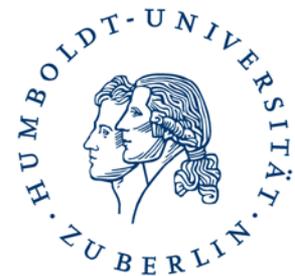


Algorithmische Bioinformatik

Boyer-Moore Algorithmus

Ulf Leser

Wissensmanagement in der
Bioinformatik



Ziele für heute

- Verständnis des Boyer-Moore Algorithmus als einer der schnellsten Stringmatching-Algorithmen in der Praxis
- Verständnis der Erweiterung zum linearen Worst-Case (und warum die praktisch irrelevant ist)
- Einige Varianten kennenlernen

Inhalt dieser Vorlesung

- Boyer-Moore Algorithmus
 - Bad Character und Good Suffix Rule
 - Preprocessing
 - Beispiel
- Erweiterung zum linearen Worst-Case
- Varianten

Boyer-Moore Algorithmus

- R.S. Boyer /J.S. Moore. „A Fast String Searching Algorithm“, Communications of the ACM, 1977
- Grundidee
 - Alignierung der Strings wie in naivem Algorithmus
 - Äußere Schleife
 - BM: P springt beim Schieben **weiter als 1**, wenn möglich
 - Vergleich von P mit Substring in T
 - Innere Schleife
 - Linearer Worst-Case: Nicht immer am **Anfang von P** starten
- Wir beginnen mit der $O(m \cdot n)$ Variante

Gerüst des Algorithmus

- Anordnung der Strings P und T
 - Erstes Zeichen von P „unter“ dem ersten von T
- Matche P und sein Gegenüber in T von **rechts nach links**
 - Also T[n] mit P[n], dann T[n-1] mit P[n-1], ...
 - Bei Mismatch oder Match für ganz P
 - Verschiebe P **um k Zeichen** nach rechts
 - Wieder von rechts nach links matchen
- Wie wird „k“ berechnet?
 - Bad Character Rule
 - Good Suffix Rule

Bad Character Rule 2

- Beobachtung

- Es sei gerade $P[n]$ mit $T[j]$ aligniert; $j \geq n$
- Sei der erste Mismatch an Position i von P
- Sei x das Zeichen an Position $j-n+i$ in T
- Sei l das am **weitesten rechts liegende Vorkommen** von x in P
- Welches sind Kandidaten für einen Match mit x in P ?
 - Fall 1: x kommt in P nicht vor: Schiebe P um i Zeichen nach rechts
 - Fall 2: $l < i$. Also kommt x in P nur vor i vor – **verschiebe P um $i-l$ Zeichen**

T xabxkabzzab**w**zzbzzb
P abz**w**ya**b**zz

↑ ↑
l i

T xabxkabzzab**w**zzbzzb
P abzwy**a**bzz

←

Wie weit können wir
jetzt schieben ?

Bad Character Rule 3

- Beobachtung

- Es sei gerade $P[n]$ mit $T[j]$ aligniert; $j \geq n$
- Sei der erste Mismatch an Position i von P
- Sei x das Zeichen an Position $j-n+i$ in T
- Sei l das am weitesten rechts liegende Vorkommen von x in P
- Welches sind Kandidaten für einen Match mit x in P ?
 - Fall 1: x kommt in P nicht vor: Schiebe P um i Zeichen nach rechts
 - Fall 2: $l < i$. x kommt in P nur vor i vor – verschiebe P um $i-l$ Zeichen
 - Fall 3: $l > i$. Das nützt uns (erst mal) nichts

T xabxkabzzabwz**zbzzb**
P abzw**abzz**

- „z“ gibt es auch rechts von i
- Was kann man noch machen?

Zusammengenommen

- Definition

Gegeben P . Dann sei $R(x)$ für alle $x \in \Sigma$ definiert als:

- $R(x) = 0$, wenn $x \notin P$
- *Sonst: $R(x) =$ „Position des **am weitesten rechts** liegenden Auftretens von x in P “*

- Platz: $O(|\Sigma|)$; Berechnung: $O(n)$ (wie?)

