Blatt 1

3. Grundlagen: Strings

Def. 3.1:

Alphabet: endl. nicht leere Menge von Symbolen

String: endl. Folge von Symbolen

$|s|$: Länge des Strings s

$|s|_x$: Anzahl des Vorkommens von x in s

ε: leerer String

Σ^n: Menge aller Strings über dem Alphabet Σ der Länge n

Σ^*: ——— " ———

Def. 3.2: seien s, t Strings über Σ

$s \cdot t = st$: Konkatenation von s und t

Def. 3.3: seien s, t Strings über Σ

s ist Teilstring von t, falls $\exists\, u, v \in \Sigma^*$, so daß $t = usv$

s ist Suffix von t, falls $\exists\, u \in \Sigma^*$, so daß $t = us$

s ist Präfix von t, falls $\exists\, v \in \Sigma^*$, so daß $t = sv$

s ist echter Teilstring, falls s Teilstring von t ist und $s \neq t$

s ist Teilsequenz von t, falls alle Buchstaben von s in der selben Reihenfolge in t vorkommen.

Beispiel: ad ist Teilsequenz von $abcd$

acb ist keine Teilsequenz von $abcd$

Def. 3.4: Σ Alphabet, $s = s_1 \dots s_n$, $s_i \in \Sigma$

$s[i,j] := s_i \dots s_j$

$s[i] := s_i$

Def. 3.5: seien s, t Strings über Σ. Dann ist eine Zerlegung von s und t gibt mit (i) $s = xy$

(ii) $t = yz$

(iii) $x \neq \varepsilon$, $z \neq \varepsilon$

(iv) (v) maximal mit (i)(ii), (iii)

Bsp. (3) dann heißt y der Overlap von s und t $ov(s,t)$, $ov(s,t) = |ov(s,t)|$

xyz die Menge von s und t $\langle s,t \rangle$

$Pref(s,t) := x$, $pref(s,t) = |Pref(s,t)|$

$Suff(s,t) := z$, $suff(s,t) = |Suff(s,t)|$

Verallgemeinerter Overlap: wie oben, aber (ii) $\overline{ov}(s,t)$

$\overline{ov}(s,t) = |\overline{ov}(s,t)|$

4. String-Algorithmen

4.1. Das String-Matching-Problem

Def. 4.1: Sei Σ ein Alphabet. Das (exakte) String-Matching-Problem ist das folgende Berechnungsproblem:

Eingabe: Zwei Strings $t = t_1 \dots t_n \in \Sigma^n$ (Text) und $p = p_1 \dots p_m \in \Sigma^m$ (Muster)

Ausgabe: die Menge $I \subseteq \{1, \dots, n-m+1\}$, sodass $i \in I$ gdw. $t_i \dots t_{i+m-1} = p$

Algorithmus 4.1.: Naives String-Matching

Eingabe: Muster $p = p_1 \dots p_m$, Text $t = t_1 \dots t_n$

```
1. I = ∅
2. for j := 0 to n-m do
       i := 1
       while p_i = t_{j+i} and i ≤ m do
           i := i+1
       if i = m+1 then
           I = I ∪ {j+1}
```

Ausgabe: I

Laufzeit: $O(m \cdot (n-m))$

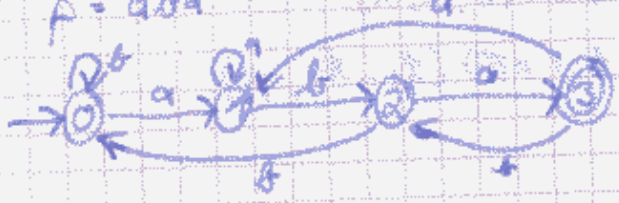
schlechtester Fall: $p = a^m$, $t = a^n$

Verbesserung der Laufzeit: Preprocessing des Musters

4.2. String-Matching-Automaten

Idee: Konstruiere EA, der für gegebene Muster $p_1 \dots p_m$ alle Texte akzeptiert, die mit p enden.

Beispiel: $p = abaa$



Def. 4.4: Sei $p = p_1 \dots p_m \in \Sigma^m$

Der **String-Matching-Automat** für p ist der EA

$$M_p (Q, \Sigma, q_0, \delta, F)$$

mit $Q = \{0, \dots, m\}$

$q_0 = 0$

$F = \{m\}$

$\delta(q, a) = \overline{\sigma\nu}(p_1 \dots p_q a, p)$ für alle $q \in Q, a \in \Sigma$

Satz 4.1: Sei $p = p_1 \dots p_m \in \Sigma^m$, $M_p = (Q, \Sigma, q_0, \delta, F)$ der String-Matching-Automat für p. Sei $t = t_1 \dots t_n \in \Sigma^n$.

Dann gilt: p ist Suffix von $t_1 \dots t_i \iff \hat{\delta}(q_0, t_1 \dots t_i) \in F$

Lemma 4.1: Sei Σ Alphabet, seien $n, m \in \mathbb{N}$, seien $x = x_1 \dots x_n \in \Sigma^n$, $y = y_1 \dots y_m \in \Sigma^m$ und $a \in \Sigma$. Sei $i = \overline{\sigma\nu}(x, y)$. Dann gilt

$$\overline{\sigma\nu}(xa, y) = \overline{\sigma\nu}(y_1 \dots y_i a, y)$$

Beweis: zeige einfacher $\overline{\sigma\nu}(xa, y) \leq \overline{\sigma\nu}(x, y) + 1$

zwei Fälle: 1. $\overline{\sigma\nu}(xa, y)$ ✓

2. $\overline{\sigma\nu}(xa, y) = r > 0 \Rightarrow y_1 \dots y_r$ Suffix von xa

$\Rightarrow y_1 \dots y_{r-1}$ Suffix von x

$\Rightarrow$ (4.1)

Zeige: $i = \overline{\sigma\nu}(x, y)$ folgt $x = x' y_1 \dots y_i$ für ein $x' \in \Sigma^*$

$$\overline{\sigma\nu}(\underbrace{x' y_1 \dots y_i}_{x} a, y) = \overline{\sigma\nu}(y_1 \dots y_i a, y).$$

□

Lemma 4.2: Sei $p = p_1 \dots p_m \in \Sigma^m$, $M_p = (Q, \Sigma, q_0, \delta, F)$ der String-Matching-Automat für p, sei $t = t_1 \dots t_n \in \Sigma^n$.
Dann gilt für alle $i \in [0, \dots, n]$

$$\hat{\delta}(q_0, t_1 \dots t_i) = \overline{\sigma v}(t_1 \dots t_i, p)$$

Beweis: Ind. über i:

$i = 0$ ✓

$i \to i+1$: Sei $q = \hat{\delta}(q_0, t_1 \dots t_i)$, dann gilt

$$\hat{\delta}(q_0, t_1 \dots t_{i+1}) = \delta(q, t_{i+1})$$

$$\overset{\text{Def. 4.4}}{\Longrightarrow} \quad = \overline{\sigma v}(p_1 \dots p_q\, t_{i+1}, p)$$

Ind. vor.: $q = \overline{\sigma v}(t_1 \dots t_i, p)$

$$\Rightarrow \hat{\delta}(q_0, t_1 \dots t_{i+1}) = \overline{\sigma v}(p_1 \dots p_q\, t_{i+1}, p)$$
$$= \overline{\sigma v}(t_1 \dots t_i\, t_{i+1}, p) \qquad \square$$

Beweis von Satz 4.1: Lemma 4.2 sowie $F = \{m\}$ $\square$

Algorithmus 4.2 Konstruktion eines String-Matching-Automaten

Eingabe: $p = p_1 \dots p_m \in \Sigma^m$

```
for q := 0 to m do
    for each a ∈ Σ do
        Berechne δ(q,a) = σv(p_1 ... p_q a, p)
```

Ausgabe: $M_p = (\{0, \dots, m\}, \Sigma, 0, \delta, \{m\})$

Algorithmus 4.3 String-Matching mit endl. Automaten

Eingabe: Text $t = t_1 \dots t_n \in \Sigma^n$ und Muster $p = p_1 \dots p_m \in \Sigma^m$

1. Berechne den String-Matching-Automaten M_p mit Alg. 4.2.
2.
```
q := q_0   I := ∅
for i := 1 to n do
    q := δ(q, t_i)
    if q ∈ F then I := I ∪ {i − m + 1}
```

Laufzeit: Berechnung von H_p: naiv: $O(|\Sigma| \cdot m^2)$

optimal: $O(|\Sigma| \cdot m)$

Schritt 2: $O(n)$

gesamt: $O(n + |\Sigma| \cdot m)$

4.3. Der Boyer-Moore Algorithmus

Idee: Verwende Suche wie im naiven Alg.

zusätzlich zwei Regeln die Vergleiche einsparen

Bad-Character-Regel: Schiebe das Muster bis zum nächsten, rechts stehenden, Vorkommen von t_j mit i im Muster

[Vergleiche $p_1 \dots p_m$ mit $t_{j+1} \dots t_{j+m}$ von hinten nach vorn, $p_i \neq t_{j+i}$ ist erste gefundene nicht übereinstimmende Position]

Zur Implementierung der Bad-Character-Regel:

Bestimme Funktion $\beta: \Sigma \to \{0, \dots, m\}$, die jedem Symbol die jeweils letzte Position in p zuordnet, falls es, 0 sonst

Lemma 4.3: Es gelte $p_{i+1} \dots p_m = t_{j+i+1} \dots t_{j+m}$ und $p_i \neq t_{j+i} = a$

dann kann das Muster für den nächsten Vergleich um $i - \beta(a)$ Positionen nach rechts verschoben werden, ohne ein Vorkommen von p in t zu verpassen.

Beweis: 1) a kommt nicht in p vor: p kann an t_{j+i} vorbeigeschoben werden

$\to$ Änderung von $i - \beta(a) = i$ Pos. möglich

2) $p(a) = k < i$

p von $i - k = i - p(a)$ Positionen verschiebbar

3) $\beta(a) = k > i$: Ignoriere Bad-Character-Regel □

Falls $p_{i+1} \dots p_m = t_{j+1+i} \dots t_{j+m}$ und

$p_i \neq t_{j+i}$, dann verschiebe das Muster um

$m - \gamma(i+1)$ Positionen nach rechts, wobei

$\gamma(i) = \max \{ 0 \le k < m \mid p_i \dots p_m$ ist Suffix von $p_1 \dots p_k$
oder $p_1 \dots p_k$ ist Suffix von $p_i \dots p_m \}$

$\gamma'(i) = \max \{ 0 \le k < m \mid p_i \dots p_m$ ist Suffix von $p_1 \dots p_k \}$

$\gamma''(i) = \max \{ 0 \le k < m \mid p_1 \dots p_k$ ist Suffix von $p_i \dots p_m \}$

Ziel: Berechne $\gamma'(i)$ für alle $i \in \{2, \dots, m\}$

<u>äquivalent</u>: Gegeben $p_1 \dots p_m$, bestimme für alle Suffixe $p_i \dots p_m$ von p für $2 \le i \le m$ das letzte Vorkommen in $p_1 \dots p_{m-1}$.

<u>äquivalent</u>: Gegeben $p_1 \dots p_m$, bestimme für alle Präfixe $p_m \dots p_i$ von $p^R = p_m \dots p_i$ für $2 \le i \le m$ das <u>erste</u> Vorkommen in $p_{m-1} \dots p_1$.

<u>Lemma 4.4</u>: Sei $M_i = (\{0, \dots, m-i+1\}, \Sigma, 0, \delta_i, \{m-i+1\})$ der String-Matching-Automat für $p_m \dots p_i$, sei $t = t_1 \dots t_n$. Dann gilt, falls $t_k \dots t_{k-m+i}$ das erste Vorkommen von $p_m \dots p_i$ in t ist, für alle $1 \le j \le k-m+i$

$$\hat{\delta}_i(q_0, t_1 \dots t_j) = \hat{\delta}_2(q_0, t_1 \dots t_j)$$

<u>Beweis</u>: Direkt aus der Konstruktion des String-Matching-Automaten. □

Idee: Simuliere die Berechnung aller M_i für $2 \le i \le m$ durch eine Berechnung von M_2.

BioInf (7) Speichere die Folge der erreichten Zustände beim Lesen von $p_{m-1} \dots p_1$, sei $q_0 : p_{m-1}, \dots, p_1$ diese Folge.

Dann gilt: $\gamma'(i) = \begin{cases} \max\{j \mid q_j = i\}, & \text{falls der Zustand } i \text{ beim Lesen von } p_{m-1}, \dots, p_1 \text{ erreicht wurde} \\ 0 & \text{sonst} \end{cases}$

Berechnung von $\gamma''(i) = \max\{0 \le k < m \mid p_m \dots p_k \text{ ist Suffix von } p_i \dots p_m\}$:

Wenn M_2 für $p_m \dots p_2$ nach dem Lesen von $p_{m-1} \dots p_1$ im Zustand q_j mit j endet, dann sind die letzten gelesenen Zeichen $p_m \dots p_j$ und j ist minimal mit dieser Eigenschaft:

$\Rightarrow p_m \dots p_j$ ist Suffix von $p_{m-1} \dots p_1$, aber für alle $j' < j$ ist $p_m \dots p_{j'}$ kein Suffix von $p_{m-1} \dots p_1$

$\Rightarrow p_j \dots p_m$ ist Präfix von $p_1 \dots p_{m+1}$ und j ist minimal mit dieser Eigenschaft.

$\Rightarrow \gamma''(i) = m - j + 1$ für alle $2 \le i \le j$

Für alle anderen Werte von i gilt $\gamma''(i) = 0$.

Algorithmus 45: Preprocessing für die Good-Suffix-Regel

Eingabe: Muster $p = p_1 \dots p_m$ über Σ

1. Konstruiere den String-Matching-Automaten $M = (Q, \Sigma, q_0, \delta, f)$ für $p_m \dots p_2$.
2. Bestimme die Zustandsfolge $q_0, q_{m-1}, \dots, q_1$, die M bei der Berechnung auf $p_{m-1}, \dots, p_1$ durchläuft.
3. for $i := 2$ to m do $\gamma'(i) := 0$
   for $j := 1$ to $m-1$ do $\gamma'(m-j+1) := q_j$

4. for $i := 2$ to m do
   if $i \leq q_1$ then
      $\sigma''(i) := m - q_1 + 1$
   else
      $\sigma''(i) := 0$

5. for $i := 2$ to m do $\sigma(i) := \max\{\sigma'(i), \sigma''(i)\}$

Ausgabe: σ

Algorithmus 4.6. Boyer - Moore - Algorithmus

Eingabe: Muster $p = p_1 \dots p_m$ und Text $t = t_1 \dots t_n$ über Σ

1. Berechne β für die Bad-Character-Regel
2. Berechne σ für die Good-Suffix-Regel
3. $I := \emptyset$
   $j := 0$
   while $j \leq n - m$ do
      $i := m$
      while $p_i = t_{j+i}$ and $i > 0$ do
         $i := i - 1$
      if $i = 0$ then $I := I \cup \{j\}$
      $j := j + \max\{i - \beta(t_{j+i}), m - \sigma(i+1)\}$

Ausgabe: I

Laufzeit: worst-case: $O(n \cdot m)$

aber: schlechtester Fall tritt sehr selten auf.

4.4. Suffix-Bäume

Def. 4.6. Sei $t = t_1 \dots t_n \in \Sigma^n$ ein Text.
Ein gerichteter Baum $T_t = (V, E)$ mit einer Wurzel r heißt **einfacher Suffix-Baum für t**, falls gilt:

(1.) T_t hat genau n Blätter, die mit $1, \dots, n$ beschriftet sind.
(2.) Die Kanten von T_t sind mit Symbolen aus Σ beschriftet.
(3.) Alle von einem inneren Knoten ausgehenden Kanten zu seinen Kindern sind mit paarweise verschiedenen Symbolen beschriftet.
(4.) Der Pfad von der Wurzel r zu dem Blatt i trägt die Beschriftung $t_i \dots t_n$ (als Konkatenation der Kantenbeschriftungen auf dem Pfad)

Frage: Existiert für jeden Text t ein einfacher Suffix-Baum?
Nein, existiert nicht, wenn ein Suffix von t gleichzeitig der Präfix eines anderen Suffixes ist.

Abhilfe: Hänge an t ein neues Symbol $\$ \notin \Sigma$ an und konstruiere den einfachen Suffix-Baum für $t\$$.

Algorithmus 4.7.: Konstruktion eines einfachen Suffix-Baums:

Eingabe: $t = t_1 \dots t_n \in \Sigma^n$

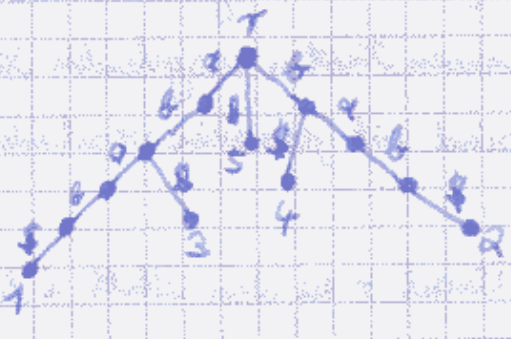
1. $t' := t\$$ für $\$ \notin \Sigma$.
2. Initialisiere $T_{t'}$ mit Wurzel r und leerer Kantenmenge
3. **for** $i = 1$ **to** n **do**
   (a) Suche von r ausgehend einen Pfad in $T_{t'}$, der mit einem maximalen Präfix $t_i \dots t_{j_i}$ beschriftet ist und in Knoten x_i endet.
   (b) Füge dem Baum einen mit $t_{j_i+1} \dots t_n\$$ beschrifteten Pfad $x_i, v_{i,j_i+1}, \dots, v_{i,n}, v_{i,n+1}$ hinzu, wobei

Beispiel ⑩ $y_{i_{j+1}}, \dots, y_{i_{n+1}}$ neue Knoten sind.

(c) Beschrifte den neuen Blattknoten $y_{i_{n+1}}$ mit i.

Ausgabe: $T_{t'}$

Beispiel: $t = abab$, $t' = abab\$$



Satz 4.3: Sei $t = t_1 \dots t_n \in \Sigma^+$, sei $\$ \notin \Sigma$.
Dann konstruiert Alg. 4.3 einen einfachen Suffix-Baum für $t\$$.

Beweis zu ①: Da die Suffixe nach fallender Länge geordnet in den Baum eingefügt werden, ist der Knoten x_i kein Blatt.
Da kein Suffix von $t\$$ ein Präfix eines anderen Suffixes sein kann, ist der im Schritt i eingefügte Pfad nicht leer.
$\Rightarrow$ der Baum hat genau $n+1$ Blätter.

zu ②: ✓ (für das Alphabet $\Sigma' = \Sigma \cup \{\$\}$)

zu ③: folgt, da der Algorithmus im i-ten Schritt zunächst den Pfad maximaler Länge sucht, der mit einem Präfix von $t_i \dots t_n\$$ beschriftet ist.

zu ④: klar nach Konstruktion des Algorithmus. □

Lösung des String-Matching-Problems mit Suffix-Bäumen:

Starte in der Wurzel und suche Pfad in T_t mit der Beschriftung p.

Falls dieser existiert, ist dieser Pfad eindeutig.

$\rightarrow$ p ist Präfix eines Suffixes von t

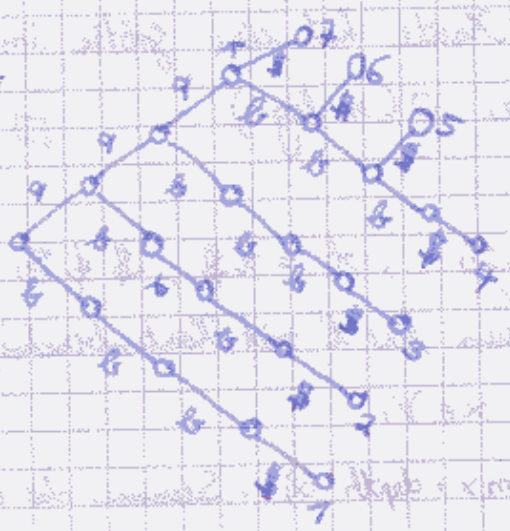
Jedes der Blätter in T_t, die unter dem Endknoten dieses Pfades hängen, entspricht einem Vorkommen von p in t.

Falls in T_t kein Pfad mit der Beschriftung p vorkommt, dann ist p kein Teilstring von t.

Problem: Der einfache Suffixe-Baum für einen Text $t = t_1 \dots t_n$ kann eine Größe in $\Omega(|t|^2 \cdot |\Sigma|)$ erreichen.

Beispiel: $t_n = a^n b^n \$$

$n = 3$:



Def. 4.7. Sei $t = t_1 \dots t_n \in \Sigma^n$ ein Text. Ein gerichteter Baum $T_t = (V, E)$ mit Wurzel r heißt kompakter Suffix-Baum für t, wenn gilt: 12.5.2003

1. Der Baum hat genau n Blätter, die mit $1, \dots, n$ beschriftet sind.
2. Jeder innere Knoten von T_t hat mindestens zwei Kinder.
3. Die Kanten sind mit Teilstrings von t beschriftet. Dabei wird jeder Teilstring der Länge k dargestellt durch seine Anfangs- und Endposition in t, falls $k \geq 2 \cdot \log_2(n \cdot |\Sigma|)$

4. Alle Beschriftungen von einem inneren Knoten aus beginnen mit paarweise verschiedenen Symbolen.
5. Der Pfad von r zu dem Blatt i trägt die Beschriftung $t_i \dots t_n$.

Lemma 4.5. Sei $t = t_1 \dots t_n \in \Sigma^n$. Ein kompakter Suffix-Baum für t hat eine Größe in $O(n \cdot \log n)$

Beweis: Jeder Suffix-Baum für t hat genau n Blätter.
Da jeder innere Knoten mindestens zwei Kinder hat, hat der Suffix-Baum höchstens $n-1$ innere Knoten haben.
$\Rightarrow$ insgesamt $\leq 2n-1$ Knoten
$\Rightarrow$ $\leq 2n-2$ Kanten
Beschriftung jeder Kante hat eine Größe in $O(\log n)$
$\Rightarrow$ Gesamtgröße $O(n \cdot \log n)$ □

Def. 4.6. Sei $t = t_1 \dots t_n \in \Sigma^n$, $t' = t\$$, $\$ \notin \Sigma$. Sei $T_{t'} = (V, E)$ ein einfacher (oder kompakter) Suffix-Baum für t' mit Kantenbeschriftung $label: E \to \Sigma^*$
Für jeden Knoten $x \in V$:

Stringtiefe von x - depth (x) : Summe der Längen der Kantenbeschriftungen auf dem Pfad von der Wurzel zu x

pathlabel (x) : Beschriftung auf dem Pfad von der Wurzel zu x.

Falls $T_{t'}$ einfacher Suffix-Baum:

Pos (x) : Minimale Beschriftung eines Blattes in dem Teilbaum mit Wurzel x.

BioInf (R3) Für eine Kante (v,x) gibt $Pos(x)+depth(v)$ die erste Position in t, an der die Beschriftung von (v,x) vorkommt.

Algorithmus 4.8 Konstruktion eines kompakten Suffix-Baums

Eingabe: String $t = t_1 \dots t_n \in \Sigma^n$

1. $t' := t\$$ mit $\$ \notin \Sigma$
   Berechne einfachen Suffix-Baum $T_{t'} = (V, E)$ mit Kantenbeschriftung label: $E \to \Sigma$ für t'.

2. Eliminiere die Knoten vom Grad 2:
   $X := \{ v \in V \mid v$ hat genau einen Nachfolger $\}$
   while $X \neq \emptyset$ **do**
   Wähle $x \in X$, sei y der Vater von x, sei z das Kind von x. Ersetze die Kanten (y,x) und (x,z) durch (y,z) mit der Beschriftung $label(y,z) = label(y,x)\,label(x,z)$
   Lösche x.

3. Komprimiere lange Kantenbeschriftungen:
   for all $e = (x,y) \in E$ **do**
   if $|label(e)| \geq 2 \cdot \log_2(n - |\Sigma|)$ **then**
   $label'(e) := [\,Pos(y) + depth(x),\ Pos(y) + depth(x) + |label(e)| - 1\,]$
   else $label'(e) := label(e)$

Ausgabe: Der konstruierte kompakte Suffix-Baum mit Kantenbeschriftung $label'$.

Algorithmus 4.3: String-Matching mit kompaktem Suffix-Baum.

Eingabe: Muster $p = p_1 \dots p_m$, Text $t = t_1 \dots t_n$ über Σ

1. Konstruiere kompakten Suffix-Baum $T_{t'}$ für $t' = t\$$ mit Wurzel r und Kantenbeschriftung label'.

2. $x := r$
   $i := 1$
   gefunden := false
   möglich := true
   while gefunden = false **and** möglich = true **do**
   | passende_Kante := false ; U = Menge der Kinder von x
   | **while** passende_Kante = false **and** $U \neq \emptyset$ **do**
   | | Wähle $v \in U$
   | | **if** label'$(x,v) = p_i \alpha$ für $\alpha \in \Sigma \cup \{\$\}$ **then**
   | | | passende_Kante := true
   | | | label := label'(x,v)
   | | **else if** label'$(x,v) = [k \dots l]$ **and** $t_k = p_i$ **then**
   | | | passende_Kante := true
   | | | $l' := \min\{l, k+m-i\}$
   | | | label := $t_k \dots t_{l'}$
   | | **else**
   | | | $U := U - \{v\}$
   | **if** ($p_i \dots p_m$ kein Präfix von label) **and** (label kein Präfix von $p_i \dots p_m$) **then**
   | | möglich := false
   | **else if** label ist Präfix von $p_i \dots p_m$ **then**
   | | $x := v$
   | | $i := i + |\text{label}|$
   | **else**
   | | $x := v$
   | | gefunden := true
   if gefunden = true **then**

BioInf (15) Bestimme die Menge I der Blattbeschriftungen in dem Teilbaum mit Wurzel x

Ausgabe: I

Satz 4.3: Algorithmus 4.3. löst das String-Matching-Problem für Text $t = t_1 \dots t_n$ und Muster $P = P_1 \dots P_m$ über Σ_t in einer Zeit in $O(n \log n + m \cdot |\Sigma| + h)$, wobei h die Anzahl der Vorkommen von P in t und $|\Sigma| \leq \frac{n}{c}$ gelte für eine Konstante $c \geq 2$.

