt 2 · Ebene 1 · Teilung

Rechteck X[0..2] × Y[0..2]



D(i,j)

	ε	G	A
ε	0	1 ←	2 ←
C	1 ↑	1 ↖	2 ↖←
A	2 ↑	2 ↖↑	1 ↖

D^rev(i,j)

	G	A	ε
C	1 ↘	1 ↓	2 ↓
A	1 →	0 ↘	1 ↓
ε	2 →	1 →	0

T(i,j) = D(i,j) + D^rev(i,j)

	ε	G	A
ε	1	2	4
C	2	1	3
A	4	3	1

T (Summe Mittelzeile)

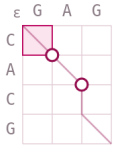
	0	1	2
T	2	1	3

Minimum von T in Spalte p=1 (Y-Position 1) → Schnittpunkt. Beide Hälften werden anschließend weiter berechnet.

d(x,y) (Editierdistanz der Sequenzen) = 1

Schritt 3 · Ebene 2 · Basisfall

Rechteck X[0..1] × Y[0..1]



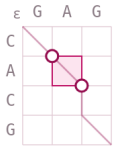
Basisfall D (direktes Alignment)

	ε	G
ε	0	1 ←
C	1 ↑	1 ↖

Eine Zeile/Spalte übrig – der Pfad (hervorgehoben) wird direkt a

Schritt 4 · Ebene 2 · Basisfall

Rechteck X[1..2] × Y[1..2]



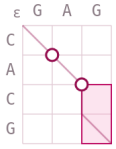
Basisfall D (direktes Alignment)

	ε	A
ε	0	1 ←
A	1 ↑	0 ↖

Eine Zeile/Spalte übrig – der Pfad (hervorgehoben) wird direkt a

Schritt 5 · Ebene 1 · Basisfall

Rechteck X[2..4] × Y[2..3]



Basisfall D (direktes Alignment)

	ε	G
ε	0	1 ←
C	1 ↑	1 ↖
G	2 ↑	1 ↖

Eine Zeile/Spalte übrig – der Pfad (hervorgehoben) wird direkt a

Figure 11.22

11.5 | Smith-Waterman example

Input

Sequences to be aligned: $A = GTAA$ and $B = GCTA$

Außerdem:

- Gap: $s(x, -) = s(-, x) = -1$
- Match: $s(x, y) = 3$ mit $x = y$
- Mismatch: $s(x, y) = 1$ mit $x \neq y$

i \ j		G	C	T	A
G					
T					
A					
A					

Figure 11.23: Matrix before the start of the algorithm.

Initialisation - (0,0)

$S(0, 0) \leftarrow 0$

i \ j		G	C	T	A
	0				
G					
T					
A					
A					

Figure 11.24: .

Initialisation - first loop

for $i \leftarrow 0$ to n **do**
  $S(i, 0) \leftarrow 0$
end for

i \ j		G	C	T	A
	0				
G	0				
T	0				
A	0				
A	0				

Initialisation - second loop

for $j \leftarrow 0$ to m **do**
  $S(0, j) \leftarrow 0$
end for

i \ j		G	C	T	A
	0	0	0	0	0
G	0				
T	0				
A	0				
A	0				

Main step - Calculation

for $j \leftarrow 1$ to n **do**
  for $i \leftarrow 1$ to n **do**

$$S(i, j) \leftarrow \max \begin{cases} S(i-1, j-1) & +s(a_i, b_j) \\ S(i-1, j) & +s(a_i, -) \\ S(i, j-1) & +s(-, b_j) \\ 0 \end{cases} = \max \begin{cases} S(i-1, j-1) & + \quad 3 & a_i = b_j \\ S(i-1, j-1) & + \ 1 & a_i \neq b_j \\ S(i-1, j) & + -1 & b_j = - \\ S(i, j-1) & + -1 & a_i = - \\ 0 \end{cases}$$

end for
end for

j = 1, i = 1:

$$S(1,1) = \max \begin{cases} S(0,0) & +3 = 3 \\ S(0,1) & +-1 = -1 \\ S(1,0) & +-1 = -1 \\ 0 \end{cases} = 3$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	3			
T	0				
A	0				
A	0				

j = 2, i = 1:

$$S(1,2) = \max \begin{cases} S(0,1) & + \ 1 & = 1 \\ S(0,2) & +-1 & = -1 \\ S(1,1) & +-1 & = 3-1 = 2 \\ 0 \end{cases} = 2$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	3	2		
T	0				
A	0				
A	0				

j = 3, i = 1:

$$S(1,3) = \max \begin{cases} S(0,2) & + \ 1 & = 1 \\ S(0,3) & +-1 & = -1 \\ S(1,2) & +-1 & = 2-1 = 1 \\ 0 \end{cases} = 1$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	3	2	1	
T	0				
A	0				
A	0				

j = 4, i = 1:

$$S(1,4) = \max \begin{cases} S(0,3) & + \ 1 & = 1 \\ S(0,4) & +-1 & = -1 \\ S(1,3) & +-1 & = 1-1 = 0 \\ 0 \end{cases} = 1$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	3	2	1	1
T	0				
A	0				
A	0				

j = 1, i = 2:

$$S(2,1) = \max \begin{cases} S(1,0) & + \ 0+1 = 1 \\ S(1,1) & +-1 = 3-1 = 2 \\ S(2,0) & +-1 = 0-1 = -1 \\ 0 \end{cases} = 2$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	3	2	1	1
T	0	2			
A	0				
A	0				

j = 2, i = 2:

$$S(2,2) = \max \begin{cases} S(1,1) & + \quad 3+1=4 \\ S(1,2) & + -1 = 2-1 = 1 \\ S(2,1) & + -1 = 1-1 = 1 \\ 0 \end{cases} = 4$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	3	2	1	1
T	0	2	4		
A	0				
A	0				

j = 2, i = 3:

$$S(2,3) = \max \begin{cases} S(1,2) & + \quad 2+3=5 \\ S(1,3) & + -1 = 1-1 = 0 \\ S(2,2) & + -1 = 4-1 = 3 \\ 0 \end{cases} = 5$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	3	2	1	1
T	0	2	4	5	
A	0				
A	0				

j = 2, i = 4:

$$S(2,4) = \max \begin{cases} S(1,3) & + \quad 1+1=2 \\ S(1,4) & + -1 = 1-1 = 0 \\ S(2,3) & + -1 = 5-1 = 4 \\ 0 \end{cases} = 4$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	3	2	1	1
T	0	2	4	5	4
A	0				
A	0				

Continue like that...

j = 2, i = 4:

$$S(4,4) = \max \begin{cases} S(3,3) + \quad 3 & = 5+3=8 \\ S(3,4) + -1 & = 8-1=7 \\ S(4,3) + -1 & = 4-1=3 \\ 0 \end{cases} = 8$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	3	2	1	1
T	0	2	4	5	4
A	0	1	3	5	8
A	0	1	2	4	8

Result

Score = 8 made up out of:

- Match G G
- Mismatch C T
- Mismatch T A
- Match A A

i \ j		G	C	T	A
	0	0	0	0	0
G	0	3	2	1	1
T	0	2	4	5	4
A	0	1	3	5	8
A	0	1	2	4	8

11.6 | Waterman-Eggert Example

Input

i \ j		G	C	T	A
	0	0	0	0	0
G	0	3	2	1	1
T	0	2	4	5	4
A	0	1	3	5	8
A	0	1	2	4	8

Take the calculated matrix from Smith-Waterman and set all cells that contribute to the optimal alignmnent to 0.

i \ j		G	C	T	A
	0	0	0	0	0
G	0	0	2	1	1
T	0	2	0	5	4
A	0	1	3	0	8
A	0	1	2	4	0

Main part - calculation

$i_start \leftarrow 1$

for $i \leftarrow i_start$ to n **do**

  for $j \leftarrow 1$ to m **do**

    if (i, j) belongs to optimal aligmnent **then**

      $S(i,j) = 0$

    else

$$S(i,j) \leftarrow max \begin{cases} 0 \\ S(i-1,j-1) + s(x[i],y[j]) \\ S(i-1,j) - s(x[i],--) \\ S(i,j-1) - s(--,y[j]) \end{cases}$$

    end if

  end for

end for

i = 1, j = 1: (1,1) belongs to optimal aligmnent, also $S(i-1,j-1)+s(x[i],y[j]) = 0$ bzw. dieser Weg darf nicht gegangen werden.

Und über den Rest bekommt man auch maximal 0, da:

$$S(1,1) \leftarrow max \begin{cases} 0 \\ S(0,1) - 1 = 0 - 1 = -1 \\ S(1,0) - 1 = 0 - 1 = -1 \end{cases}$$

i = 1, j = 2:

$$S(1,2) = \max \begin{cases} S(0,1) & + \quad 1 = 0 + 1 = 1 \\ S(0,2) & + -1 = 0 - 1 = -1 \\ S(1,1) & + -1 = 0 - 1 = -1 \\ 0 \end{cases} = 1$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	0	1	1	1
T	0	2	0	5	4
A	0	1	3	0	8
A	0	1	2	4	0

i = 1, j = 3:

$$S(1,3) = \max \begin{cases} S(0,2) & + \quad 1 = 0 + 1 = 1 \\ S(0,3) & + -1 = 0 - 1 = -1 \\ S(1,2) & + -1 = 1 - 1 = 0 \\ 0 \end{cases} = 1$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	0	1	1	1
T	0	2	0	5	4
A	0	1	3	0	8
A	0	1	2	4	0

i = 1, j = 4:

$$S(1,4) = \max \begin{cases} S(0,3) & + \quad 1 = 0 + 1 = 1 \\ S(0,4) & + -1 = 0 - 1 = -1 \\ S(1,3) & + -1 = 1 - 1 = 0 \\ 0 \end{cases} = 1$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	0	1	1	1
T	0	2	0	5	4
A	0	1	3	0	8
A	0	1	2	4	0

i = 2, j = 1:

$$S(2,1) = \max \begin{cases} S(1,0) & + \quad 1 = 0 + 1 = 1 \\ S(1,1) & + -1 = 0 - 1 = -1 \\ S(2,0) & + -1 = 0 - 1 = -1 \\ 0 \end{cases} = 1$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	0	1	1	1
T	0	1	0	5	4
A	0	1	3	0	8
A	0	1	2	4	0

i = 2, j = 2: (2,2) belongs to optimal aligmnent , also $S(i-1, j-1) + s(x[i], y[j]) = 0$ bzw. dieser Weg darf nicht gegangen werden.