- Damit

- Sei i die Position des ersten Mismatch in P
- Sei x das Zeichen in T an der entsprechenden Position
- Verschiebe P um $\max(1, i - R(x))$

- Problem: Bei **kleinem Alphabet** (DNA) wird es meistens Auftreten von x rechts von i geben

Extended Bad Character Rule

- Verschiebe zum rechtesten x in P , **das links von i liegt**

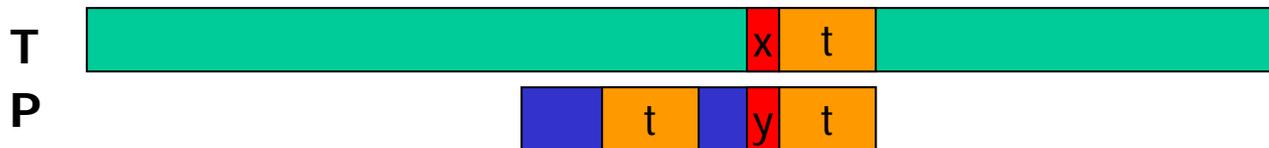
T xabxkabzzabwz**z**bzzb
P abzwy**a**bzz
 ←

T xabxkabzzabwz**z**bzzb...
P abzwy**z**abzz
 ←

- Benötigt relative Positionen
 - Für jede Position i und jedes Zeichen x : Merke Position des am weitesten rechts **aber links** von i liegenden Vorkommen von x
 - Array $[n, |\Sigma|]$ => **konstanter Lookup**, aber Platzverbrauch $O(n * |\Sigma|)$
 - Listen für jedes Zeichen mit Positionen; **linearer Platz** $O(n)$, aber was kostet der Lookup?

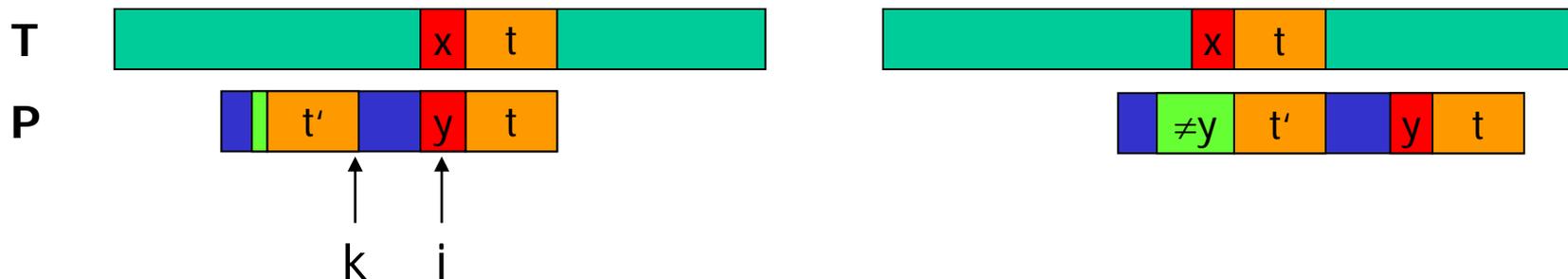
Good-Suffix Rule

- Bad Character Rule sucht nach dem ersten Mismatch
- Bis dahin haben wir aber schon ein (eventuell recht langen) **matchendes Suffix von P gefunden**
- Wenn wir über ganz P nach rechts schieben, muss dieses Suffix noch mal (in P) vorkommen, um einen Match mit dem Gegenstück in T zu haben
- Preprocessing: Vorkommen von Suffixen von P in P



Fall 1

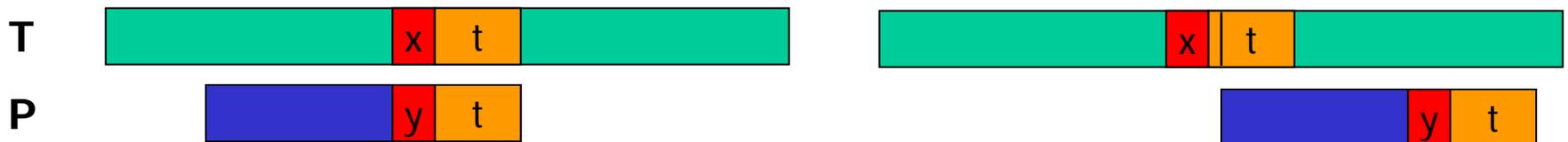
- Sei i die Position in P des ersten Mismatches (von rechts)
- Sei k das rechte Ende des am weitesten rechts liegenden Vorkommens von t in P mit $k < n$ und $P[k-|t|] \neq P[n-|t|]$ ($\neq y'$)
 - Existiert kein solches Vorkommen von t in P , dann sei $k=0$
- Dann
 - Wenn $k \neq 0$: Verschiebe P um $n-k$



- Warum fordert man nicht $P[k-|t|] = x'$?