Beweis: Korrektheit: ✓

Zeitkomplexität: Konstruktion des kompakten Suffix-Baums: $O(n \cdot \log n)$

Bestimmung des richtigen Kindes des aktuellen Knotens x:

Kantenbeschriftung der Form $[a \dots b]$ kodiert Teilstring der Länge $\in O(\log n)$, da $|\Sigma| \leq \frac{n}{c}$ und damit $2 \cdot \log_c (n - |\Sigma|) \in O(\log n)$.

$\Rightarrow$ Beim Lesen von P höchstens $O(\frac{m}{\log n})$ solcher komprimierter Kantenbeschriftungen auf einem Pfad von der Wurzel

Lesen einer Kantenbeschriftung: $O(\log n)$

Jeder aktuelle Knoten hat $\leq |\Sigma|$ Kinder, die ggf. alle überprüft werden müssen.

Teile die Kinder von x in solche Teilfolge aus, daß zunächst alle nicht-komprimierten Kanten überprüft werden.

$\Rightarrow$ nur in den Schritten, in denen auch die passende Kante eine komprimierte Beschriftung besitzt, werden komprimierte Beschriftungen gelesen.

$\Rightarrow$ Lesen der komprimierten Kantenbeschriftungen insgesamt:

$$O(\tfrac{m}{\log n} \cdot \log n \cdot |\Sigma|) = O(|\Sigma| \cdot m)$$

Lesen der nicht-komprimierten Kantenbeschriftungen insgesamt:

$O(m \cdot |\Sigma|)$

Durchsuchen des Teilbaums: $O(k)$. □

4.5 Weitere Anwendungen von Suffix-Bäumen

4.5.1. Verallgemeinerte Suffix-Bäume und das Teilstring-Problem

Def. 4.5.: Das Teilstring-Problem ist das folgende Berechnungsproblem:

Eingabe: Ein Muster P und N Texte $t_1, \dots, t_N$ über Alphabet Σ

Ausgabe: Eine Menge $I \subseteq \{1, \dots, N\}$, so daß $i \in I$ gdw. P ein Teilstring von t_i.

Idee: Verwende Suffix-Baum, der die Suffixe von $t_1, \dots, t_N$ enthält.
Konstruiere Suffix-Baum für $t_1 \$_1 t_2 \$_2 \dots t_N \$_N$, wobei $\$_i \notin \Sigma$.

14.5.2003

Lemma 4.6: Seien $t_1, \dots, t_N$ über Σ gegeben. Sei T der kompakte (nicht komprimierte) Suffix-Baum für $t' = t_1 \$_1 t_2 \$_2 \dots t_N \$_N$, wobei $\$_1, \dots, \$_N \notin \Sigma$, $\$_1, \dots, \$_N$ paarweise verschieden.

Dann treten die Trennsymbole $\$_i$, $1 \le i \le N$, in T nur in den Beschriftungen von Knoten auf, die unmittelbar zu den Blättern sind.

Beweis: Annahme: $\$_i$ liegt auf einer Kante zwischen den inneren Knoten x und y, wobei x der Vater von y sei.

$\Rightarrow$ Es ex. zwei verschiedene Suffixe $w\$_i u$ und $w\$_i v$, da y mindestens zwei Kinder hat.

Widerspruch, da $\$_i$ genau einmal in t' vorkommt. □

Def. 4.10. Seien $t_1, \dots, t_N$ über Σ gegeben, seien $\$_1, \dots, \$_N \notin \Sigma$ paarweise verschiedene Trennsymbole. Ein **verallgemeinerter Suffix-Baum für $t_1 \dots t_N$** entsteht aus einem kompakten, nicht komprimierten Suffix-Baum für $t' = t_1\$_1 \dots t_N\$_N$ durch die folgenden Schritte:

1. Ersetze jede Kantenbeschriftung der Form $u\$_i w$, $w \in (\Sigma \cup \{\$_j \mid 1 \le j \le N\})^*$ durch $u\$_i$.
2. Beschrifte jedes Blatt mit einem Paar (i,j) aus dem Index des zugehörigen Textes t_i und der Startposition j des zugehörigen Suffixes.
   i: Index des Trennsymbols auf der inzidenten Kante.
   j: an der Blattbeschriftung des kompakten Suffix-Baums für t und den Längen der Texte.
3. Komprimiere lange Kantenbeschriftungen.

Satz 4.11: Seien ein Muster $p = p_1 \dots p_n$ und N Texte $t_1 \dots t_N$ der Gesamtlänge n über Σ gegeben.

Dann läßt sich das Teilstring-Problem in linearer Zeit in $O(n \log n + m \cdot |\Sigma| + k)$ lösen, wobei k die Anzahl des Vorkommen von p in $t_1, \dots, t_N$ sei, und $|\Sigma| \le \frac{n}{c}$ für ein $c \ge 2$.

Beweis: analog zu Satz 4.3:

Konstruktion des verallgemeinerten Suffix-Baums: $O(n \cdot \log n)$

Suche des Musters p: $O(m \cdot |\Sigma|)$

Durchsuchen des Teilbaums: $O(k)$ □

Bioinf (R) 4.5.2. längste gemeinsame Teilstrings

Def. 4.11.: Das Longest-Common-Substring-Problem ist das folgende Optimierungsproblem:

Eingabe: Menge $M = \{t_1, \dots, t_N\}$ von Strings über Σ

zulässige Lösungen: Jeder String t, der Teilstring von t_i ist für alle $1 \le i \le N$

Kosten: Für zulässige Lösung t: $cost(t) = |t|$

Optimierungsziel: Maximierung

Algorithmus 4.10: Bestimmung des längsten gemeinsamen Teilstrings

Eingabe: Eine Folge $t_1, \dots, t_N$ von Strings über Σ

1. Konstruiere verallgemeinerten Suffix-Baum T für $t_1, \dots, t_N$
2. Beschrifte jeden inneren Knoten x mit einer Menge $M(x) \subseteq \{1, \dots, N\}$, so daß $i \in M(x)$ gdw. in dem Teilbaum mit Wurzel x ein Blatt mit Beschriftung (i, j) ex. für bel. j.
3. Finde unter allen inneren Knoten von T mit der Beschriftung $\{1, \dots, N\}$ einen Knoten x_{max} maximaler Stringtiefe
4. $d_{max} := pathlabel(x_{max})$

Ausgabe: d_{max}

Satz 4.5: Alg. 4.10. bestimmt den längsten gemeinsamen Teilstring von N Strings $t_1, \dots, t_N$ der Gesamtlänge n in einer Zeit in $O(n(\log n + N^2 N \cdot |\Sigma|))$

Beweis: Korrektheit: ✓

Zeitkomplexität: Konstruktion des verallgemeinerten Suffix-Baums $O(n \cdot \log n)$

Bestimmung der Mengen $M(x)$: bottom-up, von den Blättern aus:

BioInf (19) Beschriftung für Knoten x ergibt sich als Vereinigung der Mengen seiner Kinder. Jeder Knoten hat höchstens $|\Sigma|+N$ Kinder, falls v Vater eines Blattes ist, $|\Sigma|$ Kinder sonst, es gibt $O(n)$ Knoten

$\Rightarrow$ Gesamtaufwand: $O(n \cdot (|\Sigma|+N) \cdot N)$

Für jeden Knoten x speichere, ob $M(x) = \{1, \dots, N\}$ oder nicht.

$\Rightarrow$ Für jeden Knoten ist die Überprüfung der Bedingung in konstanter Zeit möglich.

$\Rightarrow$ Finden aller Knoten mit Beschriftung $\{1, \dots, N\}$ in $O(n \cdot \log n)$

Bestimmung von x_{max} und d_{max}: $O(n \cdot \log n)$

4.5.3. Effiziente Bestimmung von Overlaps

Aufgabe: Bestimmung aller paarweisen Overlaps von N Strings

Laufzeit: naiv (falls alle Strings die Länge N haben):

$$O(N^2 \cdot (\tfrac{n}{N})^2) = O(n^2)$$

Ziel: mit verallgemeinertem Suffix-Baumen: $O(n \cdot (\log n + |\Sigma| + N))$

Algorithmus 4.11.: Bestimmung aller paarweisen Overlaps

Eingabe: Eine Folge $t_1, \dots, t_N$ von N Strings über Σ

1. Konstruiere verallgemeinerten Suffix-Baum T für $t_1, \dots, t_N$ mit Wurzel r
2. Beschrifte jeden inneren Knoten x mit $L(x) \subseteq \{1, \dots, N\}$, so daß $i \in L(x)$ gdw. x inzident ist zu einer Kante mit Beschriftung $\$_i$

3. for $j := 1$ to N do

$x := r$

while x ist kein Blatt do

for all $i \in L(x)$ do

if $depth(x) < \min\{|t_i|, |t_j|\}$ then

$u(t_i, t_j) := pathlabel(x)$

$w(t_i, t_j) := depth(x)$

$x := $ Kind von x auf dem Pfad mit Beschriftung $t_j\$_j$

Ausgabe: Die Overlaps $O_v(t_i, t_j) = u(t_i, t_j)$ und ihrer Längen $o_v(t_i, t_j) = w(t_i, t_j)$ für alle $1 \le i, j \le N$

Satz 4.6: Seien N Strings $t_1, \dots, t_N$ der Gesamtlänge n über Σ gegeben. Dann bestimmt Alg. 4.11 alle paarweisen Overlaps in einer Zeit in $O(n \cdot (\log n + |\Sigma| + N))$

Beweis: Korrektheit: Falls $i \in L(x)$ für einen Knoten x, dann ist pathlabel(x) ein Suffix von t_i.

Gleichzeitig ist pathlabel(x) ist Präfix von t_j, falls x auf dem Pfad mit Beschriftung $t_j\$_j$ liegt.

$\Rightarrow$ pathlabel(x) ist Überlappung von t_i und t_j

Alle möglichen Überlappungen von t_i und t_j kommen vor als pathlabel(y) für einen Knoten y.

$\Rightarrow O_v(t_i, t_j) =$ pathlabel(y) für den tiefsten Knoten y auf dem Pfad mit Beschriftung $t_j\$_j$, dessen L-Menge den Index i enthält.

Bedingung $depth(x) < \min\{|t_i|, |t_j|\}$ stellt sicher, daß keine Überlappung berücksichtigt wird, bei der t_i Suffix von t_j ist

oder umgekehrt.

Zeitkomplexität: Konstruktion des Suffix-Baums: $O(n \cdot \log n)$

Bestimmung von $L(x)$ $\forall x$: $O(n \cdot N)$

Gesamtlänge aller untersuchten Pfade: $O(n)$

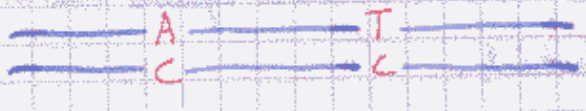
Summe der Mächtigkeiten der dabei untersuchten L-Mengen: $O(n)$

Berechnung der Pfade insgesamt $O(n \cdot (|\Sigma| + N))$

$\Rightarrow$ Insgesamt: $O(n \cdot (\log n + |\Sigma| + N))$ □

15.5.2003

5. Alignment-Verfahren



5.1. Alignment von zwei Strings

Beispiel 5.1

$S = GACGGATTATG$

$t = GATCGGAATAG$

$s' = GA-CGGATTATG$

$t' = GATCGGA\#ATA-G$

Def. 5.1: Seien $s = s_1 \ldots s_m$, $t = t_1 \ldots t_n$ zwei Strings über Σ. Sei $- \notin \Sigma$ ein spezielles Lückensymbol und $\Sigma' = \Sigma \cup \{-\}$.

$h: (\Sigma')^* \to \Sigma^*$, $h(a) = a$ $\forall a \in \Sigma$, $h(-) = \lambda$

Ein Alignment von s und t der Länge $l = \max\{m, n\}$ über Σ' ist ein Paar (s', t') für die gilt:

i) $|s'| = |t'| = l \geq \max\{m, n\}$

ii) $h(s') = s$ iii) $h(t') = t$

iv) Es gibt keine Position in der sowohl in s' als auch in t'

eine Lücke auftritt

$\forall i \in [1, \dots, L]$ $t_i' \neq -$ oder $s_i' \neq -$

4 Typen von Spalten in einem Alignment

1) Einfügung / Insertion: Der erste (obere) String besitzt in dieser Spalte eine Lücke.

2) Löschung / Deletion: Der zweite (untere) String besitzt in dieser Spalte eine Lücke.

3) Match: Die Buchstaben in dieser Spalte des Alignments sind identisch

4) Mismatch: Die Buchstaben in dieser Spalte des Alignments sind nicht identisch.

Bewertung eines Alignments

Def. 5.2: Zwei Strings s, t über Σ

Sei $p(a,b) \in \mathbb{Q}$ $\forall a, b \in \Sigma$

$g \in \mathbb{Q}$

Definiere: die Bewertung eines Alignments von s und t sei spaltenweise definiert

$\forall x, y \in \Sigma$: $\delta(x,y) = p(x,y)$

$\delta(x,-) = \delta(-,y) = g$

s', t', (s', t') Alignment von s und t

$$\delta(s', t') = \sum_{i=1}^{l} \delta(s_i', t_i')$$

Außerdem sei durch goal $\in \{\min, \max\}$ ein Optimierungsziel gegeben. Im allgemeinen sollte: $p(a,b) = p(b,a)$

Def. 5.3: Seien $s,t \in \Sigma_1$, δ eine Alignment-Bewertung, goal σ ein Optimierungsziel für δ.

Die Ähnlichkeit (Similarity) $sim_\delta(s,t)$ von s und t ist die Bewertung eines optimalen Alignments (s',t') von s und t, d.h.

$$sim_\delta(s,t) = goal_\sigma \{ \delta(s',t') \mid (s',t') \text{ Alignment von } s \text{ und } t \}$$

Übliche Bewertungsfunktionen

Edit-Distanz / Levenshtein-Distanz

$p(a,b) = 1 \quad a \neq b$

$p(a,a) = 0$

$g = 1$

goal = min

anderes Maß:

$p(a,a) = 1$

$p(a,b) = -1 \quad a \neq b$

$g = -2$

goal = max

5.12 Globale Alignments

„Ähnlichkeiten der gesamten Strings"

formal: Def. 5.4 Global-Alignment-Problem

Eingabe: s,t über Σ_1, δ eine Bewertungsfunktion, $goal_\sigma$ Optimierungsziel für δ.

zulässige Lösungen: Alle Alignments von t und s.

Kosten: $\forall$ Alignments $A = (s',t')$

$cost(A) = \delta(A)$

z.B. $goal = \frac{}{}$ im folgenden „Maximierung"

```
A T G A - - - -        -16        8
- - - - A T G A
```

```
A T G A
A T G A
```

4 (g = -2, Match = 1, Mismatch = -1) — Max.

0 (g = 1, Match = 0, Mismatch = 1) — Min.

dynamische Programmierung:

- Zerlegung eines Problems in Teilprobleme
- Setzen die Lösungen der Teilprobleme bottom-up zusammen.
- Lösungen für Teilprobleme werden zwischengespeichert

(Beispiel: Fibonacci-Zahlen)

dynamische Programmierung für das Global-Alignment-Problem

$s = s_1 \dots s_m$

$t = t_1 \dots t_n$

$(m+1) \times (n+1)$ Präfixpaare

→ Matrix M aus $(m+1) \times (n+1)$ Einträgen

$M(i,j)$: ist die Bewertung des optimalen Alignments für $s_1 \dots s_i$ und $t_1 \dots t_j$

$M(m,n)$: Bewertung des optimalen Alignments für s, t

M: Ähnlichkeitsmatrix

$s_1 \dots s_i \overset{\lambda}{\sim} \left(\begin{smallmatrix} s_1 \dots s_i \\ - \dots - \end{smallmatrix}\right)$: $M(i,0) = i \cdot g$

$t_1 \dots t_j \overset{\lambda}{\sim} \left(\begin{smallmatrix} - \dots - \\ t_1 \dots t_j \end{smallmatrix}\right)$: $M(0,j) = j \cdot g$

Initialisierung von M:

Initialisiere die erste Spalte und erste Zeile von M

mit $M(i,0) = i \cdot g \quad \forall i \in \{1,\dots,m\}$

$M(0,j) = j \cdot g \quad \forall j \in \{1,\dots,n\}$

Idee:

Rekurrenz:

$$M(i,j) = \begin{cases} M(i-1,j) + g & \text{Einfügung} \\ M(i,j-1) + g & \text{Löschung} \\ M(i-1,j-1) + p(s_i, t_j) & \text{Match/Mismatch} \end{cases}$$

Algorithmus 5.1 - Bestimmung der Ähnlichkeit

Eingabe: $S, t, \quad S = s_1 \dots s_m, \quad t = t_1 \dots t_n$

1.) Initialisierung:

```
for i=0 to m do
    for j=0 to n do
        M(i,j) := 0
```

2.) Initialisierung der Ränder:

```
for i=0 to m do
    M(i,0) := i·g
for j=0 to n do
    M(0,j) := j·g
```

3.) Auffüllen der Matrix:

for $i=1$ to n do
  for $j=1$ to n do
    $M(i,j) := \max\{ M(i-1,j)+g,$
    $M(i,j-1)+g,$
    $M(i-1,j-1)+p(s_i,t_j)\}$

Ausgabe: $sim(s,t) = M(m,n)$

Satz 5.1: Seien zwei Strings $s = s_1 \dots s_m$ und $t = t_1 \dots t_n$ über Σ gegeben. Dann läßt sich mit dem Algorithmus 5.1 (Berechnung der Ähnlichkeit) und 5.2 (Trace-Back, siehe Folie) ein optimales Alignment von s und t in Zeit $O(n \cdot m)$ bestimmen.

Beweis: Alg. 5.1: Auffüllen der Matrix $O(n \cdot m)$
Alg. 5.2: Durchlauf durch M: $O(n+m)$

Insgesamt: $O(n \cdot m)$

Berechnung aller möglichen optimalen Alignments

⇒ es können exponentiell viele optimale Alignments existieren.

z.B. $s = A^{2n}$, $t = A^n$

```
A A A A A A      A A A A A A
- A A A - -      A - - - A A
```

allg. $\binom{2n}{n}$ exponentiell viele

graphentheoretisches Problem:

Def. 5.5: $s = s_1 \dots s_m$, $t = t_1 \dots t_n$ Strings über Σ

δ Alignmentbewertung mit Parametern p und g.

ist

der Edit-Graph von s und t bezgl. δ definiert durch

$$G_g(s,t) = (V, E, c)$$

$$V := \{0,\dots,m\} \times \{0,\dots,n\}$$

$$E := \{((i,j),(i,j+1)) \mid \forall i,j\}$$
$$\cup \{((i,j),(i+1,j)) \mid \forall i,j\}$$
$$\cup \{((i,j),(i+1,j+1)) \mid \forall i,j\}$$

$$c : E \to \mathbb{Q}$$

$$c((i,j),(i,j+1)) = c((i,j),(i+1,j)) = g$$
$$c((i,j),(i+1,j+1)) = p(s_{i+1}, t_{j+1}) \quad \bullet \forall i,j$$

Lösung des Global-Alignment-Problem auf Basis des Edit-Graphen
- finde den längsten Weg von dem Knoten $(0,0)$ zum Knoten (m,n).
(Polynomielle Laufzeit)

21.5.2003

5.1.3. Lokales und Semiglobales Alignment

Lokales Alignment

Def. 5.6: Seien zwei Strings $s = s_1 \dots s_m$ und $t = t_1 \dots t_n$ über Σ gegeben, und sei δ eine Alignment-Bewertung mit Optimierungsziel Maximierung.

Ein lokales Alignment von s und t ist ein (globales) Alignment von Teilstrings $\bar{s} = s_{i_1} \dots s_{i_2}$ und $\bar{t} = t_{j_1} \dots t_{j_2}$.

Ein Alignment $A = (\bar{s}', \bar{t}')$ von $\bar{s}$ und $\bar{t}$ ist ein optimales lokales Alignment falls gilt:

$\delta(A) = \max\{ sim(\bar{s}, \bar{t}) \mid \bar{s}$ ist Teilstring von s, $\bar{t}$ ist Teilstring von $t\}$

Def. 5.7: das Local-Alignment-Problem ist das folgende Optimierungsproblem:

Eingabe: Zwei Strings s und t über Σ und Alignment-Bewertung δ mit Optimierungsziel Maximierung.

Zulässige Lösungen: Alle lokalen Alignments von s und t, d.h. alle globalen Alignments für alle möglichen Teilstrings $\bar{s}$ von s und $\bar{t}$ von t

Kosten: Für ein lokales Alignment $A = (\bar{s}, \bar{t})$ gilt $cost(A) = \delta(A)$

Optimierungsziel: Maximierung

Beispiel: $s = AAAAACTCTCTCT$

$t = GCGCGCGCAAAAA$

opt. lokales Alignment: $\overline{AAAAA}$ (CTCTCTCT)

(GCGCGCGC) $\overline{AAAAA}$

Berechnung eines opt. lokalen Alignments: dynamische Programmierung

Berechne $(n+1) \times (n+1)$-Matrix M

$M(i,j)$ = höchste Bewertung eines Alignments zwischen einem Suffix von $s_1 \dots s_i$ und einem Suffix von $t_1 \dots t_j$.

$$M(i,j) = \max \begin{cases} M(i-1, j) + \delta \\ M(i, j-1) + \delta \\ M(i-1, j-1) + p(s_i, t_j) \\ 0 \end{cases}$$

Initialisierung: Zeile 0 und Spalte 0 mit Nullen initialisieren

Bewertung eines opt. lokalen Alignments: höchster in M vorkommender Wert

Bestimmung des opt. lokalen Alignments: Traceback von höchstem Wert in M

Semiglobale Alignments

Beispiel 5.6.: $s = ACTTTATGCCTGCT$

$t = ACAGGCT$

δ: Match +1, Mismatch −1, Lücke −2

opt. globales Alignment: ACTTTATGCCTGCT
AC--A-G---GCT

Problem: t wird stark zerstückelt

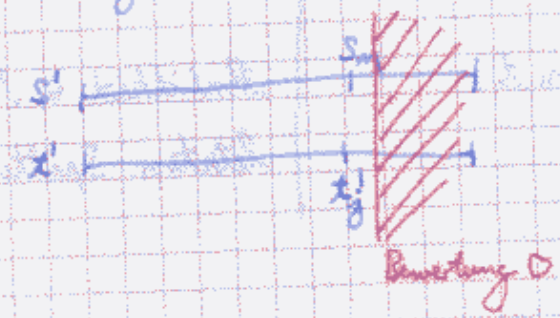
opt. lokales Alignment: ACTTTATGCCTGCT
(ACAG)GCT

„besseres Alignment": ACTTTAT-GCCTGCT
----ACAGG-CT---

Lücken vor t_1 und hinter t_n ignorieren: Bewertung 0
(statt −13 bei glob. Alignment)

1. Kostenlose Lücken am Ende von s:

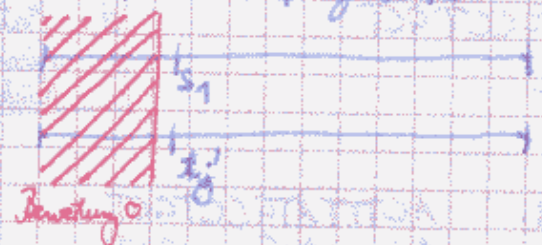
Sei (s', t') ein Alignment von s und t der Länge l, bei dem $t_j' = s_m$ gilt für ein $j < l$:



⇒ Bewertung eines globalen Alignments von s mit dem Präfix $t_1' \ldots t_j'$ von t

⇒ In der Alignment-Matrix werden in der letzten Zeile alle Bewertungen von Alignments von s mit allen Präfixen von t berechnet → Suche Maximum der letzten Zeile.

2. Kostenlose Lücken am Anfang von s:



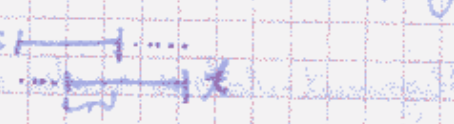
⇒ Berechne opt. globales Alignment von s mit Suffix von t

⇒ Initialisiere die erste Zeile der Matrix mit Nullen

Kostenlose Lückensymbole	Änderung des Algorithmus S.1
am Anfang von s	Initialisierung der ersten Zeile von M mit Nullen
am Ende von s	Ähnlichkeit ≙ Maximum der letzten Zeile
am Anfang von t	Initialisierung der ersten Spalte von M mit Nullen
am Ende von t	Ähnlichkeit ≙ Maximum der letzten Spalte

Beispiel: Bestimmung von überlappenden Overlaps

→ mache Lücken am Ende von s und am Anfang von t kostenlos:



5.1.4. Verallgemeinerte Bewertungsfunktion

Bewertung von Lücken:

Def. 5.5: Seien s und t zwei Strings, sei (s', t') ein Alignment von s und t.

Einen Teilstring $s'_{i+1} \dots s'_{i+k} = -^k$ mit $s'_i, s'_{i+k+1} \neq -$
bzw. einen Teilstring $t'_{j+1} \dots t'_{j+k} = -^k$ mit $t'_j, t'_{j+k+1} \neq -$
bezeichnen wir als **Lücke der Länge k**.

Bsp.: Bewertung einer Lücke der Länge k: $k \cdot g$

affine Lückenbewertung: Bewertung einer Lücke der Länge k mit

$$-(\rho + \sigma \cdot k), \quad \rho, \sigma \geq 0$$

$\Rightarrow$ Anteil $\sigma \cdot k$ proportional zur Länge der Lücke
zusätzlich Kosten für das Öffnen einer Lücke

Berechnung eines opt. Alignments mit affiner Lückenbewertung: siehe Übung

Bewertungsmatrizen:

Motivation: Vergleich von Protein-Sequenzen:
Substitution zwischen bestimmten Aminosäuren sind
wahrscheinlicher als zwischen anderen.

