

Algorithmische Bioinformatik

Knuth-Morris-Pratt Algorithmus
„Natürliche“ Erweiterung des naiven Matching



Ulf Leser
Wissensmanagement in der
Bioinformatik



Inhalt dieser Vorlesung

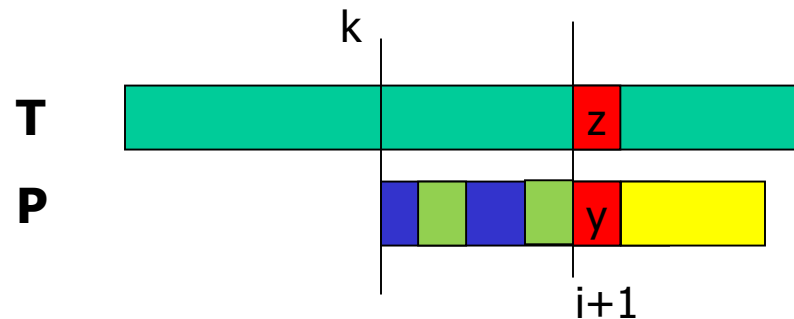
- Knuth-Morris-Pratt Algorithmus
 - Korrektheit und Komplexität
 - Patternmatching mit Automaten
- Vergleich mit anderen Algorithmen

Knuth Morris Prath

- Knuth, Donald E., James H. Morris, Jr, and Vaughan R. Pratt. "Fast pattern matching in strings." *SIAM journal on computing* 6.2 (1977): 323-350.
- Noch ein Klassiker
- Intuitive **Erweiterung des naiven Algorithmus**
 - Läuft auch von links nach rechts
 - Nutzt aber aus, wenn man beim Matchen lernt – durch Preprocessing
- Schöner Beweis der Korrektheit
- Erweiterbar zum Matching mit **mehrerer Pattern**
 - Nächste Stunde: Aho-Corasick

Grundidee

- Sei P mit T an Startposition k aligniert, links-nach-rechts
- Bei **Mismatch** an Position i+1 gilt: $T[k..k+i-1] = P[1..i]$



- Frage: Kommt ein (langes) Suffix von $P[1..i]$ vorher in P vor?
- Wenn ja: Schiebe möglichst weit (äussere Schleife)
- Und vergleiche dieses Suffix nicht nochmal (innere Schleife)

```
abcxabcgedsc
abcxabcde
  abcxabcde
```

Definitionen

- Definition

- Sei sp_i die *Länge des längsten echten Suffix von $P[1..i]$, das mit einem Präfix von P matched*
- Sei sp_i' die *Länge des längsten echten Suffix von $P[1..i]$, das mit einem Präfix von P matched und für das zusätzlich gilt: $P[i+1] \neq P[sp_i+1]$*

- Beispiel

P: abcaeabcabd
sp_i: 00010123420
 abca
 abcaeab
 abcaeabc
 abcaeabca

P: abcaeabcabd
sp_i: 00010123420
sp_i' : 00010**000**420
 abcaeabc
 abcaeabcab

KMP im Überblick

- Algorithmus
 - Preprocessing von P: Berechne sp_i
 - Matching von links nach rechts
 - Bei Match – weiter nach rechts matchen
 - Bei Mismatch (an Position $i+1$ in P) oder komplettem Match
 - **Schiebe P** um ... (Länge durch sp_i und i bestimmt - gleich)
 - **Weiter vergleichen ab ...** (gleich)
- Zwei Gewinne gegenüber naivem Matching
 - Schiebt P oft um **mehr als eine Position**
 - Vergleiche starten **meistens bei sp_i+1** (und nicht bei 1)

