Sequenzanalyse 2

by Jens Stoye

Summer semester 2026

Lecture notes by Liliana Sanfilippo

April 2026

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 . . . 24

3.3 Search Schemes and Bidirectional Indices . . . 24

4 Biologically Inspired Alignment Scores **25**

5 RNA Secondary Structure Prediction **27**

5.1 Introduction . . . 27

5.2 The Optimization Problem . . . 27

5.3 Context-Free Grammars . . . 27

5.4 The Nussinov Algorithm . . . 27

6 Pairwise Sequence Alignment in Linear Space **29**

7 Suboptimal Local Alignments **31**

8 Exact Algorithms for Sum-of-Pairs Multiple Sequence Alignment **33**

9 An Approximation Algorithm for Sum-of-Pairs Multiple Sequence Alignment **35**

10 Heuristics for Multiple Sequence Alignment **37**

11 Examples and longer explanations **39**
11.1 Nussinov example . . . 39
11.2 Smith-Waterman Algorithm . . . 46
11.2.1 Smith-Waterman example . . . 47
11.3 Waterman-Eggert Algorithm . . . 49
11.3.1 Waterman-Eggert Example . . . 50
11.4 Links . . . 53

Backmatter **i**
Definitionen . . . i
11.5 List of equations . . . iv
11.6 Algorithms . . . v
11.6.1 Detailed time complexities . . . v
Index . . . v

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:

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

and for the rising ranking function:

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

Or without using j an f:

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

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

$$\frac{d_q(x, y)}{2q} \leq d(x, y).$$

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.

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 $0 \leq i \leq l$) and each b_i is a single character (for $1 \leq i \leq l$). 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.

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.

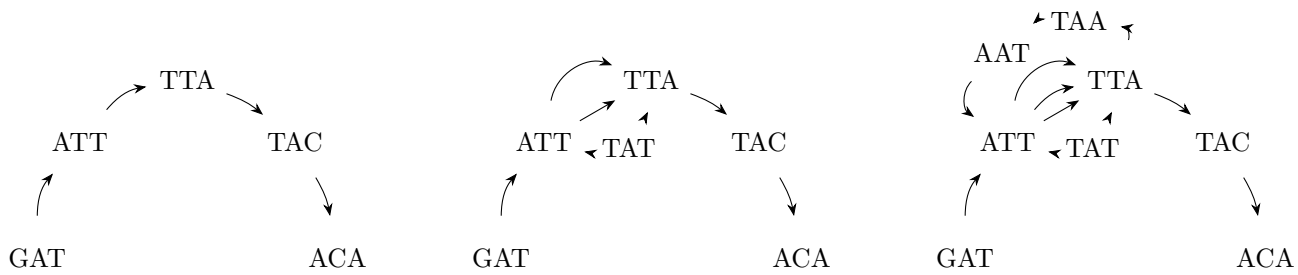
For $k = 0$, the problem is trivial. Indeed, if a sequence t that is distinct from s corresponds to a permutation of the symbols of s, then s and t clearly share the same 1-mer profile. For $k \geq 1$, an elegant answer can be found with the help of the de Bruijn subgraph $DBG(s, k)$.

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.