Konkret eine 20×20-Matrix von Zahlen, $p(a, b)$: **Bewertungsmatrix**

PAM-Matrizen:

PAM = „percent of accepted mutations"

Def. 53: Eine akzeptierte Mutation ist eine Mutation, die die Funktionsweise des Proteins nicht oder nur so wenig ändert, daß sie weiter verwirkt werden kann.

Zwei Protein-Sequenzen s und t sind eine PAM-Einheit voneinander entfernt, wenn s in t überführt wurde durch eine Folge von akzeptierten Punkt-Mutationen (Substitutionen einzelner Aminosäuren, keine Einfügungen oder Löschungen), so daß durchschnittlich eine akzeptierte Punkt-Mutation pro hundert Aminosäuren auftritt.

Beachte: Zwei Protein-Sequenzen, die k PAM-Einheiten entfernt sind, unterscheiden sich nicht notwendig an k Prozent der Stellen, da mehrere Mutationen an der gleichen Stelle auftreten können.

Ziel: Konstruiere k-PAM-Matrix für den Vergleich von Protein-Sequenzen, die k PAM-Einheiten voneinander entfernt sind.

Bestimmung einer k-PAM-Matrix im idealen Fall:

Annahmen: - Wir kennen viele Paare von homologen Protein-Sequenzen von denen wir wissen, daß sie k PAM-Einheiten voneinander entfernt sind.

- Wir kennen für jedes Paar das opt. Alignment, also die Positionen der Lücken.

Sei A die Menge aller Alignments der geg. Sequenz-Paare, sei $Sp(A)$ die Multimenge aller Spalten in A.

$$freq(a_i, a_j) = \frac{\text{Anzahl der Vorkommen von } (a_i, a_j) \text{ und } (a_j, a_i) \text{ in } \mathcal{P}(A)}{2 \cdot |\mathcal{P}(A)|}$$

$$freq(a_i) = \frac{\text{Anzahl der Vorkommen von } a_i \text{ in allen Alignments}}{\text{Gesamtlänge aller Sequenzen}}$$

$$PAM_k(i,j) = \log \frac{freq(a_i, a_j)}{freq(a_i) \cdot freq(a_j)}$$

Anschauliche Bedeutung: $PAM_k(i,j)$ beschreibt das Verhältnis zwischen der W'keit, mit der a_i durch akzeptierte Mutationen in a_j umgewandelt wird, und der W'keit, daß dieses Paar zufällig in einem Alignment auftritt.

Praktischer Ansatz:

1. Wähle Menge sehr ähnlicher Sequenzen, die von gemeinsamem Vorfahren abstammen und nur eine PAM-Einheit voneinander entfernt sind.
   → Diese Sequenzen haben ungefähr dieselbe Länge
   → Anordnung der Lücken ist „einfach" zu bestimmen
   ⟹ Bestimme 1-PAM-Matrix

2. Verwende 1-PAM-Matrix zur Berechnung von k-PAM-Matrizen:
   Sei F eine 20×20-Matrix, so daß $F(i,j)$ die W'keit angibt, daß a_i in einer PAM-Einheit zu a_j mutiert (unabhängig von der Häufigkeit des Auftretens von a_i)
   ⟹ $F^k(i,j)$ gibt die W'keit an, daß a_i in k-Einheiten zu a_j mutiert.

$$\Longrightarrow PAM_k(i,j) = \log \frac{freq(a_j) \cdot F^k(i,j)}{freq(a_i) \cdot freq(a_j)} = \log \frac{F^k(i,j)}{freq(a_j)}$$

In der Praxis: Beim Vergleich von zwei Protein-Sequenzen ist k nicht bekannt
⟹ Standardwerte: $k=40$, $k=100$, $k=250$.

5.3. Multiple Alignments

26.5.2003

5.3.1. Definition und Bewertung von multiplen Alignments

Def. 5.10: Seien k Strings $s_1 = s_{11} \dots s_{1m_1}, \dots, s_k = s_{k1} \dots s_{km_k}$ über Σ gegeben. Sei $- \notin \Sigma$ Lückensymbol, $\Sigma' := \Sigma \cup \{-\}$.

Sei $h : (\Sigma')^* \to \Sigma^*$ ein Homomorphismus, definiert durch $h(a) = a$ für alle $a \in \Sigma$, $h(-) = \lambda$.

Ein *multiples Alignment* von $s_1, \dots, s_k$ ist ein Tupel $(s_1', \dots, s_k')$ von Strings der Länge $l \geq \max\{m_i \mid 1 \leq i \leq k\}$ über Σ', so daß gilt:

(a) $|s_1'| = |s_2'| = \dots = |s_k'| = l$

(b) $h(s_i') = s_i$ für alle $1 \leq i \leq k$

(c) es gibt keine Position, an der in allen s_i' eine Lücke vorkommt.

Def. 5.11: Gegeben sei ein multiples Alignment $(s_1', \dots, s_k')$ der Länge l. Ein String $c = c_1 \dots c_l \in \Sigma^l$ heißt *Consensus* für $(s_1' \dots s_k')$, falls gilt

$$c_j = \operatorname*{argmax}_{a \in \Sigma} |\{s_{ij}' = a \mid 1 \leq i \leq k\}| \quad \text{für alle } 1 \leq j \leq l$$

Der *Abstand* eines Alignments $(s_1', \dots, s_k')$ von einem Consensus c ist definiert als

$$\text{dist}(c, (s_1', \dots, s_k')) = \sum_{j=1}^{l} |\{s_{ij}' \mid s_{ij}' \neq c_j, 1 \leq i \leq k\}|$$

Lemma 5.1: Sei $(s_1', \dots, s_k')$ ein multiples Alignment und seien c und $\bar{c}$ zwei Consensus-Strings für $(s_1', \dots, s_k')$. Dann gilt

$$dist(c, (s_1', \dots, s_k')) = dist(\bar{c}, (s_1', \dots, s_k'))$$

Beweis: Betrachte bel. Spalte j des Alignments. Falls $c_j \neq \bar{c}_j$, müssen in Spalte j die Symbole c_j und $\bar{c}_j$ gleich häufig vorkommen.

$$\Rightarrow |\{s_{ij}' \mid s_{ij}' \neq c_j, 1 \leq i \leq k\}| = |\{s_{ij}' \mid s_{ij}' \neq \bar{c}_j', 1 \leq i \leq k\}| \quad \square$$

$\Rightarrow$ Abstand zum Consensus $\hat{=}$ Abstand zu beliebigen Consensus

Beispiel: S.3:

$s_1' =$ A A T G C T
$s_2' =$ A - T T C -
$s_3' =$ - - - T C C
$c =$ A A T T C T

Abstand $1+2+1+1+0+2 = 7$

Def. 5.2: Das *Mult-Consensus-Align-Problem* ist das folgende Optimierungsproblem:

Eingabe: Eine Menge $S = \{s_1, \dots, s_k\}$ von Strings über einem Alphabet Σ.

zulässige Lösungen: alle multiplen Alignments von S.

Kosten: die Kosten eines multiplen Alignments $(s_1', \dots, s_k')$ mit Consensus c sind:

$$cost(s_1', \dots, s_k') = dist(c, (s_1', \dots, s_k'))$$

Optimierungsziel: Minimierung

Def. 5.13: Sei Σ ein Alphabet, $- \notin \Sigma$ Lückensymbol und δ Bewertungsfunktion für das Alignment von zwei Strings mit Optimierungsziel Minimierung, erwartet zum geeignete Definition für $\delta(-,-)$.

Die Bewertung δ_{SP} eines multiplen Alignments $(s_1', \dots, s_k')$ der Länge l als Summe der Paare (SP-Bewertung) für eine Spalte:

$$\delta_{SP}(s_{1j}', \dots, s_{kj}') = \sum_{i=1}^{k} \sum_{r=i+1}^{k} \delta(s_{ij}', s_{rj}')$$

$$\delta_{SP}(s_1', \dots, s_k') = \sum_{j=1}^{l} \delta_{SP}(s_{1j}', \dots, s_{kj}')$$

Beispiel 5.10:

$s_1' = AATG\,CT$

$s_2' = A-TTC-$

$s_3' = ---TCC$

δ: Edit-Distanz, d.h. $\delta(a,b)=0$, falls $a=b$, $\delta(a,b)=1$ sonst

$$\delta_{SP}(s_1', s_2', s_3') = \sum_{j=1}^{6} \sum_{i=1}^{3} \sum_{r=i+1}^{3} \delta(s_{ij}', s_{rj}')$$

$$= \sum_{j=1}^{6} \left(\delta(s_{1j}', s_{2j}') + \delta(s_{1j}', s_{3j}') + \delta(s_{2j}', s_{3j}') \right)$$

$$= 2+2+2+2+0+3 = 11.$$

Def. 5.14: Das Mult-SP-Align.-Problem ist das folgende Optimierungsproblem:

Eingabe: Eine Menge $S = \{s_1, \dots, s_k\}$ von Strings über Σ und eine Alignment-Bewertung δ mit Ziel Minimierung.

Zulässige Lösungen: Alle multiplen Alignments von S

Kosten: Die Kosten eines multiplen Alignments $(s_1', \dots, s_k')$ sind.

$cost(s_1', \dots, s_k') = \Sigma_{SP}(s_1', \dots, s_k')$

Optimierungsziel: Minimierung

5.3.2. Exakte Bestimmung multipler Alignments

Ansatz: dynamische Programmierung:
Für k Strings verwende k-dimensionales Array M, dessen Eintrag $M(i_1, \dots, i_k)$ die Bewertung eines opt. Alignments der Präfixe $s_{11} \dots s_{1i_1}, s_{21} \dots s_{2i_2}, \dots, s_{k1} \dots s_{ki_k}$ enthält.

Probleme:
- Aufwändige Datenstruktur, falls k nicht vorher bekannt.
- Laufzeit und Speicherbedarf hängen exponentiell von k ab.
  ein Eintrag des Arrays: Minimumbildung über $2^k - 1$ Terme
  (jede mögliche Kombination von Lücken in der aktuell betrachteten Spalte)
  insgesamt Zeitkomplexität $O(k^2 \cdot 2^k \cdot n^k)$.

Def. 5.15: Die Entscheidungsvariante des Mult-SP-Align-Problem ist wie folgt definiert: Das Mult-SP-Align-Problem (DMSPAP)

Eingabe: nat. Zahl k und Menge $S = \{s_1, \dots, s_k\}$ über Σ_1, eine Bewertungsfunktion $\delta: (\Sigma_1 \cup \{-\})^2 \to \mathbb{Q}$ und ein $d \in \mathbb{N}$

Ausgabe: JA, falls multiples Alignment ex., das SP-Bewertung bzgl. δ hat, die $\leq d$ ist.
NEIN, sonst.

Def. 5.16: Die Entscheidungsvariante des Problems der kürzesten gemeinsamen Supersequenz über $\Sigma_1 = \{0,1\}$ (D-(0,1)-SSP) ist wie folgt definiert:

Eingabe: $k \in \mathbb{N}$, $S = \{s_1, \dots, s_k\}$ von Strings über $\Sigma = \{0,1\}$, $m \in \mathbb{N}$

Ausgabe: JA, falls gemeinsame Supersequenz t der Strings aus S ex. mit $|t| \leq m$.

NEIN, sonst.

Lemma 5.2: D-(0,1)-SSP ist NP-vollständig. □

Satz 5.3: D-MSPAP ist NP-vollständig.

Beweis: D-MSPAP $\in$ NP ✓

Polynomialzeit-Reduktion von D-(0,1)-SSP auf D-MSPAP:

Seien $S = \{s_1, \dots, s_k\}$ Menge von Strings über $\{0,1\}$ und $m \in \mathbb{N}$ als Eingabe für D-(0,1)-SSP gegeben.

o.B.d.A. $\max\{|s_i| \mid 1 \leq i \leq k\} \leq m$

Konstruiere $n+1$ verschiedene Eingabeinstanzen für D-MSPAP und zeige, daß es eine Supersequenz der Länge $\leq m$ für S gibt gdw. eine der konstruierten Eingabeinstanzen für D-MSPAP ein multiples Alignment mit SP-Bewertung unter der geg. Schranke hat.

- Für alle $i,j \in \mathbb{N}$ mit $i+j=m$ sei $X_{i,j} = S \cup \{a^i, b^j\}$, $a, b \notin \Sigma$
- $d = (k-1) \cdot \|S\| + (2k+1) \cdot m$, wobei $\|S\| \sum_{s \in S} |s|$
- $\delta : (\{0,1,a,b,-\})^2 \to \mathbb{N}$.

	0	1	a	b	-
0	2	2	1	2	1
1	2	2	2	1	1
a	1	2	0	k	1
b	2	1	k	0	1
-	1	1	1	1	0

Blatt 83

Zeige: Die Menge S hat eine Supersequenz k mit $|k| = m$ genau dann, wenn eine der Menge $X_{i,j}$ ein multiples Alignment mit SP-Bewertung $\leq d$ bzgl. δ hat.

1. Sei für ein $X_{i,j}$ ein multiples Alignment $A = (s_1', \ldots, s_k', \alpha, \beta)$ mit SP-Bewertung $\leq d$ gegeben. Dabei sei α die zu a^i gehörige Zeile, und β die zu b^j gehörige Zeile.

Betrachte zunächst die Einschränkung A' von A auf die Zeilen $s_1', \ldots, s_k'$.

Zeige: Bewertung von A' ist genau $(k-1) \cdot \|S\|$:

Untersuche hierfür A' spaltenweise:

Spalte mit l Lücken hat Bewertung von

$$\underbrace{l\,(k-l)}_{\substack{\text{Vergl. Lücken} \\ \leftrightarrow \text{Nichtlücken}}} + \underbrace{2\,\frac{(k-l)(k-l-1)}{2}}_{0,1 \leftrightarrow 0,1} = (k-1)\cdot(k-l)$$

Sei x die Anzahl der Spalten von A' und y die Anzahl aller Lückensymbole in A'. Dann gilt $\|S\| = k \cdot x - y$.

Sei l_p die Anzahl der Lücken in Spalte p von A' für alle $1 \leq p \leq x$. Dann gilt:

$$\delta_{SP}(A') = \sum_{p=1}^{x} (k-1)\cdot(k-l_p) = (k-1)\sum_{p=1}^{x}(k-l_p)$$

$$= (k-1)\cdot(k\cdot x - y) = (k-1)\cdot\|S\|.$$

Buch: ISBN 3-519-00398-5 / Teubner

Algorithmische Grundlagen der Bioinformatik

Böckenhauer / Bongartz

28.5.2003

Zeige, daß es ein Alignment von X_{ij} mit SP-Bewertung $\leq d$ nur dann geben kann, wenn dieses Alignment die Länge m hat und in keiner Spalte a und 1 oder b und 0 gemeinsam vorkommen.

Vergleich von a mit β: Kosten von m, falls in keiner Spalte a und b gleichzeitig vorkommen.
Zusätzliche Kosten h für jede Spalte, in der a und b vorkommen.

Falls die Länge x des Alignments $< m$ ist, haben wir
Kosten von $\geq m + (m-x) \cdot h$

Vergleich von a mit $s_1', \ldots, s_k'$ bzw. von β mit $s_1', \ldots, s_k'$:
Kosten: $\geq 2 \cdot k \cdot x - z$, wobei z die Anzahl der Paare von Lückensymbolen
da a nur $x-i$ Lücken und β nur $x-j$ Lücken enthält,
gilt: $z \leq k \cdot (x-m)$, da $i+j = m$

$\Rightarrow$ Kosten: $\geq 2kx - k(x-m) = k(x+m)$
Zusätzliche Kosten für jede Spalte, in der a und 1 oder b und 0 auftreten.

$$\Rightarrow d_{SP}(A) \geq (k-1) \cdot \|S\| + \max\{m, m+k(m-x)\} + k(x+m)$$

Minimum wird erreicht für $x = m$:

$$(k-1) \cdot \|S\| + m + k(x+m) = (k-1) \cdot \|S\| + m + 2km$$
$$= d$$

→ Falls Alignment mit SP-Bewertung $\leq d$ ex., dann gilt $x = m$ und in keiner Spalte treten a und b oder a und 1 oder b und 0 gemeinsam auf.

⇒ In keiner Spalte treten 0 und 1 gemeinsam auf

⇒ Bestimme Supersequenz t von $s_1, \ldots, s_k$ der Länge m wie folgt:

$$t_\ell = \begin{cases} 0, \text{ falls } \alpha_\ell = a \\ 1, \text{ falls } \beta_\ell = b \end{cases}$$

2. Sei eine gemeinsame Supersequenz t für S der Länge m gegeben. Sei i die Anzahl der Nullen in t, j die Anzahl der Einsen in t.

Dann besitzt $x_{i,j}$ ein multiples Alignment mit SP-Bewertung $\leq d$: für jedes $p \in S$ ex. ein Alignment von p und t ohne Mismatches der Länge m.

⇒ Fasse diese Alignments zusammen, ordne die a-Symbole an a^i den 0-Spalten und die b-Symbole an b^j den 1-Spalten zu.

Analog Rechnung wie in 1. zeigt, daß dieses multiples Alignment eine Bewertung von d hat.

□

5.3.3. Zusammenfügen paarweiser Alignments

Def. 5.17: Sei $S = \{s_1, \dots, s_k\}$ eine Menge von Strings,
sei $T = \{s_{i_1}, \dots, s_{i_m}\} \subseteq S$.
Sei $A' = (s_1', \dots, s_k')$ ein multiples Alignment von S.
$A'' = (s_{i_1}'', \dots, s_{i_m}'')$ ein multiples Alignment von T.

A' heißt *kompatibel* zu A'', falls die Einschränkung von A' auf die Zeilen $i_1, \dots, i_m$, bei der alle Spalten eliminiert werden, die nur aus Lücken bestehen, gleich A'' ist.

Beispiel 5.11

```
A - C G G        A - C G G          A C G G
A - - T G   ,    A - - T G    →     A - T G
A T C G G
```

```
A - - T G        A T - G -
A T C G G   ≠    A T C G G
```

Def. 5.18: Sei $S = \{s_1, \dots, s_k\}$ eine Menge von Strings.
Ein Baum $T = (V, E)$ mit $V = \{s_1, \dots, s_k\}$, bei dem jede Kante $\{s_i, s_j\} \in E$ mit einem optimalen Alignment (s_i', s_j') beschriftet ist, heißt *Alignment-Baum* für S.

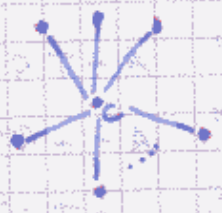
Satz 5.3. Sei $S = \{s_1, \dots, s_k\}$ eine Menge von Strings, sei $T = (V, E)$ ein Alignment-Baum für S.

Dann läßt sich ein multiples Alignment $(s_1'', \dots, s_k'')$ für S, daß mit den optimalen paarweisen Alignments (s_i', s_j') kompatibel ist für alle $(s_i, s_j) \in E$, effizient bestimmen.

□

Im folgenden: Spezialfall: T ist ein Stern

⇒ Star-Alignment



Algorithmus 5.3 - Star-Alignment

Eingabe: Menge $S = \{s_1, \ldots, s_k\}$ von Strings

1. Berechne das Zentrum c des Sterns:
   a) Bestimme optimales paarweises Alignment von s_i und s_j für alle $1 \le i, j \le k$ und berechne dessen Bewertung $\text{sim}(s_i, s_j)$
   b) Bestimme c als denjenigen String t, der $\sum_{s \in S} \text{sim}(t, s)$ minimiert.
   c) Setze T als Stern mit Zentrum c und Blättern $S - \{c\}$
2. Bestimme kompatibles multiples Alignment:
   for $i := 2$ to k do
   Bestimme ein zu T kompatibles multiples Alignment von c und $s_1, \ldots, s_i$ aus dem bereits bestimmten kompatiblen multiplen Alignment c und $s_1, \ldots, s_{i-1}$ und dem optimalen paarweisen Alignment von c und s_i nach dem Prinzip „Einmal eine Lücke – immer eine Lücke"

Ausgabe: Das zu T kompatible multiple Alignment von S.

Beispiel:

c': A T G - C A T T c'': A - T G C - A T T

s_1': A - G T C A A T s_3': A C T G T A A T T

s_2': - T C T C A - -

c''': A - T G - C - A T T

s_1''': A - - G T C - A A T

s_2''': - - T C T C - A - -

s_3''': A C T G - T A A T T

Def. 5.13: Eine Bewertungsfunktion $\delta: (\Sigma \cup \{-\})^2 \to \mathbb{Q}$ heißt gut, wenn gilt:

(i) $\delta(a,a) = 0$ für alle $a \in \Sigma \cup \{-\}$

(ii) $\delta(a,c) \leq \delta(a,b) + \delta(b,c)$ für alle $a,b,c \in \Sigma \cup \{-\}$

(Dreiecksungleichung)

Lemma 5.3: Sei δ eine gute Bewertungsfunktion. Dann gilt

$\delta(a,b) \geq 0$ für alle $a, b \in \Sigma \cup \{-\}$

Beweis: Es gilt $0 = \delta(a,a) \leq \delta(a,b) + \delta(b,a) = 2 \cdot \delta(a,b)$ für alle $a, b \in \Sigma \cup \{-\}$. □

Lemma 5.4: Sei $\delta: (\Sigma \cup \{-\})^2 \to \mathbb{Q}$ eine gute Bewertungsfunktion. Sei $S = \{c, s_1, \ldots, s_k\}$ eine Menge von Strings über Σ. Sei $T = (S, E)$ ein Stern mit Zentrum c und sei $(c', s_1', \ldots, s_k')$ ein zu T kompatibles multiples Alignment von S. Dann gilt für alle $i, j \in \{1, \ldots, k\}$

$$\delta(s_i', s_j') \overset{(1)}{\leq} \delta(s_i', c') + \delta(c', s_j') \overset{(2)}{=} \text{sim}(s_i, c) + \text{sim}(c, s_j).$$

Beweis: (1) Dreiecksungleichung

(2) folgt, da die von $(c', s_1', \ldots, s_k')$ induzierten paarweisen Alignments zwischen s_i und c sowie zwischen c und s_j optimal sind. □

Satz 5.4: Sei δ eine gute Bewertungsfunktion, sei δ_{SP} die von δ induzierte SP-Bewertungsfunktion. Sei $S = \{s_1, \dots, s_k\}$ eine Menge von Strings, sei $sim_{SP}(S)$ die SP-Bewertung eines optimalen multiplen Alignments von S.

Dann gilt für das von Alg. 5.3 berechnete multiple Alignment $(s_1', \dots, s_k')$:

$$\delta_{SP}(s_1', \dots, s_k') \leq \left(2 - \frac{2}{k}\right) \cdot sim_{SP}(s_1, \dots, s_k).$$

Beweis: Sei $(s_1'', \dots, s_k'')$ ein optimales multiples Alignment von S, d.h.

$$\delta_{SP}(s_1'', \dots, s_k'') = sim_{SP}(s_1, \dots, s_k)$$

Definiere $V(s_1', \dots, s_k') = \sum_{i=1}^{k} \sum_{j=1}^{k} \delta(s_i', s_j') = 2 \cdot \delta_{SP}(s_1', \dots, s_k')$

und $V(s_1'', \dots, s_k'') = \sum_{i=1}^{k} \sum_{j=1}^{k} \delta(s_i'', s_j'') = 2 \cdot \delta_{SP}(s_1'', \dots, s_k'')$

$$= 2 \cdot sim_{SP}(s_1, \dots, s_k)$$

Es reicht zu zeigen: $\dfrac{V(s_1', \dots, s_k')}{V(s_1'', \dots, s_k'')} \leq 2 - \frac{2}{k}$

Sei $M := \min_{t \in S} \sum_{s \in S} sim(s,t) = \sum_{s \in S} sim(c,s) = \sum_{s \in S - \{c\}} sim(c,s)$

o.B.d.A.: $c = s_k$

Nach Lemma 5.4 gilt:

$$V(s_1', \dots, s_k') = \sum_{i=1}^{k} \sum_{j=1}^{k} \delta(s_i', s_j') \leq \sum_{i=1}^{k} \sum_{j=1}^{k} \left(sim(s_i, c) + sim(s_j, \right.$$

$$= \sum_{i=1}^{k-1} \sum_{j=1}^{k-1} \left(\sim(s_i, c) + \sim(s_j, c) \right)$$

$$= \sum_{i=1}^{k-1} \sum_{j=1}^{k-1} \sim(s_i, c) + \sum_{i=1}^{k-1} \sum_{j=1}^{k-1} \sim(s_j, c)$$

$$= 2 \cdot (k-1) \cdot \sum_{i=1}^{k-1} \sim(s_i, c) = 2 \cdot (k-1) \cdot M$$

$$V(s_1'', \ldots, s_k'') = \sum_{i=1}^{k} \sum_{j=1}^{k} \delta(s_i'', s_j'') \geq \sum_{i=1}^{k} \sum_{j=1}^{k} \sim(s_i, s_j)$$

$$\geq k \cdot \sum_{j=1}^{k} \sim(c, s_j) = k \cdot M$$

$$\frac{V(s_1', \ldots, s_k')}{V(s_1'', \ldots, s_k'')} \leq \frac{2 \cdot (k-1) \cdot M}{k \cdot M} = 2 - \frac{2}{k} \qquad \square$$

2.6.2003

5.2. Heuristiken zur Datenbanksuche

5.2.1. Das FASTA-Verfahren

Ziel: Vergleiche zu suchendes Muster (Datenbankanfrage) nacheinander mit allen gespeicherten Sequenzen (Datenbank-Strings).

Prinzipielle Vorgehensweise:

1. Wähle Parameter k und suche alle exakten Matches der Länge k (Hot-Spots)

   Übliche Werte: $k = 6$ für DNA
   $k = 2$ für Proteine

2. Fasse mehrere Hot-Spots zusammen:

Betrachte Matrix M für Muster p und Text t

$$M(i,j) = \begin{cases} 1 & \text{falls } p_i = t_j \\ 0 & \text{sonst} \end{cases}$$

$\Rightarrow$ Hot-Spot $\triangleq$ Abschnitt einer Diagonalen

Suche 10 besten diagonalen Läufe, die mit Hot-Spot beginnen und enden.