Und über den Rest bekommt man auch maximal 0, da:

$$S(2,2) \leftarrow max \begin{cases} 0 \\ S(1,2) - 1 = 0 - 1 = -1 \\ S(2,1) - 1 = 0 - 1 = -1 \end{cases}$$

i = 2, j = 3:

$$S(2,3) = \max \begin{cases} S(1,2) & + \quad 3 = 1 + 3 = 4 \\ S(1,3) & + -1 = 1 - 1 = 0 \\ S(2,2) & + -1 = 0 - 1 = -1 \\ 0 \end{cases} = 4$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	0	1	1	1
T	0	1	0	4	4
A	0	1	3	0	8
A	0	1	2	4	0

i = 2, j = 4:

$$S(2,4) = \max \begin{cases} S(1,3) & + \quad 1 = 1+1 = 2 \\ S(1,4) & + -1 = 1-1 = 0 \\ S(2,3) & + -1 = 4-1 = 3 \\ 0 \end{cases} = 3$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	0	1	1	1
T	0	1	0	4	3
A	0	1	3	0	8
A	0	1	2	4	0

i = 3, j = 1:

$$S(3,1) = \max \begin{cases} S(2,0) & + \quad 1 = 0+1 = 1 \\ S(2,1) & + -1 = 1-1 = 0 \\ S(3,0) & + -1 = 0-1 = -1 \\ 0 \end{cases} = 1$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	0	1	1	1
T	0	1	0	4	3
A	0	1	3	0	8
A	0	1	2	4	0

i = 3, j = 2:

$$S(3,2) = \max \begin{cases} S(2,1) & + \quad 1 = 1+1 = 2 \\ S(2,2) & + -1 = 1-1 = -1 \\ S(3,1) & + -1 = 1-1 = 0 \\ 0 \end{cases} = 2$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	0	1	1	1
T	0	1	0	4	3
A	0	1	2	0	8
A	0	1	2	4	0

i = 3, j = 3: (3,3) belongs to optimal aligmnent $\Rightarrow S(3,3) = 0$

Aber über den Rest bekommt man 3, da:

$$S(3,3) \leftarrow max \begin{cases} 0 \\ S(2,3) - 1 = 4 - 1 = 3 \\ S(3,2) - 1 = 2 - 1 = 1 \end{cases}$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	0	1	1	1
T	0	1	0	4	3
A	0	1	2	3	8
A	0	1	2	4	0

i = 3, j = 4:

$$S(3,4) = \max \begin{cases} S(2,3) & + \quad 3 = 4+3 = 7 \\ S(2,4) & + -1 = 3-1 = 2 \\ S(3,3) & + -1 = 3-1 = 2 \\ 0 \end{cases} = 7$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	0	1	1	1
T	0	1	0	4	3
A	0	1	2	3	7
A	0	1	2	4	0

i = 4, j = 1:

$$S(4,1) = \max \begin{cases} S(3,0) & + \quad 1 = 0+1 = 1 \\ S(3,1) & + -1 = 1-1 = 0 \\ S(4,0) & + -1 = 0-1 = -1 \\ 0 \end{cases} = 1$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	0	1	1	1
T	0	1	0	4	3
A	0	1	2	3	7
A	0	1	2	4	0

i = 4, j = 2:

$$S(4,2) = \max \begin{cases} S(3,1) & + \quad 1 = 1 + 1 = 2 \\ S(3,2) & + -1 = 2 - 1 = 1 \\ S(4,1) & + -1 = 1 - 1 = 0 \\ 0 \end{cases} = 2$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	0	1	1	1
T	0	1	0	4	3
A	0	1	2	3	7
A	0	1	2	4	0

i = 4, j = 3:

$$S(4,3) = \max \begin{cases} S(3,2) & + \quad 2 + 1 = 3 \\ S(3,3) & + -1 = 3 - 1 = 2 \\ S(4,2) & + -1 = 2 - 1 = 1 \\ 0 \end{cases} = 3$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	0	1	1	1
T	0	1	0	4	3
A	0	1	2	3	7
A	0	1	2	3	0

i = 4, j = 4: (4,4) belongs to optimal aligmnent
Aber über den Rest bekommt man 6, da:

$$S(4,4) \leftarrow max \begin{cases} 0 \\ S(3,4) - 1 = 7 - 1 = 6 \\ S(4,3) - 1 = 3 - 1 = 2 \end{cases}$$

i \ j		G	C	T	A
	0	0	0	0	0
G	0	0	1	1	1
T	0	1	0	4	3
A	0	1	2	3	7
A	0	1	2	3	6

Result

i \ j		G	C	T	A
	0	0	0	0	0
G	0	0	1	1	1
T	0	1	0	4	3
A	0	1	2	3	7
A	0	1	2	3	6

Quellen: [4, 8, 6]

Comment 11.1 WARNUNG: man kann zwar bei eggert das erste Aligment nicht mehr nehmen, aber es könnte dort trtzdem noch ein Indel sein.