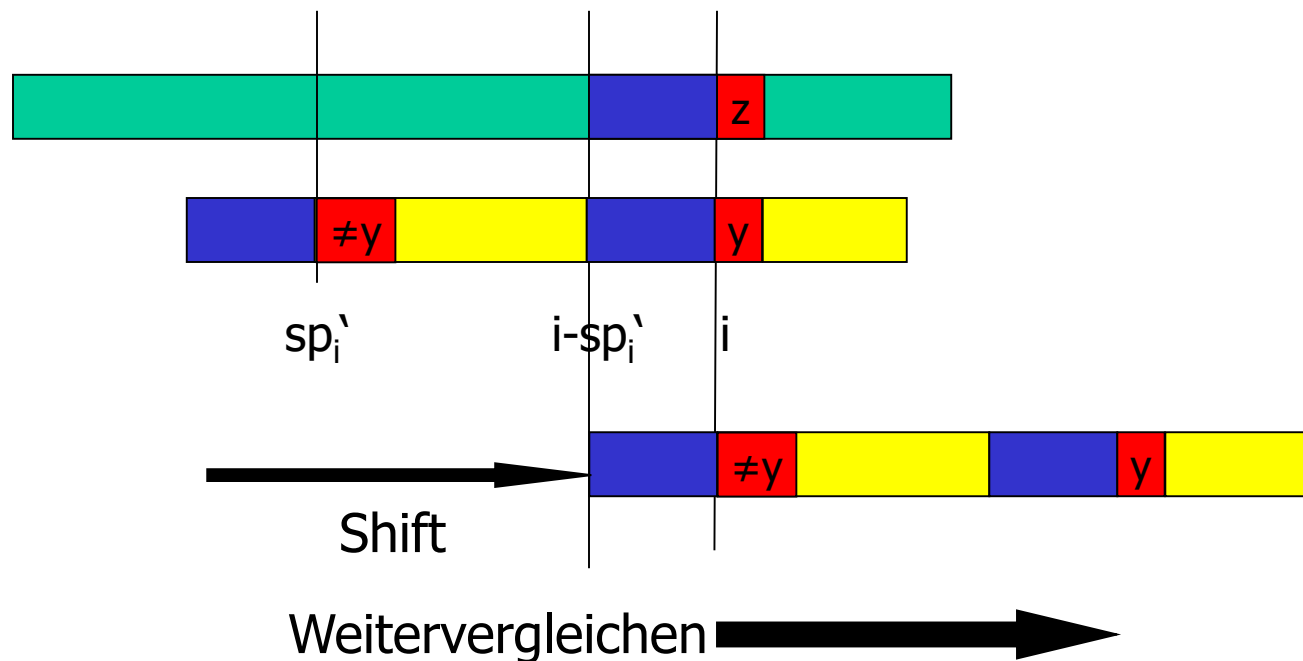
Shift Regel 1

- Wenn $P[1]$ und $T[k]$ **sofort mismatchen**
 - Schiebe P um ein Zeichen nach rechts
 - Vergleiche weiter ab $P[1]$



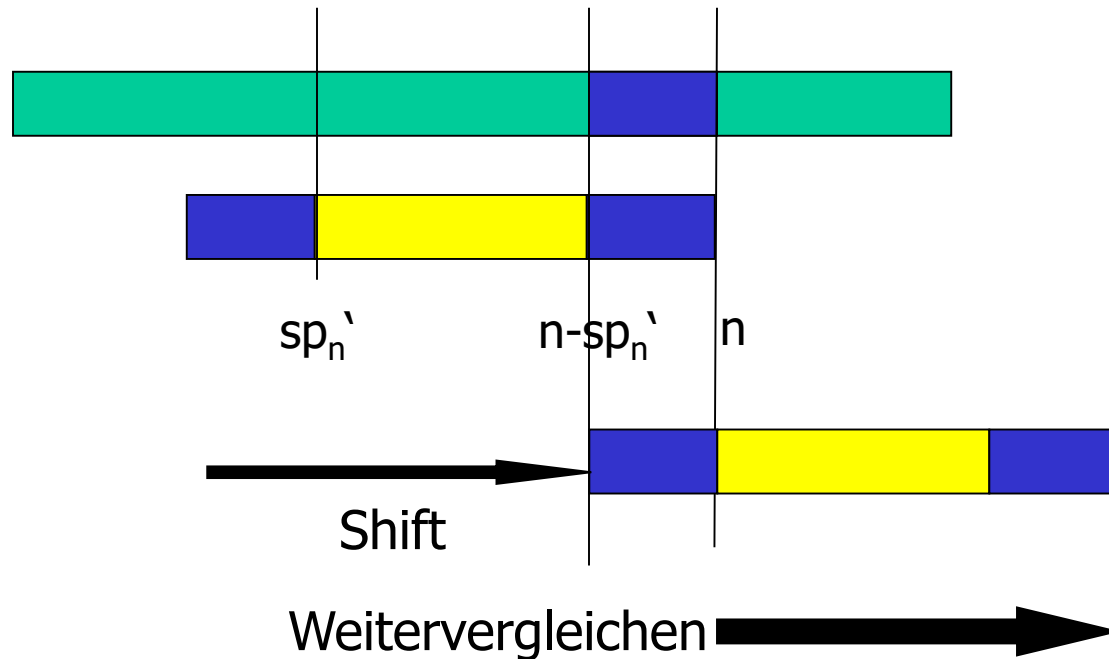
Shift Regel 2

- Wenn bei $i+1$ in P der **erste Mismatch** vorkommt
 - Schiebe P um $i - sp_i'$ Positionen nach rechts
 - Vergleiche **weiter ab $P[sp_i' + 1]$**



Shift Regel 3

- Wenn ein **kompletter Match** gefunden wird
 - Schiebe P um $n - sp_n'$ Positionen nach rechts
 - Vergleiche weiter ab $P[sp_n' + 1]$



Warum sp_i' (und nicht sp_i)?

