2 Exakter Mustervergleich *Exact String Matching*

Wir wissen, daß wichtige Biomoleküle Sequenzen über einem (kleinen) Alphabet sind, und ihre Funktion direkt oder indirekt durch diese Sequenz bestimmt ist (first fact der molekularen Biologie). Daher ist es klar/naheliegend, daß die Untersuchung auf gleiche (Teil)sequenzen oder Ähnlichkeit der Sequenzen für Fragestellungen der molekularen Bioinformatik von großer Wichtigkeit sind.

2.1 Fragestellung

gegeben sei ein *Text* T und ein *Muster* P, finde alle Vorkommen von P in T.

- Interessant ist das Durchsuchen von sehr großen Texten, daher sind effizient Methoden wichtig
- exakte Mustervergleiche in der molekularen Bioinformatik nicht der Regelfall, da oft Fehler/Abweichungen zu berücksichtigen sind

  - aber es gibt doch einige Problemstellung
  - darüberhinaus auch zum Kennen- und Verstehenlernen anderer Verfahren nützlich
  - und als Teilschritte für approximative Verfahren

2.2 Anwendungen

2.3 Definitionen

Definition 1 *Alphabet $\mathcal{A}$ ist nicht leere Menge von Zeichen*

Definition 2 $\mathcal{A}^* = \bigcup_{n \geq 0} \mathcal{A}^n$ *ist die Menge der Zeichenketten (string) oder Wörter über* $\mathcal{A}$

Definition 3 $S(i)$ *bezeichnet das i-te Zeichen von S, Indizes beginnen bei 1*

Definition 4 $S[i..j]$ *ist das Teilwort (substring) von S, das an der Position i beginnt, und bei j endet (jeweils inklusiv)*

$S[i..j] = \lambda$, *falls* $i > j$

Definition 5 $S[1..i]$ *ist Präfix (prefix) von S*
$S[j..n] == S[j..]$ *ist Suffix von S, wobei n die Länge* $|S|$ *von S ist*

Definition 6 S^r *ist der zu* S *reverse String.*

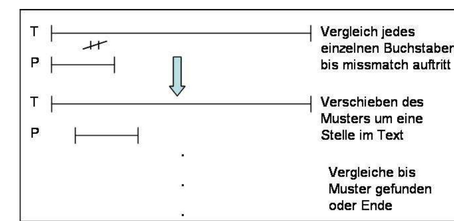
Definition 7 λ *ist der leere String* ($|\ \lambda\ | = 0$)

2.4 Der naive Algorithmus

m:= $|T|$ = Länge des Textes

n:= $|P|$ = Länge des Musters (Pattern)

$n \leq m$



Pseudocode:

```
for offset = 0 to m-n do
   found:=true;
   for j = 1 to n do
      if P(j) ≠ T(j+offset) then
         found:=false; break;
      fi
   endfor
   if found = true then 'gefunden';break; endif
endfor
```

worst-case Laufzeit: O($|T||P|$)

Es gibt (viel) bessere Verfahren, trotzdem:

- Verfahren ist sehr einfach (zu programmieren), Testen von komplizierte Algorithmen
- für große Alphabete und zufällige strings ist es im average-case nicht so schlecht (vgl)

Muster vorverarbeiten

- möglichst mehr als eine Position shiften
- ev. nach shift Vergleiche im neuen “alignment” sparen

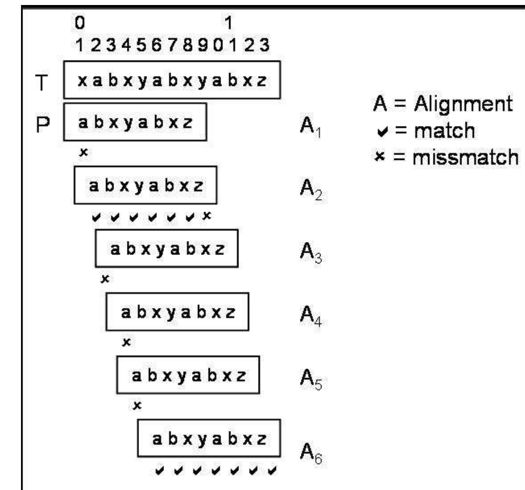
- Z-Algorithmus
- Boyer-Moore,

Text vorverarbeiten

- suffix-Bäume

2.5 Ein lineares Verfahren

Idee



2.5.1 Fundamental Preprocessing

für einen String S (das bei Anwendung für P eingesetzt wird) definieren wir:

Definition 8 *für einen String S, $2 \leq i \leq |S|$ ist*

$$Z_i(S) \;:=\; max\{p|S[i \dots i+p-1] = S[1 \dots p]\}$$

die Länge des längsten Präfixes von Suffix $S[i \dots]$, das auch Präfix von S ist (beachte: $S[i \dots i-1] = \lambda$)

