

Bioinformatik

Dynamische Programmierung



Ulf Leser
Wissensmanagement in der
Bioinformatik



Motivation

- BLAST / FASTA und Verwandte sind *die* Bioinformatik Anwendung
 - Teilweise synonym für „Bioinformatik“
- Grundlegende Gesetzmäßigkeit der Bioinformatik

Hohe Sequenzähnlichkeit bei DNA, RNA und Proteinen heißt in der Regel ähnliche Funktion bzw. Struktur

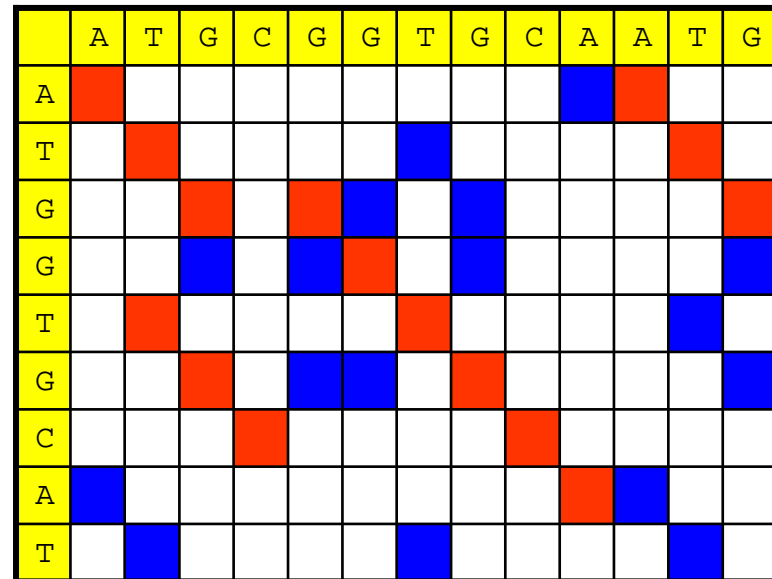
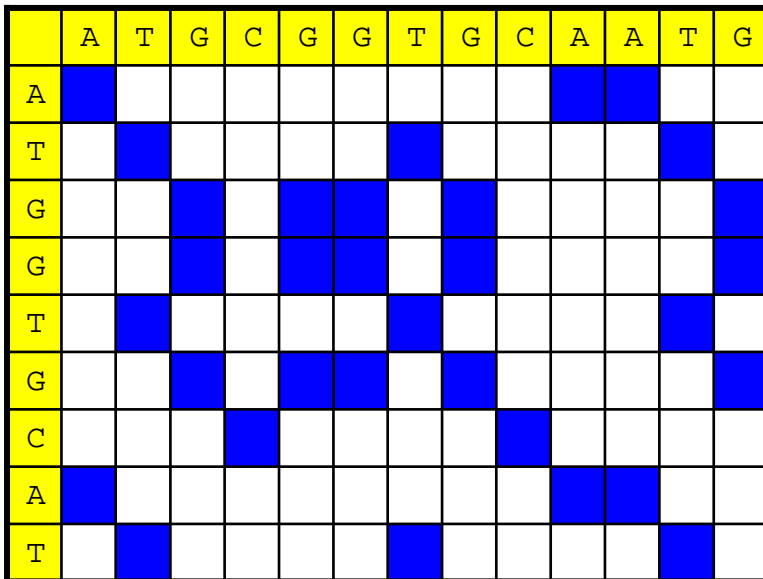
Comparative Genomics

- Bestimmung von Protein/Genfunktion in anderen Spezies wesentlich leichter als beim Menschen
 - Bakterien, Knock-out Mäuse, etc.
- Viele Gene sind hochgradig konserviert
 - Maus – Mensch: 97% Sequenzidentität
 - „Housekeeping Genes“ in allen Organismen ähnlich vorhanden
 - Die 4% „aktivsten“ (am besten verbundenen) Proteine sind in allen (bisher sequenzierten) bekannten Organismen vorhanden
- Vorwärts
 - Finden und sequenzieren eines neues Genes beim Menschen
 - Suchen nach ähnlichen Sequenzen in anderen Organismen
 - [BLAST gegen Genbank / EMBL](#)
- Rückwärts
 - Bestimmung der Funktion eines Genes einer anderen Spezies
 - Suche nach ähnlichen Sequenzen beim Menschen
 - [BLAST gegen Genbank / EMBL](#)



Dotplot und gleiche Teilstrings

- Wie erkennt man gleiche Teilstrings im Dotplot?



- Diagonalen von links-oben nach rechts-unten
 - Größter gemeinsamer Teilstring – längste Diagonale
 - Visuell bei kurzen Strings möglich

Abstandsmaße

- Approximatives Stringmatching sucht Ähnlichkeiten
 - Welcher Substring von T ist am ähnlichsten zu P?
 - Welcher String T_1, \dots, T_n ist am ähnlichsten zu T?
- Voraussetzung dafür
 - Was heißt ähnlich?
 - Was heißt „am ähnlichsten“?
- Quantifizierung des Abstandes zweier Strings
 - Definition von Ähnlichkeit ist oft eine sehr schwierige Aufgabe
 - Ähnlichkeit ist abhängig vom Gegenstand und Aufgabe
 - Wann sind sich Gesichter ähnlich - Haarfarbe zählt weniger als Augenfarbe?
 - Wann sind sich Texte ähnlich – gleiche Wörter oder gleicher Inhalt?