- Sind beide korrekt?
 - Sei $sp_i \neq sp_i'$ für ein i und bei $i+1$ tritt Mismatch auf
 - Dann ist $P[i+1] \neq T[k+i+1]$
 - Wegen $sp_i \neq sp_i'$ gilt: $P[i+1] = P[sp_i+1]$
 - Der Vergleich $P[sp_i+1]$ mit $T[k+i+1]$ ist sicher ein Mismatch
- Beobachtung
 - Es gilt: $sp_i' \leq sp_i$
 - Präfixe, die sp_i genügen, sind mindestens solange wie Präfixe für sp_i'
 - Man schiebt um $i - sp_i'$
 - Also: sp_i' erlaubt **längere Verschiebungen** als sp_i

Inhalt dieser Vorlesung

- Knuth-Morris-Pratt
- Korrektheit und Komplexität
- Patternmatching mit Automaten
- Vergleich mit anderen Algorithmen

Korrektheit der Shift-Regel

- Was passiert bei Vorkommen des Präfix **zwischen dem Präfix und dem Suffix** vor dem Mismatch?
- Versuchen wir mal

...**BZZZBCBC**EBBZZEFFGAA
BZZZ**B**C**B**DD

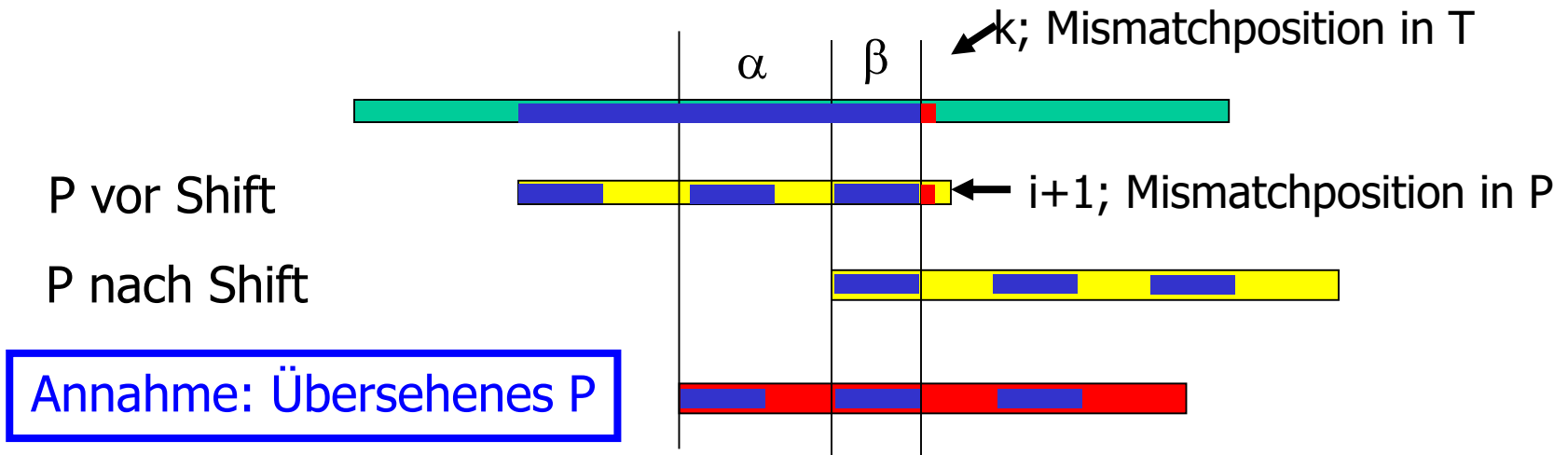
- Zwei Substrings (B) matchen mit dem Suffix vor dem Mismatch und werden mit unterschiedlichen Zeichen (Z bzw. C) fortgesetzt
- Mittleres B erzeugt (hier) keinen Match

...**BZ**Z**BC****B**CEBBZZEFFGAA
BZZZ**B**C**B**DD

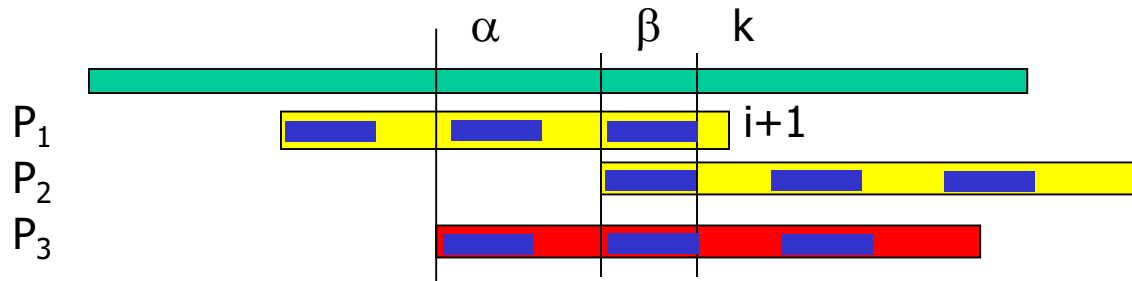
- ... aber kann das nicht auch mal klappen?

Beweis

- Theorem
 - Die Shift-Regel verschiebt nie soweit, dass ein Vorkommen von P in T übersehen wird
- Beweis
 - Wenn das anders wäre, hätten wir ...