Bewertung: Anzahl der Hot-Spots: positiv
Länge der Lücken dazwischen: negativ

Bestimme für jeden diagonalen Lauf opt. lokales Alignment

3. Setze die so berechneten Teilalignments zu längerem Alignment zusammen.

4. Berechne Alternativlösung durch lokales Alignment, beschränkt auf einen Streifen konstanter Breite um das beste lokale Alignment aus Schritt 2 herum.

Anschließend statistische Bewertung der berechneten Lösungen.

5.2.2. Das BLAST-Verfahren

Suchalgorithmus:

1. Suche ähnliche Teilstrings, sogenannte Hits, gegebener Länge w.

Übliche Werte: w = 11 für DNA
w = 3 für Proteine

2. Suche alle Paare von Hits, die Abstand $\leq d$ haben.

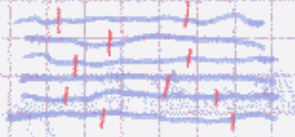
3. Erweitere die Paare von Hits an beiden Enden, bis sich die Alignment-Bewertung nicht mehr erhöht. Falls Bewertung über einem Schwellenwert S liegt:
   High-Scoring-Pair

Anschließend statistische Bewertung der Lösungen.

Teil II – DNA-Sequenzierung

Menschliches Genom: 3,5 Gbp
Direkte Sequenzierung: $\leq$ 1000 bp

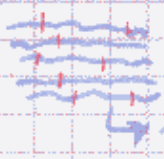
- wenige Kopien von k:



50–300 kbp

Ordnung der Fragmente geht verloren!
→ Physikalische Kartierung

- nehme ein Fragment aus einer sogenannten physikalischen Karte
  Sequenziere dieses Fragment



} DNA-Sequenzierung

~ 1000 bp

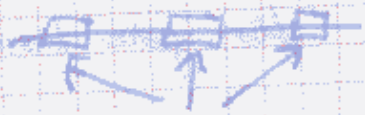
Human-Genom-Projekt Celera Genomics

7. Physikalische Kartierung:

Def. 7.1: Sei D eine DNA-Sequenz.

Eine **Physikalische Karte** von D besteht aus einer Menge M von **Markern** und einer Funktion $\rho : M \to Pot(\mathbb{N})$, die für jeden Marker dessen Position in D angibt.

$m \in M$



7.1. Restriktionsstellen-Kartierung

Verwenden: Restriktionsenzyme mit spezifischen Restriktionsstellen zum Zerschneiden (Verdau) der DNA.

→ Restriktionsstellen dienen als Marker

→ Aufgabe: Bestimme die Reihenfolge der durch den Verdau entstandenen Fragmente.

7.1.1. Das Double-Digest-Verfahren

Beachte:
- $\Delta(A), \Delta(B), \Delta(AB)$ sind Multimengen
- (idealerweise) Vollverdau, bzw. full-digest, d.h. falls eine Restriktionsstelle vorhanden ist, so wird das Molekül auch dort geschnitten.

Def. 7.2: Sei $X = \{x_1, \dots, x_n\}$ eine Multimenge mit Elementen aus $\mathbb{N} - \{0\}$. Sei π eine Anordnung dieser Elemente, $\pi = (x_{i_1}, \dots, x_{i_n})$. Dann

$$Pos(\pi) = \{0, x_{i_1}, x_{i_1} + x_{i_2}, \dots, \sum_{j=1}^{n} x_{i_j}\}$$

Positionsmenge der Anordnung π.

Umgekehrt sei Y eine Menge von Elementen aus $\mathbb{N}$ und sei

$$Y = \{y_1, \dots, y_p\}, \quad y_1 < y_2 < \dots < y_p.$$

Dann $\quad Dist(Y) = \{ |y_{i+1} - y_i| \mid i \in \{1, \dots, p-1\} \}$

Distanzmenge der Menge Y

Es gilt: $Dist(Pos(\pi)) = X$ für jede Anordnung π einer Multimenge X.

Def. 7.3: Seien A, B, C Multimengen, mit $|A| = n$, $|B| = m$.
Sei π eine Anordnung der Elemente aus A und ϕ eine Anordnung der Elemente aus B.

Das Paar (π, ϕ) heißt zulässige Lösung für A, B, C, wenn gilt:

$$Dist(Pos(\pi) \cup Pos(\phi)) = C$$

Def. 7.4: Das Double-Digest-Problem (DDP) ist das folgende Berechnungsproblem:

Eingabe: Multimengen A, B, C

Ausgabe: Ein Element aus der Menge

$$U = \{(\pi, \phi) \mid (\pi, \phi) \text{ zulässige Lösung von } A, B, C\}$$

oder den Text 0, wenn $U = \emptyset$.

$\rightsquigarrow$ die Menge U heißt auch die Menge der zulässigen Lösungen für das DDP.

naive Ansatz: Teste alle möglichen Anordnungen π und ϕ.
$\Rightarrow |A|! \cdot |B|!$ viele Möglichkeiten testen ... "zu viele"

Def. 7.5: Die Entscheidungsvariante des Double-Digest-Problems, kurz Dec-DDP, lautet wie folgt:

Eingabe: Multimengen A, B, C

Ausgabe: Ja, falls $W \geq 1$
Nein, sonst

Def. 7.6: Das Set-Partition-Problem:

Eingabe: $X = \{x_1, \ldots, x_n\}$ mit Elementen aus $\mathbb{N} - \{0\}$

Ausgabe: Ja, wenn eine Zerlegung von X in 2 disjunkte Mengen Y und Z existiert, sodass

$$\sum_{y \in Y} y = \sum_{z \in Z} z$$

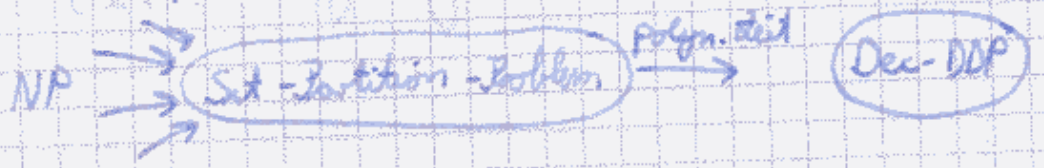
Nein, sonst

Es ist bekannt, daß das Set-Partition-Problem NP-vollständig ist.

Satz 7.1: Das Dec-DDP ist NP-vollständig!

Beweis: (i) Dec-DDP $\in$ NP ✓

(ii) Alle Probleme aus NP in polynomieller Zeit lassen sich auf Dec-DDP zurückführen.



Sei X eine Eingabe für das Set-Partition-Problem.
O.B.d.A. sei die Summe aller Werte in X gerade.

$\rightarrow$ konstruiere Eingabe für Dec-DDP (A, B, C)

BioInf (5a)

- $A := X = \{x_1, \dots, x_n\}$
- $B := \{\frac{\alpha}{2}, \frac{\alpha}{2}\}$, wobei $\alpha = \sum_{x \in X} x$
- $C := X$

zz. Es ex. eine Lösung für das Set-Partition-Problem mit Eingabe X genau dann wenn eine Lösung für das Dec-DDP mit Eingabe A, B, C (wie oben) existiert.

$\Rightarrow$ Sei X teilbar in 2 Mengen Y und Z, mit Y und Z sind disjunkt

$$\sum_{y \in Y} y = \sum_{z \in Z} z$$

Dann ist $\pi = (P(Y), P'(Z))$ und $\phi = (\frac{\alpha}{2}, \frac{\alpha}{2})$ mit P, P' beliebige Anordnungen, eine Lösung für das Dec-DDP.

$\Leftarrow$ Sei (π, ϕ) eine zulässige Lösung für das Dec-DDP. Sei $\pi = (x_{j_1}, \dots, x_{j_n})$ dann existiert ein Index n_0 mit

$$\sum_{i=1}^{n_0} x_{j_i} = \sum_{i=n_0+1}^{n} x_{j_i}$$

da diese Aufteilung wegen $B = \{\frac{\alpha}{2}, \frac{\alpha}{2}\}$ notwendig ist.

4.6.2003

7.12. Das Partial-Digest-Problem

→ unvollständiges Verdau (partial-digest)

→ Multimenge $\Delta p(A)$ im Gegensatz DDP

Im Folgenden gehen wir von idealen Daten aus.

Def. 7.7 Die ermittelte Datenmenge $\Delta p(A)$ heißt **ideal**, wenn sie die Länge jedes durch Restriktionsstellen bzw. Enden des Moleküls begrenzten Fragments enthält.

D.h. wir haben Schnittstellen (bzw. Endpunkte)

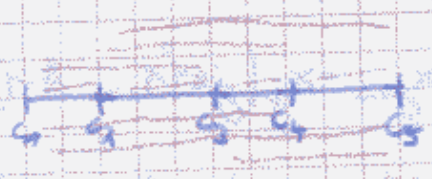
$$c_1 < c_2 < \ldots < c_k$$

dann gilt:

$$\Delta p(A) = \{ c_j - c_i \mid 1 \le i < j = k \}$$

→ eine ideale Multimenge für das PD enthält

$\binom{k}{2}$ Elemente (k ist die Anzahl der Restriktionsstellen + Endpunkte)



Def. 7.8: Sei A eine Multimenge mit $\binom{k}{2}$ Elementen aus $\mathbb{N} - \{0\}$, sei $P = \{x_1, \ldots, x_k\}$ eine Menge von natürlichen Zahlen mit $x_1 = 0$ und $x_1 < x_2 < \ldots < x_k$

Dann heißt P auch **Punktmenge**.

Zu P lässt sich eine Multimenge alles paarweisen Distanzen definieren:

$$\mathrm{Dist}_P(P) = \{ x_j - x_i \mid 1 \le i < j \le k \}$$

Eine Punktmenge P heißt zulässige Lösung für A falls

$$Dist_P(P) = A$$

Def. 7.9: Partial-Digest-Problem (PDP)

Eingabe: Multimenge A mit $\binom{k}{2}$ Elementen aus $\mathbb{N} - \{0\}$

Ausgabe: Ein Element aus

$$\mathcal{A} = \{P \mid P \text{ ist zulässige Lösung für } A\}$$

oder 0, falls $\mathcal{A} = \emptyset$.

Beachte: Im Gegensatz zum DDP ist hier "keine Anordnung" von den Elementen in A gesucht.

Def. 7.10: Sei A eine Multimenge mit $\binom{k}{2}$ Elementen aus $\mathbb{N} - \{0\}$. Sei $P = \{x_1, \dots, x_k\}$ eine zulässige Lösung für A, mit $x_1 = 0$ und $x_1 < x_2 < \dots < x_k$. Dann bezeichne

$$level_P(i) = \{x_{j+i} - x_j \mid j \in \{1, \dots, k-i\}\} \subseteq A$$

die Multimenge von Distanzen deren Endpunkte den Abstand i in der Lösung P besitzen.

Bemerkungen 7.1: A Multimenge und P zulässige Lösung (wie oben)

- $|level_P(i)| = k - i$
- $level_P(k)$: Distanzen zwischen benachbarten

Restriktionsstellen $\rightarrow$ full digest

$\rightarrow$ atomare Distanzen

- $level_P(k-1) \triangleq$ maximale Distanz in A, Länge des betrachteten Moleküls.
- Level bilden "disjunkte" Zerlegung von A:

$$level_P(1) \cup level_P(2) \cup \dots \cup level_P(k-1) = A$$

Naiver Ansatz:
- wähle $k-1$ atomare Distanzen aus A $\binom{k}{k-1}$
- Überprüfe alle Anordnungen: $(k-1)!$

$\Rightarrow$ Laufzeit: $O\left[(k-1)!\binom{k}{k-1}\right]$ exponentiell

- Ermittle atomare Distanzen durch full-digest-Experiment bestimmen $\Rightarrow O((k-1)!)$

Backtracking:
- sukzessive Spezifikation von Teillösungen
- falls eine Spezifikation nicht "erfolgreich"
  $\rightarrow$ Backtracking-Schritt, nehme letzte Spezifikation zurück

Hier: Teillösungen $\triangleq$ festgelegte Positionen in der Punktmenge.

Notation: $\gamma \in \mathbb{N}$, Multimenge $X = \{x_1, \dots, x_n\}$

$$\delta(\gamma, X) = \{|x-\gamma| \mid x \in X\}$$

schematische Darstellung des Backtracking-Algorithmus: (Idee)

1) platziere längste Distanz
   $\rightarrow$ Intervall, in dem alle weiteren Punkte liegen, festgelegt.

2) Für die jeweils längste verbleibene Distanz:
- überprüfe, ob Platzierung am linken Rand möglich
  → falls ja, platziere links
- ansonsten, überprüfe, ob Platzierung am rechten Rand möglich
  → falls ja, platziere rechts
- ansonsten: Backtracking-Schritt

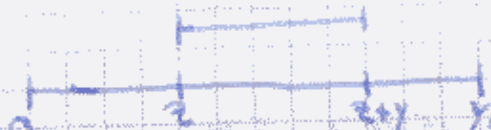
3) Gebe Lösung aus, wenn alle Distanzen erfolgreich platziert wurden.

Satz 7.2: Sei A eine Eingabe für das PDP.

Falls eine zulässige Lösung für das PDP mit Eingabe A existiert, dann berechnet der Algorithmus 7.1 eine solche.

Beweis: Der Algorithmus 7.1 untersucht alle Lösungsmöglichkeiten, bei denen die jeweils längste verbleibende Distanz y entweder am linken oder rechten Rand platziert wird.
→ zeige: damit werden alle möglichen Lösungen durchlaufen.

Sei $X = [0, x_2, x_3, \ldots, y_{max}]$ die bisher konstruierte Teillösung, A Restmenge mit verbleibenden Distanzen, y momentan größte Distanz.



Angenommen die Distanz y könnte "in der Mitte" platziert werden, also in einem Intervall $[z, z+y]$ $(z \neq 0,\ z+y \neq y_{max})$

- wenn $z+y \notin X$, so muß $z+y \in A$
  → aber $z+y > y$ ⇒ y ist nicht größtes Element in A ↯
- wenn $z \notin X$ ⇒ $y_{max} - z \in A$
  es gilt $y_{max} - z > y = y$ ist nicht größtes Element in A ↯
- wenn $z \in X$ und $z+y \in X$
  ⇒ notwendige Distanz der Länge 0 in A ↯

□

Satz 7.3: Der Algorithmus 7.1 hat im schlechtesten Fall eine Laufzeit von $O(2^k \cdot k \log k)$ für eine Eingabe mit $\binom{k}{2}$ Elementen.

Beweis:
- Initialisierung: $O(1)$
- Sortierung von A: $O(k^2 \cdot \log k^2) = O(k^2 \cdot \log k)$
- Platzierung von k-1 Distanzen, wobei Platzierung am rechten und linken Rand möglich sind.
  → 2^{k-1} Platzierungsmöglichkeiten
  - Funktion δ liefert $O(k)$ zu überprüfende Distanzen
    Aufwand ob Platzierung möglich / Platzierung durchführen
    $O(k \cdot \log k^2) = O(k \cdot \log k)$
    Aufwand für das Backtracking (analog zur Platzierung)
    $O(k \cdot \log k)$
    ⇒ $O(k^2 \cdot \log k + 2^{k-1} \cdot k \cdot \log k) = O(2^k \cdot k \log k)$

Algorithmus 7.1 hat im schlechtesten Fall exponentiellen Aufwand (es existieren entsprechende Eingaben)

Satz 7.4: Sei die Anzahl der Backtracking-Schritte, die Algorithmus 7.1 benötigt, unabhängig von k für eine Eingabe A der Größe $\binom{k}{2}$, so kann die Laufzeit durch $O(k^2 \log k)$ abgeschätzt werden.

Gemessen an der Eingabegröße $\binom{k}{2}$ ergibt sich dann die Laufzeit $O(n \cdot \log n)$ $\left(n = \binom{k}{2}\right)$.

Beweis: Platziere $O(k)$ Elemente (nicht wie in Satz 7.3 2^{k-1})
(analog zu Beweis von Satz 7.3)

Zusammenfassend:

DDP: einfaches Experiment $\longleftrightarrow$ "schweres" Problem

PDP: "komplexes" Experiment $\longleftrightarrow$ "leichtes" Problem
(guter Algorithmus)

7.2. Methode des Fingerabdrucks und Kartierung durch Hybridisierung

17.6.2003

Konzept des Fingerprinting

- Ordne jedem Fragment spezifische Eigenschaften zu
  Fingerprints / Fingerabdrücke

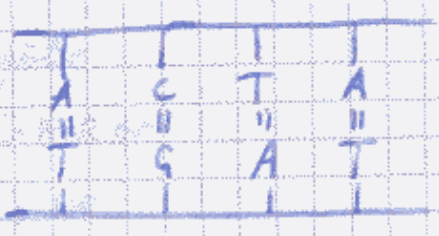
- Fingerprints sind leicht zu ermitteln
- Zwei Fragmente überlappen einander ⟺ ähnliche Fingerprints

Kandidaten für Fingerprints

- Restriktionsstellen-Kartierung
- Fragmentgrößen nach Verdau durch Restriktionsenzyme
- Hybridisierungsdaten

Exkurs: Biologische Grundlagen: Hybridisierung - Klonierung - DNA-Chips

Hybridisierung: Aneinanderlagerung (komplementärer) Nucleinsäure-stränge



⇒ Test, ob ein bestimmter Teilstring in einem Nucleinsäurestrang auftritt.

Ziel: Viele Hybridisierungsexperimente gleichzeitig durchführen

Davor: Vervielfältigung von DNA

1.) Klonierung:
- Einbau der zu kopierenden DNA (Insert) in einen Vektororganismus (Host)
- Replikation des Host replikiert auch das Insert
- Extraktion des Insert aus dem Host

Länge des Inserts: 15-50 kbp für Bakterien / Viren
mehrere Millionen für künstliche Chromosomen

Problem: z.B. Übereinstimmungen mit des Host-DNA.

2) Polymerase-Kettenreaktion (1983)

Schritt 1: Tube im Reagenzglas
- zu kopierende DNA d
- Primer P_1 und P_2 (DNA-Stränge (Einzelstränge), die komplementär zum Anfang, bzw. zum Ende von d sind).
- alle Nucleotide in ausreichender Menge
- DNA-Polymerase (Enzym, das aus einem Primer entsprechend einer Folge zukommen einen kompletten Molekülstrang aufbaut).

Schritt 2: (zyklisch beliebig oft)
- denaturieren: DNA (durch Erhitzen werden die beiden Einzelstränge voneinander getrennt).
- abkühlen → Primer lagern sich an die Einzelstränge an.
→ DNA-Polymerase verlängert die Primer zu einem vollständigen komplementären Strang.

n Iterationen → bis zu 2^n Kopien

Voraussetzung: Kenntnis der Primer

Problem: Fehler in frühen Iterationen können sich exponentiell fortpflanzen.

DNA-Chips:

- s, t DNA-Sequenzen (einzelsträngig)

  Aufgabe: Teste, ob t in s als Teilstring vorkommt

  → Führe Hybridisierungsexperiment von s und t' durch, wobei t' die zu t komplementäre Sequenz ist.

- $s, t_1, \ldots, t_n$ DNA-Sequenzen (einzelsträngig)

  Teste „parallel", ob $t_1, \ldots, t_n$ als Teilstring in s auftreten

  → 1) Verankere $t_1', \ldots, t_n'$ auf einer Oberfläche (Positionen von t_1' bis t_n' sind bekannt).

  $t_1', \ldots, t_n' \triangleq$ Probes / Sonden

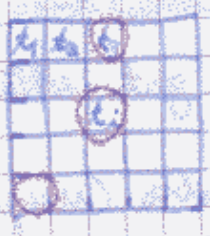
  Oberfläche mit Probes $\triangleq$ DNA-Chip

  2) Gebe markierte (z.B. fluoreszierende) Kopien der zu untersuchenden DNA auf den Chip

  - Kopien der DNA $\triangleq$ Clones
  - Clones hybridisieren mit entsprechenden Probes

  3) Wasche nicht-hybridisierten Clones (‚abwaschen')

  4) Bestimme (anhand der Markierungen), wo Hybridisierung stattfanden.



Fehlertypen:
- falsch positiv: Experiment meldet Hybridisierung von s und t_i', obwohl t_i kein Teilstring von s ist.
- falsch negativ: Experiment meldet keine Hybridisierung von s und t_i', obwohl t_i ein Teilstring von s ist.

Beachte: unbekannt, wie oft t_i als Teilstring in s auftritt.

Methode 7.3 Kartierung durch Hybridisierung

Gegeben: zu untersuchende DNA d

1) Ensemble mit hoher Wahrscheinlichkeit überlappender Fragmente von d.
2) Ensemble Kopien der Fragmente → Clones
3) $C = \{c_1, \dots, c_n\}$ Clones (clone-library), weitere Menge von Proben $P = \{p_1, \dots, p_m\}$
4) Führe alle Hybridisierungsexperimente (c_i, p_j), $1 \le i \le n$, $1 \le j \le m$

Ausgabe: $(n \times m)$-Hybridisierungsmatrix H

$$H(i,j) = \begin{cases} 1, & \text{falls } c_i \text{ mit } p_j \text{ hybridisiert} \\ 0, & \text{sonst} \end{cases}$$

Ziel: Aus H die ursprüngliche Anordnung der Clones ableiten.

Def. 7.11: Das Problem der Kartierung durch Hybridisierung (KdH):

Eingabe: $(n \times m)$-Hybridisierungsmatrix.
Ausgabe: Anordnung der Clones/Proben, die diese Hybridisierungsmatrix möglichst gut erklärt.

- Kartierung mit eindeutigen Proben (unique Probes) ←
- Kartierung mit mehrdeutigen Proben (non-unique Probes)

7.2.1 Kartierung mit eindeutigen Proben

Keine Eindeutigkeit → STS-Proben

Aufgabe: Suche eine Permutation der Proben, die der realen Anordnung entspricht.

Def. 7.12: Sei A eine $(m \times n)$-Matrix mit Einträgen aus $\{0,1\}$.
A hat die Eigenschaft der aufeinanderfolgenden Einsen / Consecutive Ones Property (C1P), falls eine Permutation Π der Spalten von A existiert, so daß in jeder Zeile keine Null zwischen zwei Einsen steht.
Falls A schon diese Form besitzt, dann hat A die Consecutive Ones Form (C1F).

Aufgabe: Besitzt eine Hybridisierungsmatrix die C1P, wenn ja berechne eine entsprechende Permutation.

naiv → teste alle Permutationen → $O(n!)$ Hilft nicht!

PQ-Bäume:

Def. 7.13: Sei $U = \{u_1, \dots, u_n\}$ eine endliche Menge von Elementen. Ein PQ-Baum über U eine Struktur $T = (V, E, r, B, \text{label}, \text{type})$ mit:

(i) (V,E) ist ein gerichteter Baum
(ii) $r \in V$ ist die Wurzel von (V,E)
(iii) $B \subseteq V$ ist die Menge der Blattknoten
(iv) label: $B \to U$ ist eine bijektive Abbildung der Blätter auf
(v) type: $V - B \to \{P, Q\}$ ist eine Zuordnung der inneren Knoten zu einem P oder Q

Def. 7.14: Sei U eine Menge, T ein PQ-Baum über U und $(v_1, \dots, v_k)$ die Blätter von T entsprechend ihrer Anordnung von links nach rechts.

Front von T: $Front(T) = (label(v_1), \dots, label(v_k))$

Def. 7.15: Sei T ein PQ-Baum über einer Menge $U = \{u_1, \dots, u_n\}$

legale Operationen:

(i) Die Anordnung der Kinder eines P-Knotens darf beliebig verändert werden.

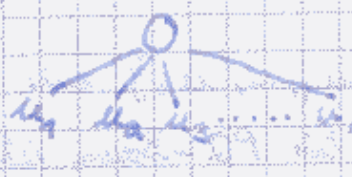
(ii) Die Anordnung der Kinder eines Q-Knotens darf invertiert werden:

$$v_1 v_2 \dots v_k \longrightarrow v_k v_{k-1} \dots v_2 v_1$$

konsistente Permutationen:

$Consist(T) = \{\pi \mid \pi$ ist eine Permutation über U, π ergibt sich als Front von T durch eine Folge legaler Operationen$\}$

universellen PQ-Baum: $U = \{u_1, \dots, u_n\}$



→ stellt alle möglichen Permutationen der Elemente in U dar.

Eingabe: Eine Menge $U = \{u_1, \dots, u_n\}$ paarweise verschiedene Elemente (Proben) und eine Menge von Restriktionen $R \in Pot(U)$ (Einsen in den Zeilen der Hybridisierungsmatrix)

Ausgabe: Alle Permutationen Π der Elemente aus U, so daß für alle Restriktionen R gilt:
Die Elemente aus R folgen aufeinander.

Lösung:
- Start mit universellem PQ-Baum über U
- Für jede Restriktion transformiere den aktuellen PQ-Baum, so daß die Menge der konsistenten Permutationen die Restriktion R erfüllt.

Satz 7.5: Sei M eine $(n \times m)$-Matrix mit Einträgen $\{0,1\}$ und sei k die Anzahl der Einsen in M.
Dann existiert ein Algorithmus (basierend auf PQ-Bäumen), der das C1-Problem in Zeit $O(n+m+k)$ löst.

7.2.2 Kartierung mit eindeutigen Proben und Fehlern

15.6.2005

Angenommen, die Anordnung der Proben in dem untersuchten Molekül ist bekannt.

Was folgt daraus für bestimmte Fehlertypen?

- falsch negativ: 0000 1111111 1 111111 00
- falsch positiv: 00 1 0000 1111111110000
- Chimären: 00001111 1 0000 1111000

Lücken $\triangleq$ Block von 0, der durch Einsen begrenzt wird.