Editskripte

- Definition

Ein *Editskript* e für zwei Strings A, B aus $\Sigma^* = \Sigma \cup \{ "_\}$ ist eine Sequenz von Editieroperationen

- I (*Einfügen* eines Zeichen $c \in \Sigma$ in A)
 - Dargestellt als Lücke in A ; das neue Zeichen erscheint in B
- D (*Löschen* eines Zeichen c in A)
 - Dargestellt als Lücke in B ; das alte Zeichen erscheint in A
- R (*Ersetzen* eines Zeichen in A mit einem anderen Zeichen in B)
- M (*Match*, d.h., gleiche Zeichen in A und B an dieser Stelle)

so, dass $e(A) = B$

- Beispiel: $A = \text{„ATGTA“}$, $B = \text{„AGTGTC“}$

– MIMMMR	IRMMMDI
A_TGTA	_ATGTA_
AGTGTC	AGTGT_C

Editabstand

- Offensichtlich gibt es für A, B ziemlich viele Editskripte
- Definition
 - Die *Länge eines Editskript* ist die Anzahl von Operationen o im Skript mit $o \in \{I, R, D\}$
 - Der *Editabstand* zweier Strings A, B ist die Länge des kürzesten Editskript für A, B
- Bemerkung
 - Matchen zählt nicht – interessant sind nur die Änderungen
 - Anderer Name: [Levenshtein-Abstand](#)
 - Es gibt oft verschiedene kürzeste Editskripte
 - | | | | | | | |
|----------|----------|----------|----------|----------|----------|----------|
| I | M | M | M | M | M | D |
| _ | A | G | A | G | A | _ |
| G | A | G | A | G | A | _ |
 - | | | | | | | |
|----------|----------|----------|----------|----------|----------|----------|
| D | M | M | M | M | M | I |
| A | G | A | G | A | G | _ |
| _ | G | A | G | A | G | A |

Alignment

- Andere Darstellung: **Alignments**
- Definition
 - Ein (*globales*) **Alignment** zweier Strings A, B ist eine Untereinanderanordnung von A und B , jeweils mit beliebigen zusätzlichen Leerzeichen (`_`), ohne dass zwei Leerzeichen untereinander stehen
 - Achtung: Untereinanderstehende Zeichen müssen nicht matchen
 - Der **Alignmentscore** eines Alignment ist die Anzahl von Leerzeichen und Mismatches
 - Der **Alignmentabstand** zweier Strings A, B ist der minimale Alignmentscore aller Alignments der beiden Strings

- Beispiele

–	A_TGT_A	A_T_GTA	_AGAGAG	AGAGAG_
	AGTGTC_	_AGTGTC	GAGAGA_	_GAGAGA

Score: 3

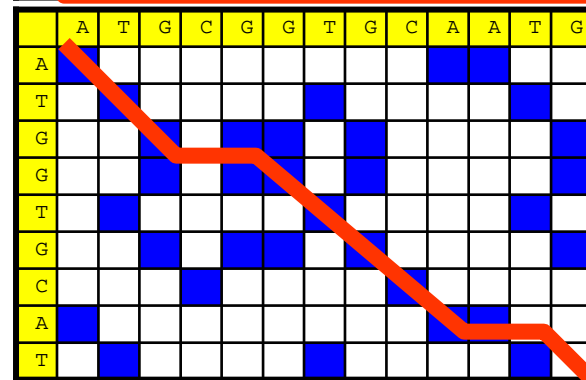
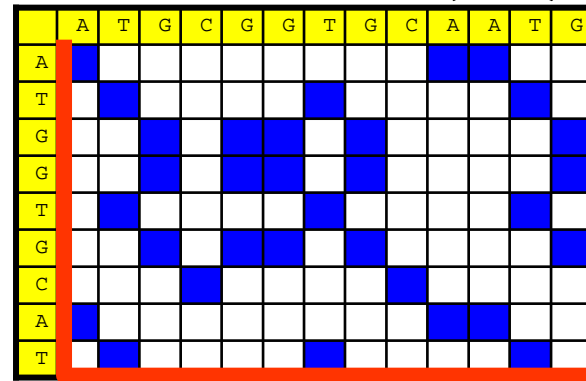
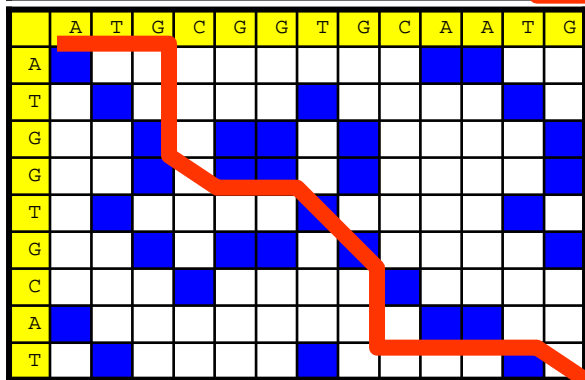
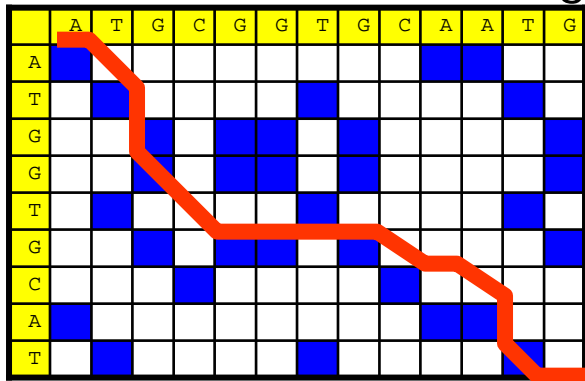
5

2

2

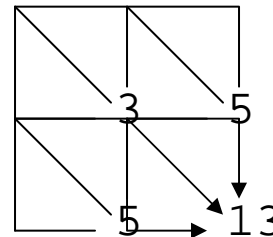
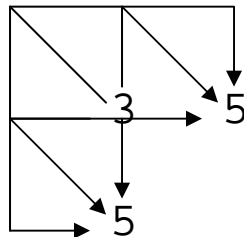
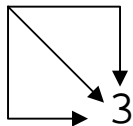
Alignments und Dotplots

- Sei $|A|=m$, $|B|=n$
- Betrachte **Pfade** im Dotplot von $(1,1)$ nach (m,n)
 - Pfad startet in linker oberer Ecke
 - Erlaubte Schritte: nach rechts, nach unten und nach rechts-unten (diagonal)
 - Pfad: Zusammenhänge Menge von Schritten bis (m,n)