Bemerkung 1 *falls klar notieren wir Z_i statt $Z_i(S)$*

Definition 9 *für ein $Z_i(S) > 0$ heißt das Intervall $[i, i + Z_i(S) - 1]$ die Z-Box an (der Position) i*

Definition 10 *für $2 \leq i \leq |S|$ definieren wir $[l_i, r_i]$ als die Grenzen derjenigen Z-Box, die vor (inklusive) i beginnt und von diesen die maximale rechte Grenze hat:*

$V_i := \{[a_j, b_j] \mid [a_j, b_j]$ *ist Z-Box an* $a_j \wedge a_j \leq i\}$

(die Menge der Grenzen aller Z-Boxen, die vor oder bei i beginnen)

$$[l_i, r_j] := \begin{cases} \underset{[a_j,b_j] \in V_i}{\operatorname{argmax}} \; b_j, & \textit{falls } V_j \neq \emptyset \\ [0,0] & \textit{sonst} \end{cases}$$

Bemerkung 2 *für $i < \tilde{i}$ gilt stets $V_i \subseteq V_{\tilde{i}}$ (wegen der Definition), und daher $r_i \leq r_{\tilde{i}}$*

Z-Algorithmus

```
// |S| ≥ 2
l:=r:=0;   // l,r enthalten jeweils l_{i-1}, r_{i-1}
Z[2] := prefix( S, S[2 .. ]).length();
if Z[2]>0 then
   l:=2; r:=2+Z[2]-1;
fi
```



```
for i=3 to |S| do
   if i>r then      //Fall I
      Z[i]:=prefix(S, S[i .. ]).length();
      if Z[i]>0 then   l:=i; r:=i+Z[i]-1;   fi
   else //Fall II
      k:=i-l+1;
      if Z[k]<r-i+1 then   //|β| = r − i + 1
         Z[i]:=Z[k]
      else
         aux:=prefix(S[r-i+2 .. ],S[r+1 .. ]).length();
         Z[i]:=r-i+1+aux;
         l:=i; r:=i+Z[i]-1;
      fi
   fi
endfor
```

prefix(A,B) liefert das längste gemeinsame Prefix von A und B

Satz: dieser Algorithmus ist eine korrekte Berechnung für Z_i

Begründung:

i=2 explizite Berechnung

i>2 I explizite Berechnung
II evtl. partiell in der Übung

Satz: der Algorithmus ist linear in der Länge des Strings $\Theta(|S|)$ (worst-case)

Beweis:

$$\begin{aligned} T_{preproc}(|S|) &= T_{\text{for-Schleife ohne Vergleiche in prefix}}(|S|) + T_{\text{Vergleich bei prefix}}(|S|) \\ &= \Theta(|S| * const) + T_{\text{Vergleich bei prefix}}(|S|) \\ &= \Theta(|S|) + O(|S|) \\ &= \Theta(|S|) \end{aligned}$$

Schleife ohne prefix: konstant je Durchlauf

Vergleiche:

a) die zum mismatch führen:
maximal einmal pro Aufruf von Prefix()
$\rightarrow$ insgesamt $O(|S|)$

b) die erfolgreich sind:

- wir führen nie für eine Position x, für die wir einen Vergleich $S(x) = S(y), (x > y)$ durchgeführt haben, der zum Erfolg führte, wieder einen Vergleich mit Zeichen $S(z), z < x$ durch
(oder anders ausgedrückt: jeder erfolgreiche Vergleich erhöht – indirekt – r)
- also auch begrenzt durch die Länge des Strings S

2.5.2 Exaktes Matching mittels des Z-Algorithmus

- betrachten P\$T mit $\$ \notin \mathcal{A}$ (Trennzeichen)
- berechne Z_i für diesen String (in $\Theta(|T|)$, da $|T| \geq |P|$)
(beachte: $\forall i : Z_i \leq n$)
- für $i > (n + 1)$ gilt:

$Z_i = n$
$\Leftrightarrow P\$T[i \ldots i + n - 1]$ ist Prefix von P\$T, $n = |P|$
$\Leftrightarrow T[i - (n + 1) \ldots i - 2] = P$
$\Leftrightarrow$ P kommt an Position i-(n+1) in T vor

Satz dies liefert exaktes Matching in $\Theta(|T|)$

2.6 Boyer Moore

2.6.1 right-to-left-scan

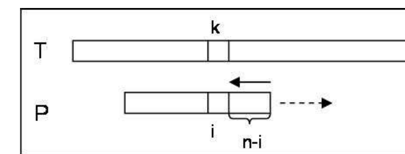
2.6.2 Bad Character Rule

Definition 11

für $a \in \mathcal{A}$ ist R(x) die am weitesten rechts auftretende Position von $x \in P$ oder 0, falls $x \notin P$