Ziel: Suche Permutation der Spalten in der Hybridisierungsmatrix, so daß die Anzahl der Lücken minimiert wird.

Def. 7.16: Das Problem der minimalen Lückenanzahl, MinLM:

Eingabe: $(n \times m)$-Matrix über $\{0,1\}$

Zulässige Lösungen: Für alle Eingaben A,

$$\mathcal{U}(A) = \{(i_1, \ldots, i_m) \mid \text{Permutationen der Spalten in } A\}$$

Kosten: für eine zulässige Lösung $\pi \in \mathcal{U}(A)$

$cost(\pi, A)$ = Anzahl der Lücken in der Matrix A_π, die durch Permutation der Spalten entsprechend π resultiert.

Optimierungsart: Minimierung

Annahme: In der betrachteten Matrix existiert keine Zeile, die nur aus Nullen besteht.

Def. 7.17: Sei A eine binäre $(n \times m)$-Matrix. Dann ist der Spaltenabstandsgraph $G_\Delta = (V, E, c)$ definiert durch

- $V = \{1, \ldots, m\}$
- $E = \{\{i,j\} \mid 1 \le i,j \le m, i \neq j\}$
- $c : E \to \mathbb{N}$, $c(i,j) = \underbrace{|\{k \mid 1 \le k \le n, A(k,i) \neq A(k,j)\}|}_{\text{Hamming-Distanz}}$

→ Wenn eine Lücke in einer Zeile auftritt

→ entlang des entsprechenden Pfades in G_Δ zwei Bitwechsel vor, Kosten erhöhen sich um 2.

Umformulierung MinLM auf ein Problem über G_Δ:

"Finde einen Pfad in G_Δ, der jeden Knoten genau einmal besucht und minimale Kosten besitzt"

Problem: - Falls die Permutation mit einem Einserblock beginnt und endet, so tragen nur die Lücken zu den Kosten des Pfades bei.

- Falls die Permutation mit einem Nullenblock beginnt und endet, so tragen die Lücken und der 0-1-Wechsel am Anfang und der 1-0-Wechsel am Ende zu den Kosten bei.

→ vermeidbar!

Idee: Füge eine spezielle Spalte, die nur aus Nullen besteht hinzu und betrachte statt Pfaden Kreise!

Satz 7.6: - A binäre $(n \times m)$ Matrix
- A' binäre $(n \times (m+1))$ Matrix (A ergänzt um Nullspalte p')
- G_Δ ist der zu A' gehörende Spaltenabstandsgraph
- π Permutation der Spalten in A.
- A_π die aus Anwendung der Permutation resultierende Matrix
- K_π sei nun der Kreis $p'(\pi_1 \dots \pi_m)p'$ in G_Δ

$\Rightarrow \text{cost}(K_\pi) = 2 \cdot l + 2 \cdot n$

wobei l die Anzahl der Lücken in A_π ist.

Beweis: folgt aus der Hinzunahme der Spalte p'.

$\Rightarrow$ MinLM auf den Spaltenabstandsgraphen entspricht TSP.

Frage: Besitzt G_Δ eine bestimmte Eigenschaft?

→ die Kosten auf den Kanten von G_Δ erfüllen die Dreiecksungleichung.

$v_1, v_2, v_3 \in G_\Delta$:

$$c(\{v_1, v_3\}) \leq c(\{v_1, v_2\}) + c(\{v_2, v_3\})$$

TSP auf Graphen mit Δ-Ungleichung:

$\frac{3}{2}$-Approximation (Christofides)

→ 2-Approximation

Spannbaum-Algorithmus für Δ-TSP

Satz 31: Der Spannbaum-Algorithmus ist ein 2-Approximationsalgorithmus für das Δ-TSP.

Beweis: 1) Laufzeit: - Berechnung des min. Spannbaums $O(|E| \cdot \log|V|)$
- Tiefensuche $O(|V|)$
} polynomiell

2) Approximationsgüte

Sei H_{opt} eine optimale Lösung für eine Eingabe $G = (V, E, d)$

- $cost(T) \leq cost(H_{opt})$, denn die Löschung einer Kante aus einem beliebigen Hamilton-Kreis ergibt einen Spannbaum. Alle Kantenkosten sind positiv.

- Sei W der Pfad, der bei der Tiefensuche durchlaufen wird $cost(W) = 2 \cdot cost(T)$, da jede Kante des Spannbaums genau 2 mal durchlaufen wird.

Beispiel (62) - H entspricht W, wobei Knoten nicht mehrfach besucht werden, sondern durch kurze Kante überbrückt.

Wegen Dreiecksungleichung

$$cost(H) \leq cost(W)$$

- Insgesamt: $cost(H) \leq cost(W) = 2 \cdot cost(T) \leq 2 \cdot cost(H_{opt})$

8 Bestimmung der Basensequenz

Gelelektrophorese und Kettenabbruch-Methode

- Gelelektrophorese:
  - Trennung von DNA-Molekülen entsprechend ihrer Länge
  - DNA-Moleküle sind negativ geladen
    → in einem elektrischen Feld wandern sie in Richtung des positiven Pols.
  - Idee: Gebe Gemisch zu trennende DNA auf Gel-artigen Träger
    - lege elektrisches Feld an
      → große Moleküle → langsame Wanderung im Gel
      → kleine Moleküle → schnelle Wanderung im Gel
      → Wanderungsgeschwindigkeit ist antiproportional zur Größe des Moleküls.

Kettenabbruchmethode:

- Verfahren zur (direkten) Sequenzierung von DNA mittels Gelelektrophorese

Bioinf (70) Methode:

Gegeben: 4 Reagenzgläser, beschriftet mit A, C, G, T

1) Erzeuge viele Kopien des zu sequenzierenden DNA (Einzelstränge)
2) Verteile die Kopien auf Reagenzgläser.
3) Gebe in jedes Reagenzglas $I \in \{A, C, G, T\}$ alle Nucleotide außer I hinein (also gebe C, G, T in (A))
4) Gebe in jedes Reagenzglas $I \in \{A, C, G, T\}$ in einem bestimmten Verhältnis:
   - Nucleotid I
   - chemisch veränderte Form von I, an das der Aufbau eines komplementären Strangs durch DNA-Polymerase abbricht.
5) Gebe in jedes Reagenzglas
   - DNA-Polymerase und Primer
   → es werden komplementäre Stränge synthetisiert
   → mit hoher Wahrscheinlichkeit enthält I alle Einzelstränge, die auf dem Nucleotid I enden (gegen dem Einbau der chem. veränderten Form)
6) Gebe die Inhalte des Reagenzgläser nebeneinander auf ein Gel und starte die Gelelektrophorese
   → Lange Stränge entsprechend ihrer Länge
   → Lese die Sequenz ab (Read)

→ Hiermit können Moleküle bis zu 1000 bp lang sequenziert werden.

8.1. Shotgun-Sequenzierung

Modul 8.1: Shotgun-Sequenzierung

Eingabe: Ein DNA-Molekül D

1) Erzeuge Kopien $C = \{D_1, \ldots, D_m\}$ von D
2) Zerlege jede Kopie in kleine Fragmente (zufällig)
   → Menge einander überlappender DNA-Fragmente
3) Ermittle die Sequenz der Fragmente (Kettenabbruch-Methode) (oder Anfangsstücke der Fragmente)
   → Eine Menge von Strings über dem Alphabet
   $\Sigma_{DNA} = \{A, C, G, T\}$

Ausgabe: Die Menge der ermittelten DNA-Fragment-Sequenzen
$S = \{s_1, \ldots, s_n\}$

Definition: Fragment-Assembly-Problem

Eingabe: Menge von Strings $S = \{s_1, \ldots, s_n\}$

Ausgabe: Anordnung der Strings, die der ursprünglichen Anordnung in dem Molekül entspricht.

Beispiel: DNA der Länge $L = 100$ kbp

Fragmentanzahl $n = 1500$

durchschnittliche Länge der Fragmente $f = 500$ bp

Datenmenge: $n \cdot f = 750$ kbp

durchschnittliche Überdeckung / Coverage: $7{,}5 = \frac{n \cdot f}{L}$

23.6.03

Schema zur Lösung des Fragment-Assembly-Problem:

1.) Overlap-Bestimmung: - Überlappungen zwischen den einzelnen Strings ermitteln
- Überlappungen müssen nicht notwendig Suffix-Präfix-Paare sein ⇒ Alignments

2.) Layout: Anordnung der Strings ⇒ semiglobales multiples Alignment

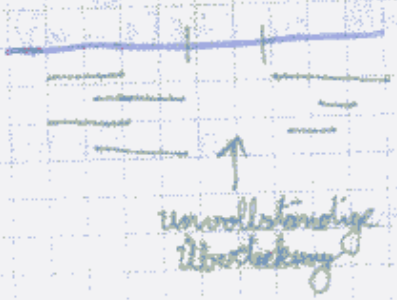
3.) Consensus: Bestimmung der Sequenz aus dem Layout.

→ Ziel: Modellbildung für die Layout-Phase.

8.1.1. Fehlerquellen und Probleme beim Fragment-Assembly-

- Sequenzierungsfehler:
  - Einfügung / Insertion: Einfügen einer Base wo diese nicht vorhanden ist
    AGTAXCA
  - Löschung / Deletion: Löschen einer Base
    AGTATTGCA
  - Ersetzung / Substitution: Ersetzen einer Base durch eine andere
    ACCTGAAC
    ↓
- Chimäre: Fragmente, die aus unterschiedlichen Bereichen der Ursprungs-DNA stammen und sich fälschlicherweise zusammengelagert haben.

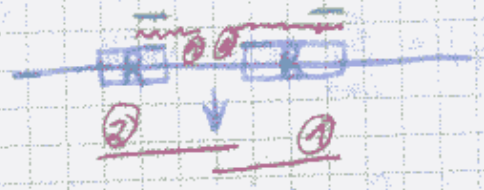
- unvollständige Überdeckung der Ursprungs-DNA



z.B. da Fragment toxische Wirkung auf einen Wirtsorganismus hat.

- Unbekannte Orientierung der Fragmente

- Repeats: Teilstrings der DNA, die an mehreren verschiedenen Stellen identisch (fast identisch) auftreten
  Große Varianz über die Länge, Häufigkeit, Ähnlichkeit der Repeats
  Problematik: - Überlappung wegen Repeats



- Anordnung der Repeats unklar

- Länge der Ursprungs-DNA ins Modell einbinden.

⇒ ideale Daten ≙ Kreis der vorangegangen Teilstrings Teil für Teil auf.

8.1.2. Shortest-Common-Superstring-Problem

Def. 8.2: Das Shortest-Common-Superstring-Problem (SCS) ist das folgende Optimierungsproblem:

Eingabe: Eine Menge $S = \{s_1, \dots, s_n\}$ von Strings über einem Alphabet Σ.

zulässige Lösungen: Jedes Superstring w von S, d.h. w enthält alle s_i ($i \in \{1,\dots,n\}$) als Teilstring

Kosten: Länge von w, $cost(w) = |w|$

Optimierungsziel: Minimierung

Def. 8.3: Eine Menge S von Strings heißt teilstringfrei, wenn kein Paar $s,t \in S$ existiert mit $s \neq t$ und s ist Teilstring von t.

Def. 8.4: Der triviale Superstring w_T einer Menge $S = \{s_1,\dots,s_n\}$ entspricht der Konkatenation aller Strings in S, also

$$w_T = s_1 \cdot s_2 \cdot \dots \cdot s_n$$

Die Länge des trivialen Superstrings

$$|w_T| = \sum_{i=1}^{n} |s_i| = \overline{|S|}\cdot|S|$$

Def. 8.5: Sei w ein Superstring einer Menge $S = \{s_1,\dots,s_n\}$. Die Kompression von w ist definiert als

$$comp(w,S) = |S| - |w|$$

Def. 8.6: Das Maximum-Compression-Version-Superstring-Problem (MCSS)

Eingabe: $S = \{s_1,\dots,s_n\}$ von Strings

zulässige Lösungen: Jedes Superstring w von S.

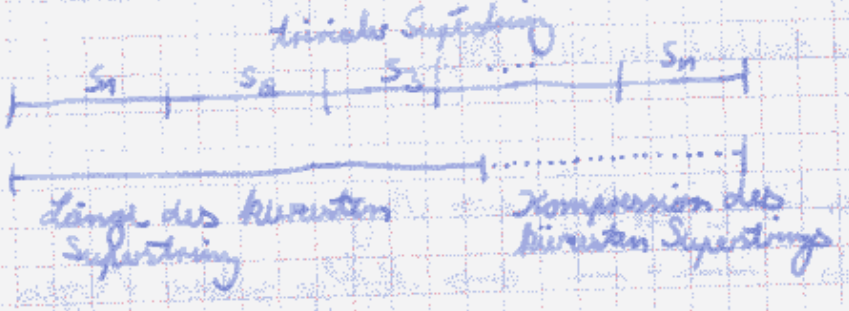
Kosten: Die Kompression des Superstrings

$$cost(w) = comp(w,S)$$

Optimierungsziel: Maximierung

SCS: Shortest-Common-Superstring-Problem mit Längenmaß

MCCS: Shortest-Common-Superstring-Problem mit Kompressionsmaß



Zur Erinnerung: Seien s,t Strings

- die Menge $\langle s,t\rangle$ von sind t die Verschmelzung zweier Strings mit maximalen Überlappungsbereich, $s=uv$, $t=vw$, v maximal $\quad \langle s,t\rangle = uvw$

- Overlap, $Ov(s,t)=v$, $\quad ov(s,t)=|v|$
- Präfix, $Pref(s,t)=u$, $\quad pref(s,t)=|u|$ (Distanz)
- (Suffix, $Suff(s,t)=w$, $\quad suff(s,t)=|w|$)

$\rightarrow pref(s,t)=|s|-ov(s,t)$

Def. 8.7: $S=\{s_1,\dots,s_n\}$ von Strings

Overlap-Graph $G_{ov}(S)$, gewichtet, gewichtet, vollständiger Graph (V,E,c)

$V:=S$, $E:=V^2$

$c: E\rightarrow \mathbb{N}$ mit $c(s_i,s_j)=ov(s_i,s_j)$ $\forall s_i,s_j\in V$

analog: Distanz-Graph $G_{pref}(S)$

$V:=S$, $E:=V^2$

$c: E\rightarrow \mathbb{N}$ mit $c(s_i,s_j)=pref(s_i,s_j)$

Satz 8.1: Sei $S = \{s_1, \dots, s_n\}$ Menge von Strings über Σ.
die Erstellung eines Overlap- bzw. Distanzgraphen aus S
benötigt eine Laufzeit in $O(n \cdot \|S\|)$.

- Kante in $G_{ov}(S)$ $[G_{pref}(S)]$ $\leftrightarrow$ Menge
- Kantenfolge in $\dots$ $\leftrightarrow$ Hintereinanderschreiben der Mengen
$(s_1, s_2, s_3, \dots, s_n)$ gerichteter Pfad in $G_{ov}(S)$ $[G_{pref}(S)]$

$$\updownarrow$$

$$\langle s_1, s_2, s_3, \dots, s_n \rangle = \overrightarrow{pref}(s_1, s_2)\, \overrightarrow{pref}(s_2, s_3) \dots \overrightarrow{pref}(s_{n-1}, s_n)\, s_n$$

Def 8.5: Sei $S = \{s_1, \dots, s_n\}$ eine Menge von Strings
π eine Anordnung der Strings.
Dann ist die durch π induzierte Superstring

$$w_\pi = \langle s_{i_1}, s_{i_2}, \dots, s_{i_n} \rangle$$

$$\left(\pi = (s_{i_1}, s_{i_2}, \dots, s_{i_n})\right)$$

$\rightarrow$ Zurückführung des SCS auf TSP

$$Opt_{TSP} \leq Opt_{SCS} - ov_{min} \leq Opt_{SCS}$$

Satz 8.2: Die Entscheidungsvariante für das SCS ist NP-vollständig!
$\rightarrow$ Reduktion auf das HK-Problem.

Algorithmus 8.1: Greedy-Superstring

Eingabe: Eine Menge von Strings $S = \{s_1, \dots, s_n\}$

while $|S| > 1$

1) Bestimme $s_i, s_j \in S$, $s_i \neq s_j$ mit dem maximalen Overlap aller Strings in S.

2) Sei $s' = \langle s_i, s_j \rangle$ der Menge von s_i, s_j

3) Lösche s_i, s_j aus S und füge s' hinzu

Ausgabe: Den einzigen verbleibenden String $\text{"Greedy"} \in S$.

30.6.03

Greedy-Algorithmus bzgl. Overlap-Graphen

- wähle immer die Kante (s_i, s_j) mit dem größten Gewicht, die keine Schleife ist.
- verschmelze die Knoten s_i und s_j zu einem einzigen Knoten $\langle s_i, s_j \rangle$
- lösche alle ausgehenden Kanten von s_i und alle eingehenden Kanten von s_j

$\Rightarrow$ bis nur noch ein Knoten verbleibt.

...noch abstrakter:

$\rightsquigarrow$ Hamiltonscher Pfad im Overlap-Graphen

Beispiel 8.3:

$S = \{$cababaa, caba, aadda, aabca, aacab$\}$

$\langle$caba, ababaa$\rangle$ = cababaa

$\{$aadda, aabca, aacab, cababaa$\}$

$\langle$aacab, cababaa$\rangle$ = aacababaa

$\{$aadda, aabca, aacababaa$\}$

$\langle aacabababaa, aaddbb\rangle = aacababaaaaddbb$

$\{aa, bca, aacababaaaaddbb\}$

$\langle aabca, aacababaaaddd\rangle = aabcaacabababaddbb$

Länge 16, Kompression von 9

Satz 8.3: Der Algorithmus Greedy-Superstring benötigt für eine Eingabe $S=\{s_1,\dots,s_n\}$ eine Laufzeit in $O(n\cdot\|S\|)$.

Beweis: (verwende die Analogie der Berechnung eines Hamiltonschen Pfades im Overlap-Graphen)

$O(n\cdot\|S\|)$:
- Erstellung des Overlap-Graphen ($\rightarrow$ Satz 8.1) $\Rightarrow O(n\cdot\|S\|)$
- Sortierung der Kanten entsprechend ihres Gewichtes (Overlap)
  $O(n^2\log(n))$ $\Rightarrow$ mit Bin-Sort $\underbrace{O(n^2+\|S\|)}_{\leq O(n\cdot\|S\|)}$
  $\Rightarrow$ Adjazenzmatrix und geordnete Liste der Kanten

$O(n^2+\|S\|) \Rightarrow O(n\cdot\|S\|+\|S\|) \Rightarrow O(n\cdot\|S\|)$

zu Schritt 1: Zugriff auf die Kante mit max. Gewicht $\Rightarrow O(1)$

zu Schritt 2 und 3: gewähltes Paar (s_i, s_j)

$O(n)$:
- Verbot von ausgehenden Kanten von s_i $\rightarrow$ markiere die i-te Zeile
- Verbot von eingehenden Kanten von s_j $\rightarrow$ markiere die j-te Spalte
- Verbot eines Zyklus: $\rightarrow$ markiere die Position (j,i) in der Adjazenzmatrix

- Lösche markierte Elemente vom Beginn der geordneten Liste
  $\Rightarrow O(n)$

- (Protokollieren der durchgeführten Merge $O(1)$)

$\Rightarrow$ Die Schritte 1 bis 3 werden $O(n)$-mal durchgeführt

$\Rightarrow$ Insgesamt: $O(n \cdot |S|) + O(n) \cdot O(n)$
$= O(n \cdot |S|)$ □

Beispiel 8.4:

$S = \{c(ab)^m, (ba)^m, (ab)^m c\}$

Greedy: $\langle c(ab)^m, (ab)^m c \rangle = c(ab)^m c$
$\langle (ba)^m, c(ab)^m c \rangle = \underbrace{(ba)^m c (ab)^m c}_{2m+2+2m = 4m+2}$

Optimal: $cabab\,ab \ldots ab$
$babab\,ab \ldots aba$
$ababab \ldots ababc$
$\Rightarrow ca(ba)^m bc$ Länge $2m+4$

Approximationsgüte $= \frac{4m+2}{2m+4} \xrightarrow[m \to \infty]{} 2$

Satz 8.4: Der Algorithmus Greedy-Superstring ist ein 4-Approximationsalgorithmus für das SCS. □

Satz 8.5: Der Algorithmus Greedy-Superstring ist ein 2-Approximationsalgorithmus für das MCCS. □

Satz 8.6: Der Algorithmus Greedy-Superstring ist ein 3-Approximationsalgorithmus für das MCCS.

Beweis: $w_{opt} = \langle s_{i_1}, \dots, s_{i_n} \rangle$, $comp(w_{opt}) \triangleq \sum$ der Overlaps des Merges

$w_{greedy} = \langle s_{j_1}, \dots, s_{j_n} \rangle$

„Ein Merge von Greedy kann maximal 3 Merges der optimalen Lösung verhindern!"

$m = \langle s_{j_k}, s_{j_{k+1}} \rangle$

1) m tritt auch in der optimalen Lösung auf.
   $\Rightarrow$ kein Merge verhindert

2) $m = \langle s_{i_\ell}, s_{i_{\ell+x}} \rangle$
   $\Rightarrow$ zwei Merges werden verhindert, nämlich
   $\langle s_{i_\ell}, s_{i_{\ell+1}} \rangle$ und $\langle s_{i_{\ell+x-1}}, s_{i_{\ell+x}} \rangle$

3) $m = \langle s_{i_{\ell+x}}, s_{i_\ell} \rangle$
   $\Rightarrow$ drei Merges werden verhindert, nämlich
   $\langle s_{i_{\ell+x}}, s_{i_{\ell+x+1}} \rangle, \langle s_{i_{\ell-1}}, s_{i_\ell} \rangle$, ein Merge der Form $\langle s_{i_\ell}, s_{i_{\ell+1}} \rangle \dots \langle s_{i_{\ell+x-1}}, s_{i_{\ell+x}} \rangle$ (Zyklus)

$\rightsquigarrow$ Da Greedy die Kante mit max. Overlap wählt, ergibt sich ein Superstring, der mindestens 1/3 des Overlaps der optimalen Lösung besitzt. □

Def. 8.10: Sei $G = (V, E, c)$ ein vollständiger, gewichteter, gerichteter Graph mit Gewichtsfunktion $c: E \to \mathbb{N}$.
Ein Cycle-Cover CC von G besteht aus einer Menge gerichteter Kreise in G, $CC = \{C_1, \dots, C_k\}$, so daß jeder Knoten in G in genau einem Kreis C_i vorkommt.

Die Kosten cost(CC) eines Cycle-Covers CC mit $C=\{c_1,\dots,c_k\}$ entsprechen der Summe der Kantengewichte in den einzelnen Kreisen

$$cost(CC) := \sum_{i=1}^{k} \sum_{e \in c_i} c(e)$$

Ein minimaler CC_{min} ist ein Cycle-Cover mit minimalen Kosten.

⇝ Berechnung min. Cycle-Cover ist in polynomieller Zeit möglich.

Algorithmus 6.2 Cycle-Cover-Superstring-Algorithmus

Eingabe: $S=\{s_1,\dots,s_n\}$, $G_{pref}(S)$ den zugehörigen Distanzgraphen.

1) Berechne den min. Cycle-Cover C von $G_{pref}(S)$
2) Für jeden Kreis $c \in C$ wähle (beliebig) einen Repräsentanten r_c.
   Die Menge der Repräsentanten $R=\{r_c \mid c \in C\}$
3) Bestimme den durch R induzierten Teilgraphen G' von $G_{pref}(S)$
4) Berechne min. Cycle-Cover C' auf G'.
5) In jedem Kreis $c'=(c'_1,\dots,c'_{k_c}) \in C'$ lösche die Kante, die im Overlap-Graphen das minimale Gewicht in c' besitzt.

   Sei o.B.d.A. (c'_{k_c}, c'_1) diese Kante.
   Bezeichnen wir die Strings in dem entstehenden Pfad

   $$\forall c' \in C' : \quad w_{c'} = \langle c'_1,\dots,c'_{k_c} \rangle$$

6) Konkateniere alle diese Strings $w_{c'}$ für alle $c' \in C'$ und bezeichne den resultierenden String mit s'.
7) Sei $c=(c_1,\dots,c_{k_c}) \in C$. O.B.d.A. sei $r_c = c_1$.
   Ersetze jeden Repräsentanten r_c in s' durch die Konkatenation aller Präfixe in Kreis c_1, also durch

   $$Pref(r_c, c_2)\, Pref(c_2, c_3) \cdots Pref(c_{k_c-1}, c_{k_c})\, Pref(c_{k_c}, r_c)\, r_c$$

BioInf (SS2) Bezeichne den resultierenden String mit w.

Ausgabe: Die Superstring w.

Satz 8.7: Algorithmus Cycle-Cover-Superstring ist ein 3-Approximationsalgorithmus für SCS.

27.2003

Beweis: Skizze:

(i) Kosten min Cycle-Covers auf dem Distanzgraphen ≤ Länge des kürzesten Superstrings.

(ii) Abschätzen der Länge des Strings u' (Schritt 5 Alg.)