CHAPTER 12

Übersichten, Listen und Co.

12.1 | List of equations

Wichtig

- **Q-gramme**:
  - **Q-gram distance** (page 6, equation (1.2))

$$d_q(x,y) := \Sigma_{z \in \Sigma^q} |p_q(x)_z - p_q(y)_z|$$

  - **Q-gram Distanz als filter für Edit-Distanz** (page 10, equation (1.9))

$$\frac{d_q(x,y)}{2q} \leq d(x,y) = d_q(x,y) \leq d(x,y) \cdot 2q$$

- **Ranking**:
  - **Rising ranking function** (page 7, equation (1.4))

$$r(z) = \Sigma_{i=1}^{q} r_\Sigma(z[i]) \cdot \Sigma^{i-1}$$

  - **Falling ranking function** (page 7, equation (1.3))

$$r(z) = \Sigma_{i=1}^{q} r_\Sigma(z[i]) \cdot \sigma^{q-i}$$

  - **Rising unranking function** (page 8, equation (1.6))

$$\frac{r}{\sigma} = r', \quad \mathrm{R}\ c$$

  - **Falling unranking function** (page 8, equation (1.5))

$$\frac{r}{\sigma^{q-1}} = c, \quad \mathrm{R}\ r'$$

  - **Update function for rising ranking** (page 9, equation (1.8))

$$r(zb) = \lfloor \frac{r(au)}{\sigma} \rfloor + r_\Sigma(b) \cdot \sigma^{q-1}$$

- **Update function for falling ranking** (page 9, equation (1.7))

$$r(ub) = (r(au) \mod \sigma^{q-1}) \cdot \sigma + r_\Sigma(b)$$

- **Pairwise Sequence Alignment**:
  - **Total cost matrix for two sequences x and y of lengths m and n** (page 45, equation (6.1))

$$T(i,j) = \min \left\{ \text{cost}(A++B) \middle| \begin{array}{l} A \text{ is an alignment of the prefixes } x[1\dots i] \text{ and } y[1\dots j] \\ B \text{ is an alignment of the suffixes } x[i+1\dots m] \text{ and } y[j+1\dots n] \end{array} \right\}$$

  - **Additional cost matrix for two sequences x and y of lengths m and n** (page 46, equation (6.2))

$$C(i,j) = T(i,j) - d(x,y)$$

- **log-Odds Score Matrices**:
  - **Log-odds score matrix** (page 36, equation (4.3))

$$\text{score}_t(a,b) := \log\left(\frac{M_t(a,b)}{\pi_a \cdot \pi_b}\right)$$

  - **Frequency vector** (page 35, equation (4.2))

$$\pi = (\pi_a)_{a\in\Sigma}$$

- **RNA Secondary Structure Prediction**:
  - **Best simple structure score** (page 40, equation (5.1))

$$\text{score}(S^*) = \max\{\text{score}(S) \mid S \text{ is a simple structure on } s\}$$

  - **Best score calculation for the Nussinov Algorithmus** (page 42, equation (5.2))

$$M(i,j) = \max \begin{cases} M(i+1,j) & (\text{case 1: } S \to .S) \\ M(i,j-1) & (\text{case 2: } S \to S.) \\ M(i+1,j-1) + \mu(i,j) & (\text{case 3: } S \to (S)) \\ \max_{i+1<k<j}\{M(i,k-1) + M(k,j)\} & (\text{all cases 4: } S \to SS) \end{cases}$$

- **Suboptimal Local Alignments**
  - **Waterman-Eggert-Algorithm** (page 53, equation (7.1))

$$S(i,j) = max \begin{cases} 0 \\ S(i-1,j-1) + s(x[i],y[j]) \\ S(i-1,j) - s(x[i],--) \\ S(i,j-1) - s(--,y[j]) \end{cases}$$