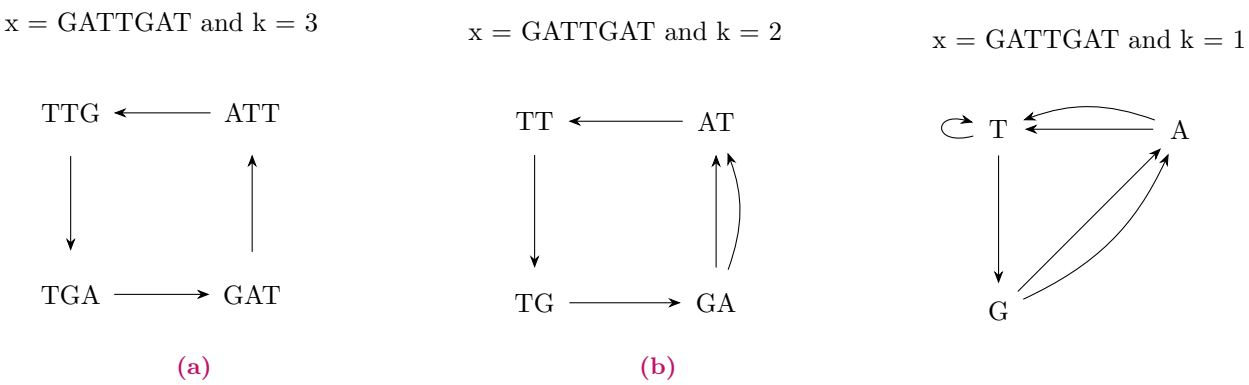


Figure 2.2: De Bruijn subgraphs for the same sequence but with k varying from 3 to 1.

Example 2.2

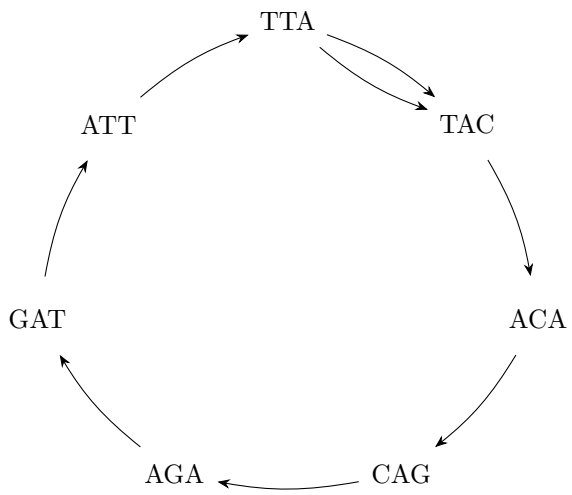


Figure 2.3: It could also be on big circle.

Example 2.3 Additional example from lecture

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

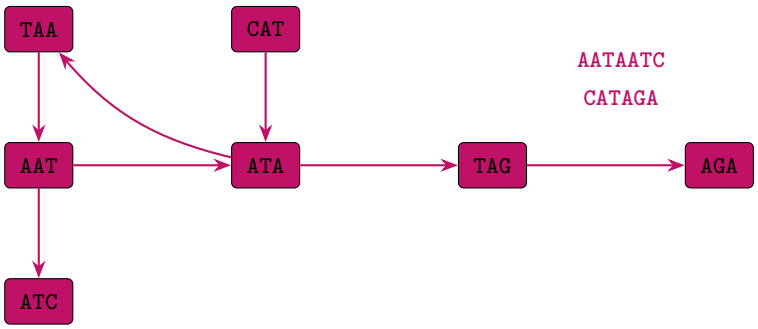
kein Kreis ist, und bei 2.1b muss es dann mehrere geben, weil da eine Schleife drin ist, aber NEIN der Pfad ist die Reihenfolge der Knoten und nicht der Kanten, also gibt es nur einen. Man kann nicht unterscheiden, ob man zuerst die obere oder untere Kante der Schleife zuerst gegangen ist. Gibt also 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).