(iii) Abschätzen der Länge von w' bzgl. Cycle-Cover C' auf G' und dem Overlap der gelöschten Kanten

(iv) Abschätzen der Länge von w bzgl. Cycle-Cover C', Cycle-Cover C und dem Overlap der gelöschten Kanten

(v) Abschätzen des Overlaps durch den Cycle-Cover C

Bezeichnungen:

- $Opt_{SCS}(S)$ ≙ Länge des kürzesten Superstrings für S
- $Opt_{CC}(G)$ ≙ Kosten eines optimalen Cycle-Covers für G
- c : Kreis im Cycle-Cover C von G
- c' : Kreis im Cycle-Cover C' von G'
- $min\text{-}ov_{c'}$ ≙ Kosten des minimalen Overlap in $c' \in C'$.
- $min\text{-}ov_{C'}$ ≙ Summe der $min\text{-}ov_{c'}$ $\forall c' \in C'$
- $sum\text{-}ov_{c'}$ ≙ Summe aller Overlaps in einem Kreis $c' \in C'$
- $sum\text{-}ov_{C'}$ ≙ Summe der $sum\text{-}ov_{c'}$ $\forall c' \in C'$

zu (i): Kreis $(s_{i_1}, \dots, s_{i_k}) \rightsquigarrow$ String $\langle s_{i_1}, \dots, s_{i_k} \rangle$

$$\text{cost}((s_{i_1}, \dots, s_{i_k})) = \text{pref}(s_{i_1}, s_{i_2}) + \dots + \text{pref}(s_{i_{k-1}}, s_{i_k}) + \text{pref}(s_{i_k}, s_{i_1})$$

$$\text{cost}(\langle s_{i_1}, \dots, s_{i_k} \rangle) = \text{pref}(s_{i_1}, s_{i_2}) + \dots + \text{pref}(s_{i_{k-1}}, s_{i_k}) + |s_{i_k}|$$

Da $\text{pref}(s_{i_k}, s_{i_1}) \leq |s_{i_k}|$

$$\Rightarrow \text{cost}(C) = \text{opt}_{CC}(G) \leq \text{opt}_{SCS}(S) \quad (1)$$

$$\Rightarrow \text{cost}(C') = \text{opt}_{CC}(G') \leq \text{opt}_{SCS}(R) \quad (2)$$

Da G' Teilgraph von G ist:

$$\Rightarrow \text{cost}(C') \leq \text{cost}(C) \quad (3)$$

zu (ii): Aufbrechen eines Kreises $C' = (c'_1, \dots, c'_k) \in C'$ an einer Kante

o.B.d.A (c'_k, c'_1) mit min. Overlap.

$$|u_{C'}| = \text{pref}(c'_1, c'_2) + \dots + \text{pref}(c'_{k-1}, c'_k) + |c'_k|$$

$$= \underbrace{\text{pref}(c'_1, c'_2) + \dots + \text{pref}(c'_{k-1}, c'_k) + \text{pref}(c'_k, c'_1)}_{= \text{cost}(C')} + |c'_k| - \text{pref}(c'_k, c'_1)$$

$$= \text{cost}(C') + \underbrace{|c'_k| - \text{pref}(c'_k, c'_1)}_{= \text{ov}(c'_k, c'_1)}$$

$$= \text{cost}(C') + \min\text{-ov}_{C'}$$

$$\Rightarrow |u_{C'}| = \text{cost}(C') + \min\text{-ov}_{C'} \quad (4)$$

zu (iii): w' ergibt sich als Konkatenation aller w_c'

$$\Rightarrow |w'| \leq \underbrace{\sum_{c' \in C'} \text{cost}(c')}_{\text{cost}(C')} + \underbrace{\sum_{c' \in C'} \text{min-or}_{c'}}_{\text{min-or}_{C'}} \tag{5}$$

zu (iv): Ersetzen τ_c in w' durch

$$\tau_c \to \underbrace{\text{Pref}(\tau_c, c_{i_2}) \dots \text{Pref}(c_{i_{k_c-1}}, c_{i_{k_c}}) \text{Pref}(c_{i_{k_c}}, \tau_c)\tau_c}_{\text{dieses Wort entspricht dem Kanten des Kreises cost}(c)}$$

$\Rightarrow$ Verlängerung um $\text{cost}(c) \quad \forall c \in C$

$$\Rightarrow |w| \leq |w'| + \text{cost}(C)$$
$$\overset{(5)}{\leq} \text{cost}(C') + \text{min-or}_{C'} + \text{cost}(C)$$
$$\overset{(3)}{\leq} 2 \cdot \text{cost}(C) + \underbrace{\text{min-or}_{C'}} \tag{6}$$

zu (v): Jeder Kreis besteht aus mindestens 2 Kanten

$$\Rightarrow \text{min-or}_{C'} \leq \tfrac{1}{2} \cdot \text{sum-or}_{C'} \tag{7}$$
$$\Rightarrow |w| \leq 2 \cdot \text{cost}(C) + \tfrac{1}{2} \cdot \text{sum-or}_{C'}$$

Hilfsaussage:

Lemma 5.1: Seien c_1 und c_2 Kreise in einem minimalen Cycle-Cover und $s_1 \in c_1$ und $s_2 \in c_2$ zwei Strings in diesem Kreisen, dann gilt:

$$\text{or}(s_1, s_2) \leq \text{cost}(c_1) + \text{cost}(c_2)$$

(ohne Beweis) $\square$

Da alle Knoten in einem Kreis $c' \in C'$ zu unterschiedlichen Knoten in C gehören, folgt:

$$\text{sum-ov}_{C'} \leq 2 \cdot \text{cost}(C) \qquad (8)$$

mit (8) und (1) folgt:

$$|w| \leq 2 \cdot \text{cost}(C) + \tfrac{1}{k} \cdot \text{sum-ov}_{C'}$$

$$\overset{(8)}{\leq} 3 \cdot \text{cost}(C)$$

$$\overset{(1)}{\leq} 3 \cdot \text{opt}_{SCS}(S) \qquad \square$$

8.1.3 Das Reconstruction-Modell

Ziel: Einbringung von Sequenzierungsfehlern und der unbekannten Orientierung.

Def. 8.11: Seien s, t Strings über Σ. Dann ist die Teilstring-Edit-Distanz $\text{ed}_S(s,t)$ definiert als

$$\text{ed}_S(s,t) = \min_{x \in \text{Substr}(t)} \text{ed}(s,x),$$

wobei $\text{Substr}(t)$ die Menge aller Teilstrings von t und ed die übliche Edit-Distanz (siehe Alignment)

Def. 8.12: Das Reconstruction-Problem

Eingabe: $S = \{s_1, \dots, s_n\}$, $\varepsilon \in [0,1]_{\mathbb{R}}$ (Fehlertoleranzwert)

zulässige Lsg: Jeder String w, sodass für alle $i = 1, \dots, n$

$$\min\{\text{ed}_S(s_i, w), \text{ed}_S(\bar{s}_i, w)\} \leq \varepsilon \cdot |s_i|$$

wobei $\overline{s_i}$ das revers Komplement von s_i ist.

Kosten: Die Länge von v.

Ziel: Minimierung

Satz 8.5 Das Rekonstruktion-Problem ist NP-schwer. □

8.2 Sequenzierung durch Hybridisierung

Methode 8.2: Sequenzierung durch Hybridisierung (SBH)

Eingabe: Die zu sequenzierende DNA und $l \in \mathbb{N}$

1) Erzeuge einen DNA-Chip mit allen verschiedenen Proben der Länge l.
2) Erzeuge Kopien der DNA.
3) Führe Hybridisierungsexperiment durch

Ausgabe: Menge $S = \{s_1, \dots, s_n\} \subseteq \Sigma_{DNA}^l$ der Länge l, die als Teilstrings in der DNA auftreten.
(→ Spektrum (der DNA))

Def. 8.16: Sei w ein String und $S = \{s_1, \dots, s_n\}$ ein Spektrum mit Strings der Länge l.
w kompatibel zu S, falls w jeden String aus S als Teilstring enthält und keinen anderen String der Länge l enthält.

w einfach-kompatibel zu S, falls w kompatibel zu S und w keinen String in S mehrfach enthält.

→ Ziel: Finde einen kompatiblen String w.

Def. 8.18:

Sei S ein Spektrum mit Strings der Länge l.

Spektrum-Graph $G_{Spektrum}(S) = (V, E, label)$

- $V := \Sigma^{l-1}$ (alle Strings der Länge $l-1$)
- $E = \{(x,y) \mid x,y \in V$, es exist. ein $s \in S$ mit $\langle x,y \rangle = s\}$.
- $label((x,y)) = y_{l-1}$

Sei $x_1, x_2, \dots, x_k$ ein Pfad in $G_{Spektrum}(S)$

$pathlabel(x_1, x_2, \dots, x_k) = x_1\, label((x_1,x_2)) \dots label((x_{k-1}, x_k))$

Eulerscher Pfad := Ein Pfad in einem Graphen, der jede Kante genau einmal durchläuft. (Pfad nicht notwendig zustandsdisjunkt)

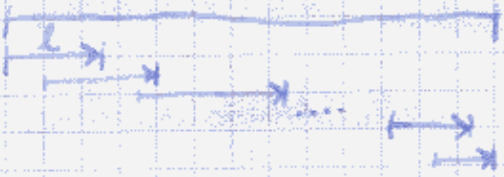
Eulerscher Kreis := Ein Kreis in einem Graphen, der jede Kante genau einmal durchläuft.

Graph G, der einen Eulerschen Kreis enthält, heißt eulersch.

Satz 8.19: Sei S ein Spektrum mit Strings der Länge l. Ein String w ist genau dann einfach-kompatibel zu S, wenn es der Beschriftung eines Eulerschen Pfades im Spektrum-Graphen entspricht.

Beweis: $t \in S$, $t = x\,\text{label}((x,y))\,y$ $\quad x,y \in V$

$\Rightarrow$ v ein einfach-kompatibler String.



„Jeder Block der Länge l in v entspricht einer Kante in $G_{\text{Spektrum}}(S)$"

$\Leftarrow$ Das Pathlabel eines Eulerschen Pfades entspricht einem einfach-kompatiblen String. $\square$

$\Rightarrow$ Eulerkreise / Eulerpfade sind effizient berechenbar.

$\Rightarrow$ es ist effizient möglich einfach-kompatible Strings zu einem Spektrum zu finden S

- Fehlerbehandlung?
- Große Anzahl verschiedener Eulerpfade!

7.7.2003

9. Bestimmung von Signalen in DNA-Sequenzen

Ziel: Finde „interessante" Regionen in der DNA, z.B.

- Restriktionsstellen
- Gene
- Bindungsstellen von Proteinen
- ...

Ansätze: meist statistische Methoden

3.1 Gleiche und ähnliche Teilstrings

Ziel: Bestimmung von Bindungsstellen, d.h. von Regionen der DNA, an denen sich ein bestimmtes Protein anlagern kann, um z.B. die Transkription eines Gens zu steuern.

Biologisches Experiment: liefert DNA-Fragment der Länge m, in dem sich (mit hoher W'keit) eine Bindungsstelle der Länge $l \ll m$ befindet.

Problem: Finde in den DNA-Fragmenten mehrere solche Experimente einen möglichst langen gemeinsamen Teilstring (magisches Wort).

Exakte Lösung: mit Suffix-Bäumen (Alg. 4.10)

Probleme:
- Komplizierte Muster: Restriktionsstelle des Restriktionsenzyms XcmI: CCA*********TGG
- Mismatches: Finde magisches Wort, das in allen DNA-Fragmenten ungefähr vorkommt
  ⇒ Spezialfall eines lokalen multiplen Alignments ohne Lücken.

Def. 3.1: Sein $S = s_1, \dots, s_m$ und $t = t_1, \dots, t_m$ zwei Strings der Länge m. Der Hamming-Abstand $d_H(s,t)$ von s und t ist definiert als die Anzahl der Positionen $1 \le i \le m$ mit $s_i \neq t_i$.

Def. 3.2: Das Consensus-String-Problem:

Eingabe: Menge von n Strings $\{s_1, \dots, s_n\} \in \Sigma^m$, $l \in \mathbb{N}$

zulässige Lösungen: Alle $(n+1)$-Tupel $(t, t_1, \dots, t_n)$, wobei t_i Teilstrings der Länge l von s_i für $1 \le i \le n$. Der String $t \in \Sigma^l$ wird Median-String genannt.

Kosten: $\text{cost}(t, t_1, \ldots, t_n) = \sum_{i=1}^{n} d_H(t, t_i)$

Optimierungsziel: Minimierung

Satz 3.1: Das Consensus-String-Problem ist NP-schwer. □

$\Rightarrow$ Approximations-Algorithmus:

Algorithmus 3.1 Consensus-String-Approximation

Eingabe: Menge $S = \{s_1, \ldots, s_n\} \in \Sigma^m$, $k, r \in \mathbb{N}$

1. Initialisierung: $c' := \infty$
   $u' := \lambda$
   for $i = 1$ to n do $v_i' := \lambda$

2. for all $(u_1, \ldots, u_r)$, wobei $u_i \in \Sigma^k$ für alle i ein Teilstring eines Strings in S ist do
   Berechne u als den Consensus von $u_1, \ldots, u_r$ (Def. 5.11)
   for $i := 1$ to n do
   Berechne v_i als den Teilstring von s_i mit minimalem Hamming-Abstand zu u
   $c := \sum_{i=1}^{n} d_H(u, v_i)$
   if $c < c'$ then
   $c' := c$
   $u' := u$
   for $i := 1$ to n do $v_i' := v_i$

Ausgabe: $(u', v_1', \ldots, v_n')$ mit Kosten c'

Satz 3.2: Der Alg. 3.1 ist für jedes $r \geq 3$ ein polynomielle Approximationsalg. für das Consensus-String-Problem mit einer Approximationsgüte von

$$1 + O\left(\sqrt{\frac{\log r}{r}}\right)$$

und einer Laufzeit in $O\left((m-l+1)^{r+1} \cdot n^{r+1} \cdot l\right)$

Beweisidee: Laufzeit: Es gibt n Strings in S und $m-l+1$ mögliche Anfangspositionen in jedem der Strings aus S für jedes s_i

$\Rightarrow (n \cdot (m-l+1))^r$ Möglichkeiten, $(s_{i_1}, \ldots, s_{i_r})$ zu wählen.

Approximationsgüte: Sei $S = \{s_1, \ldots, s_n\} \in \Sigma^m$, $l, r \in \mathbb{N}$, $r \geq 3$

Sei $(s, t_1, \ldots, t_n)$ eine opt. Lösung mit Kosten

$$c_{opt} = \sum_{i=1}^{n} d_H(s, t_i)$$

Für alle $(i_1, \ldots, i_r) \in \{1, \ldots, n\}^r$ sei $s_{i_1 \ldots i_r}$ ein Consensus von $t_{i_1}, \ldots, t_{i_r}$ und es sei $c_{i_1 \ldots i_r} = \sum_{i=1}^{n} d_H(s_{i_1 \ldots i_r}, t_i)$

Idee: Approximiere den opt. Consensus s durch ein $s_{i_1, \ldots, i_r}$ für ein $(i_1, \ldots, i_r)$.

Man kann zeigen, daß sich für r unabhängig gleichverteilt zufällig gewählte Stellen $i_1, \ldots, i_r$ die Abschätzung von $c_{i_1 \ldots i_r}$ wie folgt abschätzen läßt:

$$E[c_{i_1 \ldots i_r}] \leq \left(1 + O\left(\sqrt{\frac{\log r}{r}}\right)\right) \cdot c_{opt}$$

$\Rightarrow$ es exist. $i_1, \dots, i_r$ mit

$$c_{i_1 \dots i_r} \leq \left(1 + O\left(\sqrt{\frac{\log^r}{n}}\right)\right) \cdot c_{opt}$$

Der Alg. hat alle möglichen r-Tupel untersucht $\Rightarrow$ Behauptung $\square$

9.3. Häufige und seltene Teilstrings

Ziel: Häufigkeit der vorkommenden Teilstrings analysieren

Idee: Teilstrings, die signifikant häufiger oder seltener auftreten, werden als interessante Regionen klin.

Beispiel: DNA von Bakteriophagen:
4-6 Basenpaare lange Teilstrings, die den Bindungsstellen von Restriktionsenzymen entsprechen, treten signifikant seltener auf.

Problem: Für jeden Teilstring t der Länge l in einem gegebenen String s der Länge n vergleiche die Anzahl der Vorkommen $h(t)$ mit der erwarteten Anzahl von Vorkommen in einem zufälligen String der Länge n.

$\Rightarrow$ Berechne Erwartungswert und Varianz

Alg. 9.5: Häufigkeits-Analyse für Teilstrings

Eingabe: String s der Länge n, $l \in \mathbb{N}$

for alle $t \in \Sigma^l$ do

- Bestimme die Anzahl $h(t)$ der Vorkommen von t in s.
- Bestimme die erwartete Anzahl der Vorkommen von t in einem zufälligen String der Länge n und deren Varianz.

Ausgabe: Alle Strings t, deren tatsächliche Häufigkeit signifikant von der erwarteten Häufigkeit abweicht.

Es gibt: $\mathrm{Var}(X) = E[(X - E[X])^2]$

Problem: Varianz hängt von dem konstanten String selbst ab, nicht nur von seiner Länge.

Def. 9.7: Sei $t = t_1 \dots t_\ell$ ein String. Die Autokorrelation von t ist ein Binärstring $c(t) = c_0^{(t)} \dots c_{\ell-1}^{(t)}$, wobei

$$c_i^{(t)} = \begin{cases} 1 & \text{falls } t_1 \dots t_{\ell-i} = t_{i+1} \dots t_\ell \\ 0 & \text{sonst} \end{cases}$$

Das Autokorrelationspolynom von t ist definiert als:

$$\mathrm{corr}_t(x) = \sum_{i=0}^{\ell-1} c_i^{(t)} \cdot x^i$$

Beachte: $c_0^{(t)} = 1$ für alle t

Vereinbarung: der String s sei zyklisch, d.h. wir zählen auch Vorkommen $t_1 \dots t_\ell = s_{m-\ell+1} \dots s_m s_1 \dots s_{\ell-1}$

Satz 9.6: Sei Σ ein Alphabet der Größe k, sei $s_1, \dots, s_m$ ein zyklischer Bernoulli-String der Länge m, d.h. jedes s_i sei zufällig gleichverteilt und unabhängig aus Σ gewählt.

Sei $t \in \Sigma^\ell$ ein in s zu suchendes Muster.

Sei für $1 \le i \le m$ die Zufallsvariable X_i def. durch

$$X_i = \begin{cases} 1 & \text{falls } t \text{ an Position } i \text{ von } s \text{ beginnt} \\ 0 & \text{sonst} \end{cases}$$

Dann ist die Anzahl der Vorkommen von t in s durch

BioInf (34)

die Zufallsvariable $X = \sum_{i=1}^{m} X_i$ gegeben.

Sei $p = E[X_i] = \frac{1}{4^\ell}$. Dann gilt

(a) $E[X] = m \cdot p$

(b) $\mathrm{Var}[X] = p \cdot m \cdot \left(2\,\mathrm{corr}_\ell\left(\frac{1}{4}\right) - (2\ell-1)p - 1\right)$.

Beweis: (a.) Linearität des Erwartungswerts. ✓

(b.) $\mathrm{Var}[X] = E[X^2] - E[X]^2$

$$= \sum_{1 \le i,j \le m} \left(E[X_i \cdot X_j] - E[X_i] \cdot E[X_j]\right)$$

Sei $d(i,j)$ der kürzeste Abstand zwischen den Positionen i und j in dem zyklischen String s.

Dann gilt:

$$\mathrm{Var}[X] = \underbrace{\sum_{\substack{1 \le i,j \le m \\ d(i,j) > \ell}} \left(E[X_i \cdot X_j] - E[X_i] \cdot E[X_j]\right)}_{S_1}$$

$$+ \underbrace{\sum_{\substack{1 \le i,j \le m \\ d(i,j) = 0}} \left(E[X_i \cdot X_j] - E[X_i] \cdot E[X_j]\right)}_{S_2}$$

$$+ \underbrace{\sum_{\substack{1 \le i,j \le m \\ 0 < d(i,j) \le \ell}} \left(E[X_i \cdot X_j] - E[X_i] \cdot E[X_j]\right)}_{S_3}$$

$S_1 = 0$, da X_i und X_j unabhängig sind.

$$S_2 = \sum_{1 \le i \le m} \left(\underbrace{E[X_i \cdot X_i]}_{E[X_i] = p} - \underbrace{E[X_i] \cdot E[X_i]}_{p^2}\right) = m \cdot (p - p^2)$$

$$S_3 = \sum_{i=1}^{m} \sum_{r=1}^{l-1} \sum_{d(i,j)=r} \left(E[X_i \cdot X_j] - E[X_i] \cdot E[X_j] \right)$$

Wenn $c_r^{(t)} = 0$, dann ist $X_i \cdot X_{i+r} = 0$ für alle i

Wenn $c_r^{(t)} = 1$, dann ist es möglich, daß t in s an den Pos. i und $i+r$ beginnt:

$E[X_i \cdot X_{i+r}]$ = Produkt aus der W'keit des Auftretens von t an der Pos. $i+r$ und der W'keit, daß

$$s_i \dots s_{i+r-1} = t_1 \dots t_r$$

$$\Rightarrow E[X_i \cdot X_{i+r}] = c_r^{(t)} \cdot p \cdot \frac{1}{k^r}$$

Für jedes i gibt es genau zwei Positionen j mit $d(i,j) = r$.

$$\Rightarrow \sum_{i=1}^{m} \sum_{r=1}^{l-1} \sum_{d(i,j)=r} E[X_i \cdot X_j] = \sum_{i=1}^{m} \sum_{r=1}^{l-1} 2 \cdot p \cdot c_r^{(t)} \cdot \frac{1}{k^r}$$

$$= \sum_{i=1}^{m} 2p \cdot \left(\text{corr}_t\left(\frac{1}{k}\right) - 1 \right)$$

$c_0^{(t)} \cdot \frac{1}{k^0}$

$$\Rightarrow \sum_{i=1}^{m} \sum_{r=1}^{l-1} \sum_{d(i,j)=r} \left(E[X_i \cdot X_j] - E[X_i] \cdot E[X_j] \right)$$

$$= \sum_{i=1}^{m} 2p \cdot \left(\left(\text{corr}_t\left(\frac{1}{k}\right) - 1 \right) - 2(l-1) \cdot p^2 \right)$$

$$= p \cdot m \cdot \left(2 \, \text{corr}_t\left(\frac{1}{k}\right) - 2 - 2(l-1)p \right)$$

$$\Rightarrow \text{Var}[X] = p \cdot m \cdot \left(2 \cdot \text{corr}_t\left(\frac{1}{k}\right) - 2 - 2(l-1) \cdot p \right) + m(p - p^2)$$

$$= p \cdot m \cdot \left(2 \cdot \text{corr}_t\left(\frac{1}{k}\right) - (2l-1) \cdot p - 1 \right) \cdot$$

□

3.4. Hidden-Markov-Modelle

Ziel: Auffinden von sog. CG-Inseln, d.h. Bereichen in einer DNA-Sequenz, in denen CG wesentlich häufiger vorkommt als im Rest der DNA-Sequenz.

Def 3.8: Ein Hidden-Markov-Modell (HMM) ist ein Quintupel $\mathcal{M} = (\Sigma, Q, q_0, \delta, \gamma)$, wobei

- Σ Alphabet
- Q endl. Zustandsmenge
- $q_0 \in Q$ Anfangszustand
- δ eine $(|Q| \times |Q|)$-Matrix von Transitionswahrscheinlichkeiten
- γ eine $(|Q|-1) \times |\Sigma|$-Matrix von Emissionswahrscheinlichkeiten

$\delta(p,q) \triangleq$ W'keit des Übergangs von Zustand p nach Zustand q

$$\delta(p,p) = 0 \text{ und } \sum_{q \in Q} \delta(p,q) = 1 \text{ für alle } p \in Q$$

$\gamma(q,a) \triangleq$ W'keit, daß im Zustand q das Symbol a ausgegeben wird für $q \in Q - \{q_0\}$, $a \in \Sigma$

$$\sum_{a \in \Sigma} \gamma(q,a) = 1$$

Ein Pfad in $\mathcal{M}$ ist eine Folge $\pi = q_0, q_1, \ldots, q_n$ von Zuständen.

Beispiel 9.2: Würfel-Experiment

Für eine Reihe von Würfen wird heimlich ein fairer Würfel verwendet, der jede Zahl $\in \{1,\dots,6\}$ mit W'keit $\frac{1}{6}$ würfelt, und heimlich ein unfairer Würfel, der 6 mit W'keit $\frac{1}{2}$ und jede Zahl $\in \{1,\dots,5\}$ mit W'keit $\frac{1}{10}$ würfelt. Der Würfel wird mit W'keit $\frac{1}{20}$ gewechselt. Am Anfang wird mit W'keit $\frac{1}{2}$ einer der Würfel gewählt.

Lemma 9.3: Sei $M = (\Sigma, Q, q_0, \delta, \eta)$ ein HMM, sei $\pi = q_0, q_1, \dots, q_n$ ein Pfad in M und sei $x = x_1 \dots x_n \in \Sigma^n$. Dann gilt:

$$\text{Prob}[x \wedge \pi] = \prod_{i=1}^{n} \left(\delta(q_{i-1}, q_i) \cdot \eta(q_i, x_i)\right).$$

Beweis:

$$\text{Prob}[\pi] = \prod_{i=1}^{n} \delta(q_{i-1}, q_i)$$

$$\text{Prob}[x \mid \pi] = \prod_{i=1}^{n} \eta(q_i, x_i)$$

$$\text{Prob}[x \wedge \pi] = \text{Prob}[\pi] \cdot \text{Prob}[x \mid \pi] \qquad \square$$

Beispiel 9.3: Würfelexperiment aus Bsp. 9.2

$\pi = q_0, U, U, F \qquad x = 666$

$$\text{Prob}[x \wedge \pi] = \prod_{i=1}^{3} \left(\delta(q_{i-1}, q_i) \cdot M(q_i, x_i)\right)$$

$$= \delta(q_0, U) \cdot \eta(U, 6) \cdot \delta(U, U) \cdot \eta(U, 6) \cdot \delta(U, F) \cdot \eta(F,$$

$$= \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{19}{20} \cdot \frac{1}{2} \cdot \frac{1}{20} \cdot \frac{1}{6}$$

$$= \frac{19}{19.200} \approx 0,00099.$$

Def. 3.3: Das HMM-Dekodier-Problem:

Eingabe: HMM $\mathcal{H} = (\Sigma, Q, q_0, \delta, \eta)$, $x = x_1 \dots x_n \in \Sigma^n$

zulässige Lösungen: Alle Pfade $\pi = q_0, q_1, \dots, q_n$ der Länge n in $\mathcal{H}$.