Fall 2

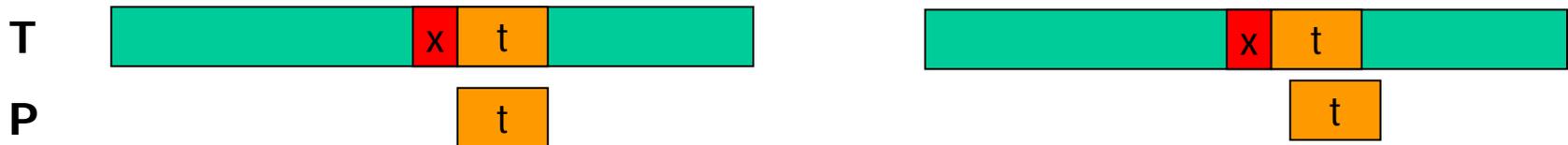
- Dann
 - Wenn $k \neq 0$: Verschiebe P um $n-k$
 - Wenn $k=0$ und $P \neq t$: Verschiebe P um $n-|t|+1$



- Man kann unter Umständen weiter verschieben
 - Was müssten wir dazu wissen?

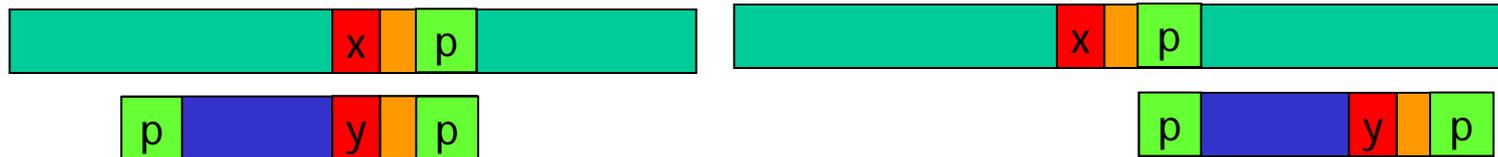
Fall 3

- Dann
 - Wenn $k \neq 0$: Verschiebe P um $n-k$
 - Wenn $k=0$ und $P \neq t$: Verschiebe P um $n-|t|+1$
 - Wenn $k=0$ und $P=t$: Verschiebe P um 1



Hinweis

- Was heißt „... unter Umständen weiter...“?
 - Betrachte das **längste Präfix von P, das auch Suffix von P ist** (p)
 - Muss kürzer als t sein
 - Ist im voraus berechenbar



- Details: Siehe Gusfield, $l'(i)$ -Werte

Average-Case

- Aber: Worst Case ist weiterhin $O(m \cdot n)$
- Aber: Der Average-Case ist linear
- Tatsächlich: $\sim m/c$ Zeichenvergleiche
 - c hängt ab von $|\Sigma|$, Häufigkeitsverteilung von Zeichen, etc.
 - P wird oft um viele Zeichen auf einmal verschoben
 - Submatches (t) sind meistens kurz – kurze innere Schleifen
 - „Normaler“ Text erlaubt große Sprünge
 - Großes Alphabet, z.B. Englisch, Proteinsequenzen, Chinesisch

Inhalt dieser Vorlesung

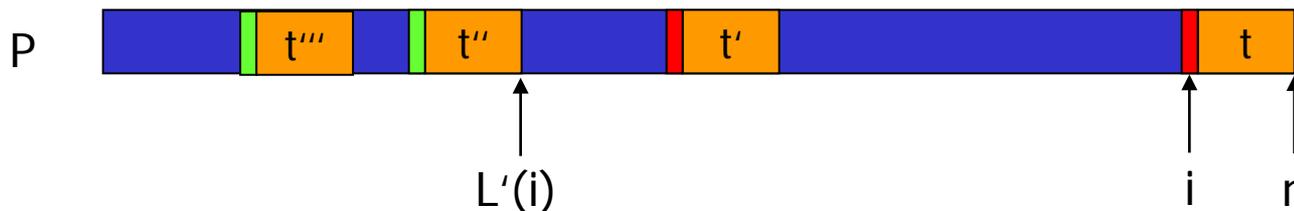
- Boyer-Moore Algorithmus
 - Bad Character und Good Suffix Rule
 - [Preprocessing](#)
 - Beispiel
- Erweiterung zum linearen Worst-Case
- Weitere Algorithmen