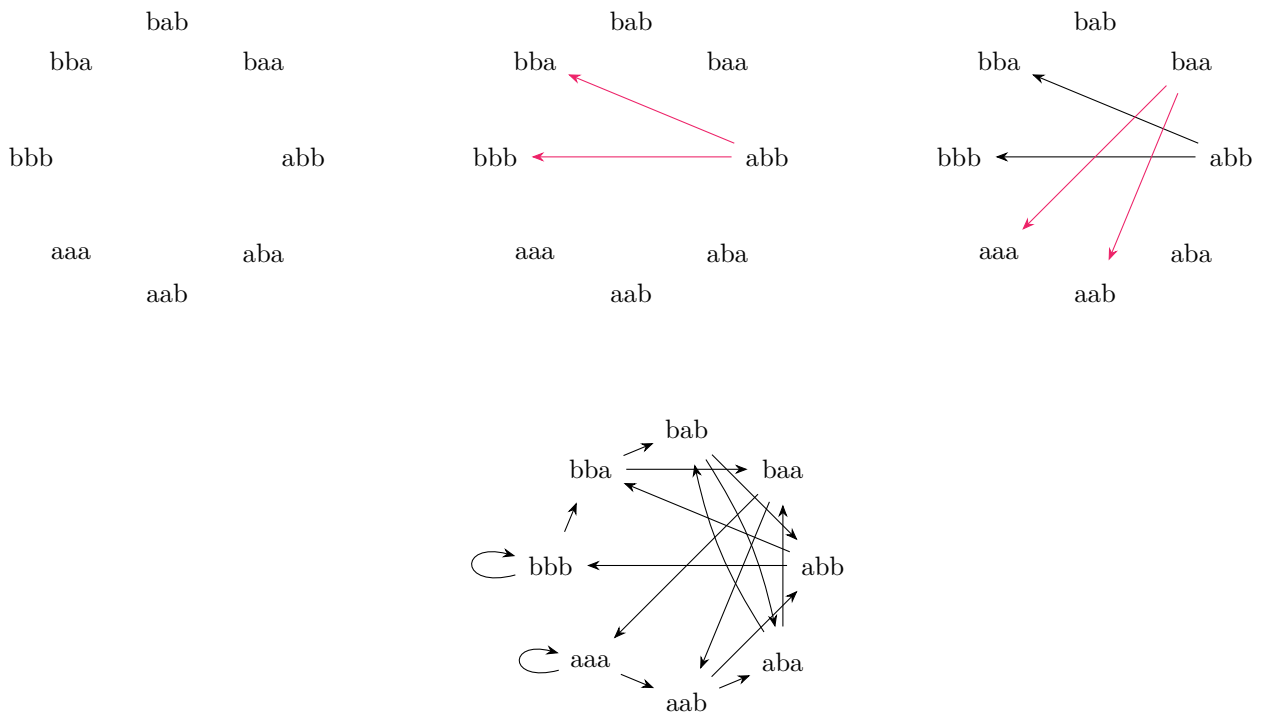
2.3 | Variants of de Bruijn Graphs

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



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.

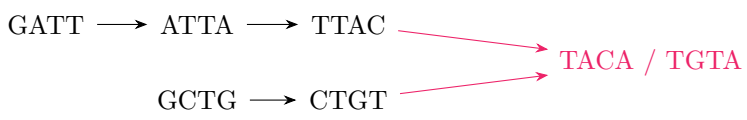
Problem 2.1 In most bioinformatics applications where DNA sequence reads come from a sequencing machine, it is not clear from which strand of the DNA double helix a read originates.