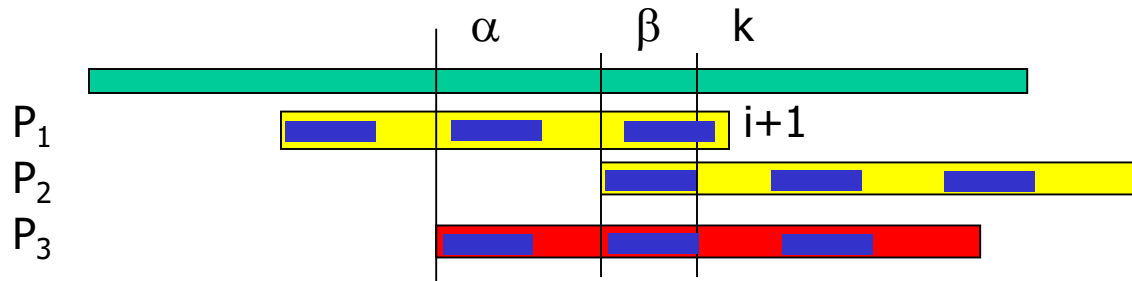


Beweis 2



- Wir zeigen **Widerspruch zur Definition** von sp_i
 - β ist Präfix von P ; $|\beta| = sp_i$
 - P_1 und P_3 matchen mit T bis $k-1$; also ist
 - $\alpha\beta$ Suffix von $P[1..i]$ (in P_1)
 - $\alpha\beta$ Präfix von P (in P_3)
 - Das Zeichen nach $\alpha\beta$ in P_1 ist ein Mismatch: $P[i+1] \neq T[k]$
 - Weil P_3 matched, muss aber $P[|\alpha\beta|+1] = T[k]$ sein
 - Also müsste gelten: $P[i+1] \neq P[|\alpha\beta|+1]$

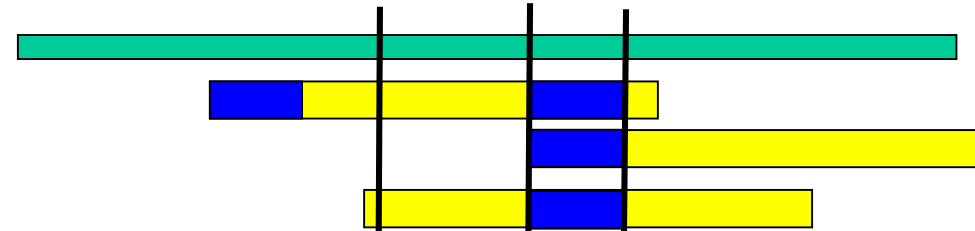
Beweis 3



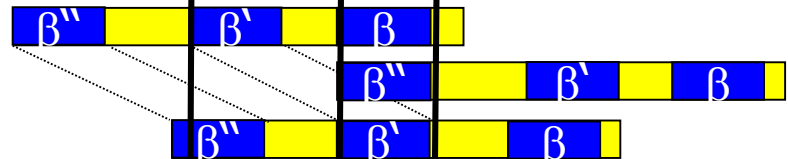
- Weiterhin muss gelten $|\alpha| > 0$
 - sonst ist $P_2 = P_3$ und wir haben nichts übersehen
- Damit
 - $\alpha\beta$ ist Präfix von P
 - $\alpha\beta$ ist Suffix von $P[1..i]$
 - $P[i+1] \neq P[|\alpha\beta|+1]$
 - $|\alpha\beta| > |\beta| = sp_i'$
- $\alpha\beta$ erfüllt alle Voraussetzungen für sp_i' , ist aber länger
- **Widerspruch zur Definition von sp_i'**

Mehrere β

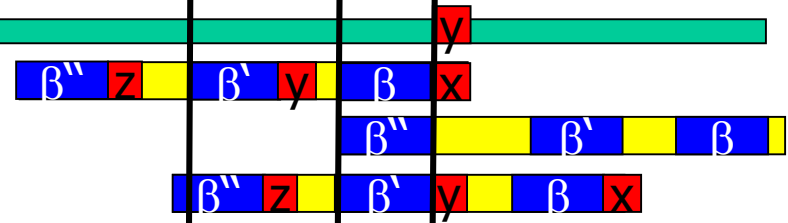
Ausgangssituation



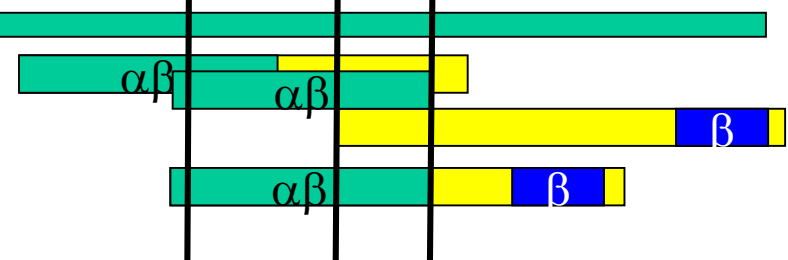
Sei β' das mittlere Vorkommen von β zwischen β und β''



Weil P3 matcht, müssen nach β und β' unterschiedliche Zeichen kommen ($x \neq y$)



P1 und P3 matchen tw. denselben Teilstring in T; $\alpha\beta$ ist also Präfix von P und Suffix von $P[1\dots i]$