Preprocessing

- Woher wissen wir, **wo und ob t in P** noch vorkommt?
- Gesucht: Zu jedem Suffix t von P den am **weitesten rechts liegenden Endpunkt** ($L'(i)$) eines weiteren Vorkommens von t in P
- Definition

Sei $L'(i)$ der *größte Wert* für den gilt:

- *Bedingung 1:* $P[L'(i)-|t|+1 .. L'(i)] = P[i+1..n]$
- *Bedingung 2:* $L'(i) < n$
- *Bedingung 3:* $P[L'(i)-|t|] \neq P[i]$ (*Strong good suffix*)
- $L'(i) = 0$, falls kein solcher Teilstring existiert



Anders gesagt

- Gesucht: Zu jedem Suffix von P suchen wir den nächsten (von rechts nach links) identischen Substring von P, den man nicht weiter nach links verlängern kann
- Erinnerung Z-Boxen: „... zu jedem Präfix von P alle identischen Substrings von P (von links nach rechts), die man nicht weiter verlängern kann ...“
- Das GSR-Preprocessing ist also (fast) eine „invertierte“ Z-Box Berechnung
 - Unterschied: Es kann viele Präfix-Substring Paare geben, wir suchen aber nur eines davon

Zwischenschritt

- Definition

Sei $N(j)$ die Länge des längsten Suffix von $P[1..j]$, das auch Suffix von P ist

- Beispiel

<p style="text-align: center;">dcabcabdabdab $N(j) = 0002002005000$</p>
--

- j ist also **potentiell** das gesuchte rechte Ende eines t
- Berechnung: Z-Boxen auf **umgedrehtem P** ($=P^r$)

Ableitung von $L'(i)$ aus $N(j)$

- $N(j)$ geben die **Länge von längsten Suffixen** an, die links von j in P vorkommen
- $L'(i)$ sucht das am weitesten rechts liegende Auftreten von Suffixen der **Länge $n-i$**
 - i gibt die Länge des Suffixes vor, nach dem wir suchen
 - Suffix darf sich nicht verlängern lassen, sonst hätten wir bei Schieben von P wieder denselben Mismatch
- Damit ist **$L'(i)$ der größte Wert j für den gilt: $N(j)=n-i+1$**
 - Alle Positionen mit $N(j)=n-i+1$ stehen für ein (nicht verlängerbares) Suffix der gewünschten Länge
 - Davon interessiert uns das am weitesten rechts liegende
 - Und das entspricht dem größten j

Beispiel

1234567890123
dcabcabdabdab
$N(j) = 0002002005000$

- Suchen wir $L'(12)$
 - Also Suffixe „ab“
 - Zeichen vor diesem „ab“ darf nicht „d“ sein
 - Das sind alle Positionen j mit $N(j) = 2$
 - Denn dort liegt ein längstes Suffix der Länge 2
 - Davon das „rechtteste“ ist unser Treffer
 - Also: $L'(12) = 7$

Zusammen: Preprocessing

- Gegeben: P
 - Berechne N-Werte durch Z-Box auf P^r
 - Berechne L'-Werte durch (ohne string good suffix)

```
for i=1 to n
    L'(i) := 0;
for j=1 to n-1
    i := n-N(j);
    if (i ≤ n) then L'(i) := j;
end for;
```

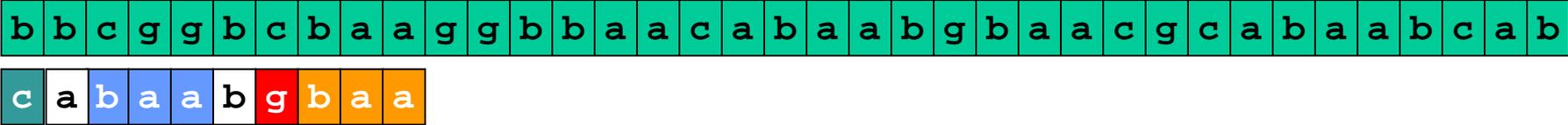
$$N(j) = n - i$$

- Komplexität: $O(n)$
 - Z-Box ist $O(n)$
 - L'-Werte ist $O(n)$

Boyer-Moore

```
compute L'(i);
compute R(x) for each x∈Σ;                                // Simple BCR
k := n;                                                    // Runs thru T
while (k≤m) do
    align P with T with right end k;
    match P and T from right to left until
        mismatch:      Compute shift s1 using BCR and R(x);
                       Compute shift s2 using GSR and L'(i);
                       k := k + max(s1, s2);
        P matched:    print k;
                       k := k + 1;                          // Could be impr.
end while;
```