Algorithmus

- Naives Verfahren um den besten Pfad zu finden
 - Alle Pfade aufzählen
 - Das sind **exponentiell viele**



- Nur Pfade „um“ die Hauptdiagonale: $> 3^{\min(m,n)}$
- **Inakzeptable Laufzeit**
- Tatsächliche Komplexität des Problems: $O(m^2)$
- Trick: **Dynamische Programmierung**

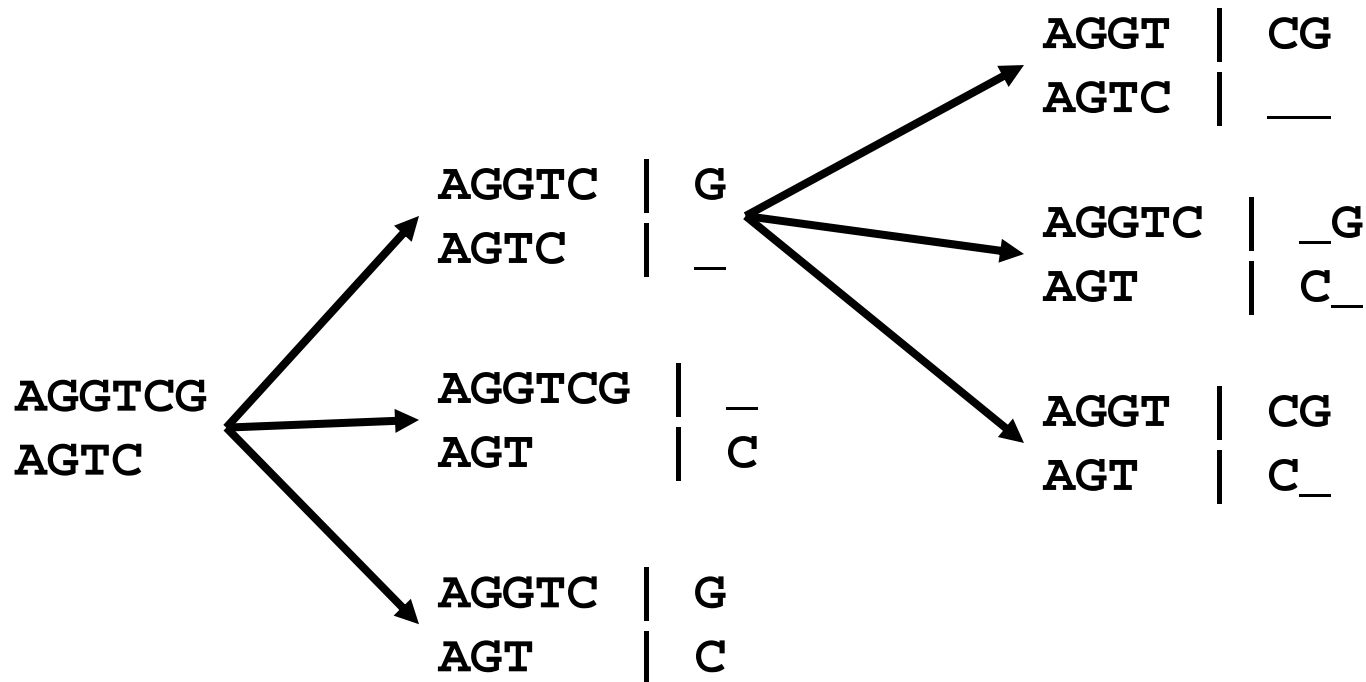
Inhalt dieser Vorlesung

- Editabstand durch dynamische Programmierung
- Varianten
 - Editgraphen
 - Needleman-Wunsch Algorithmus
 - Gewichtete Editabstände
 - Ähnlichkeit

Vorgehen

- Rekursive Formulierung
- Dynamische Programmierung: Bottom-Up statt Top-Down
- Andere Sichtweisen auf dasselbe Problem
 - Edit-Graphen, Ähnlichkeit, Needleman-Wunsch
- Andere, eng verwandte Probleme
 - End-Free Alignment, Lokales Alignment, Alignment mit Gaps, Alignment mit Substitutionsmatrizen
- Large-Scale: Datenbanksuche und heuristisches Alignment
- Mehr Sequenzen: Multiples Alignment
- Von Sequenzen zu Bäumen: Phylogenie

Rekursive Betrachtung von Alignments



Editabstände

- Definition

Gegeben zwei Strings A , B mit $|A|=n$, $|B|=m$

- *Funktion $\text{dist}(A,B)$ berechne den Editabstand von A , B*
- *Funktion $d(i,j)$, $0 \leq i \leq n$ und $0 \leq j \leq m$, berechne den Editabstand zwischen $A[1..i]$ und $B[1..j]$*

- Bemerkungen

- Jedes R kann durch $\{I,D\}$ ersetzt werden; also werden R bevorzugt
- Offensichtlich gilt: $d(n,m) = \text{dist}(A,B)$
- $d(i,j)$ dient zur rekursiven Berechnung von $\text{dist}(A,B)$

Rekursive Berechnung 1

- Wir betrachten die Berechnung von $d(i,j)$ für A,B
 - Wir haben die optimalen Editskript für $A[1..i_0]$ mit $B[1..j_0]$, $i_0 \leq i \wedge j_0 \leq j \wedge \neg(i_0 = i \wedge j_0 = j)$, berechnet
 - Wie kann das Editskript weitergeführt werden?
- Fallunterscheidung
 - 1. Insertion in A (oder Deletion in B)
 - Situation:
 ...I
 XXX_
 XXXT
 - Also benutzen wir ein Zeichen mehr von B
 - $d(i,j-1)$ ist der Editabstand von $A[1..i]$ zu $B[1..j-1]$
 - Symbolisiert durch die XXX
 - Damit: $d(i,j) = d(i, j-1) + 1$

