Bioinformatik

Alignment mit linearem Platzbedarf

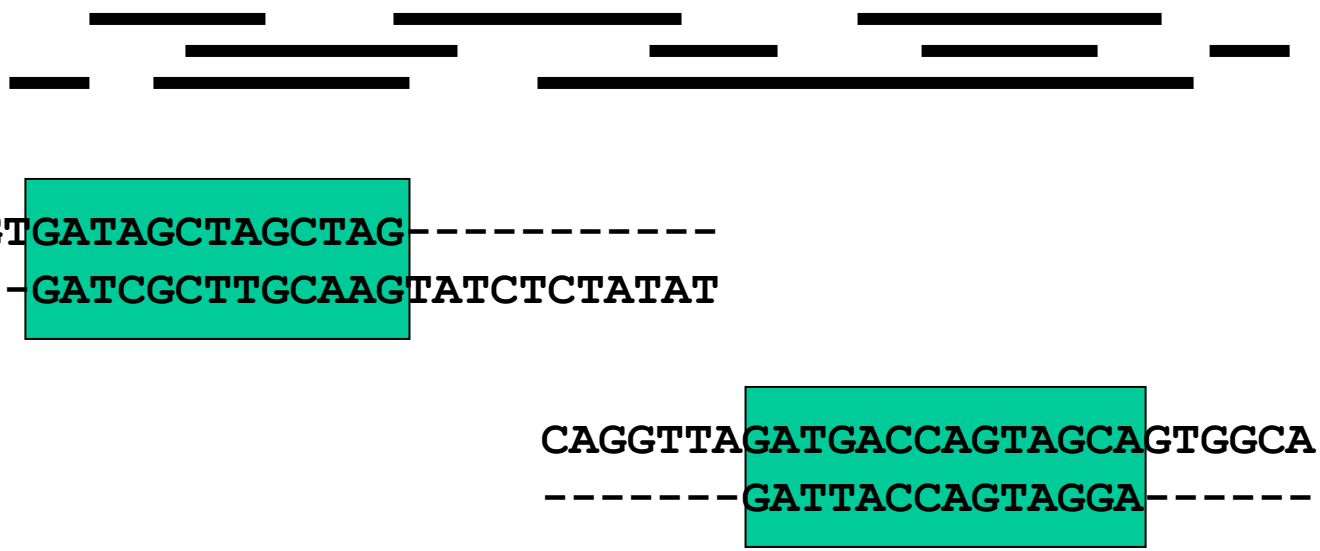
K-Band Alignment

Ulf Leser

Wissensmanagement in der Bioinformatik

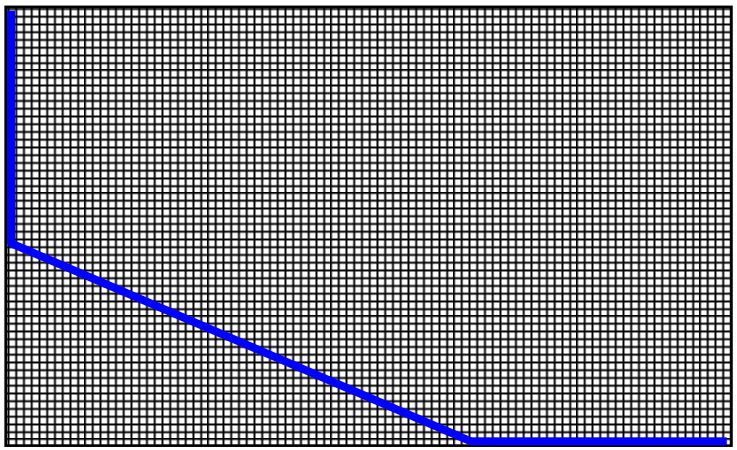
Spezialfall Assembly

- Assembly beim Shotgun-Sequenzieren braucht keine Alignments der kompletten Strings