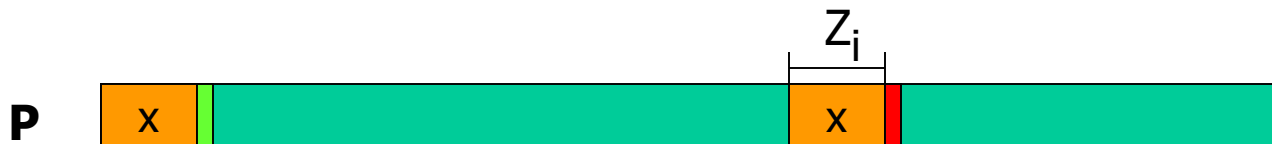
Wenn es also ein β' gibt, das zu einem Match führen würde, dann muss es wegen der Überlappung in T ein längeres Präfix/Suffix geben

Komplexität von KMP

- Theorem
 - Die Suchphase des KMP benötigt *höchstens $2m$ Zeichenvergleiche, ist als $O(m)$*
- Beweis
 - Mismatches
 - Pro Phase gibt es maximal einen Mismatch
 - Es gibt m Phasen, also maximal m Mismatches
 - Matches
 - Jedes Compare beginnt in T entweder
 - ... am letzten Zeichen des letzten Vergleichs (bei Mismatch), oder
 - ... am Zeichen rechts vom letzten Zeichen des letzten Vergleichs (bei vollständigem Match)
 - Damit wird kein Zeichen mehr als einmal positiv verglichen

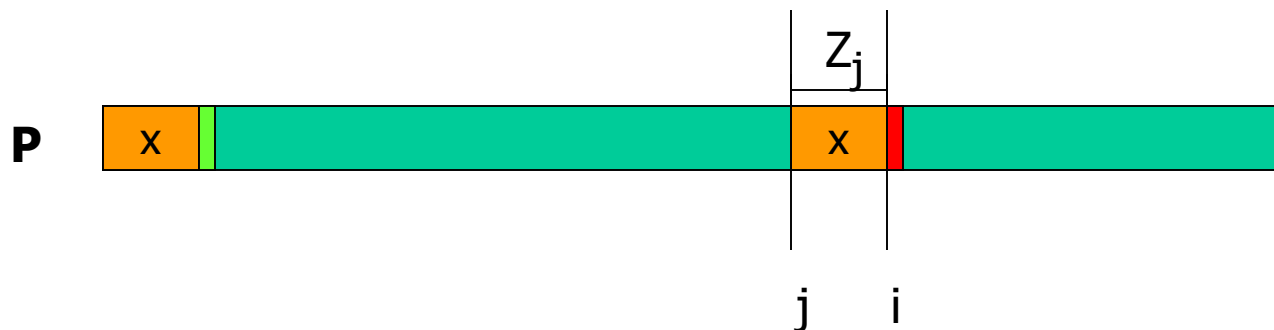
Preprocessing

- Gesucht: sp_i Werte
- Wir führen das Preprocessing auf Z-Boxen zurück
- Erinnerung (Z-Box)
 - Sei $i > 1$. Dann ist Z_i die Länge des *größten Substrings* x in P mit
 - $x = P[i..i+|x|-1]$ (x startet an Position i in P)
 - $P[i..i+|x|-1] = P[1..|x|-1]$ (x ist auch Präfix von P)
 - x heißt *Z-Box* von P an Position i mit Länge $Z_i(P)$



Verwendung der Z-Boxen

- Für alle i suchen wir echte Suffixe von $P[1..i]$, die auch Präfixe von P sind
 - ... und sich nicht verlängern lassen (sp_i)
 - Also für jede Position die **längste Z-Box**, die an ihr **endet**



Formal

- Theorem

- Für $i > 1$ sei $j > 1$ die am *weitesten links stehende Position* in P für die gilt: $i = j + Z_j - 1$
- Wenn j existiert, dann ist $sp_i' = Z_j = i - j + 1$
- Sonst $sp_i' = 0$

- Beweis

- Wenn j existiert, ist Z_j ein längstes Suffix von $P[1..i]$, das auch Präfix von P ist
 - Sonst wären Z -Boxen falsch berechnet
- Wenn j nicht existiert, matched kein Suffix von $P[1..i]$ mit einem Präfix von P
- qed.

Berechnung

```
for i = 1 to n           // Initialisierung
    spi := 0;
end for;
compute Z-Boxes on P;
for j = n downto 2      // Spätere (weiter links) Treffer
    i := j + Zj - 1;   // überschreiben frühere
    spi := Zj;
end for;
```

- Damit
 - Berechnung Z-Boxen ist $O(n)$
 - Berechnung sp_i ist $O(n)$
 - KMP Shift/Compare ist $O(m)$
- KMP ist $O(m+n)$

Shift-Regel:

Mismatch bei $i+1$

Schiebe um $i - sp_i'$

Vergleiche ab $sp_i' + 1$

Beispiel

Pos:	1	2	3	4	5	6	7	8
P:	G	C	A	G	C	T	A	G
sp_i' :	0	0	0	0	2	0	0	1

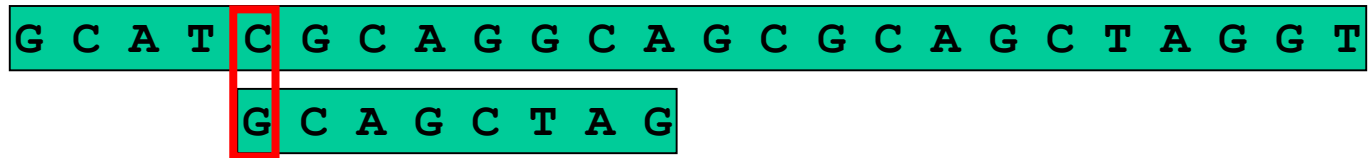
Schiebe um $3-0=3$
Vergleiche ab $0+1$



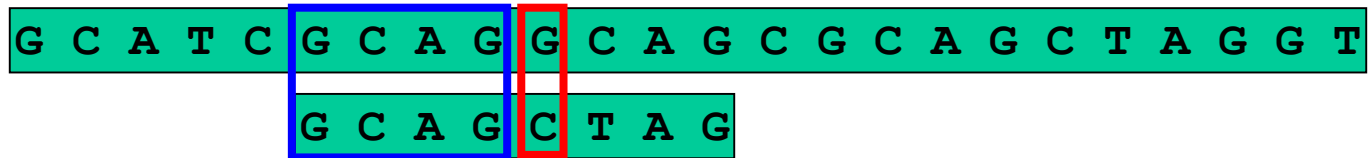
Schiebe um 1
Vergleiche ab 1



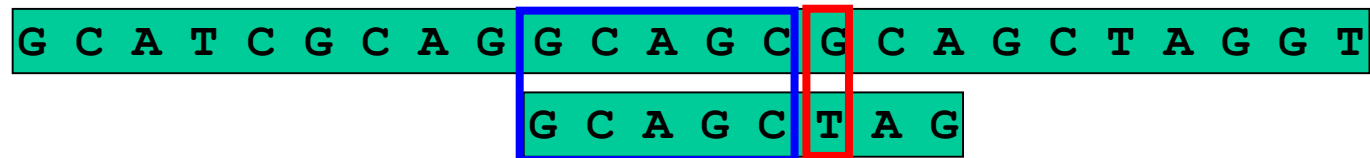
Schiebe um 1
Vergleiche ab 1



Schiebe um $4-0=4$
Vergleiche ab $0+1$



Schiebe um $5-2=3$
Vergleiche ab $2+1$



Shift-Regel:

Mismatch bei $i+1$

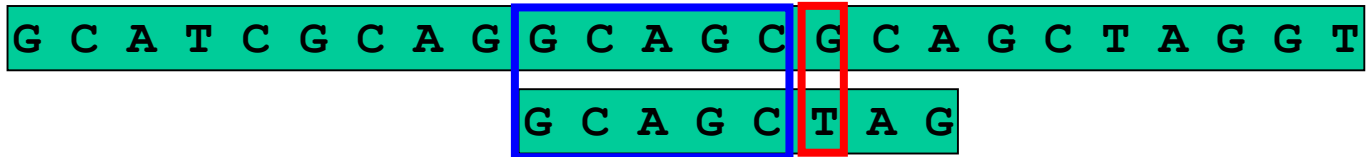
Schiebe um $i-sp_i'$

Vergleiche ab $sp_i'+1$

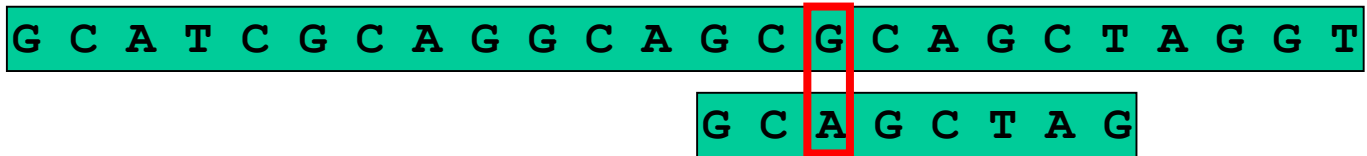
Beispiel 2

Pos:	1	2	3	4	5	6	7	8
P:	G	C	A	G	C	T	A	G
sp_i' :	0	0	0	0	2	0	0	1

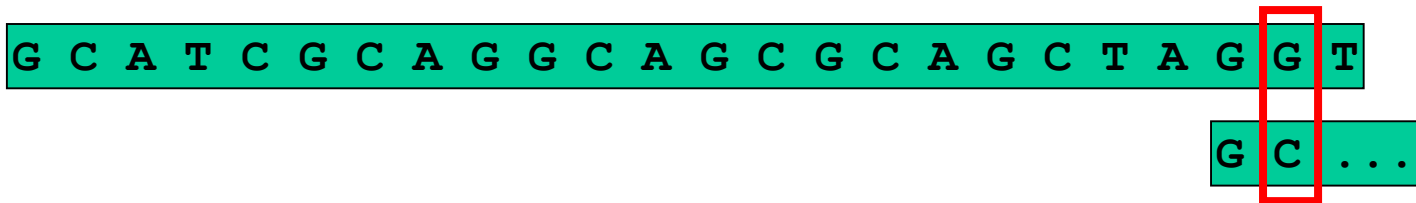
Schiebe um $5-2=3$
Vergleiche ab $2+1$



Schiebe um $2-0=2$
Vergleiche ab $0+1$



Schiebe um $8-1=7$
Vergleiche ab $1+1$



Kompletter KMP Algorithmus

```
compute spi`;  
h := 1;           // Next comparison in T  
i := 1;           // Next comparison in P  
while h+(n-i)<=m do  
    while P[i]=T[h] and i<=n do  
        i++;  
        h++;  
    end while;  
    if i=n+1 then      // Match  
        print h-n;  
    else                // Mismatch at h/i  
        if i=1 then  
            i++;      // First comp. fails - move 1 pos  
            h++;  
        end if;  
        // Comparison in T will continue at position h  
        // Comparison in P will continue after shift  
        i:=spi-1` +1;      // i is mismatch pos. in P  
    end while;
```