$R(x) := \max \quad \{i | 1 \leq i \leq n \wedge P(i) = x\} \cup \{0\}$

bad character shift rule für eine gegebene Verschiebung von P gegen T



- match für $P[i + 1..n]$
- mismatch für $P(i)$

verschiebe P um $\max\{1\ ,\ i - R(T(k))\}$

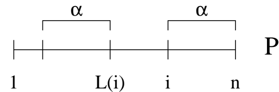
Erweiterung zur extended bad-character-shift-rule

2.6.3 (Strong) good-suffix-rule

2.6.4 Formalisierung und Vorverarbeitung

Definition 12 *(für good suffix rule, Fall Ia = mis match)*

sei P string für $1 \leq i \leq n = |P|$ definieren wir L(i) als die maximal (d.h. die am weitesten rechts stehende Position) $< n$, so dass



- $P[i \mathinner{..} n]$ *ein Suffix von* $P[1..L(i)]$ *ist,*
  d.h. $\alpha = P[i \mathinner{..} n] = P[L(i) - (n-i) \mathinner{..} L(i)]$
- *oder* $L(i) = 0$*, falls es keine solche Position gibt*

formal: $L(i) = \max \quad \{0\} \cup \{j | j < n \wedge (P[i \mathinner{..} n] = P[j - (n-i) \mathinner{..} j)\}$

Definition 13
von $L'(i)$ analog, allerdings wird zusätzlich gefordert, dass das Zeichen vor den zwei Kopien von α – falls beide im String – ungleich sind:

$$L'(i) := \max \quad \{0\} \cup \{j | j < n \quad \wedge (P[i \mathinner{..} n] = P[j - (n-i) \mathinner{..} j] \\ \wedge (P(i-1) \neq P(j - (n-i) - 1) \ \vee \ j - (n-i) - 1 = 0)\}$$

Definition 14 *(für good suffix rule, Fall II = match oder Fall Ib)*

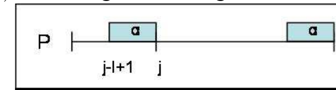
sei P string, $l'(i)$ ist die Länge des längsten Präfixes von P, das auch echtes Suffix von $P[i \mathinner{..} n]$ ist

falls kein solches Präfix existiert ist l'(i)=0

formal: $l'(i) := \max \quad \{0\} \cup \left\{ j | 1 \leq j \leq \underbrace{n - i + 1}_{|P[i \mathinner{..} n]|} \wedge P[1 \mathinner{..} j] = P[n - j + 1 \mathinner{..} n] \right\}$

Definition 15
sei P string, für $1 \leq j < n$ ist

$N_j(P)$ *die Länge des längsten Suffixes von P[1 .. j], das auch Suffix von P ist*



$$N_j(P) := \max\{0\} \cup \left\{ l | 1 \leq l \leq j \wedge (\underbrace{P[j - l + 1 \mathinner{..} j]}_{\alpha} = \underbrace{P[n - l + 1 \mathinner{..} n]}_{\alpha}) \right\}$$

Berechung von $L(i) und L'(i)$

for i=1 to n do L'(i) := 0 endfor

for j=1 to n-1 do
  if N_j(P)>0 then
    $i := n - N_j(P) + 1$
    $L'(i) := j$
  endif
endfor

L(2):=L'(2)
for i=3 to n do
  $L(i) := \max\{L(i-1), L'(i)\}$
endfor

strong good suffix shift rule

für eine gegebene Verschiebung von P gegen T trete

- ein Missmatch auf: $P(i-1) \neq T(k-1), i \leq n$
  → falls $L'(i) > 0$, dann verschiebe um $n - L'(i)$-Positionen
  → falls $L'(i) = 0$, dann verschiebe um $n - l'(i)$-Positionen
- ein Missmatch auf: $P(n) \neq T(k)$
  → dann verschiebe um eine Position
- ein Vorkommen von P in T auf
  → dann verschiebe um $n - l'(2)$-Positionen

2.6.5 Kompletter Boyer Moore

//Vorverarbeitung
berechne

- Z_i für 1 $\leq$ i $\leq$ n
- N_i, L'(i), l'(i) für 2 $\leq$ i $\leq$ n
- $R(x), x \in \mathcal{A}$


```
//Vergleich
h:=n    //rechter Rand von P in T
while h ≤ m do
  i:=n //Vergleichsposition in P
  k:=h //Vergleichsposition in T
  while i > 0 ∧ P(i) = T(k) do
    i–;k–;
  endwhile
  if i==0 then
    printf'Vorkommen' //Vorkommmen von P in T
    h+ = n − l'(2)
  else
    h+ = max{Verschiebug aus b.c.s.r., Verschiebung aus g.s.s.r.}
  endif
endwhile
```