- **Multiple Sequence Alignment / Divide-and-Conquer Alignment**:
  - **Multiple additional cost of a cut** $(c_1, c_2, \dots, c_k)$ (page **??**, equation (**??**))

$$C(c_1, c_2, \dots, c_k) := \sum_{1\le p<q\le k} C_{(p,q)}(c_p, c_q)$$

- **Segment Identification**:
  - **Weight of a diagonal** (page 71, equation (10.2))

$$w_D = -\ln(P_D)$$

- **DIALIGN**:
  - **Overlap weight of a diagonal** D (page 72, equation (10.3))

$$\mathrm{olw}(D) = w_D + \sum_{E \in \mathcal{D}} \tilde{w}(D, E)$$

Nicht wichtig zu lernen, da intuitiv

- Occurrence count

12.2 | Algorithms

Name	Aufgabe	Laufzeit	Speicherplatz
Ranking function	Aufgabe TODO	$O(q)$	-
Q-gram profile	Aufgabe	$O(\#q - grams + m + n)$ or $O(n)$ if m $\leq$ n and $q = \left\lfloor \frac{\log(m)\log(c)}{\log(\sigma)} \right\rfloor$	$O(\#q - grams)$
Sellers' algorithm	Approximate String Matching	?	$O(m)$ (no need to store back pointers and the entire alignment matrix D)
Hirschberg's Algorithm	optimales globales Alignment zweier Sequenzen; geringer Speicherbedarf	$O(mn)$	$O(min\{m, n\})$
Nussinov Algorithmus	RNA Secondary Structure Prediction	$O(n^3)$	$O(n^2)$
Waterman-Eggert-Algorithm	lokales Alignment; mehrere nicht überlappende lokale Alignments	$O(m^2n^2)$	? type
Waterman-Smith		$O(mn)$	

12.2.1 | Detailed time complexities

Aspect	Complexity	Because...
Initialise empty array of q-gram profile	$O(\#entries)$	1 step per entry (= q-gram)
Count q-grams in string x of length m (create q-gram profile of x)	$O(m)$	compute hash of first q-gram in $O(m)$, then use rolling hash to update in $O(1)$ for each remaining position
Initialize q-gram distance variable	$O(1)$	trivial
Summarize something (e.g. absolute differences for the occurrence counts)	$O(\#entries)$	Iteration through entries, 1 step per entry

12.3 | Alignment categories and types

Global vs local Bei global und lokal geht es um die Art von Algorithmus, die genutzt wird bzw. welche Bereiche von Sequenzen aligniert werden sollen. [3]

Weitere Beschreichsabgrenzungen (oder so...) wären noch:

- Semi-global alignment (/ free-end-gap alignment)

Pairwise vs Multiple Sequence Hierbei geht es um den Alignment-TYP, bzw. wie viele Sequenzen aligned werden sollen. [3]

Optimal vs. Suboptimal Hierbei geht es um die Qualität des alignments (in bezug auf das Scoring-Modell!)

free end-gap und so Dabwei geht es um die Gap-Kosten bzw. wie mit welchen Arten von Gaps umgegangen werden soll.

- Gap-free alignment
- Linear gap penalty
- Affine gap penalty
- Free end gaps / end-gap-free

Overlapping vs non-overlapping Das ist letztendlich eine weitere Beschränkung unter dem größeren Schirm davon Alternativen/Mehrere Alignments zu finden.

In ähnlicher Motivation könnte man nooch listen:

- k-best alignments
- multiple optimal alignments

Backmatter

Definitionen

c-approximation An algorithm for a cost minimization problem for $c \geq 1$ where it is guaranteed that its solution has cost at most c times the optimal solution. For a maximization problem, an algorithm is called a c-approximation if it is guaranteed that its solution has score at least $1/c$ times the optimal solution.. 64

q-gram . 3, 5–10, 15, 116

s-basepair a set of two different numbers $\{i, j\} \subset \{1, \ldots, n\}$ such that $\{s[i], s[j]\} \in \{\{\mathtt{A}, \mathtt{U}\}, \{\mathtt{C}, \mathtt{G}\}, \{\mathtt{G}, \mathtt{U}\}\}$. Basepairs must bridge at least $\delta \geq 0$ positions (e.g. $\delta = 3$); to account for this, we have the additional constraint $|i - j| > \delta$ for a basepair $\{i, j\}$.. 39

additional cost matrix $C(i, j) = T(i, j) - d(x, y)$ where "++" denotes concatenation of alignments.. 46, 114

alignment distance The alignment distance of two sequences $x, y \in \sum^*$ is defined as $d(x, y) := min\{cost(A) : A \in \mathcal{A}*$ is an alignment of x and $y\}$. 5

alphabet encoding ranking function . 6–9

approximate string matching problem todo. 23

background frequencies Die Einträge des frequency vector π; gewinnbar als Randverteilungen (Marginals) von M_t: $\pi_a = \sum_{b \in \Sigma} M_t(a, b)$ und $\pi_b = \sum_{a \in \Sigma} M_t(a, b)$ (wegen Symmetrie). 35

bidirected de Bruijn subgraph Erweitert den klassischen de-Bruijn-Graphen, indem jedes k-Mer mit seinem Reverse-Komplement identifiziert wird. Beide Sequenzen werden somit durch denselben Knoten repräsentiert. Diese Darstellung berücksichtigt, dass bei DNA-Sequenzierungen häufig nicht bekannt ist, von welchem Strang der DNA-Doppelhelix ein Read stammt. Die übrige Struktur des de-Bruijn-Graphen, insbesondere die Definition der Kanten, bleibt unverändert.. 19