Kosten: Für zul. Lsg. π: $\text{cost}(\pi) = \text{Prob}[x|\pi]$

Optimierungsziel: Maximierung

Lösungsansatz: Dynamische Programmierung:

Bestimme für alle Zustände und alle Präfixe von x den wahrscheinlichsten Pfad, der in diesem Zustand endet und diesem Präfix emittiert

Lemma 3.4: Sei $\mathcal{H} = (\Sigma, Q, q_0, \delta, \eta)$ ein HMM und sei $x = x_1 \dots x_n \in \Sigma^n$.

Sei $\sigma_q(i)$ die W'keit des wahrscheinlichsten Pfades für $x_1 \dots x_i$, der in q endet, für alle $q \in Q$, für alle $0 \le i \le n$.

Dann gilt:

$$\sigma_{q_0}(0) = 1$$

$$\sigma_q(0) = 0 \quad \text{für } q \in Q - \{q_0\} \tag{3.2}$$

und für alle $q \in Q - \{q_0\}$ und für alle $1 \le i \le n$ gilt:

$$\sigma_q(i) = \eta(q, x_i) \cdot \max_{p \in Q} \{\sigma_p(i-1) \cdot \delta(p, q)\} \tag{3.3}$$

Beweis: (9.2) ✓

(9.3): Sei $\pi = q_0, q_1, \dots, q_i$ der Pfad mit der höchsten Emissionswahrscheinlichkeit für $x_1 \dots x_i$, der in q_i endet.

$$\Rightarrow \sigma_{q_i}(i) = \sigma_{q_{i-1}}(i-1) \cdot \delta(q_{i-1}, q_i) \cdot \eta(q_i, x_i)$$

Da π der Pfad mit der höchsten Emissionswahrscheinlichkeit für $x_1 \dots x_i$ ist, der in q_i endet, kann kein Pfad mit anderem vorletzten Zustand eine höhere Emissionswahrscheinlichkeit aufweisen.

$$\Rightarrow \sigma_{q_i}(i) = \eta(q_i, x_i) \cdot \max_{p \in Q} \left(\sigma_p(i-1) \cdot \delta(p, q_i) \right). \qquad \square$$

$\Rightarrow$ Viterbi-Algorithmus

Satz 9.7: Der Viterbi-Algorithmus löst das HMM-Dekodier-Problem für ein HMM mit k Zuständen und einem String der Länge n in Zeit $O(n \cdot k^2)$.

Beweis: Korrektheit: Lemma 3.4.

Laufzeit: Initialisierung: k

Schritt 2: $n \cdot k \cdot k = n \cdot k^2$

Schritt 3: $O(n)$ $\qquad \square$

In der Praxis: Verwende Logarithmen der Wahrscheinlichkeiten

$\Rightarrow$ - Zahlen weniger nahe bei 0 $\rightarrow$ weniger Rundungsfehler

- Additionen statt Multiplikationen $\rightarrow$ schneller

$$\Rightarrow \sigma_q(i) = \log \eta(q, x_i) + \max_{p \in Q} \left(\sigma_p(i-1) + \log \delta(p, q) \right).$$

Modellierung des CG-Insel-Problems als HMM:

$M_{CG} = (\Sigma_{DNA}, Q, \pi_0, \delta, \eta)$

- $\Sigma_{DNA} = \{A, C, G, T\}$

- $Q = \{\pi_0, A^+, C^+, G^+, T^+, A^-, C^-, G^-, T^-\}$

- Transitionswahrscheinlichkeiten aus Statistiken, also durch Auswerten von DNA-Sequenzen in denen man die CG-Inseln schon kennt.

π_0 klein $+ \curvearrowright -$ kleiner als $- \curvearrowright +$

$+ \curvearrowright +$ kleiner als $- \curvearrowright -$

$\delta(C^+, G^+) \gg \delta(C^-, G^-)$

- $\eta(A^+, A) = \eta(A^-, A) = 1$
  $\eta(C^+, C) = \eta(C^-, C) = 1$
  $\eta(G^+, G) = \eta(G^-, G) = 1$
  $\eta(T^+, T) = \eta(T^-, T) = 1$

11. Phylogenetische Bäume

Ziel: Rekonstruktion von Verwandtschaftsbeziehungen von z.B. Lebewesen (Taxa)

→ Konstruktion eines phylogenetischen Baumes (Phylogenie)

Blätter ≙ Taxa

innere Knoten ≙ Vorfahren

Abstand im Baum ≙ Grad der Verwandtschaft

Meist: Binärbaum

Wurzel des Baumes ≙ gemeinsamer Vorfahr aller Taxa

14.7.2003

Verschiedene Modelle phylogenetischer Bäume:

- Verzweigungsgrad: meist Binärbäume
- gerichtete oder ungerichtete Bäumen
- Kantenlängen spezifiziert oder nur Topologie des Baums

Verschiedene zur Verfügung stehende Daten:

- Distanzmaß, das je zwei Taxa einen Abstand zuordnet.
  z.B. Alignment-Bewertung.
- Menge von Merkmalen
  z.B. phänotypische Merkmale
  oder: multiples Alignment von Gen-Sequenzen, jede Position definiert ein Merkmal.

11.1. Ultrametrische Distanzen

Def. 11.1: Sei A eine Menge von Taxa, sei $d: A \times A \to \mathbb{Q}^{\geq 0}$.
Dann ist d eine Metrik auf A, wenn gilt:

(i) $d(a,b) = 0 \iff a = b$ für alle $a, b \in A$

(ii) $d(a,b) = d(b,a)$ für alle $a, b \in A$ (Symmetrie)

(iii) $d(a,b) \leq d(a,c) + d(c,b)$ für alle $a, b, c \in A$ (Dreiecksungl.)

Def. 11.2: Sei A eine Menge von Taxa, sei $d: A \times A \to \mathbb{R}^{\geq 0}$ eine Metrik auf A. Dann ist d eine Ultrametrik auf A, wenn zusätzlich die folgende Drei-Punkt-Bedingung gilt:

Für alle $a, b, c \in A$ sind zwei der Distanzen $d(a,b)$, $d(a,c)$, $d(b,c)$ gleich und nicht kleiner als die dritte.

$d(a,b) \leq d(a,c) = d(b,c)$ oder

$d(a,c) \leq d(a,b) = d(b,c)$ oder

$d(b,c) \leq d(a,b) = d(a,c)$.

Ziel: Bestimme phylogenetischen Baum mit einer Wurzel, bei dem auch die Kantenlängen bekannt sind, so daß die Pfadlängen von der Wurzel zu einem beliebigen Blatt gleich sind.

→ ultrametrischer Baum

ideales Evolutionsmodell, in dem die Evolutionsgeschwindigkeit in jedem Ast des Baumes genau gleich ist.

Def. 11.3: Sei $A = \{a_1, \dots, a_n\}$ eine Menge von Taxa, ein gerichteter kantengewichteter Baum $T = (V, E, d)$ mit einer Wurzel r und Kantenbewertung $d: E \to \mathbb{Q}^{>0}$ ist ein ultrametrischer Baum für A, wenn gilt:

(i) T ist binärer Baum.

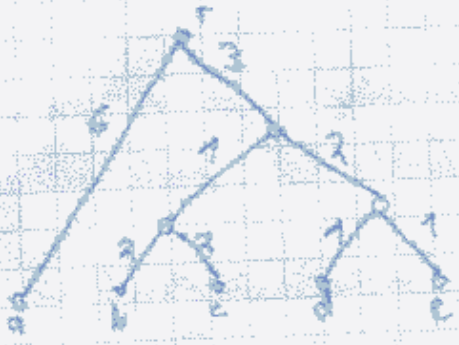
(ii) T hat genau n Blätter, die mit den Taxa beschriftet sind.

(iii) Die Summe der Kantenbeschriftungen auf jedem Pfad von der Wurzel zu einem beliebigen Blatt ist gleich.

Die Distanz zwischen $x, y \in V$ ist die Summe der Kantenbeschriftungen auf dem Pfad von x nach y in T.

Bezeichnung: $dist_T(x,y)$

Beispiel 11.1:



Lemma 11.1: Sei A eine Menge von Taxa, sei $T = (V, E, d)$ ein ultrametrischer Baum für A. Dann definiert die Distanzen $dist_T$ zwischen den Taxa eine Ultrametrik auf A.

Beweis: Seien a, b, c Blätter von T.
Zeige, daß $dist_T(a,b)$, $dist_T(a,c)$, $dist_T(b,c)$ die Drei-Punkt-Bedingung erfüllen.

1. Fall: $a = b$

$\Rightarrow 0 = dist_T(a,b) \leq dist_T(a,c) = dist_T(b,c)$ ✓

2. Fall: a, b, c paarweise verschieden.

$dist_T(r,a) = dist_T(r,b) = dist_T(r,c)$
(folgt aus (11.1))

$dist_T(v,a) = dist_T(v,b)$

$\Rightarrow dist_T(a,c) = dist_T(a,v) + dist_T(v,c) = dist_T(b,v) + dist_T(v,c) = dist_T(b,c)$

noch zu zeigen: $dist_T(a,b) \leq dist(a,c)$
aus (11.1) folgt: $dist_T(u,a) = dist_T(u,c)$

$\Rightarrow dist_T(a,c) = 2 \cdot dist_T(a,u)$
$= 2 \cdot (dist_T(a,v) + dist_T(v,u))$
$\geq 2 \cdot dist_T(a,v)$
$= dist_T(a,b)$ □

andere Richtung: Gegeben eine Menge A von Taxa mit ultrametrischer Distanzfunktion δ, finde ultrametrischen Baum:

Idee: Konstruiere Baum, sodaß Knoten $\hat{=}$ Teilmengen von A
- Starte mit allen einelementigen Teilmengen von A ($\rightarrow$ Blätter)
- Berechne paarweisen Abstand
- Solange möglich, wähle zwei Menge X, Y ohne Vorfahren mit minimalem Abstand, füge $X \cup Y$ als neuen Knoten hinzu, verbinde ihn mit X und Y, und berechne Abstand zu allen anderen Knoten.

$\rightarrow$ nach $n-1$ Schritten entsteht Baum mit Wurzel A.

$\rightarrow$ UPGMA-Algorithmus (Alg. 11.1)

Satz 11.1: Der Algorithmus 11.1 berechnet für eine gegebene Menge $A = \{a_1, \dots, a_n\}$ von Taxa und eine ultrametrische Distanzfunktion δ auf A einen ultrametrischen Baum für A in einer Zeit $O(n^3)$.

Beweis: Korrektheit: offenbar konstruiert der Alg. einen Baum, in dem der Abstand von der Wurzel zu einem beliebigen Blatt immer gleich ist.

Zeige, daß die in 2.(f) berechneten Kantengewichte nicht negativ werden. Es reicht aus, für neu konstruierten Knoten D und seine Kinder C_1 und C_2 zu zeigen, daß $height(D) \geq height(C_1)$ und $height(D) \geq height(C_2)$

Sei D_i der in der i-ten Iteration von Schritt 2 neu hinzugefügte Knoten, seien $C_{1,i}$, $C_{2,i}$ dessen Kinder, V_i die Menge der Knoten nach Iteration i, T_i die Menge T nach Iteration i.

Zeige: Für alle $1 \le i \le n-1$ und für alle $X \in V_{i-1}$:

$$height(D_i) \geq height(X)$$

Vollst. Ind. über i: $i=0$ ✓

Ind. Schritt: zeige $height(D_{i+1}) \geq height(D_i)$

$\Rightarrow$ zu zeigen: $dist(C_{1,i+1}, C_{2,i+1}) \geq dist(C_{1,i}, C_{2,i})$

1. Fall: D_i kein Kind von D_{i+1}:

$\Rightarrow C_{1,i+1}, C_{2,i+1}, C_{1,i}, C_{2,i} \in T_{i-1}$

Da in der i-ten Iteration in Schritt 2(a) $C_{1,i}$ und $C_{2,i}$ als diejenigen mit minimalem Abstand gewählt wurden, folgt (11.2).

2. Fall: D_i ist Kind von D_{i+1}: o.B.d.A. $D_i = C_{1,i+1}$

$\Rightarrow C_{2,i+1}, C_{1,i}, C_{2,i} \in T_{i-1}$

$\Rightarrow dist(C_{1,i}, C_{2,i}) \leq dist(C_{1,i}, C_{2,i+1})$ und
$dist(C_{1,i}, C_{2,i}) \leq dist(C_{2,i}, C_{2,i+1})$

$\overset{2(a)}{\Rightarrow} dist(C_{1,i+1}, C_{2,i+1}) = dist(D_i, C_{2,i+1})$

$$= \frac{dist(C_{1,i}, C_{2,i+1}) + dist(C_{2,i}, C_{2,i+1})}{2}$$

$$\geq \frac{dist(C_{1,i}, C_{2,i}) + dist(C_{1,i}, C_{2,i})}{2}$$

$$= dist(C_{1,i}, C_{2,i}).$$

□

Laufzeit: Schritt 1: $O(n^3)$

Schritt 2: $n-1$ Durchläufe

pro Durchlauf: 2(a): $O(n^2)$
(b): $O(1)$
(c): $O(n)$
(d): $O(1)$
(e): $O(1)$
(f): $O(1)$

→ gesamt: $O(n^2)$

$O(n^3)$

12. Additive Bäume

Problem: Reale Daten sind häufig nicht ultrametrisch

→ schwächere Voraussetzung für das Distanzmaß,
anderes Modell: Taxa an beliebigen Knoten des Baums.

Def. 11.4: Sei A eine Menge von n Taxa, sei $\delta: A \times A \to \mathbb{Q}^{\geq 0}$ eine Metrik auf A, $T = (V, E, d)$ ein kantenbewerteter Baum mit $A \subseteq V$.

Für alle $a, b \in A$ sei $dist(a,b)$ die Summe der Kantenbewertungen auf dem Pfad von a nach b in T.

T heißt **additiver Baum für A und δ**, wenn $dist(a,b) = \delta(a,b)$ gilt, für alle $a, b \in A$.

Problem: Gegeben A und δ, ex. ein additiver Baum für A und δ?
Falls ja, wie kann man ihn berechnen?

Def. 11.5: Sei A eine Menge von Taxa, δ Metrik auf A.

Ein additiver Baum $T=(V,E,d)$ für A und δ heißt *kompakter additiver Baum* für A und δ, falls $V=A$.

Def. 11.6: Das *Compact-Add-Tree-Problem* ist das folgende Berechnungsproblem:

Eingabe: Menge A von Taxa, Metrik $\delta: A\times A \to \mathbb{Q}^{\geq 0}$.

Ausgabe: kompakter additiver Baum für A und δ, falls ex., Fehlermeldung sonst.

16.7.2003

Def. 11.7: Sei A eine Menge von Taxa, δ metrisches Distanzmaß auf A. Der *Distanz-Graph für A und δ* ist der vollständige, kantengewichtete Graph $G(A,\delta)=(V,E,d)$ mit $V=A$ und $d(a,b)=\delta(a,b)$ für alle $a,b\in A$.

Satz 11.2: Sei A eine Menge von Taxa, δ Metrik auf A. Falls ein kompakter additiver Baum T für A und δ existiert, dann ist T der eindeutig bestimmte minimale Spannbaum von $G(A,\delta)$.

Beweis: Sei T ein kompakter additiver Baum für A und δ.

Zeige: keine Kante, die nicht in T enthalten ist, kann in einem minimalen Spannbaum enthalten sein.

Sei $e=\{x,y\}$ eine nicht in T enthaltene Kante.

- Pfad von x nach y in T hat Gesamtgewicht von $\delta(x,y)$
- Alle Kantengewichte in T sind >0.

$\Rightarrow$ Jede Kante auf dem Pfad von x nach y in T hat Gewicht $< d(x,y)$.

Annahme: e ist in minimalen Spannbaum $T' = (A, E')$ enthalten.

Sei $G' = (A, E' - \{e\})$, seien S und S' die Zusammenhangskomponenten von G', o.B.d.A. $x \in S$, $y \in S'$.

Sei e' die Kante auf dem Pfad P von x nach y in T, die von S nach S' verläuft.

$\Rightarrow e' \neq e$, $e' \notin T'$

Setze $T'' = \big(A, (E' - \{e\}) \cup \{e'\}\big)$. Dann ist T'' Spannbaum von $G(A,\delta)$

$\Rightarrow T'$ hat echt geringere Kosten als T'. $\lightning$ □

Lemma 11.2: Sei $G = (V, E, d)$ ein vollständiger kantenbewerteter Graph, $U \subsetneq V$, $\{u,v\}$ Kante minimaler Kosten mit $u \in U$, $v \in V \setminus U$.

Dann gibt es einen min. Spannbaum von G, der $\{u,v\}$ enthält.

Beweis: Annahme: kein min. Spannbaum enthält $\{u,v\}$

Sei T ein min. Spannbaum. Dann enthält $T \cup \{u,v\}$ einen Kreis.

$\Rightarrow$ Da u und v auch in T verbunden sind, ex. andere Kante $\{u',v'\}$ mit $u' \in U$ und $v' \in V \setminus U$.

$(T \cup \{u,v\}) \setminus \{u',v'\}$ ist auch ein Spannbaum von G, der nicht teurer ist als T. $\lightning$ □

Satz 11.3: Der Algorithmus 11.2 löst das Compact-Add-Tree-Problem für eine Menge A von n Taxa und eine Metrik d auf A in Zeit in $O(n^3 \cdot \log n)$.

Beweis: Korrektheit: In jedem Durchlauf durch die While-Schleife gilt die Aussage von Lemma 11.2., also berechnet der Alg. einen minimalen Spannbaum, falls dieser eindeutig ist.

Laufzeit: Bestimmung des Distanzgraphen: $O(n^3)$

Sortieren der Kantengewichte: $O(n^2 \cdot \log n)$

While-Schleife: $O(n^3)$ □

11.4. Das Parsimony-Prinzip und die Quartett-Methode

Ziel: Phylogenie bestimmen mit Hilfe der DNA-Sequenzen monologer Gene als Merkmale.

Taxa: Menge gleichlanger Strings

$\triangleq$ DNA-Sequenzen einer Menge homologer Gene.

Merkmale: Spalten eines mult. Alignments dieser Strings

Ziel: Bestimme Topologie eines binären phyl. Baums ohne Wurzel, dessen Blätter den Taxa entsprechen.

Zwei Schritte: 1. Bestimme Kostenmaß für gegebene Topologie für gegebene Menge von Taxa und Merkmalen.

2. Bestimme Topologie mit geringsten Kosten.

Def. 11.12: Sei $S = \{s_1, \dots, s_n\}$ eine Menge von Taxa. Ein *ungewurzelter phylogenetischer Baum* für S ist ein ungewurzelter Binärbaum ohne Wurzel, der genau n Blätter hat, die (bijektiv) mit den Taxa aus S beschriftet sind.

Def. 11.13: Das *Parsimony-Problem*:

Eingabe: Menge $S = \{s_1, \dots, s_n\}$ von Strings der Länge k über Σ und ein ungewurzelter phylogenetischer Baum $T = (V, E)$ für S.

Zulässige Lösungen: für eine Eingabe-Instanz ist jede Funktion $\beta: V \to \Sigma^k$ eine zulässige Lösung, wobei die Blätter auf die in der Eingabe gegebenen Strings abgebildet werden.

Kosten: für zulässige Lösungen β:

$$cost(\beta) = \sum_{\{x,y\} \in E} dist_H(\beta(x), \beta(y))$$

($dist_H$ = Hamming-Abstand)

Optimierung: Minimierung

Idee: zur Lösung des Parsimony-Problems:

- Füge an beliebiger Stelle eine Wurzel in den Baum ein.
- Durchlaufe den Baum von den Blättern her, speichere für jeden Knoten Menge möglicher Beschriftungen.
- Durchlaufe den Baum von der Wurzel aus und wähle für jeden Knoten eine der möglichen Beschriftungen aus

- Entfernt die Wurzel

⟹ Fitch-Algorithmus

Satz 11.7: Der Alg. MS löst das Parsimony-Problem in Zeit in $O(n \cdot k)$.

Beweis: Korrektheit: klar

Laufzeit: Sowohl bei dem Bottom-Up Durchlauf, wie auch bei dem Top-Down Durchlauf wird jeder Knoten für jedes Merkmal genau einmal betrachtet $\rightarrow O(n \cdot k)$. □

Ziel: Finde ungerichteten phylogenetischen Baum, der die Parsimonie-Bewertung minimiert.

Def. 11.11: Min-Par-Top-Problem

Eingabe: Menge S von n Strings der Länge k

Zulässige Lösungen: Jeder ungerichtete phylogenetische Baum für S.

Kosten: optimale Parsimony-Bewertung

Optimierungsziel: Minimierung

Naiver Ansatz: Alle möglichen Topologien durchprobieren:

$\rightarrow$ exponentielle Laufzeit!

Satz 11.8: Für alle $n \geq 3$ beträgt die Anzahl der nicht isomorphen ungerichteten phylogenetischen Bäume für n Taxa

$$\prod_{i=3}^{n} (2i-5) = \frac{(2n-4)!}{2^{n-2} \cdot (n-2)!}$$

Beweisidee: Binärbaum ohne Wurzel mit n Blättern hat genau $2n-3$ Kanten.

$n=3$: genau ein Binärbaum:

Binärbaum mit n Blättern läßt sich aus Binärbaum mit $n-1$ Blättern konstruieren: $2(n-1)-3 = 2n-5$ Möglichkeiten. □

Kein effizienter Algorithmus für Min-Par-Inf.-Problem bekannt

→ Heuristiken

Idee: Phylogenie für Menge S von n Taxa zusammensetzen aus phylogenetischen Bäumen für Teilmengen von S.

→ Teilmengen der Größe 4

→ Quartett-Methode

Def. 11.15: Ein Quartett ist ein ungerichteter phylogenetischer Baum für 4 Taxa.

Für $S=\{a,b,c,d\}$ gibt es genau 3 verschiedene Quartette.

optimales Quartett: Quartett mit opt. Parsimony-Bewertung.

Def. 11.16: S Menge von n Taxa, T ungerichteter phylogenetischer Baum für S, sei $S'=\{a,b,c,d\}\subseteq S$, $Q=(a,b;c,d)$ Quartett für S'.

Sei P_1 der Pfad von a nach b in T, P_2 der Pfad von c nach d in T.

Q heißt konsistent zu T, falls P_1 und P_2 disjunkt sind.

Satz 11.3: Sei S eine Menge von Taxa, T unger. phyl. Baum für S.

Sei Q_T die Menge aller zu T konsistenten Quartette.

Dann läßt sich T aus Q_T in polynomieller Zeit eindeutig konstruieren.

Ansatz: Bestimme für jede 4-elementige Teilmenge der Taxa das Quartett mit optimaler Parsimony-Bewertung.

Dann bestimme den ungerichteten phylogenetischen Baum, der zu den meisten Quartetten konsistent ist.

Def. 11.17: Max-Quartett-Consist-Problem:

Eingabe: Menge S von n Taxa

zulässige Lösungen: Jeder ungerichtete phyl. Baum T für S.

Kosten: Für einen solchen Baum T entsprechen die Kosten der Anzahl aller optimalen Quartette für Teilmengen von S, die zu T konsistent sind.

Optimierungsziel: Maximierung

Satz 11.10: Das Max-Quartett-Consist-Problem ist NP-schwer. □

Aber: beliebig gut approximierbar.
→ aufwändiger Algorithmus

Hier: Heuristik: Quartett-Puzzling

Algorithmus 11.6: Quartett-Puzzling

Eingabe: Menge S von n Taxa

1. Berechne für jede 4-elem. Teilmenge $S' \subseteq S$ das optimale Quartett $Q(S')$.
2. Wähle zufällige Reihenfolge $a_1, \dots, a_n$ der Elemente in S.
3. $T := Q(\{a_1, \dots a_4\})$

4. for $i := 5$ to n do
   - Initialisiere die Kosten aller Kanten in T mit 0.
   - Für alle $S' = \{b_1, b_2, b_3\} \in \{a_1, \ldots, a_{i-1}\}$, so daß $Q(\{b_1, b_2, b_3, a_i\})$ die Form $(b_1, b_2; b_3, a_i)$ hat, erhöhe die Kantenkosten auf dem Pfad von b_1 nach b_2 in T jeweils um 1.
   - Wähle eine Kante $\{x, y\}$ in T mit minimalen Kosten, lösche diese und füge einen neuen Knoten ein, der mit x, y und a_i verbunden ist.

Ausgabe: Der ungerichtete phyl. Baum T für S.

Beispiel 11.8: $S = \{a, b, c, d, e\}$, $(a, b; c, d)$, $(a, b; c, e)$
$(a, d; b, e)$, $(a, c; d, e)$, $(b, d; c, e)$

10. Vergleich von Genomen

bisher: Vergleich von DNA-Sequenzen, basierend auf lokale Mutationen (Löschen, hinzufügen oder ändern einer einzelnen Base).

jetzt: Mutationen auf höherer Ebene:
betrachten die DNA-Sequenz nur zwischen den einzelnen Genen
→ nur die Reihenfolge der Gene verändert
→ Genom. Rearrangements

Motivation: bei eng verwandten Arten sind die Gene fast identisch, aber Anordnung der Gene variiert.

Spezialfall: mitochondriale Genome
(fast) nur Reversals als Genome Rearrangements,
nur ein Chromosom.