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

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

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



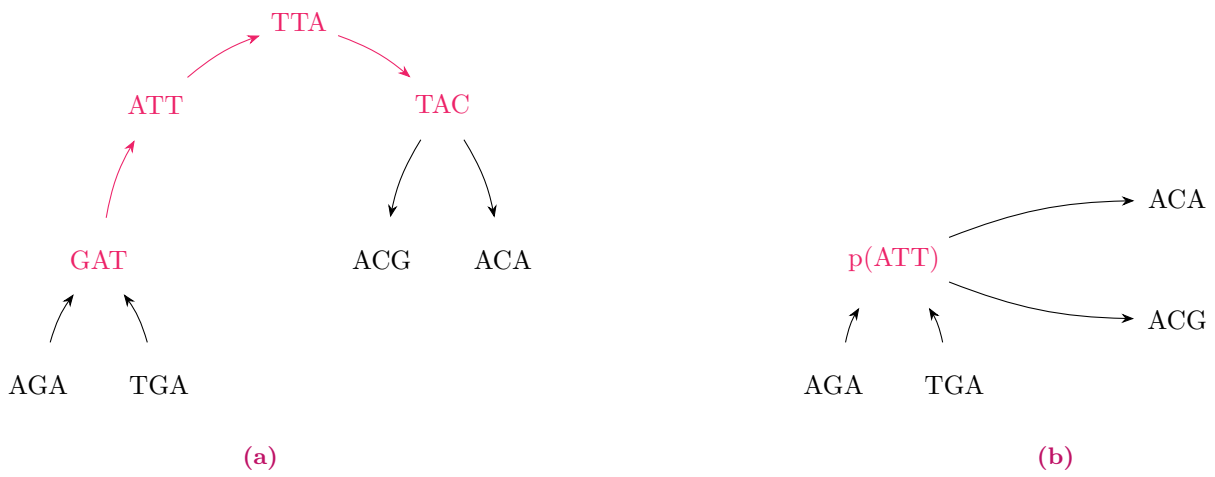
Definition 2.4 Bidirected de Bruijn subgraph Erweitert den klassischen de-Bruijn-Graphen, indem jedes k-Mer mit seinem Reverse-Komplement identifiziert wird. Beide Sequenzen werden somit durch denselben Knoten repräsentiert. Diese Darstellung berücksichtigt, dass bei DNA-Sequenzierungen häufig nicht 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..

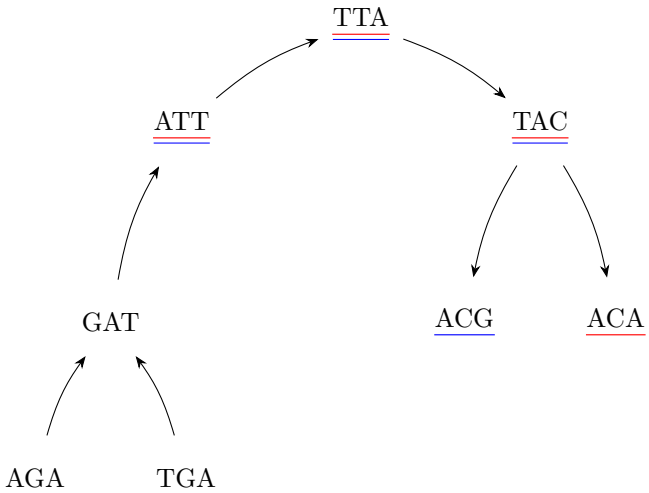
Example 2.7 for compacted de Bruijn subgraph



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



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 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.

CHAPTER 3

Approximative String Matching

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:

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.

Definition 3.2 k-approximate matches of x in y

3.1 | Sellers' Algorithm

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 ??, 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 $j = 1, \ldots, n$.

---

todo

Some remarks about Sellers' algorithm:

Since we only report end positions and costs, there is no need to store back pointers and the entire alignment matrix D in Algorithm ??. 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.

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 $([j, j+1], 1)$.

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.

3.2 | Ukkonen's Cutoff Algorithm

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 $_j$ 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.

Definition 3.3 last essential index todo

The last essential index $_j$ of column C_j is $_j := (C_j) = \max\{i | C_j(i) \leq k\}$. 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 $_0$ 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 $i >_j - 1$ and costs are non-negative, these cells provide candidate values $> k for C_j(i)$ for $i >_{j-1} + 1$ 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 $i >_{j-1} + 1$, we know that the last essential index $_j$ of C_j occurs before that i. Algorithm ?? shows the details.

todo

Example 3.1 TODO S 19

3.3 | Search Schemes and Bidirectional Indices

CHAPTER 4

Biologically Inspired Alignment Scores

CHAPTER 5

RNA Secondary Structure Prediction

5.1 | Introduction

TODO

5.2 | The Optimization Problem

TODO

5.3 | Context-Free Grammars

TODO

5.4 | The Nussinov Algorithm

Example calculation for GGUCCA on page 39.