bidirectional indices Textindizes, die einen partiellen Match flexibel *sowohl nach links als auch nach rechts* verlängern können, wie es Search Schemes erfordern. Beispiele: Affix-Bäume/-Arrays, bidirektionale BWT, bidirektionaler Wavelet-Index, br-index, bidirektionaler Move-Index. 32

colored de Bruijn subgraph Erweitert den de-Bruijn-Graphen um Informationen über die Herkunft der k-Mere. Hierzu werden die Eingabesequenzen in mehrere Gruppen, beispielsweise verschiedene Genome oder Datensätze, unterteilt. Jeder Gruppe wird eine Farbe zugewiesen. Ein Knoten erhält die Menge aller Farben der Gruppen, in denen das entsprechende k-Mer vorkommt. Dadurch lässt sich nachvollziehen, in welchen Sequenzmengen ein k-Mer enthalten ist. Colored de-Bruijn-Graphen eignen sich insbesondere für den Vergleich mehrerer Genome, erfordern jedoch aufgrund der Speicherung der Farbinformationen deutlich mehr Speicher als klassische de-Bruijn-Graphen.. 20, 21

compacted de Bruijn subgraph Entsteht durch das Zusammenfassen nicht verzweigender Pfade eines de-Bruijn-Graphen. Besitzt eine Folge aufeinanderfolgender Knoten jeweils genau einen Nachfolger und einen Vorgänger

(mit Ausnahme der Endknoten), so wird diese zu einem einzelnen Knoten, einem sogenannten *Unitig*, zusammengefasst. Dieser repräsentiert ein längeres Teilstück der ursprünglichen Sequenz. Durch diese Kompaktierung werden Speicherbedarf und häufig auch die Laufzeit nachfolgender Algorithmen reduziert, ohne dass die wesentliche Struktur des Graphen verloren geht.. 20

context-free grammar A quadruple $G = (N, T, P, S)$, where

- N is a finite set of *non-terminal symbols*,
- T is a finite set of *terminal symbols*, $N \cap T = \emptyset$,
- P is a finite subset of $N \times (N \cup T)^*$, the *productions*, which are rules for transforming a non-terminal symbol into a (possibly empty) sequence of terminal and non-terminal symbols, and
- $S \in N$ is the *start symbol*.

. 41

de Bruijn graph The graph that contains one vertex for each k-mer (and therefore σ^k vertices) and a directed edge for each overlap of length k − 1 (and therefore σ^{k+1} edges). 18

de Bruijn subgraph The de Bruijn subgraph $DBG(s, k) = (V, E)$ for a given sequence s of dimension k (positive integer $\leq |s|$) is a directed multigraph whose vertices V correspond to the distinct k-mers of $s, s_{k,1}, \cdots, s_{k,|s|-k+1}$, and whose edges are derived from the (k + 1)-mers of s as follows: Let a (k + 1)-mer of s be denoted as $s[i \ldots i + k] = aub$, where a and b are symbols and $|u| = k - 1$, then a directed edge is contained in E, linking vertex au to vertex ub.. iv, 15, 16, 18–21

decision problem A "yes/no question" with just two problem solutions, $S = \{\mathtt{T}, \mathtt{F}\}$ (*true/false*).. 61, 62

dense . 6

dimension of a de Bruijn subgraph The positive integer $\leq |s|$ in a $DBG(s, k) = (V, E)$. 15

edit distance ... 5, 10, 12, 13

eulerian path . 16

fixed evolutionary time period Zeitraum t, über den die Ersetzung einzelner Aminosäuren verfolgt wird. t ist so zu wählen, dass die Score-Matrix zum Problem passt: Für Sequenzen mit gemeinsamem Vorfahren vor $\approx t/2$ sollte eine aus Paaren mit Divergenz t konstruierte Matrix genutzt werden. 35

frequency vector Vektor $\pi = (\pi_a)_{a \in \Sigma}$ der natürlich vorkommenden Aminosäuren mit $\pi_a \geq 0$ für alle $a \in \Sigma$ und $\sum_{a \in \Sigma} \pi_a = 1$. Für zwei zufällige Positionen ist die Wahrscheinlichkeit des geordneten Paares (a, b) gleich $\pi_a \cdot \pi_b$. 35, 114

general similarity score Allgemeine Ähnlichkeitsfunktion, bei der Matches positiv, Mismatches und Indels negativ bewertet werden – im Gegensatz zu einem Distanzmodell, in dem ein Match stets Kosten 0 hat. Nützlich insbesondere fürs lokale Alignment, wo Distanzfunktionen ungeeignet sind. 34