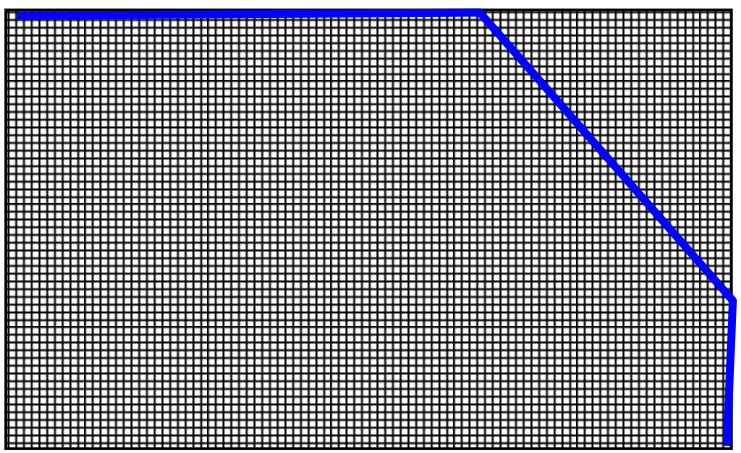


- Modellierung?
  - Spaces am Anfang / Ende des Alignments sind umsonst

End-Free Pfade



```
_______AAAA_AAAAAAAAAA_AAAAAAAAAAAA
BBBBBBBB__BBBB_BBBB_BBBBBBB________
```



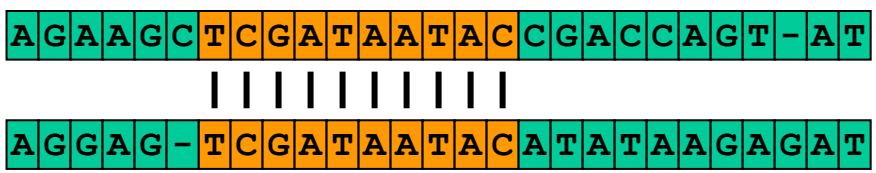
```
AAAAAAAAAAAAAAAA_AA_AAAAAA________
________________BBBBBBBB__BBBB_BBBB
```

Lokale Alignments

- Definition. *Gegeben zwei Strings A, B.*
  - *Finde Substrings $a \in A$, $b \in B$ so dass*

$$sim(a,b) = \max_{\forall a' \in A, b' \in B} \left(sim(a', b') \right)$$

  - *Das vom (globalen) Alignment von a und b induzierte Alignment von A und B heißt lokales Alignment*
- Bemerkung
  - Lokale Alignments sind unempfindlich gegen unterschiedliche lange Strings
  - Wichtigkeit der „Blockung" hängt von Scoring Funktion ab
- Beispiel
  - Lokales A. findet den identischen Substring
  - Das ist die biologisch wichtige Information