Rekursive Berechnung 2

- Wir betrachten die Berechnung von $d(i,j)$ für A,B
 - Wir haben die optimalen Editskript für $A[1..i_0]$ mit $B[1..j_0]$, $i_0 \leq i \wedge j_0 \leq j \wedge \neg(i_0 = i \wedge j_0 = j)$, berechnet
 - Wie kann das Editskript weitergeführt werden?
- Fallunterscheidung
 - 2. Deletion in A (oder Insertion in B)
 - Situation:
 $\dots D$
 $XXXXT$
 $XXX_$
 - Umgekehrte Situation
 - Wir benutzen ein Zeichen mehr von A
 - $d(i-1, j)$ ist der Editabstand von $A[1..i-1]$ zu $B[1..j]$
 - Damit: $d(i, j) = d(i-1, j) + 1$

Rekursive Berechnung 3

- Wir betrachten die Berechnung von $d(i,j)$ für A,B
 - Wir haben die optimalen Editskript für $A[1..i_0]$ mit $B[1..j_0]$, $i_0 \leq i \wedge j_0 \leq j \wedge \neg(i_0 = i \wedge j_0 = j)$, berechnet
 - Wie kann das Editskript weitergeführt werden?
- Fallunterscheidung
 - 3. Match
 - Situation:
 $\dots M$
 $XXX Y$
 $XXX Y$
 - Wir benutzen ein Zeichen mehr von A und eines mehr von B
 - Match kostet nichts
 - Damit: $d(i,j) = d(i-1, j-1)$

Rekursive Berechnung 4

- Wir betrachten die Berechnung von $d(i,j)$ für A,B
 - Wir haben die optimalen Editskript für $A[1..i_0]$ mit $B[1..j_0]$, $i_0 \leq i \wedge j_0 \leq j \wedge \neg(i_0 = i \wedge j_0 = j)$, berechnet
 - Wie kann das Editskript weitergeführt werden?
- Fallunterscheidung
 - 4. Mismatch
 - Situation:
 $\dots R$
 $XXX Y$
 $XXX Z$
 - Wir benutzen ein Zeichen mehr von A und eines mehr von B
 - Mismatch kostet 1
 - Damit: $d(i,j) = d(i-1, j-1) + 1$

Rekursionsgleichung

- Wir leiten das nächste Symbol im Editskript aus schon bekannten Editabständen ab
- Wir suchen das **kürzeste Skript**, also

$$d(i, j) = \min \left\{ \begin{array}{l} d(i, j-1) + 1 \\ d(i-1, j) + 1 \\ d(i-1, j-1) + t(i, j) \end{array} \right\}$$

$$t(i, j) = \begin{cases} 1: & \text{wenn } A[i] \neq B[j] \\ 0: & \text{sonst} \end{cases}$$

Randbedingungen

- **Randbedingungen** nicht vergessen
 - $d(i,0) = i$
 - Um $A[1..i]$ zu „“ zu transformieren braucht man i Deletions
 - $d(0,j) = j$
 - Um $A[1..0]$ zu $B[1..j]$ zu transformieren braucht man j Insertions

Zusammen

- Theorem

- Der *Editabstand zweier Strings* A, B mit $|A|=n$, $|B|=m$ berechnet sich mit Startbedingung

$$d(i, 0) = i \quad d(0, j) = j$$

als $d(n, m)$ mit folgender *Rekursionsgleichung*

$$d(i, j) = \min \left\{ \begin{array}{l} d(i, j-1) + 1 \\ d(i-1, j) + 1 \\ d(i-1, j-1) + t(i, j) \end{array} \right\}$$

- Beweis

- Per Konstruktion

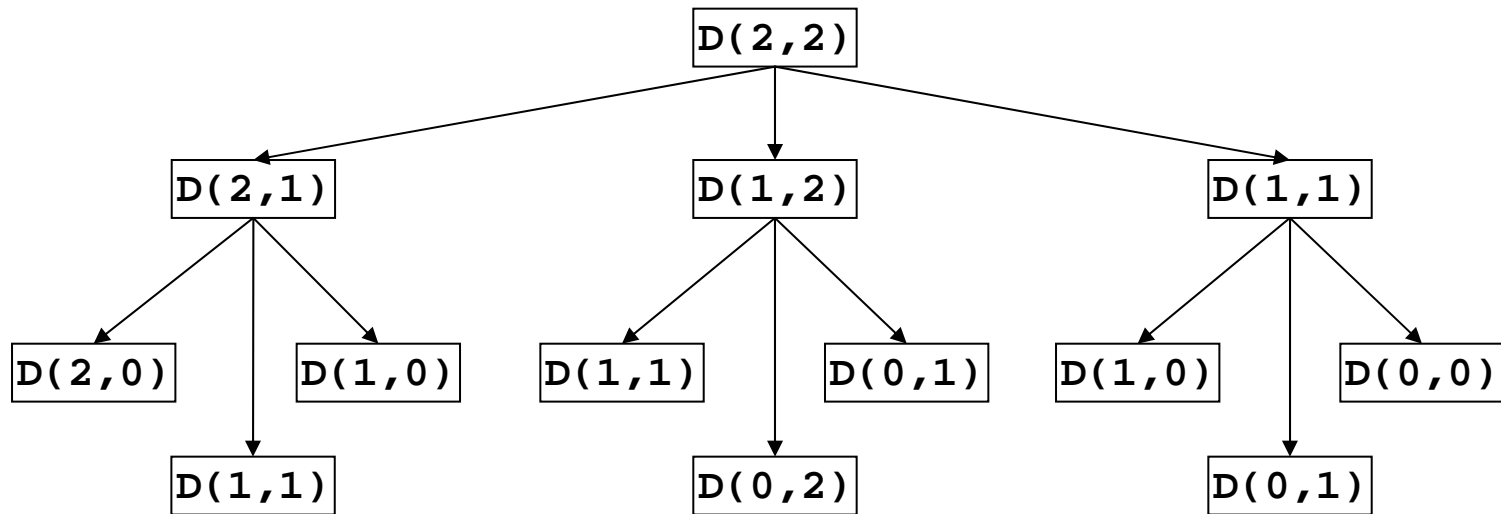
Rekursiver Algorithmus

```
function d(i,j) {
    if (i = 0)          return j;
    else if (j = 0)    return i;
    else
        return min (   d(i-1,j) + 1,
                       d(i,j-1) + 1,
                       d(i-1,j-1) + t(A[i],B[j]));
}
function t(c1, c2) {
    if (c1 = c2)    return 0;
    else               return 1;
}
```