Beispiel mit EBCR



EBCR wins



Mit BCR Schieben um 1 (geht immer)



GSR wins



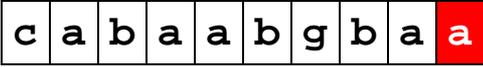
GSR wins



Mit /ohne l'



- Match
- Good suffix
- Mismatch
- Ext. Bad character



Inhalt dieser Vorlesung

- Boyer-Moore Algorithmus
 - Bad Character und Good Suffix Rule
 - Preprocessing
 - Beispiel
- Erweiterung zum linearen Worst-Case
- Varianten

BM mit linearem Worst-Case

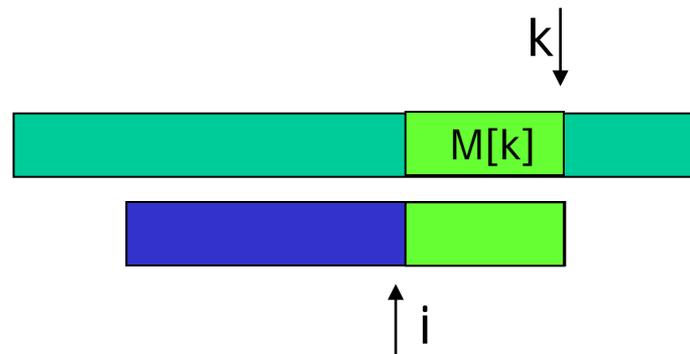
- Apostolico, Giancarlo (1986). "The Boyer-Moore-Galil string searching strategies revisited" *SIAM Journal on Computing*
- Stellt sicher, dass **jedes Zeichen in T maximal einen Match** erzeugt
- Das reicht zum linearen Worst Case (Beweis analog Z-Box)
 - Nach jedem Mismatch schieben wir um mindestens 1 – also haben wir insgesamt maximal m Mismatches
 - Außerdem maximal m Matches (siehe oben)
 - Also ist der Algorithmus $O(m)$

Vorarbeiten

- Erinnerung: $N(j)$ ist die Länge des längsten Suffix von $P[1..j]$, das auch Suffix von P ist
- Boyer-Moore Grundgerüst: Shift/compare Phasen
 - Eine Phase besteht aus Verschieben von P (shift) und Matchen von rechts nach links bis Mismatch oder vollständiger Match (compare)
 - Dann Shift berechnen und nächste Phase starten, bis Ende von T erreicht