CHAPTER 6

Pairwise Sequence Alignment in Linear Space

CHAPTER 7

Suboptimal Local Alignments

CHAPTER 8

Exact Algorithms for Sum-of-Pairs Multiple Sequence Alignment

CHAPTER 9

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

CHAPTER 10

Heuristics for Multiple Sequence Alignment

CHAPTER 11

Examples and longer explanations

11.1 | 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

todo what does it

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: $\mathbf{i = 2}$ (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: $\mathbf{i = 3}$ (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: $\mathbf{i = 4}$ (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: $\mathbf{i = 5}$ (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)

Warning!

Following the instruction in the script, this would be the result:

$M =$

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	
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		unpaired	unpaired	init		
4				unpaired	init	
5					unpaired	init
6						

(b) $T(2,3) \leftarrow$ "unpaired"

Figure 11.13: $\mathbf{d = 1}$ und $\mathbf{i = 2}$ (second iteration)

But this is (apparently) wrong.

$M =$

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	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		case 3	unpaired	init		
4			unpaired	unpaired	init	
5					unpaired	init
6						

(b) $T(3,4) \leftarrow$ "unpaired"

Figure 11.14: $\mathbf{d = 1}$ 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		1	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		case 3	unpaired	init		
4			unpaired	unpaired	init	
5				unpaired	unpaired	init
6						

(b) $T(4,5) \leftarrow$ "unpaired"

Figure 11.15: $\mathbf{d = 1}$ 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		1	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		case 3	unpaired	init		
4			unpaired	unpaired	init	
5				unpaired	unpaired	init
6					unpaired	

(b) $T(5,6) \leftarrow$ "unpaired"

Figure 11.16: $\mathbf{d = 1}$ 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

$$M(1,3) = max \begin{cases} M(2,3) & = 0 \\ M(1,2) & = 0 \\ M(2,2) + \mu(2,4) = M(2,2) + score(\{G,U\}) = 0 + 1 & = 1 \\ max_{2<k<3}\{M(1,???) + M(???,3)\} & = NaN \end{cases} = 1$$

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	0	0				
2	0	0	0			
3	1	1	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	case 3	case 3	unpaired	init		
4			unpaired	unpaired	init	
5				unpaired	unpaired	init
6					unpaired	

(b) $T(1,3) \leftarrow$ case

Figure 11.17: $\mathbf{d = 2}$ und $\mathbf{i = 1}$ (first iteration)

We are loking at $M(2,4)$

Is $j - i \leq \delta$?

→ No $4 - 2 = 2 > 1$

Is $\{s[2], s[4]\} = \{G, C\}$ a valid base pair?

→ 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	1	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	case 3	unpaired	init		
4		case 3	unpaired	unpaired	init	
5				unpaired	unpaired	init
6					unpaired	

(b) $T(2,4) \leftarrow$ case

Figure 11.18: $\mathbf{d = 2}$ 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	1	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	case 3	unpaired	init		
4		case 3	unpaired	unpaired	init	
5			case 1-3	unpaired	unpaired	init
6					unpaired	

(b) $T(3,5) \leftarrow$ case

Figure 11.19: $\mathbf{d = 2}$ 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	1	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	case 3	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: $\mathbf{d = 2}$ und $\mathbf{i = 4}$ (fourth iteration)

$M(5,7)$ **does not exist**

Iteration 2 and 3

were taken from `https://rna.informatik.uni-freiburg.de/Teaching/index.jsp?toolName=Nussinov` and checked with `https://iba-amrt.vlabs.ac.in/exp/rna-nussinov/simulation/index.html`. 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	1	0	0		
4	2	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	case 3	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 11.21: $\mathbf{d = 2..3}$ 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

$$M(1,6) = max \begin{cases} M(2,6) & = 1 \\ M(1,5) & = 2 \\ M(2,5) + \mu(1,6) = M(2,5) + score(\{G,A\}) = 1 + 0 & = 0 \\ max_{2<k<6}\{M(1,k-1) + M(k,6)\} = * & = 2 \end{cases} = 2$$

$$\begin{aligned} *max_{2<k<6}\{M(1,k-1) + M(k,6)\} &= max\{M(1,3-1) + M(3,6), M(1,4-1) + M(4,6), M(1,5-1) + M(5,6)\} \\ &= max\{M(1,2) + M(3,6), M(1,3) + M(4,6), M(1,4) + M(5,6)\} \\ &= max\{0 + 1, 1 + 0, 2 + 0\} \\ &= max\{1, 1, 2\} \\ &= 2 \quad mit\ k = 5 \end{aligned}$$