- Komplexität?
 - Für n,m erfolgen 3 Aufrufe, die wiederum jeweils 3 Aufrufe auslösen, die ...
 - Komplexität damit mindestens $O(3^{\min(n,m)})$

Sicher nicht optimal

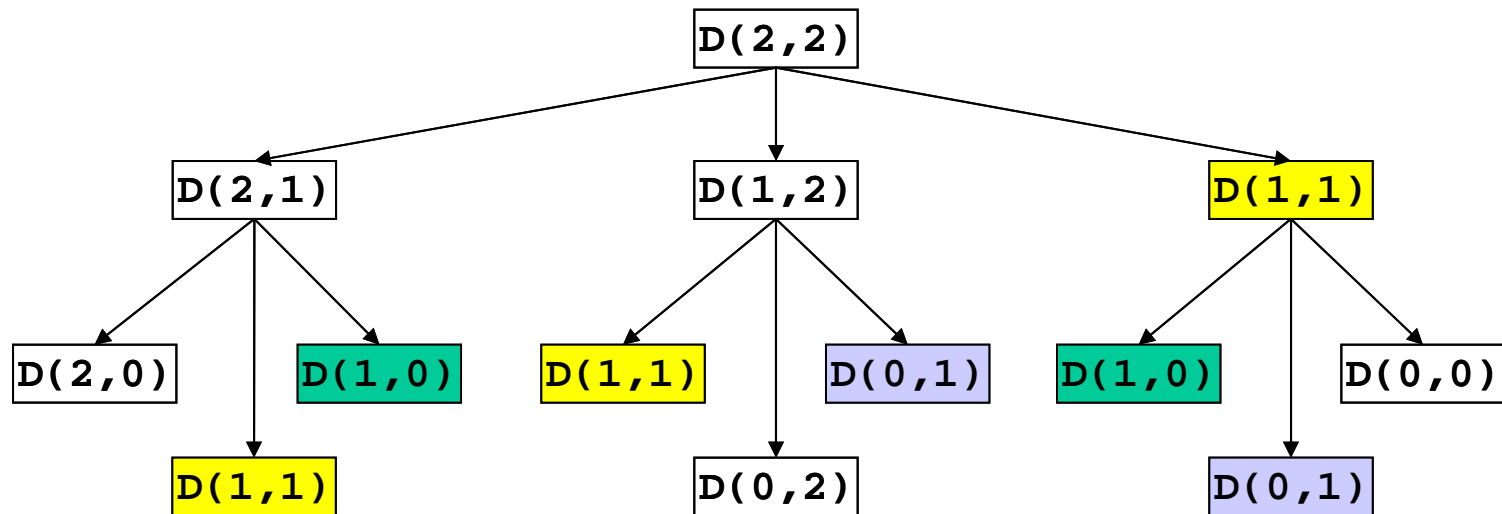
- Aufrufbaum mit Parametern



- Optimierungspotential?

Sicher nicht optimal

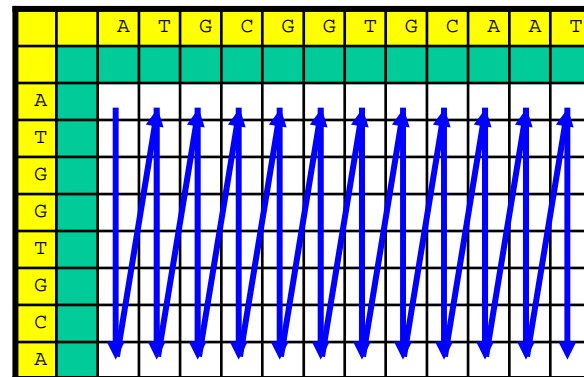
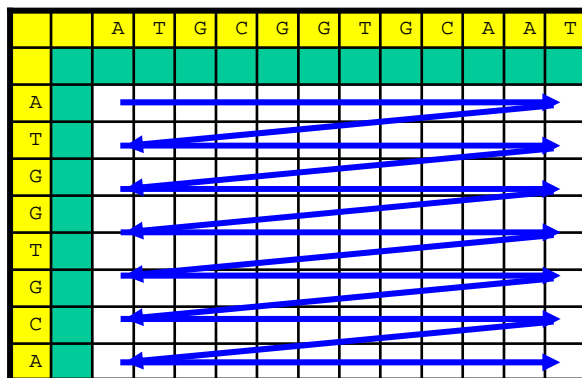
- Durch die Rekursionsgleichung werden viele Teillösungen mehrfach berechnet



- Es gibt nur $(n+1) * (m+1)$ verschiedene Aufrufe
- Wie kann man die redundanten Berechnungen sparen?

Tabellarische Berechnung

- Grundidee
 - Speichern der Teillösungen in Tabelle
 - Bei Berechnung Wiederverwendung wo immer möglich
- Aufbau der Tabelle: Bottom-Up (statt rekursiv Top-Down)
 - **Initialisierung** mit festen Werten $d(i,0)$ und $d(0,j)$
 - **Sukzessive Berechnung** von $d(i,j)$ mit steigendem i,j
 - Für $d(i,j)$ brauchen wir $d(i,j-1)$, $d(i-1,j)$ und $d(i-1,j-1)$
 - Verschiedene Reihenfolgen möglich



Beispiel

$$d(i, j) = \min \left\{ \begin{array}{l} d(i, j-1) + 1 \\ d(i-1, j) + 1 \\ d(i-1, j-1) + t(i, j) \end{array} \right\}$$

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1							
T	2							
G	3							
G	4							

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0						
T	2							
G	3							
G	4							

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2							
G	3							
G	4							

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3							
G	4							

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3	2	1	0	1	2	3	4
G	4							

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3	2	1	0	1	2	3	4
G	4	3	2	1	1	1	2	3

Was ist gewonnen?