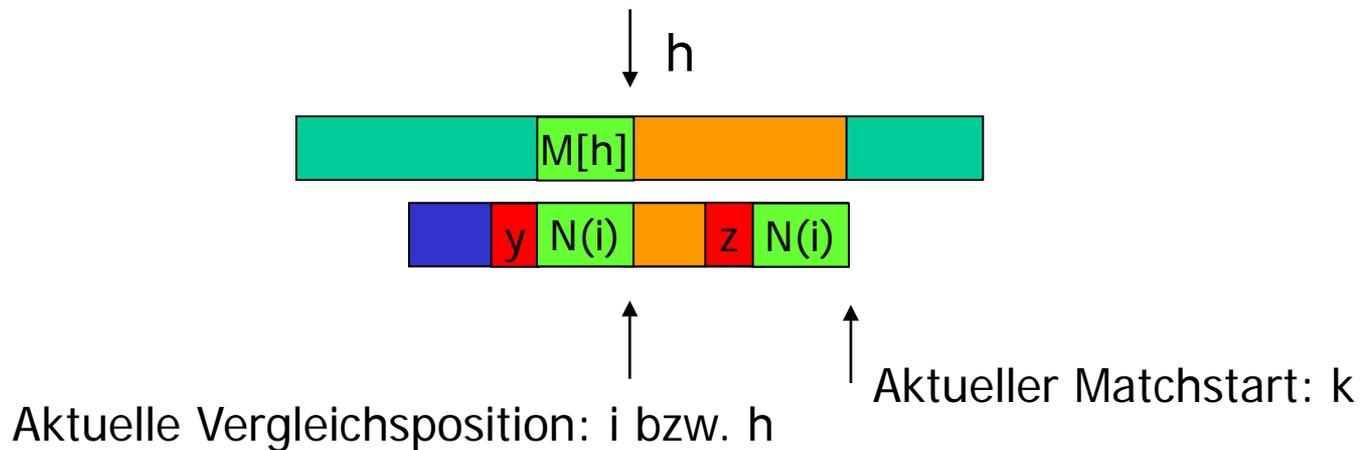
M: Suffixe matchen Suffixe

- Sei M ein Array $\text{int}[m]$, initialisiert mit -1
 - Wir beginnen eine Phase
 - Sei k das rechte Ende von P in T
 - Wir matchen nach links
 - Sei i die Position des Mismatches in P
 - Dann matched das Suffix von P der Länge $n-i$ mit einem Suffix des Substrings $T[..k]$
 - $P[i+1..]=T[k-n+i..k]$
 - Die Länge merken wir uns in $M[k]$



M: Suffixe matchen Suffixe

- M wird wie folgt benutzt
 - Später vergleichen wir $M[h]$ und $N(i)$
 - $N(i)$ sind Suffixe von P in P
 - $M[h]$ sind Suffixe von P in T
 - Da wir immer nach links vergleichen, aber nach rechts schieben, kennen wir in einer Phase an Position k schon alle $M[i]$ Werte mit $i < k$



Details – 1

- Wir beginnen eine Phase an Position k in T
 - Wir schieben wie beim „normalen“ Boyer-Moore
 - Wir **sparen im Compare-Schritt** (innere Schleife optimiert)
- Wir laufen von rechts nach links (h durch T und i durch P)
- Beachte: Nicht alle $M[k]$ werden berechnet, also kann es sein dass $M[k'] = -1$ mit $k' < k$
- Fall 1: Wenn $M[h] = -1$ oder $M[h] = N(i) = 0$, dann
 - $T[h] = P[i]$ und $i = 1$: Das ist ein **vollständiger Match**; beende Phase
 - $T[h] = P[i]$ und $i > 1$: Weiter nach links matchen ($h--$; $i--$)
 - $T[h] \neq P[i]$: Kein Match; setze $M[k] = k - h$; beende Phase

Details – 2

- Fall 2: Wenn $M[h] < N(i)$

- Sei $h' = k - M[h] + 1$, $i' = i - M[h] + 1$

- $A = T[h'..h] = P[i'..i]$

- Wegen $M[h]$ bzw. $N(i)$

- Auch: $A = P[n - M[h] + 1..]$

- Links von h' bzw. i' kommt

- $X = P[i' - 1] = P[n - M[h]]$ wegen $N(i) > M[h]$

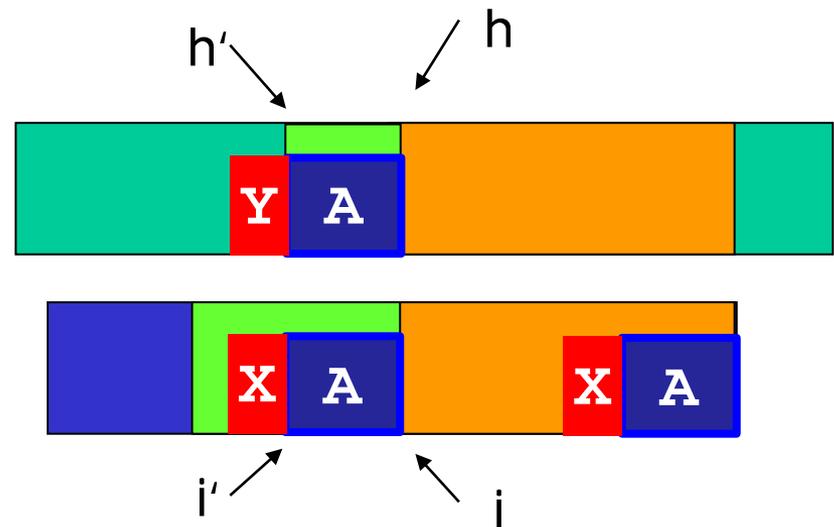
- $Y = T[h' - 1] \neq P[n - M[h]] = X$ weil $M[k]$ längstes Suffix mit Ende h ist