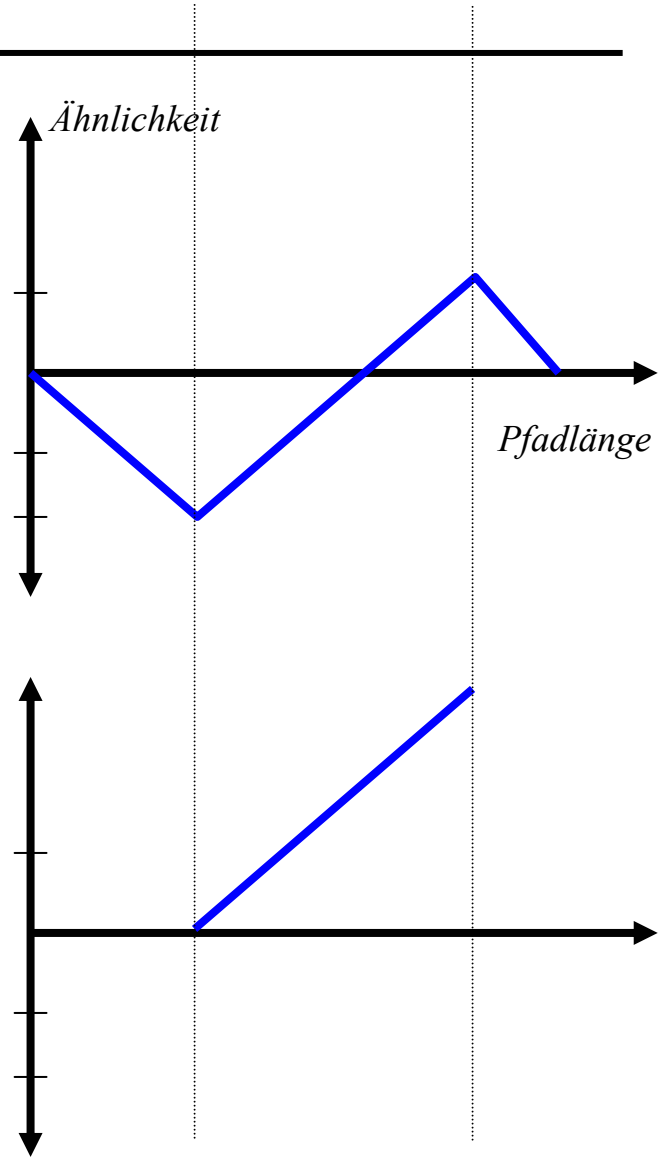


Match: +1
I/R/D: -1

Beispiel

		A	T	G	T	G	G
	0	-1	-2	-3	-4	-5	-6
G				-1			
T					0		
G						1	
A							0

		A	T	G	T	G	G
	0	-1	0	-3	-4	-5	-6
G				1			
T					2		
G						3	
A							0



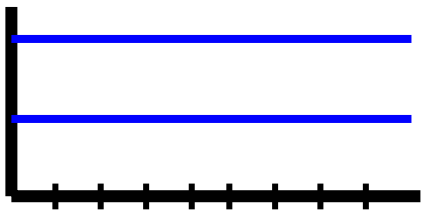
Lösen des lokales Suffixalignmentproblems

- Theorem.
  Gegeben Strings A,B. Dann gilt

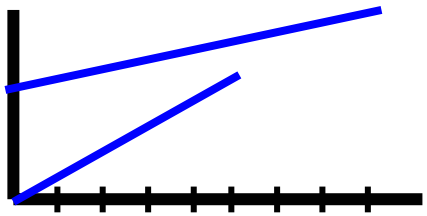
$$v(i,j) = \max \left\{ \begin{array}{c} 0 \\ d(i,j-1) + s(\_, B[i]) \\ d(i-1,j) + s(A[i], \_) \\ d(i-1,j-1) + s(A[i], B[j]) \end{array} \right\}$$

- Beweisidee
  - Beweis läuft sehr ähnlich zum Beweis der ursprünglichen Rekursionsformel
  - Einzige Ausnahme ist die „0" – der Reset
- Traceback
  - Starte beim maximalen Wert in der Matrix (nicht notwendigerweise am Rand)
  - Verfolge beliebigen Pfad bis zu einer Zelle mit Wert 0