⇒ Modelliere Abfolge der Gene durch Permutation

Ziel: Sortiere Permutation durch Reversals

Zwei Modelle: - mit bekannter Leserichtung
→ gerichtete Permutation

- mit unbekannter Leserichtung
→ ungerichtete Permutationen

10.2. Sortieren ungerichteter Permutationen

Def. 10.1: Sei $\pi = (\pi_1, \dots, \pi_n)$ Permutation der Ordnung n

Für $1 \le i < j \le n$ ist ein (i,j)-Reversal eine Permutation $\rho(i,j)$, so daß gilt:

$$\pi \cdot \rho(i,j) = (\pi_1, \dots, \pi_{i-1}, \pi_j, \pi_{j-1}, \dots, \pi_{i+1}, \pi_i, \pi_{j+1}, \dots, \pi_n)$$

(i,i)-Reversal: identische Permutation

(j,i)-Reversal = (i,j)-Reversal

Def. 10.2: Sortieren einer Permutation durch Reversals, MinSR-Problem

Eingabe: $n \in \mathbb{N}$, Permutation $\pi = (\pi_1, \dots, \pi_n)$ der Ordnung n

Zulässige Lösungen: Jede Folge $\rho_1, \dots, \rho_t$ von Reversals, so daß $\pi \rho_1 \dots \rho_t = (1, \dots, n)$

Kosten: Anzahl t der Reversals

Ziel: Minimierung

Satz 10.1: Das MinSR-Problem ist NP-schwer. □

→ Approx. alg. mit Güte 2

Def. 10.3: Sei $\pi = (\pi_1, \dots, \pi_n)$ eine Permutation, definiere $\pi_0 = 0$, $\pi_{n+1} = n+1$

$ext(\pi) = (\pi_0, \pi_1, \dots, \pi_n, \pi_{n+1})$ erweiterte Darstellung von π.

Def. 10.4: Sei $\pi = (\pi_1, \dots, \pi_n)$ Permutation. Ein Breakpoint von π ist ein Paar $(i, i+1) \in \{0, \dots, n\} \times \{1, \dots, n+1\}$ von Positionen, so daß

$$|\pi_i - \pi_{i+1}| \neq 1$$

$bp(\pi)$ Anzahl der Breakpoints von π

Beispiel: $0 \mid 4\ 3\ 2 \mid 7 \mid 1 \mid 5\ 6 \mid 8\ 9$

Lemma 10.1: Sei π eine Perm. der Ordnung n. Dann sind mindestens $\left\lceil \frac{bp(\pi)}{2} \right\rceil$ Reversals nötig, um π zu sortieren.

Beweis: Die identische Permutation hat keine Breakpoints.
Jedes Reversal eliminiert höchstens zwei Breakpoints.

→ zu: Finde Folge von Reversals, die π in die identische Permutation sortiert und im Durchschnitt 1 Breakpoint pro Reversal eliminiert.

Def. 10.5: Sei π Permutation der Ordnung n. Sei $k = bp(\pi)$, seien $(i_1, i_1+1), \dots, (i_k, i_k+1)$, $i_1 < \dots < i_k$ die Breakpoints von π.

Dann heißen die $k+1$ Folgen

$$S_0 = (\pi_0, \dots, \pi_{i_1}),\ S_1 = (\pi_{i_1+1}, \dots, \pi_{i_2}), \dots, S_k = (\pi_{i_{k+1}}, \dots, \pi_{n+1})$$

die Strips von π.

Der Strip S_j heißt aufsteigend, falls $\pi_{i_j+1} < \dots < \pi_{i_j+n}$ gilt,

absteigend, falls $\pi_{i_j+n} > \dots > \pi_{i_j+1}$.

Einelementige Strips sind absteigend, außer S_0 und S_k.

Lemma 10.2: Sei π eine Permutation der Ordnung n, sei $k \in \{0, \dots, n+1\}$

(a) Falls k in einem absteigenden Strip liegt und $k-1$ in einem aufsteigenden Strip liegt, dann ex. ein Reversal ρ, so daß $bp(\pi\rho) < bp(\pi)$.

(b) Falls l in einem absteigenden Strip liegt und $l+1$ in einem aufsteigenden Strip, dann ex. ein Reversal σ, so daß $bp(\pi\sigma) < bp(\pi)$.

Lemma 10.3: Sei π eine Permutation mit einem absteigenden Strip. Dann ex. ein Reversal ρ, so daß $bp(\pi\rho) < bp(\pi)$.

Beweis: Zähle k als das kleinste Element in einem abst. Strip.
Beh. folgt mit Lemma 10.2(a). □

Lemma 10.4: Sei π Perm. ohne absteigenden Strip. Dann ist π die identische Permutation, oder es ex. Reversal ρ, so daß $\pi\rho$ einen absteigenden Strip enthält und $bp(\pi\rho) \leq bp(\pi)$

Beweis: Sei π nicht die identische Perm. Dann hat π mind. 2 Breakpoints
Sowohl $\pi_0 = 0$ als auch $\pi_{n+1} = n+1$ liegen in aufst. Strips.
$\Rightarrow$ ex. mind. zwei aufst. Strips $S_1 = (\pi_0, \ldots, \pi_i) = (0, \ldots, i)$ und
$S_2 = (\pi_j, \ldots, \pi_{n+1}) = (j, \ldots, n+1)$
mit $j > i+1$.

23.7.2003

Lemma 10.5: Sei π eine Permutation mit absteigendem Strip. Sei k das kleinste Element in einem absteigendem Strip und l das größte.

ρ das Reversal, daß $k-1$ neben k platziert
σ das Reversal, daß $l+1$ neben l platziert

Wenn sowohl $\pi\rho$, als auch $\pi\sigma$ keine abst. Strips enthält,
dann gilt $\rho = \sigma$ und $bp(\pi\rho) = bp(\pi) - 2$

Beweis: Situation (b) und (c) liegt vor.

k muß in dem von σ umgedrehten Intervall liegen.
l muß in dem von ρ umgedrehten Intervall liegen.

Annahme: $\rho \neq \sigma$

$\Rightarrow$ es exist. ein Strip S, der nur von einem der beiden Reversals umgedreht wird.
falls S aufsteigend in π, dann absteigend in $\pi\rho$ oder $\pi\sigma$
falls S absteigend in π, dann aufsteigend in $\pi\rho$ oder $\pi\sigma$ ✓

$\Rightarrow \rho = \sigma$ eliminiert zwei Breakpoints. □

Satz 10.2: Der Alg. 10.1 ist ein 2-Approximations Algorithmus für das MinSR-Problem.

Beweis: Reversal am Anfang der Berechnung kann vermieden werden, mit letztem Reversal, das zwei Breakpoints eliminieren muß, da keine Permutation mit genau einem Breakpoint exist.

Rest folgt aus Lemma 10.1 bis 10.5.

Laufzeit: $O(n^2)$

Best bekannte Approximation: 1,375

bisher: Betrachtung der Moleküle als Strings
→ aber DNA/Proteine sind keine Strings, sondern räumliche Moleküle

- Funktion der Moleküle ergibt sich durch die räumliche Struktur.

⇒

12. Höherdimensionale Strukturen von Molekülen

- Labormethoden: Röntgen-Kristallographie
⇒ teuer, zeitaufwendig

- bei vielen Molekülen gibt es einen Zusammenhang zwischen String-Darstellung und der räumlichen Struktur

⇒ Ziel: Vorhersage der räumlichen Struktur durch die Betrachtung des eigentlichen Strings.

12.1. Sekundärstruktur von RNA

Strukturhierarchie: • A A G C U C — Einzelstrang mit Basen Adenin, Guanin, Cytosin, Uracil

↝ String über $\Sigma_{RNA} = \{A, C, G, U\}$

⇒ Primärstruktur der RNA.

•  Zusammenlagerung des RNA-Strangs durch Wasserstoffbrücken zwischen den Basen.

⇒ Sekundärstruktur der RNA

Def. 12.1: Sei $\tau = \tau_1 \dots \tau_n$ die Primärstruktur eines RNA.

die Sekundärstruktur von τ, $SecStruct_\tau$ ist eine Menge von Index-Paaren

$$SecStruct_\tau \subseteq \{(i,j) \mid 1 \le i < j \le n\}$$

mit der Bedeutung Base τ_i und τ_j sind miteinander verbunden.

üblich Anforderungen:

(i) jeder Index k kommt in höchstens einem Paar in $SecStruct_\tau$ vor.

nicht zugelassen

(ii) jedes Paar $\in SecStruct_\tau$ ist ein Watson-Crick-Paar $\{(A,U),(C,G)\}$ oder $\{(G,U)\}$ $\Rightarrow$ gültige Basenpaare

(iii)

Für jedes Paar in $SecStruct_\tau$ (i,j) gilt $j - i \ge 4$.

- Tertiärstruktur $\hat{=}$ Lage der Atome im 3-dim. Raum.

12.1.1. Minimierung der freien Energie

- freie Energie $\hat{=}$ Potential eines Molküls durch Bindungen

Energie freimachen

- stabile Moleküle besitzen geringe freie Energie

Ziel: Berechne Sekundärstruktur mit minimaler freier Energie.

1. Ansatz: Algorithmus von Nussinov

Annahme: freie Energie ~ Anzahl der Basenpaarungen in der Sekundärstruktur.

$\Rightarrow$ desto mehr Basenpaarungen, desto stabiler

Idee: Dynamische Programmierung
- Betrachte optimale Strukturen für Teilstrings
- Setze daraus sukzessive die optimale Struktur für größere Teilstrings zusammen.

Notation:
- $\tau = \tau_1 \dots \tau_n$ String (Primärstruktur der RNA)
- $S_{i,j} \triangleq$ opt. Sekundärstruktur für den Teilstring $\tau_i \dots \tau_j$
- $BP(S_{i,j}) = |S_{i,j}|$ (Anzahl der Basenpaare in $S_{i,j}$)
- $\delta(\tau_i, \tau_j) = \begin{cases} 1 & \text{falls } (\tau_i, \tau_j) \text{ gültiges Basenpaar} \\ 0 & \text{sonst} \end{cases}$

Beachte:
- Der Algorithmus 2.1 berechnet lediglich die Anzahl der Basenpaare in der optimalen Sekundärstruktur.
- Ermittlung der Sekundärstruktur ist durch ein Trace-Back-Verfahren möglich.

Verfeinerung:
- Gewichte die Basenpaarungen entsprechend ihrer Bindungsstärke.
- $f_e(\tau_i, \tau_j) \triangleq$ freie Energie der Basenpaarung (τ_i, τ_j) (negativ)
- $E(S_{i,j}) \triangleq$ minimale freie Energie von $S_{i,j}$

$\rightarrow$ Minimierung der freien Energie.

Ersetze in Alg. 2.1 die Rekurrenz durch

$$E(S_{i,j}) := \min \begin{cases} E(S_{i+1,j}) & \text{(i)} \\ E(S_{i,j-1}) & \text{(ii)} \\ E(S_{i+1,j-1}) + f_e(\tau_i, \tau_j) & \text{(iii)} \\ \min_k \{ E(S_{i,k}) + E(S_{k+1,j}) \} & \text{(iv)} \end{cases}$$

Skript (123)

„Beschränkung auf gültige Basenpaare entfällt"

Laufzeit: - Berechnung einer (n×n)-Matrix → $O(n^3)$

- Jeder Eintrag: (i), (ii), (iii) → $O(1)$

(v) → $O(n)$

⇒ $O(n^3)$

Erweiterung Turnbr: Algorithmus von Zuker

- Verfeinerung der Berechnung der freien Energie

→ Betrachtung der freien Energie von Substrukturen.

Def. 122:

Sei r Primärstruktur und $(i,j) \in SecStruct_r$

Wir sagen

- eine Base k ist <u>erreichbar</u> von (i,j), falls $i<k<j$ und kein Basenpaar $(i',j') \in SecStruct_r$ existiert mit

$i < i' < k < j' < j$

- ein Basenpaar (l,m) ist <u>erreichbar von (i,j)</u>, falls

$i<l<m<j$ sind

kein Basenpaar $(i',j') \in SecStruct_r$ existiert mit,

$i < i' < l < m < j' < j$

<u>Def. 123:</u> - r Primärstruktur, $SecStruct_r$ (Sekundärstruktur) <u>28.7.2003</u>

$(i,j) \in SecStruct_r$

durch (i,j) induzierte Teilstrukturen

- <u>Stacked Pair:</u> falls $(i+1, j-1) \in SecStruct_r$

⇒ unmittelbar aufeinanderfolgende Folge von Basenpaaren (Stem).

- Hairpin-Loop: falls von (i,j) kein Basenpaar in $SecStruct_T$ erreichbar ist.
- Bulge: falls ein von (i,j) erreichbares Basenpaar $(i',j') \in SecStruct_T$ existiert, so daß entweder $i'-i>1$ oder $j-j'>1$.
- Interior Loop: falls ein von (i,j) erreichbares Basenpaar $(i',j') \in SecStruct_T$ existiert, so daß sowohl $i'-i>1$ und $j-j'>1$
- Multiple Loop: falls von (i,j) mehr als ein Basenpaar erreichbar ist.

Bemerkung: - Größe einer Substruktur $\triangleq$ Anzahl der erreichbaren Basen
- Loops $\triangleq$ Hairpin, Loop, Bulge, Interior Loops, Multiple Loops.

Stem / Stacked Pairs $\Rightarrow$ stabilisierend (negative freie Energie)
Loops $\Rightarrow$ destabilisierend ("positive" freie Energie)

Annahme: kein Pseudoknot

Idee: dynamische Programmierung
2. Teil (iii) des Alg. von Nussinov
Betrachte die Energie der induzierten Substruktur und nicht nur die der Basenpaare.

Berechnung: $L_{i,j}$:= durch das Basenpaar (r_i, r_j) induzierte Substruktur mit minimaler freier Energie

Basisfall: (Alg. 122)

Bestimmung von $E(L_{i,j})$

Notation: $e_{\text{experimentell}}(\text{Substruktur})$ $\triangleq$ freie Energie der Substruktur bestimmt

$b(r_i, r_j) \triangleq$ freie Energie des Basenpaares (r_i, r_j)

Laufzeit: - Berechnung $(n \times n)$-Matrix $O(n^3)$

- jeder Eintrag (i), (ii) $O(1)$
  (iv) $O(n)$
  (iii) (a), (b) $O(1)$
  (c), (d) $O(n)$
  (e) $O(n^2)$

$\Rightarrow$ Gesamt: $O(n^4)$

$\rightarrow$ Literatur $O(n^3)$

12.3 Strukturvorhersage für Proteine

- wichtig, besonders auch für med. Forschung
- Strukturebenen der Proteine:
  - Primärstruktur: Darstellung als String (Abfolge der Aminosäuren entlang einer Polypeptidkette)
  - Sekundärstruktur: Wechselwirkungen zwischen den Atomen des Rückgrats einer Polypeptidkette
    (grob) 2 Typen:
    - $\rightarrow$ Helices
    - $\rightarrow$ Faltblätter
    - ($\rightarrow$ Schlingen $\triangleq$ alle Abschnitte in der Primärstruktur, die keine Helices oder Faltblätter sind).
  - Tertiärstruktur: (3-dimensionale Struktur)
    Zusammenfassung von Sekundärstrukturen zu
    - Motiven
    - Domänen (besitzen spezifische Funktion, aktives Zentrum)
  - Quartärstruktur: Aufbau eines Proteins aus Untereinheiten und evtl. aus zusätzlichen, molekularen Bausteinen (z.B. Hämoglobin)

12.3.1 Gitter-Modell (Minimierung der freien Energie)

- 3 dim. Modell: ~ Festlegung der Lage aller Atome im Raum,
  Spezifikation der Winkel, Bindungslängen

- Abstraktion der Strukturen hier:
  - Betrachte Aminosäuren (punktförmig) statt Atome
  - Winkel nur 90°, 180°, (270°)
  - Bindungslängen einheitlich

  ⇒ Einbettung der Aminosäuresequenz (Strings) in ein Gitter.

- Abstraktion der freien Energie:
  - grobe Einteilung der Aminosäuren in
    - hydrophob (wasserabweisend, H, 1)
    - hydrophil (wasserliebend, P, 0)
  - häufig kommt es zur Ausbildung eines hydrophoben Molekülkerns.

    ⇒ maximiere Anzahl der Nebenbeziehungen zwischen hydrophoben Aminosäuren.

Def. 12.20: Sei $s = s_1 \dots s_n$ über $\{0,1\}$.
Eine Abbildung $\gamma : \{1, \dots, n\} \to \mathbb{Z}^d$ heißt Einbettung von s in ein Gitter $\mathbb{Z}^d$ ($d \in \{2,3\}$), wenn
- alle in s benachbarten Symbole sind auch in $\mathbb{Z}^d$ benachbart.
- keine zwei Positionen in s werden demselben Gitterpunkt zugeordnet.

  ~> self-avoiding walk

- Falls γ einer Position im Gitter den Buchstaben 1 zuordnet, dann heißt diese Position mit 1 belegt (analog für 0)

Def. 12.21: Sei $s = s_1 \dots s_n$ über $\{0,1\}$ und ψ Einbettung von s in $\mathbb{Z}^d$.

Zwei Positionen i und j sind **verbundene Nachbarn**, wenn $(i-j) = 1$.

Zwei Positionen i und j sind **topologische Nachbarn**, wenn

$$\|\psi(i) - \psi(j)\| = 1$$

- Wir nennen eine Position i mit $s_i = 1$ eine **Eins-Position**
- Wir nennen ein Positionspaar (i,j) von zwei Eins-Positionen, wobei i,j topologische Nachbarn sind **1-1-Paar**, **Kontakt**.

Def. 12.22: Gegeben sei ein Gitter $\mathbb{Z}^d$.

Das d-Max-1-1-Problem ist definiert durch

Eingabe: $s = s_1 \dots s_n \in \{0,1\}^*$

zulässige Lösungen: jede Einbettung ψ von s in $\mathbb{Z}^d$

Kosten: Anzahl der 1-1-Paare

Ziel: Maximierung

Satz 12.6: Das 2-Max-1-1-Problem ist NP-schwer (analog 3-Max-1-1). □

$\Rightarrow$ Einschränken auf $\mathbb{Z}^3$.

$\rightarrow$ Approximationsalgorithmus.

Lemma 12.1: Sei $s = s_1 \dots s_n \in \{0,1\}^*$ und ψ eine Einbettung von s in das Gitter $\mathbb{Z}^3$.

(i) Jede Position (außer den Endpunkten) in s hat max. 2 topologische Nachbarn.

(ii) Für jedes 1-1-Paar (i,j) gilt, daß i gerade und j ungerade oder i ungerade und j gerade ist.

Beweis: (i) ✓

(ii) Färbungsargument:

- Färbe das Gitter mit zwei Farben (benachbarte Knoten besitzen verschiedene Farben).
  ↝ "Schachbrettmuster"

- Betrachte 1-1-Paar (i,j) und zugehörige Knoten $\gamma(i) = (x_1,y_1)$, $\gamma(j) = (x_2,y_2)$

- Da (x_1,y_1) und (x_2,y_2) topologische Nachbarn sind, besitzen sie unterschiedliche Farben.

- Auf dem Pfad von (x_1,y_1) und (x_2,y_2) wechseln die Farben der Knoten ständig ab.

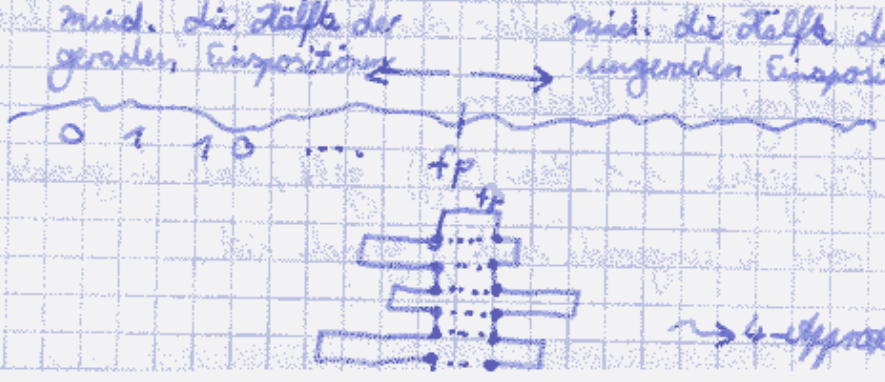
⇒ Pfad beginnt mit der einen Farbe und endet mit der anderen (insbesondere gilt das auch für den durch γ induzierten Pfad).

⇒ ungerade Anzahl von Knoten auf einem solchen Pfad

⇒ i genau dann gerade, wenn j ungerade ist. □

23.7.2003 Approximationsalgorithmus für 2-Nae-1-1-Problem

- Annahme: die Endpositionen des einzubettenden Strings sind $\neq 1$.



↝ 4-Approximation

Aufteilung der Einspositionen entsprechend ihrer Parität

X := die Menge der Einspositionen mit gerade/ungerade Parität.

Y := die Menge der Einspositionen mit ungerade/gerade Parität.

wobei: $|X| \leq |Y|$

$\Rightarrow XY$ - Partitionierung

Satz 12.7: Sei $s = s_1 \dots s_n \in \{0,1\}^*$.

Sei $Opt(s)$ eine optimale Lösung des 2-Area-1-1-Problems.

X, Y eine Partitionierung wie oben.

Dann gilt: $cost(Opt(s)) \leq 2 \cdot |X|$

Beweis: folgt direkt aus Lemma 12.1.

Def. 12.23: Sei $s = s_1 \dots s_n \in \{0,1\}^*$, XY-Partitionierung.

Dann heißt eine Position $fp \in \{1, \dots, n\}$ Faltungspunkt wenn gilt:

- $|\underbrace{\{i \in X \mid i \leq fp\}}_{X'}| \geq \frac{|X|}{2}$ und $|\underbrace{\{j \in Y \mid fp < j\}}_{Y'}| \geq \frac{|X|}{2}$

oder

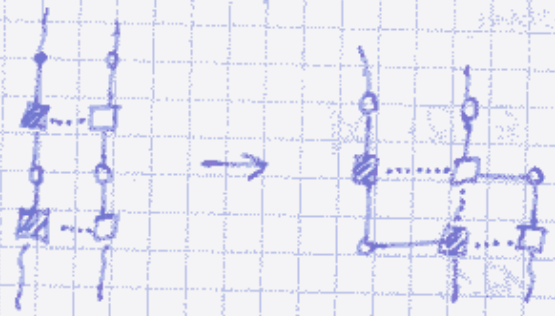
- $|\underbrace{\{j \in Y \mid j \leq fp\}}_{Y'}| \geq \frac{|X|}{2}$ und $|\underbrace{\{i \in X \mid fp < i\}}_{X'}| \geq \frac{|X|}{2}$

Satz 12.8: Der Algorithmus 12.7 ist ein 4-Approximationsalgorithmus für das 2-Haar-1-1-Problem für eine Eingabe S mit "Nicht-Eis-Endpositionen"

Beweis: - Algorithmus 12.7 bildet $|X|/2$ 1-1 Paare
- Opt hat $2|X|$ 1-1 Paare

$\Rightarrow \frac{2|X|}{|X|/2} = 4$ Approximationsgüte von 4. □

Idee für besseren Approximation:



$\Rightarrow$ 3-Approximation

(SODA 2002, A. Hauptmann)

12.3.2 Protein-Threading

Idee: Tertiärstruktur durch Alignment der Primärstruktur mit bereits bekannten Tertiärstrukturen zu bestimmen

Annahme: Viele verschiedene Primärstrukturen, aber nur "wenige" verschiedene Tertiärstrukturen.

→ Modell für Tertiärstruktur

Def. 12.24: Strukturmodell / Kernmodell

Sei s ein String. Ein Strukturmodell für s ist ein 5-Tupel

$M = (m, c, \lambda, l_{min}, l_{max})$, mit den folgenden Eigenschaften:

- Die Struktur von s beinhaltet m Kernregionen (Kern $\hat{=}$ Helix, Faltblatt, Motiv, Domäne) $C_1, \dots, C_m$
- Länge der Kernregion C_i ist durch c_i gegeben
  $c = (c_1, \dots, c_m)$
- Die Kernregionen c_i und c_{i+1} sind durch eine Schlife λ_i miteinander verbunden.
  $\lambda = (\lambda_0, \dots, \lambda_m)$
- min. Länge der Schleife λ_i: l_i^{min} : $l_{min} = (l_0^{min}, \dots, l_m^{min})$
- max. Länge der Schlife λ_i : l_i^{max} : $l_{max} = (l_0^{max}, \dots, l_m^{max})$

Insbesondere gilt:

- $|S| = \lambda_0 + \sum_{i=1}^{m} (c_i + \lambda_i)$
- $l_i^{min} \leq \lambda_i \leq l_i^{max}$

"Einpassen eines Strings (Primärstruktur) in ein Kernmodell"
$\hat{=}$ Threading

Def. 12.25: s String, $M = (m, c, \mathcal{R}, l_{min}, l_{max})$ Threadmodell von s.

s' String (mit unbekannter Struktur)

Ein Threading T von s' in M ein m-Tupel

$$T = (t_1, \dots, t_m)$$

mit

Anordnungs-Anforderungen:

$$\begin{cases} 1 + l_0^{min} \le t_1 \le 1 + l_0^{max} \\ t_i + c_i + l_i^{min} \le t_{i+1} \le t_i + c_i + l_i^{max} & \forall i,\ 1 \le i \le m \\ t_m + c_m + l_m^{min} \le |s'| + 1 \le t_m + c_m + l_m^{max} \end{cases}$$

$$1 + \sum_{j<i} (c_j + l_j^{min}) \le t_i \le |s'| + 1 - \sum_{j \ge i} (c_j + l_j^{min})$$

(Abstandsanforderung)

Bewertung eines Threadings:

- Bewertung wie bei Zuordnung eines bestimmten Kerns
  → polynomiell
- Verschiebungen werden im Kernprozess
  → „schwieriger"
- Zuordnung zum x_{cm}: $g_1(i, t_i)$
- Verschiebungen erwünscht: Kerne: $g_r(i_1, \dots, i_r, t_{i_1}, \dots, t_{i_r})$

hier: $r = 2$

Def. 12.26: Protein-Threading-Problem

Eingabe: Strukturmodell $H = (m, c, \Sigma, l_{min}, l_{max})$ eines String s.
ein String s'
zwei Funktionen g_1, g_2

zulässige Lösungen: Alle möglichen Threadings $T = (t_1, \ldots, t_m)$

Kosten: $cost(T) = \sum_{i=1}^{m} g_1(i, t_i) + \sum_{i=1}^{m} \sum_{j=i+1}^{m} g_2(i, j, t_i, t_j)$

Optimierungsziel: Minimierung

→ Protein-Threading-Problem ist NP-schwer
(Reduktion von MaxCut)

→ Branch & Bound-Ansatz