Hirschberg's Algorithm . 116

k-approximate matches $i^*(C) := \max\{\, i \mid C(i) \leq k \,\}$. 23

last essential index todo. 27

left-to-right partition $Plr(x, y) = (z_0, \langle b_1 \rangle, z_1, \langle b_2 \rangle, ..., z_{\ell 1}, \langle b_\ell \rangle, z_\ell)$ of x with respect to y is the partition defined by the condition that for all $i \in [1, \ell], z_{i1} b_i$ is not a substring of y.. 11, 12

likelihood ratio Verhältnis $M_t(a,b)/(\pi_a \cdot \pi_b)$, das die Wahrscheinlichkeit des Paares (a,b) im Evolutionsmodell zu der im Zufallsmodell in Beziehung setzt. Werte > 1 bedeuten Ähnlichkeit, Werte < 1 Unähnlichkeit von a und b. 36

log-odds score matrix Additive Score-Matrix bzgl. Zeitraum t, definiert als Logarithmus der likelihood ratio: $\text{score}_t(a,b) := \log\left(\frac{M_t(a,b)}{\pi_a \cdot \pi_b}\right)$ (Wichtige Familien: PAM(t) und BLOSUM(s)). 36, 114

maximal matches distance $\delta(x||y) := min\{|P| : P \text{ is a partition of } x \text{ with respect to } y\}$. 11–13

MinU search scheme Search Scheme (Renders et al., 2024), dessen Fehlerbereiche $(L[i], U[i])$ langsamer wachsen als beim pigeonhole-Schema und das daher effizienter ist. Es ist *lossless*, d. h. kein Match mit bis zu k Fehlern wird verpasst (alle Fehlerkonfigurationen $(e_1, \ldots, e_p)$ mit $\sum e_i \leq k$ sind abgedeckt). 31

multiple additional cost The additional cost matrix $C_{(p,q)}$ for sequence pair (s_p, s_q), $p < q$: $C(c_1, c_2, \ldots, c_k) := \sum_{1 \leq p < q \leq k} C_{(p,q)}(c_p, c_q)$. 68, 114

non-overlapping suboptimal local alignment An alignment that has no match or mismatch in common with any higher-scoring (sub)optimal local alignment. (Two alignments A and A' have a match or mismatch in common (they overlap) if there is at least one pair of positions (i,j) such that $x[i]$ is aligned with $y[j]$ both in A and in A'.). 51

NP-complete . 61–63

NP-hard . 62

occurrence count Let $x \in \Sigma^m$ and choose $q \in [1,m]$. The occurrence count of a q-gram $z \in \Sigma^q$ in x is the number $Occ(x,z) := |i \in [1, m-q+1] : x[i...i+q-1] = z|$. 6

optimal cut A family of cut positions $(c_1, c_2, \ldots, c_k)$ of the sequences $s_1, s_2, \ldots, s_k$ of lengths $n_1, n_2, \ldots, n_k$ for which exists an alignment A of the prefixes $(s_1[1 \ldots c_1], s_2[1 \ldots c_2], \ldots, s_k[1 \ldots c_k])$ and an alignment B of the suffixes $(s_1[c_1+1 \ldots n_1], s_2[c_2+1 \ldots n_2], \ldots, s_k[c_k+1 \ldots n_k])$ such that the concatenation $A{+}{+}B$ is an optimal alignment of $s_1, s_2, \ldots, s_k$.. 67

optimization problem Addresses the minimization of a cost or maximization of a score function. 61, 64

overlap weight Weight with the set of all diagonals $\mathcal{D}$ and where $\tilde{w}(D_1, D_2) := w_{D_3}$ is the weight of the (implied) overlap D_3 of the two diagonals D_1 and D_2: $\text{olw}(D) = w_D + \sum_{E \in \mathcal{D}} \tilde{w}(D,E)$. 72, 115

p-value of the diagonal $P_D(l_D, s_D) = \sum_{i=s_D}^{l_D} \binom{l_D}{i} p^i (1-p)^{l_D - i}$. 70

pigeonhole principle search scheme Zerlegt das Muster x bei Fehlerschwelle k in $p = k+1$ Teile $x_1, \ldots, x_{k+1}$. Da bei $\leq k$ Fehlern mindestens ein Teil exakt vorkommen muss, besteht das Schema aus $k+1$ Suchen S_i: Teil x_i exakt matchen, dann Backtracking nach links und rechts bis insgesamt k Fehler erreicht sind. 31

problem Binary relation on a set I of problem instances and a set S of problem solutions. 61

q-gram distance The q-gram distance of $x \in \Sigma^m$ and $y \in \Sigma^n$ is defined for $1 \leq q \leq min\{m,n\}$ as: $d_q(x,y) := \sum_{z \in \Sigma^q} |p_q(x)_z - p_q(y)_z|$. 5, 6, 8, 10, 12, 13, 15, 113

q-gram Distanz als filter für Edit-Distanz . 10, 113

q-gram profile Let $\sigma = |\sum|$. The q-gram profile of x is a σ^q-element vector $p_q(x) := (p_q(x)_z)_{z \in \sum^q}$, indexed by all q-grams, defined by $p_q(x)_z := Occ(x,z)$. 6, 8, 15, 116