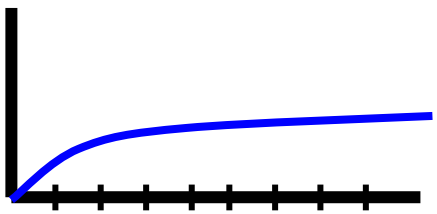
Klassen von Gapscorefunktionen



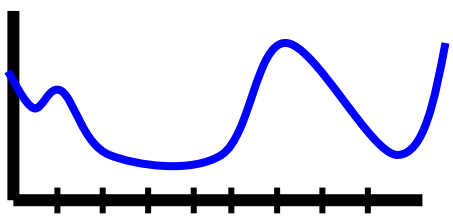
- Konstanter Gapscore



- Linearer Gapscore

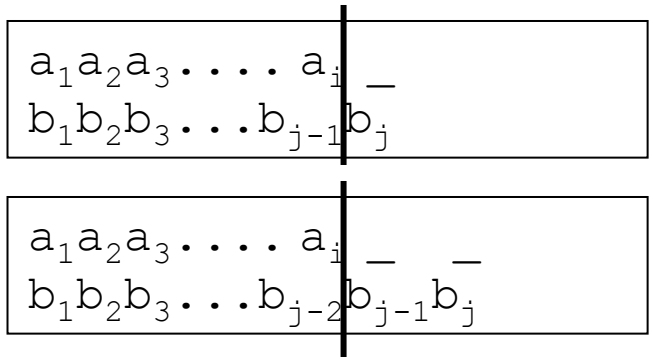


- Konvexer Gapscore

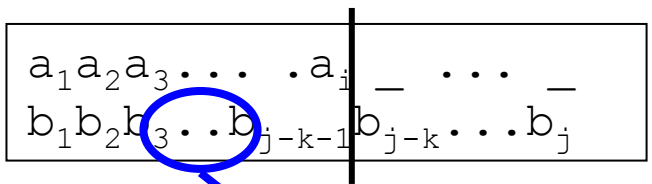


- Beliebiger Gapscore

Beliebige Gapscorefunktion: Insertions



Allgemeiner Fall



Hier liegen natürlich entsprechend mehr Leerzeichen in B verborgen

Drei Rekursionsgleichungen

$$E(i,j) = \max_{1 \le k \le j-1} \left(V(i,k) - w(j-k) \right)$$

$$F(i,j) = \max_{1 \le l \le i-1} \left(V(l,j) - w(i-l) \right)$$

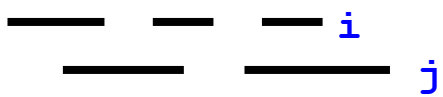
$$G(i,j) = V(i-1, j-1) + s(A[i], B[j])$$

$$V(i,j) = \max(E(i,j), F(i,j), G(i,j))$$

Komplexität

- Theorem
  Gegeben Strings A,B mit |A|=n, |B|=m. Für beliebige Gapscorefunktionen kann das optimale Alignment in $O(n^2m+nm^2)$ berechnet werden
- Beweis
  - Berechnung der Werte in Tabelle ähnlich Alignierung ohne Gaps
  - In jeder Zelle berechnen wir 3 Werte: E(i,j), F(i,j), G(i,j)
  - Dazu müssen die Zellen (i-1,j-1), (i,1...j-1) und (1...i-1,j) betrachtet werden
  - Für fixes i (eine Spalte füllen) betrachten wir also Σj für $1 \leq j \leq i-1$ Zellen. Das ist $O(i^2)$
  - Dito für fixe j – $O(j^2)$
  - Wir haben n Zeilen und m Spalten
  - Daraus folgt: $O(n*m^2 + m*n^2)$

Lineares Gapscorefunktion: Insertions



- Zwei Fälle
  - Gap beginnt in A an Position i

$$E(i,j) = V(i,j-1) - w_f - w_s$$

  - Gap in A wird fortgesetzt

$$E(i,j) = E(i,j-1) - w_f$$

- Zusammen (für Insertion in A)

$$E(i,j) = \max\left(V(i,j-1) - w_f - w_s, E(i,j-1) - w_f\right)$$

Komplexität

- Theorem
  *Gegeben Strings A,B mit |A|=n, |B|=m. Für lineare Gapscorefunktionen kann das optimale Alignment in O(n*m) berechnet werden*
- Beweis
  - Wir berechnen in jeder Zelle die Werte E(i,j), F(i,j), G(i,j)
  - Für die Berechnung jeder Zelle müssen nur konstant viele andere Zellen betrachtet werden
  - Daraus folgt: O(n*m)