- Also sind die nächsten $M[h]$ Vergleiche Matches und danach kommt ein Mismatch

- Setze $M[k] := k - h + M[h]$

- Phase beenden (mit Mismatch)

- (PS: Gusfield matched ab h' / i' weiter, was nicht notwendig ist)



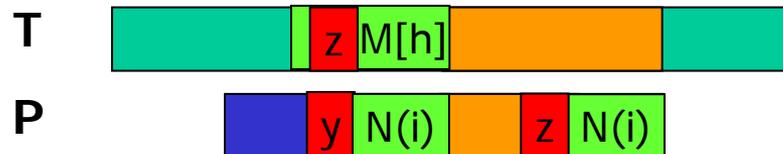
Details – 3

- Fall 3: Wenn $M[h] \geq N(i)$ und $N(i)=i>0$
 - In T liegt ein String, der bei h endet,
 - ... der länger ist als N(i)
 - ... und dessen Suffix mit N(i) matched
 - ... und N(i) ist so lang, dass der Teil in P vor i und der Teil nach i (haben wir schon gesehen) zusammen ein kompletter Match sind
 - Vorkommen von P melden
 - Setze $M[k] := n$
 - Gusfield setzt auf k-h (kürzer ist nie schlimmer) wg einfacherem Beweis
 - Phase beenden (mit Match)



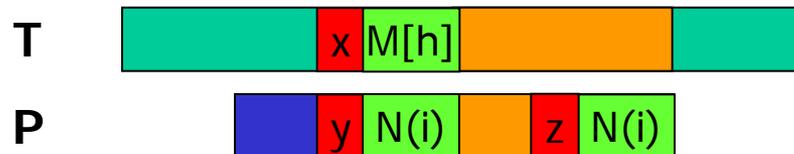
Details – 4

- Wenn $M[h] > N(i)$ und $N(i) < i$
 - $P[i-N(i)+1..]$ matched mit Substring in T , der an Position h endet
 - Danach kommt **garantiert ein Mismatch**
 - Denn wegen $M[h]$ kommt ein Zeichen, das mit der Verlängerung des Suffixes von P matched
 - Aber wegen $N(i)$ können wir den Substring in P , der an i endet, nicht weiter nach links mit dem Suffix von P matchen
 - Setze $M[k] := k-h+N(i)$
 - Phase beenden (mit Mismatch)



Details – 5

- Wenn $M[h]=N(i)$ und $N(i) < i$
 - Wir können wieder Matches überspringen
 - Denn $P[i-N(i)+1..]$ matched mit dem Substring in T , der an Position k endet
 - Was danach kommt, wissen wir nicht
 - Sprung: $h := h-M[h]; i := i-M[h];$
 - Dann weiter nach links matchen
 - Phase läuft weiter



Beweis

- Die Kurzform
 - Matches
 - Zeichen, die wir matchen, führen zur Erhöhung von $M[k]$
 - Wenn $M[h] > 0$, dann haben wir einen Substring in T schon gesehen und gematched
 - In all diesen Fällen überspringen wir die nächsten $M[h]$ Zeichen oder beenden die Phase
 - Mismatches
 - Nach jedem Mismatch beenden wir die Phase

Inhalt dieser Vorlesung

- Boyer-Moore Algorithmus
 - Bad Character und Good Suffix Rule
 - Preprocessing
 - Beispiel
- Erweiterung zum linearen Worst-Case
- Varianten

Two Faster Variants

- BM-Horspool
 - Drop the good suffix rule – makes algorithm slower in practice
 - Rarely shifts longer than EBCR, needs time to compute the shift
 - Instead of looking at the mismatch character x , always look at the symbol in T aligned to the last position of P
 - Generates longer shifts on average (i is maximal)
- BM-Sunday
 - Instead of looking at the mismatch character x , always look at the symbol in T after the symbol aligned to the last position of P
 - Generates even longer shifts on average
- Details: Navaro, G. and Raffinot, M. (2002). "Flexible Pattern Matching in Strings", Cambridge University Press.

Selbsttest

- Erklären Sie den Boyer-Moore Algorithmus
- Wann ist welche der beiden Regeln besser?
- Wie viel Speicherplatz braucht man für die EBCR?
- An welchen Stellen ändert die Sunday-Variante den Algorithmus?
- Wie kann man Häufigkeiten von Zeichen zu weiteren Optimierungen ausnutzen?
- Welche Häufigkeiten sind dabei entscheidend? Wo sollte man messen?