ranking algorithm An algorithm that implements a ranking function. 6

ranking function A bijective function $r : \mathcal{X} \to [0, |\mathcal{X}| - 1]$ for a finite set $\mathcal{X}$. v, vi, 6, 7, 9, 10, 116

right-to-left partition $Prl(x, y) = (z_0, \langle b_1 \rangle, z_1, ..., \langle b_{\ell 1} \rangle, z_{\ell 1}, \langle b_\ell \rangle, z_\ell)$ of x with respect to y is the partition defined by the condition that for all $i \in [1, \ell], b_i z_i$ is not a substring of y.. 12

search problem Addresses the minimization of a cost or maximization of a score function. 61

sensible score function Funktion score : $\mathcal{A}(\Sigma) \to \mathbb{R}$ mit $\text{score}\left(\binom{a}{a}\right) \geq 0 \geq \text{score}\left(\binom{a}{-}\right) = \text{score}\left(\binom{-}{a}\right)$, $\text{score}\left(\binom{a}{a}\right) \geq \text{score}\left(\binom{a}{c}\right) = \text{score}\left(\binom{c}{a}\right)$ und $\text{score}\left(\binom{a}{c}\right) \geq \text{score}\left(\binom{a}{-}\right) + \text{score}\left(\binom{-}{c}\right)$ (Symmetrie; Selbst-Ähnlichkeit maximal; Substitution besser als Deletion+Insertion). 34

simple structure on s A set $S = \{b_1, b_2, \ldots, b_N\}$ of s-basepairs with $N \geq 0$ and the following properties:

- Either $N \leq 1$, or
- if $N \geq 2$ and $\{i, j\} \in S$ and $\{k, l\} \in S$ are two different basepairs such that $i < j$ and $k < l$, then $i < k < l < j$ or $k < i < j < l$ (nested basepairs) or $i < j < k < l$ or $k < l < i < j$ (separated basepairs).

. 39

third-bits Score-Einheit: Ein Score ist in *bits*, wenn er per $\log_2(\cdot)$ gebildet wird; wird er zusätzlich mit dem Faktor 3 multipliziert (und gerundet), erhält man Scores in third-bits. 36

total cost matrix $T(i, j) = \min \left\{ \text{cost}(A + +B) \middle| \begin{array}{l} A \text{ is an alignment of the prefixes } x[1 \ldots i] \text{ and } y[1 \ldots j] \\ B \text{ is an alignment of the suffixes } x[i+1 \ldots m] \text{ and } y[j+1 \ldots n] \end{array} \right\}$ where "++" denotes concatenation of alignments.. 45, 114

transversion/transition cost Biologisch motivierte Kostenfunktion auf dem DNA-Alphabet: Purin/Purin- und Pyrimidin/Pyrimidin-Ersetzungen (Transitionen) kosten 1, Purin/Pyrimidin-Ersetzungen (Transversionen) kosten 2; oft mit Indel-Kosten 3. Reflektiert, dass Transitionen wahrscheinlicher sind als Transversionen. 33

unranking algorithm An algorithm that implements a unranking function. 6

unranking function The inverse of a ranking function. vi, 6

update function for falling ranking . 9, 114

update function for rising ranking . 9, 113

weight of a diagonal $w_D = -\ln(P_D)$ (Wobei P_D der p-value of the diagonal ist). 71, 115

Index

BLAST, 13

Bibliography

[1] Franzili. *Sellers.kt – Implementation of the Sellers Algorithm*. https://github.com/Franzili/Sellers-Algorithm/blob/main/src/main/kotlin/Sellers.kt. GitHub repository, Zugriff am 31. Juli 2026. 2026.

[2] *Sequence Analysis 2*. Lecture notes Summer 2026.

[3] Sanju Tamang. *Sequence Alignment – Definition, Types, Methods, Uses*. https://microbenotes.com/local-global-multiple-sequence-alignment/. Microbe Notes, online. May 2024.

[4] University of Dundee. *Smith-Waterman Algorithm*. https://www.compbio.dundee.ac.uk/papers/water/node5.html. Online; Zugriff am 31. Juli 2026.

[5] University of Freiburg. *Nussinov Algorithm Teaching Tool*. https://rna.informatik.uni-freiburg.de/Teaching/index.jsp?toolName=Nussinov. Online; Zugriff am 31. Juli 2026.

[6] University of Freiburg. *Smith-Waterman Teaching Tool*. https://rna.informatik.uni-freiburg.de/Teaching/index.jsp?toolName=Smith-Waterman. Online; Zugriff am 31. Juli 2026.

[7] Wikipedia contributors. *Approximate string matching — Wikipedia, The Free Encyclopedia*. [Online; accessed 31-July-2026]. 2026. URL: https://en.wikipedia.org/w/index.php?title=Approximate_string_matching&oldid=1351934189.

[8] Wikipedia contributors. *Needleman–Wunsch-Algorithmus*. https://w.wiki/ZAL. Online; Zugriff am 31. Juli 2026.