Inhalt dieser Vorlesung

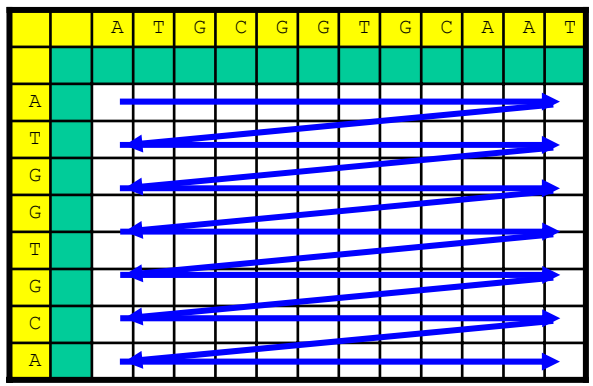
- Alignment in linearem Platz
- K-Banded Alignment
  - Alignment in (meistens) weniger als O(n*m)

Alignment mit linearem Platzbedarf

- Bisherige Algorithmen haben O(m*n) Zeit- und O(m*n) Platzbedarf
  - Woher kommt der Platzbedarf?
- Platzbedarf ist ein Problem für Alignierung großer Sequenzen (Genom-Genom)
- Gesucht: Speicherplatzeffizienterer Algorithmus

Editabstand in O(n) Space

- Editabstand kann man leicht in linearem Platz berechnen



- Für Zeile i+1 sind nur Werte von Zeile i sowie Spalte 0 notwendig
  - Berechnung des Editabstand also in O(n) Space möglich
- Aber: Keine Berechnung des tatsächlichen Alignments mehr möglich (kein Traceback)

Alignment in O(n) Space

- Klassische Erweiterung für Algorithmen basierend auf dynamischer Programmierung
  - Hirschberg: „Algorithms for the longest common subsequence problem", Journal of the ACM 24, 1977
- Grundidee
  - Rekursive Zerlegung des Problems in viele kleinere
  - Diesen werden in linearem Platz und quadratischer Zeit gelöst
  - Die Gesamtlösung wird aus den Teillösungen zusammengesetzt
  - Laufzeitkomplexität wird nicht schlechter
- Beweise: Gusfield, pp. 256-259
  - Wir beschränken uns auf die Idee
- Zuerst einige Vorarbeiten

Strings und reverse Strings

- Definition
  - *Mit A^r bezeichnen wir das Reverse eines Strings A*
  - *$A^r[1..i]$ bezeichnet entsprechend die ersten i Zeichen von A^r*
  - *Für Strings A,B sei $v^r(i,j)=sim(A^r[1..i], B^r[1..j])$*
- Bemerkung
  - Offensichtlich gilt : $v^r(i,j)=sim(A[n-i..n],B[m-j..m])$
  - Berechnung von v^r kann exakt wie die Berechnung von v erfolgen

```
A  ............ATGCGGT
B  ..............GGTCGTAG

Ar TGGCGTA............
Br GATGCTGG..............
```