$M =$

j / i	G_1	G_2	U_3	C_4	C_5	A_6
1	0	0				
2	0	0	0			
3	1	1	0	0		
4	2	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	case 3	unpaired	init		
4	–	case 3	unpaired	unpaired	init	
5	–	–	case 1-3	unpaired	unpaired	init
6	case 3/ 4 (k=5)	–	–	case 1-3	unpaired	

(b) $T(4,6) \leftarrow$ case

Figure 11.22: $\mathbf{d = 5}$ und $\mathbf{i = 1}$ (first iteration)

11.2 | Smith-Waterman Algorithm

Algorithm 1 Smith-Waterman Darstellung nach Skript

Sequences to be aligned: $A = a_1a_2...a_n$ and $B = b_1b_2...b_m$
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 2 Smith-Waterman Darstellung angepasst

Sequences to be aligned: $A = a_1a_2...a_n$ and $B = b_1b_2...b_m$
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

11.2.1 | 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**

$$S(i, j) \leftarrow \max \begin{cases} S(i-1, j-1) & +s(a_i, b_j) \\ S(i-1, j) & +s(a_i, -) \\ S(i, j-1) & +s(-, b_j) \\ 0 \end{cases} = \max \begin{cases} S(i-1, j-1) & + \quad 3 \quad a_i = b_j \\ S(i-1, j-1) & + \quad 1 \quad a_i \neq b_j \\ S(i-1, j) & + -1 \quad b_j = - \\ S(i, j-1) & + -1 \quad a_i = - \\ 0 \end{cases}$$

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

11.3 | Waterman-Eggert Algorithm

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

Algorithm 3 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 alignmnent **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

11.3.1 | 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 alignmnent **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

11.4 | Links

- https://www.compbio.dundee.ac.uk/papers/water/node5.html
- https://w.wiki/ZAL
- https://rna.informatik.uni-freiburg.de/Teaching/index.jsp?toolName=Smith-Waterman

WARNUNG: man kann zwar bei eggert das erste Aligment nicht mehr nehmen, aber es könnte dort trtzdem noch ein Indel sein.

Backmatter

Definitionen

q-gram . v, 3, 5–10, 15

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

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

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

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.. iii, 15, 16, 18–21

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

k-approximate matches . 23

last essential index todo. 24

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

maximal matches distance $\delta(x||y) := min\{|P| : P$ is a partition of x with respect to $y\}$. 11–13

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

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: $d_q(x, y) := \sum_{z \in \Sigma^q} |p_q(x)_z - p_q(y)_z|$. iv, 5, 6, 8–10, 12, 13, 15

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 $p_q(x)_z := Occ(x, z)$. v, 6, 8, 15

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 $\mathcal{X}$. iv, v, 6, 7, 9, 10

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.. 11, 12

unranking algorithm An algorithm that implements a unranking function. 6

unranking function The inverse of a ranking function. iv, 6

11.5 | List of equations

Wichtig

- **Q-gram distance** (page 6, equation (1.2))

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

- **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))

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

- **Falling unranking function** (page 8, equation (1.5))

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

Nicht wichtig zu lernen, da intuitiv

- Occurrence count

11.6 | Algorithms

Name	Aufgabe	Laufzeit	Speicherplatz
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 $q = \left\lfloor \frac{\log(m)\log(c)}{\log(\sigma)} \right\rfloor$	$O(\#q - grams)$

11.6.1 | 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

Index

BLAST, 13