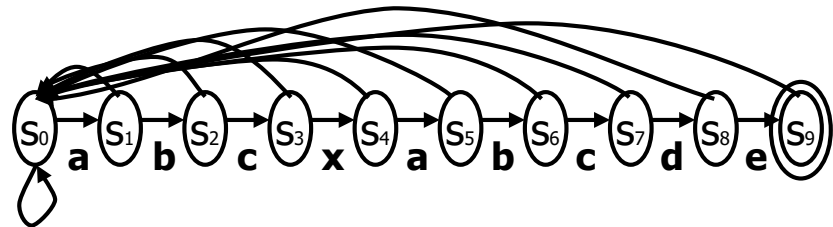

Inhalt dieser Vorlesung

- Knuth-Morris-Pratt
- Korrektheit und Komplexität
- **Patternmatching mit Automaten**
- Vergleich mit anderen Algorithmen

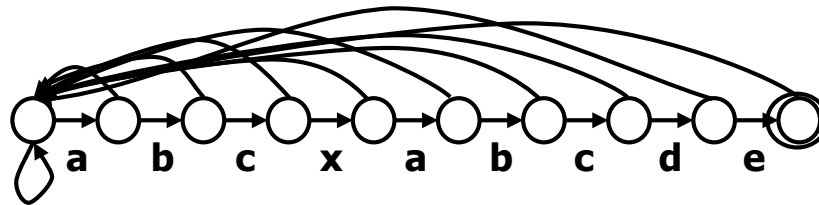
Pattern Matching mit Automaten

- Naiver Algorithmus als **DEA**
 - Konstruiere einen Automaten A aus dem Pattern
 - A hat $n+1$ Zustände, nummeriert mit s_i , $0 \leq i \leq n$
 - Jedes s_i mit $i < n$ hat zwei ausgehende Kanten
 - Eine beschriftet mit $P[i+1]$ und führt zu s_{i+1}
 - Eine für alle anderen Symbole und führt zu s_0
 - (Solche Kanten zeigen wir nicht – verkürzte Darstellung)
 - s_n ist einziger akzeptierender Zustand, s_0 ist Startzustand
 - Der Endzustand hat eine Kante zum Startzustand

P = abcxabcde



Matching



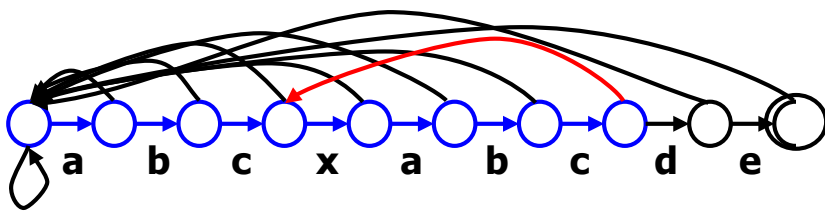
abcxabcgedsc

- `j:=1; s:=Startzustand`
- `while j≤m-n \\ Äußere Schleife`
 - `i:=0;`
 - `while true \\ Innere Schleife`
 - `Gehe von s zu s' gemäß T[j+i];`
 - `if s'=Endzustand: Match;`
 - `if s'=Startzustand: Break;`
 - `i++; s := s';`
 - `end while;`
 - `j++;`
- `end while;`

Komplexität:
 $O(m*n)$

Beobachtung

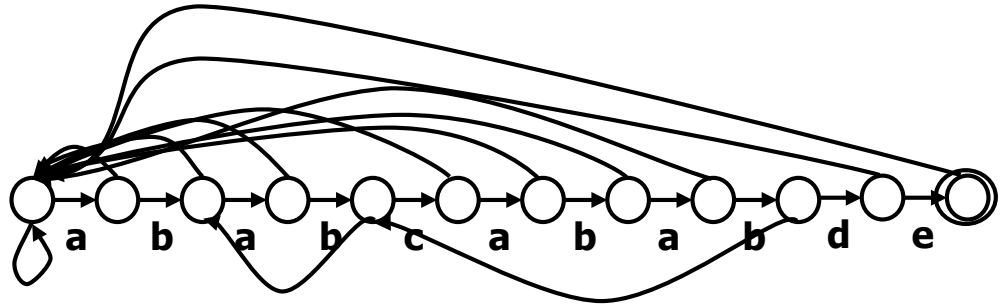
- Statt zum Startzustand zu springen können wir die sp_i Werte ausnutzen
 - Weiter schieben: Bei Mismatch an $i+1$ springe zum Zustand sp_i
 - Niemals doppelt positiv matchen: Nur eine Schleife j



```
j:=1; s:=Startzustand
while j ≤ m-n
  If s hat_kante mit T[j] oder
  s=Startzustand
    j++;
  Gehe von s zu s' gemäß T[j];
  if s'≠Endzustand: Match an j-n;
  s := s';
end while;
```