Problemhalbierung

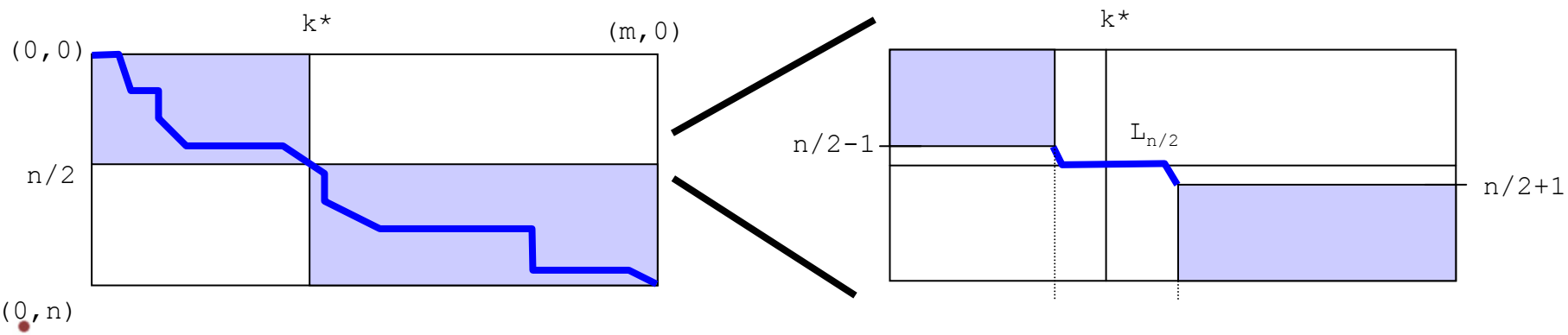
- Lemma.
  Gegeben A,B

$$v(n,m) = \max_{0 \le k \le m} \left(v(n/2,k) + v^r(n/2, m-k) \right)$$

- Beweisidee / Intuition
  - Wir alignieren
    - A[1..n/2] mit B[1..k] vorwärts
    - A[n/2+1..n] mit B[k+1..m] rückwärts
  - Durch das laufende k erwischen wir auf alle Fälle den optimalen Pfad durch die Matrix
- Bemerkung
  - Das Problem wird also bzgl. n (=|A|) halbiert

Teilpfade

- Definition
  - *Sei k* das k für das das Maximum v(n,m) erreicht wird*
  - *Sei L der Pfad von (0,0) bis (n,m)*
  - *Sei $L_{n/2}$ der Pfad zwischen dem letzten Knoten in der Zeile n/2-1 und dem ersten Knoten in n/2+1*
- Lemma
  - *L und $L_{n/2}$ müssen k* enthalten*
- Beweisidee: L muss irgendwo die Zeile n/2 passieren

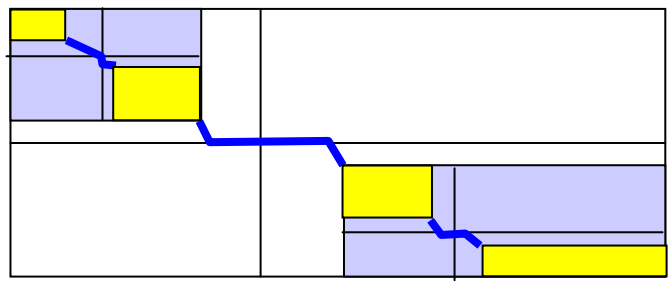


Folgerungen

- Lemma
  - *k* kann in Zeit O(m*n) und Platz O(m) berechnet werden*
  - *$L_{n/2}$ kann ebenfalls in Zeit O(m*n) und Platz O(m) berechnet werden*
- Beweisidee
  - Berechne Matrix zeilenweise vorwärts von 0 bis n/2-1; speichere jeweils nur die letzte Zeile
    - O(m*n) Zeit, O(m) Platz
  - Berechne Matrix zeilenweise rückwärts von n bis n/2; speichere jeweils nur die letzte Zeile
    - O(m*n) Zeit, O(m) Platz
  - Mit den Werten v(n/2, 1..m) und v^r(n/2, 1...m) kann man k* finden (Summe maximieren)
    - O(m) Zeit, kein Platzverbrauch
  - Der Pfad $L_{n/2}$ wird gefunden durch Traceback von Zelle (n/2,k*) bis zu einer Zelle in Zeile n/2-1
    - O(m) Zeit, kein Platzverbrauch

Rekursion

- Wo sind wir?
  - Wir halten die gewünschten Komplexitätsschranken
  - Wir haben das Mittelstück von L berechnet
- Damit können wir rekursiv absteigen
  - Löse rekursiv die Probleme
    - für A[1..n/2] / B[1..k*] und
    - für A[n/2+1..n] / B[k*+1..m]



- Platzkomplexität steigt nicht
- Zeitkomplexität steigt auch nicht
  - Zeitbedarf steigt um Faktor 2
  - Beweis: Gusfield