	A	T	G	C	G	G	T	
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3	2	1	0	1	2	3	4
G	4	3	2	1	1	1	2	3

- Editabstand von ATGG, ATGCGGT ist 3
- Wir suchen aber ein Alignment, nicht nur den Abstand
- Extraktion aus der Tabelle durch „Traceback“
 - Bei Berechnung von $d(i,j)$ behalte Pointer auf minimale Vorgängerzelle(n)
 - Die muss nicht eindeutig sein

	A	T	G	C	G	G	T	
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3	2	1	0	1	2	3	4
G	4	3	2	1	1	1	2	3

	A	T	G	C	G	G	T	
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3	2	1	0	1	2	3	4
G	4	3	2	1	1	1	2	3

	A	T	G	C	G	G	T	
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3	2	1	0	1	2	3	4
G	4	3	2	1	1	1	2	3

Vom Pfad zum Alignment

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3	2	1	0	1	2	3	4
G	4	3	2	1	1	1	2	3

- Jeder Pfad von (n,m) nach $(1,1)$ ist ein optimales Alignment
 - Starte von $(1,1)$
 - Nach rechts: Deletion in A
 - Nach unten: Insertion in A
 - Diagonal: Match/Replace

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3	2	1	0	1	2	3	4
G	4	3	2	1	1	1	2	3

ATGCGGT
ATG_G__

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3	2	1	0	1	2	3	4
G	4	3	2	1	1	1	2	3

ATGCGGT
AT__GG_

Beispiel 2

	0	1	2	3	4	5	6	7	
		w	r	i	t	e	r	s	
0		0	1	2	3	4	5	6	7
1	v	1	1	2	3	4	5	6	7
2	i	2	2	2	2	3	4	5	7
3	n	3	3	3	3	3	4	5	6
4	t	4	4	4	4	3	4	5	6
5	n	5	5	5	5	4	4	5	6
6	e	6	6	6	6	5	4	5	6
7	r	7	7	6	7	6	5	4	5

	0	1	2	3	4	5	6	7	
		w	r	i	t	e	r	s	
0		0	1	2	3	4	5	6	7
1	v	1	1	2	3	4	5	6	7
2	i	2	2	2	2	3	4	5	7
3	n	3	3	3	3	3	4	5	6
4	t	4	4	4	4	3	4	5	6
5	n	5	5	5	5	4	4	5	6
6	e	6	6	6	6	5	4	5	6
7	r	7	7	6	7	6	5	4	5

WRIT_ERS
VINTNER_

	0	1	2	3	4	5	6	7	
		w	r	i	t	e	r	s	
0		0	1	2	3	4	5	6	7
1	v	1	1	2	3	4	5	6	7
2	i	2	2	2	2	3	4	5	7
3	n	3	3	3	3	3	4	5	6
4	t	4	4	4	4	3	4	5	6
5	n	5	5	5	5	4	4	5	6
6	e	6	6	6	6	5	4	5	6
7	r	7	7	6	7	6	5	4	5

WRI_T_ERS
V_INTNER_

	0	1	2	3	4	5	6	7	
		w	r	i	t	e	r	s	
0		0	1	2	3	4	5	6	7
1	v	1	1	2	3	4	5	6	7
2	i	2	2	2	2	3	4	5	7
3	n	3	3	3	3	3	4	5	6
4	t	4	4	4	4	3	4	5	6
5	n	5	5	5	5	4	4	5	6
6	e	6	6	6	6	5	4	5	6
7	r	7	7	6	7	6	5	4	5

WRI_T_ERS
VINTNER

Komplexität

- Aufbau der Tabelle
 - Zur Berechnung einer Zelle muss man genau drei andere Zellen betrachten
 - Konstante Zeit pro Betrachtung
 - $m \cdot n$ Zellen
 - Insgesamt: $O(m \cdot n)$
- Traceback (für ein Alignment)
 - Man kann einen beliebigen Pfad wählen
 - Es muss einen Pfad von (n, m) nach $(1, 1)$ geben
 - Jede Zelle hat mindestens einen Pointer
 - Keine Zelle zeigt aus der Tabelle hinaus
 - Worst-Case Pfadlänge ist $O(m+n)$
- Zusammen
 - $O(m \cdot n)$ (für $m \cdot n > m+n$)

Verbesserung

- Brauchen wir die Pointer?

Algorithmus ohne Pointer

- Setze $i=n$, $j=m$
- 1. Bestimme alle Herkunftszellen von (i,j)
 - Je nachdem ob $d(i,j)=d(i-1,j)+1$, $d(i-1,j)+1$, ...
 - Es gibt immer eine oder mehrere Herkunftszellen geben
- 2. Wähle eine Herkunftszelle aus
 - Setze i,j neu
 - Baue simultan Alignment auf
 - Pfade haben keine Sackgassen
- 3. Wenn $(i,j) \neq (0,0)$, dann springe zu 1

Komplexität bleibt unverändert

$O(m+n)$ Schritte mit jeweils max. 3 Berechnungen

Prinzip des dynamischen Programmierens

- Ziel: Optimierung einer Zielfunktion (Editabstand)
- Voraussetzung: Optimale Gesamtlösungen setzen sich aus optimalen Teillösungen zusammen
- Drei Bestandteile
 - Berechnung „größerer“ Probleme aus **optimalen Lösungen** für kleinere Probleme
 - Hier: Rekursionsgleichung
 - **Zwischenspeichern der Teillösungen**
 - Hier: Tabelle mit Werten $d(i,j)$
 - **Rückverfolgung** der Gesamtlösung aus Tabelle
 - Hier: Traceback des Alignments
- Weitere Beispiele
 - Kürzeste Wege in Graphen
 - Berechnung der optimalen Joinreihenfolge in RDBMS