Komplexeres Beispiel

$P = ababcababde$



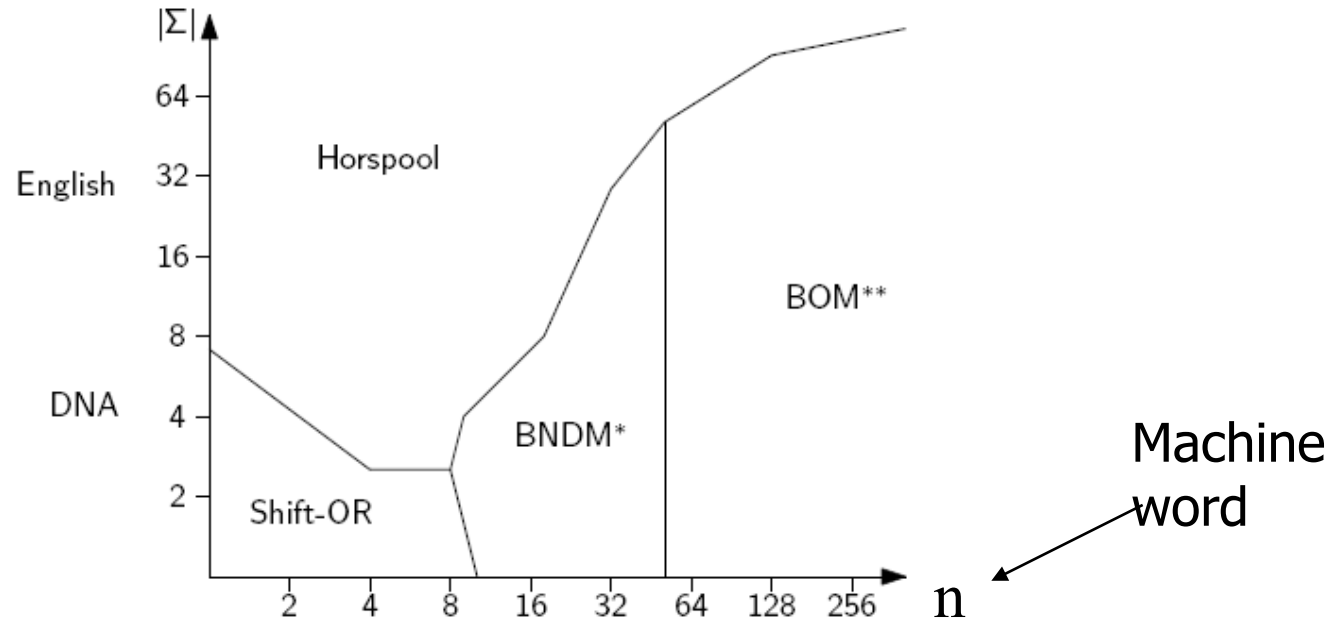
- Übergänge, die durch einen Mismatch beschriftet werden, nennt man auch **failure links**
- Beachte: In einem Schritt können mehrere failure Links hintereinander abgelaufen werden

Vergleich

	Z-Box	Boyer-Moore (Apostolico-Giancarlo)	Knuth-Morris-Pratt
Preprocessing	$O(m+n)$	$O(n)$	$O(n)$
Suche	$O(m)$	$O(m)$	$O(m)$
Gesamt	$O(m+n)$	$O(m+n)$	$O(m+n)$
Größe Alphabet	<ul style="list-style-type: none"> Praktisch unabhängig von Σ 	<ul style="list-style-type: none"> Je größer, desto besser – EBCR führt zu großen Sprüngen EBCR greift selten bei kleinen Alphabeten 	<ul style="list-style-type: none"> Praktisch unabhängig von Σ
Bemerkung	<ul style="list-style-type: none"> AC und WC Komplexität gleich 	<ul style="list-style-type: none"> Best Case ist $O(m/n)$ 	<ul style="list-style-type: none"> AC und WC Komplexität gleich Erweiterbar auf mehrere Pattern

Empirical Comparison [NR02]

(Gonzalo Navarro & Mathieu Raffinot, 2002)



- Shift-OR: Bit-wise parallelization (only small alphabets)
- BNDM: Backward nondeterministic DAWG Matching (automata-based)
- BOM: Backward Oracle Matching (automata-based)

Selbsttest

- Beweisen Sie die Korrektheit des KMP
- Begründen Sie, warum kein Online-Algorithmus weniger als $\sim m/n$ Vergleiche benötigen kann
 - Und wie könnte man trotzdem schneller werden?
- Warum darf die GSR nicht wie der KMP zum linksten Vorkommen des „good suffix“ schieben?
- Wie würde dieser Automat aussehen, wenn Sie die sp_i statt den sp_i' Werten verwenden?