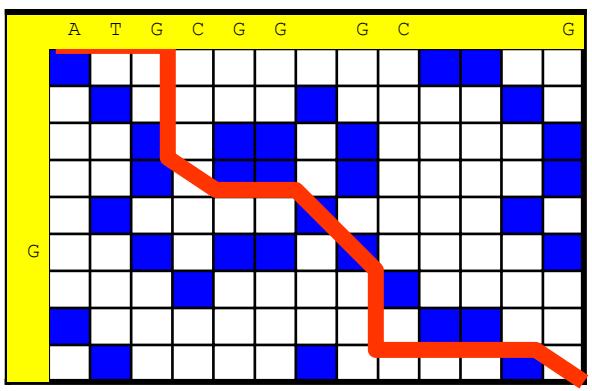
K-Band Algorithmus

- Den Platz haben wir damit drastisch reduziert
- Kann man auch was an der Laufzeit machen?
  - Im allgemeinen Fall nicht
  - Aber oft sucht man nur besonders gute Alignments
    - Z.B. Forensik, Vaterschaftstests, Suche nach homologen Gensequenzen
    - Ein Treffer ist entweder fast identisch oder uninteressant

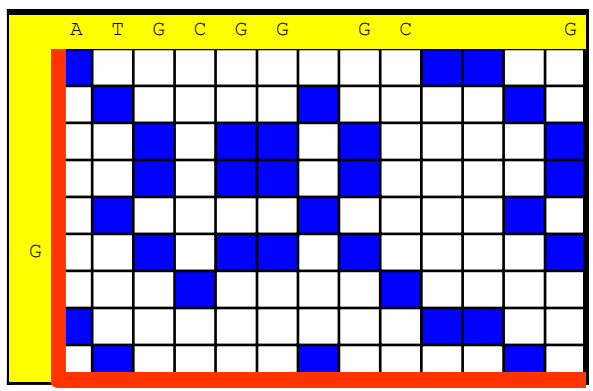
Gute Alignments

- Im Folgendne
  - Wir maximieren Ähnlichkeit: s ist der Score eines Matches, b der einer Insertion oder Deletion (i.d.R. negativ), 0 der Score eines Mismatch
  - Wir nehmen |A|=|B|=n an
- Der bestmögliche Score eines Alignments für A, B ist also n*s, der schlechtestmögliche 2*b
- Gute Alignments müssen eng an der Hauptdiagonale bleiben
  - Jedes Abzweigen kostet – wir müssen auch wieder zurück
- Können wir irgendwie nur die Diagonale entlang laufen?

```
ATG___CGGTG__CAATG
___ATGG__TGCA____T
```




```
___________ATGCGGTGCAATG
ATGGTGCCAT_____________
```

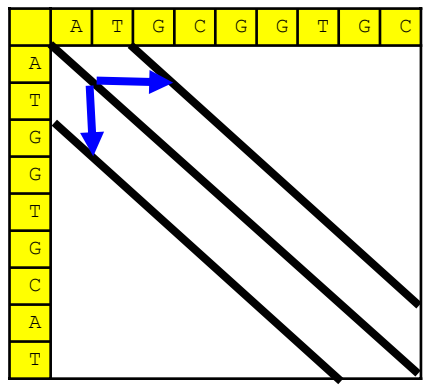


K-Band Algorithmus

- Wir erlauben ein Abweichungen nur um +k/-k Schritte
- Algorithmus
  - Berechnet das beste globale Alignment innerhalb des Bandes der Breite 2*k

```
for i= 1 to n do
  for j= i-k to i+k do
    if (j<1) or (j>n) break;
    M[i,j]= M[i-1,j-1] + t(A[i],B[j]);
    if inband(i-1,j) then
      M[i,j]= max( M[i,j], M[i-1,j]+b);
    if inband(i,j-1) then
      M[i,j]= max( M[i,j], M[i,j-1]+b);
  end for;
end for;
return M[n,n]
```

Beispiel: k=2



Eigenschaften

- Ein zusammenhängendes Gap eines Alignments im k-Band kann höchstens 2k Leerzeichen lang sein
  - (Ist auch die Gesamtanzahl von Leerzeichen beschränkt?)
  - Wenn wir also Alignments mit höchstens z langen Gaps suchen, reicht k=z/2
- Komplexität des K-Band Algorithmus?
  - Für jede Zelle sind maximal 3 Zugriffe, 1 Addition und ein paar Vergleiche notwendig
  - Wir berechnen O(2k*n) Zellen
  - Also: O(k*n)