Zwischenstand

- Logisch äquivalent
 - Alignment
 - Editskripte
 - Pfadgüte
- Tabellarische, Bottom-Up Berechnung
 - Prinzip der dynamischen Programmierung
 - Berechnet Editabstand und damit Alignmentsscore

Nächste Themen

- Varianten von Alignments
 - Editgraphen
 - Needleman-Wunsch Algorithmus
 - Gewichteter Editabstände
 - Ähnlichkeit

Analogie: Kürzeste Wege in Editgraphen

- Definition

Ein *Editgraph* für A, B mit $|A|=n$, $|B|=m$ ist

- Ein Graph mit $m \cdot n$ Knoten, beschriftet mit (i, j) für $0 \leq i \leq n$, $0 \leq j \leq m$
- Kanten und Gewichte
 - $(i, j-1) \rightarrow (i, j)$ mit Gewicht 1
 - $(i-1, j) \rightarrow (i, j)$ mit Gewicht 1
 - $(i-1, j-1) \rightarrow (i, j)$ mit Gewicht 0 gdw. $A[i]=B[j]$; sonst 1

- Es gilt

- Alle „leichtesten“ Wege von $(1, 1)$ bis (n, m) sind optimale Alignments
- Umkehrung gilt auch

Needleman-Wunsch Algorithmus

- Historisch der **erste Algorithmus**

- S.B. Needleman, C.D. Wunsch, „A general method applicable to the search for similarities in the amino acid sequence of two proteins“, J. of Molecular Biology, 48, 1970

- Drei Schritte

- Initialisiere Matrix mit 0/1 bei Mismatch/Match
 - Dotplot
- Berechne Zellwerte von rechts unten nach links oben mit

$$d(i, j) = d(i, j) + \max[d(i+1, j+1), d(i+2, j+1), \dots, d(n, j+1), d(i+1, j+2), \dots, d(i+1, m)]$$

Bestimme Alignment

- Dritter Schritt
 - Bestimme größten Wert auf den Rändern $(i,1)$ und $(1,j)$
 - Sei dieser an Position (k,l)
 - Baue Pfade von (k,l)
 - Springe zum **größten Wert auf den Grenzen des Rechtecks rechts unter (k,l)**
 - Gibt es den größten Wert mehrmals -> mehrere Pfade
 - Wenn das ein Diagonalschritt ist, verlängere im Alignment beide Zeilen um das nächste Zeichen
 - Wenn das kein Diagonalschritt ist, verlängere im Alignment nur eine Zeile um das nächste Zeichen und die andere um entsprechend viele Leerzeichen
 - Mindestens eine Koordinate erhöht sich nur um 1
 - Iteriere, bis entweder $k=n$ oder $l=m$
 - Alignment links und rechts mit Leerzeichen füllen

Beispiel

Initialisierung

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

Berechnung

	A	T	G	C	G	G	T
A	4	2	2	2	1	1	0
T	2	3	2	2	1	0	1
G	1	1	2	1	2	1	0
G	0	0	1	0	1	1	0

Pfade

	A	T	G	C	G	G	T
A	4	2	2	2	1	1	0
T	2	3	2	2	1	0	1
G	1	1	2	1	2	1	0
G	0	0	1	0	1	1	0

ATGCGGT
ATG_G__

	A	T	G	C	G	G	T
A	4	2	2	2	1	1	0
T	2	3	2	2	1	0	1
G	1	1	2	1	2	1	0
G	0	0	1	0	1	1	0

ATGCGGT
ATG__G_

	A	T	G	C	G	G	T
A	4	2	2	2	1	1	0
T	2	3	2	2	1	0	1
G	1	1	2	1	2	1	0
G	0	0	1	0	1	1	0

ATGCGGT
AT__GG_

Komplexität

- Komplexität (Sei $n=m$)
 - Initialisierung ist $O(n^2)$
 - Berechnung der Zellwerte
 - $n \cdot n$ Zellen
 - Maximum betrachtet jeweils $(n-i) + (n-j) + 1 = O(n)$ Zellen
 - Zusammen: $O(n^3)$
 - Pfadbestimmung ist $O(n^2)$
 - Insgesamt sind es n (eigentlich: $\min(m,n)$) Sprünge
 - Bei jedem Sprung werden $(n-i) + (n-j) + 1 = O(n)$ Zellen durchsucht
 - Zusammen: $O(n^3)$
- Algorithmus wird nicht mehr benutzt

Warum ist Needleman/Wunsch so schlecht?

- Was passiert?
 - Algorithmus **maximiert die Matches** (1'er) über alle Pfade
 - Dabei werden aber pro Teillösungen viele weitere Pfade betrachtet
 - man kann ja mitten in die Kanten springen
 - Grundoperationen des Alignments sind damit nicht Leerzeichen+Buchstabe und Buchstabe+Buchstabe, sondern
 - Buchstabe + Buchstabe
 - Leerzeichen + 3 Buchstaben
 - 2 Leerzeichen + 4 Buchstaben
 - 3 Leerzeichen + 5 Buchstaben
 - ...
 - Wir betrachten also unnötig viele und unnötig komplizierte Operationen

Gewichtete Editabstände

- Unser Bewertung von Abstand liegt zugrunde
 - Matching ist umsonst
 - Löschen / Einfügen / Vertauschen ist gleich teuer (1)
- Das muss nicht so sein
 - Aminosäuren sind mehr oder weniger ähnlich
 - Vertauschungen haben unterschiedlich starke Wirkung
 - Ersetzungen (R) sollten unterschiedliche Gewichte haben
 - Lange Gaps wegen Crossing-Over-Events nicht linear schlimmer als kurze Gaps
 - I/D auf DNA bewirkt Frameshift – viel schlimmer als R
- Man braucht flexiblere **Bewertungsschemata**