Optimalität

- Theorem
  Gegeben Strings A, B mit |A|=|B|. Sei $d_K(A,B)$ der optimale K-Band Score für A und B. Wenn
  $d_K(A,B) \geq s*(n-k-1)+2b*(k+1)$, dann ist $d_K(A,B)=d(A,B)$.
- Beweis
  - Wenn das optimale Alignment im k-Band läuft, gilt auf alle Fälle $d_K=d(A,B)$
  - Wenn nicht, dann muss es irgendwo aus dem K-Band laufen. Im optimalen Fall haben wir dann n-k-1 Matches und dann k+1 Leerzeichen zum Verlassen des Bandes und weitere k+1 Leerzeichen, um am Ende noch (n,n) zu erreichen
  - Der bestmögliche Score außerhalb des K-Bandes ist also s*(n-k-1)+2b*(k+1)
  - Wenn also $d_k \geq s*(n-k-1)+2b*(k+1)$, muss das optimale Alignment im K-Band laufen und damit durch den K-Band Algorithmus gefunden werden
  - qed.

Iteratives K-Band

- Das können wir ausnutzen, um das optimale Alignment iterativ zu finden

```
k = 1;
while (true) do
        compute d_k;                    // Costs O(k*n)
        if d_k ≥ s(n-k-1)+2b(k+1) then
                return d_k;
        else
                k = 2*k;
        end if;
end while;
```

Komplexität

- Theorem.
  Sei d=d(A,B). Der iterative K-Band Algorithmus benötigt $O(sn^2-dn)$ Laufzeit.
- Beweis
  - d_k wird mit wachsendem k nie kleiner, sondern höchstens größer
  - Der Algorithmus stoppt, wenn $d_k \geq s(n-k-1)+2b(k+1)$, also

$$k \geq \frac{sn - d_k}{s-2b} - 1$$

  - Bis dahin wurden O(1n+2n+4n+...+kn) ~ O(2kn) Berechnungen durchgeführt
  - Wenn wir bei k stoppen, dann kann bei k/2 die Abbruchbedingung noch nicht erfüllt gewesen sein, und damit gilt:

$$\frac{k}{2} < \frac{sn - d_{k/2}}{s-2b} - 1$$

Komplexität - 2

- Betrachten wir den vorletzten Schritt k/2. Zwei Möglichkeiten
  - $d_{k/2}=d_k=d$.
    $$k < 2\left(\frac{sn-d}{s-2b}-1\right)$$
  - $d_{k/2}<d_k=d$. Dann haben wir mit k/2 das optimale Alignment noch nicht gefunden, weil es mehr als k/2 Leerzeichen hat. Damit muss gelten:
    $$d \le s(n-k/2-1)+2b(k/2+1)$$
  - Und damit:
    $$k \le 2\left(\frac{sn-d}{s-2b}-1\right)$$
- Die Berechnungszeit 2kn können wir damit nach oben abschätzen durch:
  $$2nk \le 2n*2\left(\frac{sn-d}{s-2b}-1\right)=4n\left(\frac{sn-d}{s-2b}-1\right)$$
- Und das ist $O(sn^2-dn)$
- qed.

Zusammen

- K-Band Algorithmus hat Komplexität $O(sn^2-dn)$
- Setzen wir z.B. s=1
  - Dann kann d maximal n sein
  - Sehr ähnliche Sequenzen erreichen Wert nahe bei d
  - Dann läuft der Algorithmus auch sehr schnell
- K-Band also umso besser, je ähnlicher die Sequenzen sind
  - Gut, um das schnell festzustellen (Algorithmus mit kleinen k laufen lassen)
  - Schlecht, um irgendwelche Alignments zu berechnen