Allgemeines Bewertungsschema

- Operationen werden **unterschiedlich bestraft**
 - Einfügen: c_i
 - Löschen: c_d
 - Match: m
 - Replace: $r(x,y)$, wobei $x=A[i]$ und $y=B[j]$
- $r(x,y)$ entnimmt man einer **Substitutionsmatrix**
 - Eigenes Thema – später (PAM, BLOSUM)
 - Beispiel

	A	C	G	T
A	0	1	1	1
C	1	0	.5	1
G	1	.5	0	1
T	1	1	1	0

- Bemerkung
 - $r(x,y)$ sollte immer kleiner als c_i+c_d sein, da I+D ein R simulieren

Bewertungsschema im Algorithmus

- Algorithmus muss nur marginal verändert werden

$$d(i,0) = i * c_d \quad d(0,j) = j * c_i$$

$$d(i,j) = \min \left\{ \begin{array}{l} d(i,j-1) + c_i \\ d(i-1,j) + c_d \\ d(i-1,j-1) + r(i,j) \end{array} \right\}$$

- Alle Eigenschaften bleiben erhalten
 - Optimalität, tabellarische Berechnung, Pfade, ...

Beispiel

Scoring $r=2, i/d=3$

	0	1	2	3	4	5	6	7
		w	r	i	t	e	r	s
0	0	3	6	9	12	15	18	21
1	v	3	2	5	8	11	14	17
2	i	6	5	4	5	8	11	14
3	n	9	8	7	6	7	10	13
4	t	12	11	10	9	6	9	12
5	n	15	14	13	12	9	8	11
6	e	18	17	16	15	12	9	10
7	r	21	20	17	18	15	12	9

VINTNER_
WRIT_ERS VINTNER
WRITERS

Scoring $r=1, i/d=1$

	0	1	2	3	4	5	6	7
		w	r	i	t	e	r	s
0	0	1	2	3	4	5	6	7
1	v	1	1	2	3	4	5	6
2	i	2	2	2	2	3	4	5
3	n	3	3	3	3	3	4	5
4	t	4	4	4	4	3	4	5
5	n	5	5	5	5	4	4	5
6	e	6	6	6	6	5	4	5
7	r	7	7	6	7	6	5	4

VINTNER V_INTNER_
WRI_T_ERS WRI_T_ERS

VINTNER_
WRIT_ERS

Ähnlichkeit

- Welche Frage will man eigentlich beantworten?
 - Wie weit entfernt sind diese beiden Sequenzen
 - Wie ähnlich sind sich zwei Sequenzen
- Ähnlichkeit ist die andere Seite der Medaille
 - Je ähnlicher zwei Strings, desto geringer ist der Abstand
 - „Gut“ ist also: kleiner Abstand, große Ähnlichkeit
 - Man maximiert für die Ähnlichkeit und Minimiert für den Abstand
 - **Logisch ist Ähnlichkeit und Abstand äquivalent**
- Differenzierte Betrachtung
 - Ähnlichkeit einzelner Zeichen / Basen / Aminosäuren
 - Ähnliche Zeichen – positive Werte
 - Unähnliche Zeichen – negative Werte

Formal

- Definition

Gegeben Alphabet $\Sigma' = \Sigma \cup _$ und ein Alignment zweier Strings A, B der Länge n

- Eine *Scoringfunktion* ist eine Funktion $s: \Sigma' \times \Sigma' \rightarrow \text{Integer}$
- Die *Ähnlichkeit* des Alignments von A und B bzgl. s ist

$$\text{sim}(A, B) = \sum_{i=1}^n s(A[i], B[i])$$

- Bemerkung

- Wir suchen im Allgemeinen wieder das Alignment mit der größten Ähnlichkeit
- Die Parameter c_i , c_d und m sind nun Teil der Tabelle

Beispiel

$$\Sigma' = \{A, C, G, T, _ \}$$

	A	C	G	T	_
A	4	-2	-2	-2	0
C		4	-2	-2	-2
G			4	-1	0
T				4	-2
_					

A	C	_	G	T	C
A	G	G	T	_	C

= 3

A	C	G	T	C
A	G	G	T	C

= 14

Rekursionsgleichung

- Wieder nur kleine Veränderung
 - Welche?

$$d(i,0) = \sum_{i=1}^n s(A[i], _) \quad d(0, j) = \sum_{i=1}^m s(_, B[j])$$

$$d(i, j) = \max \left\{ \begin{array}{l} d(i, j-1) + s(_, B[j]) \\ d(i-1, j) + s(A[i], _) \\ d(i-1, j-1) + s(A[i], B[j]) \end{array} \right\}$$

Logisch äquivalent

- Alignment
 - Untereinanderordnung von Strings mit Einfügen von Spaces
 - Alignmentscore zählt Mismatches und Spaces
 - Alignmentabstand: Minimiert Alignmentscore
- Editskripte
 - Operationen auf Zeichen
 - Länge eines Editskript zählt Operationen R, I, D
 - Dynamische Programmierung: Berechnet Editabstand
- Ähnlichkeit
 - Differenzierte Betrachtung der „Ähnlichkeit“ von Zeichen
 - Ähnlichkeitsmaß summiert Zeichenähnlichkeiten
 - Dynamische Programmierung: Maximiert Ähnlichkeit

Zusammenfassung

- Berechnung eines optimalen globalen Alignments hat **quadratische Komplexität**
 - Mittels dynamischer Programmierung
 - Tabelle aufbauen, Pfad zurückverfolgen
- **Platzbedarf ist auch quadratisch**
 - Das ist kritisch
 - Wir werden Algorithmen mit linearem Platzbedarf kennen lernen
- Diverse Erweiterungen
 - Berücksichtigen die **Biologie der Aufgabe**: Mutationswahrscheinlichkeiten, Aminosäureähnlichkeiten, Evolutionsmodell, ...
 - Hohe praktische Relevanz: Erweiterungen machen approximatives Stringmatching erst anwendbar in der Bioinformatik