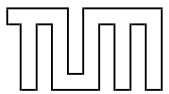
Algorithmische Bioinformatik 1

Dr. Hanjo Täubig

Lehrstuhl für Effiziente Algorithmen
(Prof. Dr. Ernst W. Mayr)
Institut für Informatik
Technische Universität München

Sommersemester 2009



Übersicht

Erweiterung des Aho-Corasick-Algorithmus

- Wie ist der Algorithmus von Aho-Corasick zu erweitern, wenn ein Suchwort aus S ein echtes Teilwort eines anderen Suchwortes aus S sein darf?
- Dann ist es möglich, dass der Algorithmus beim Auftreten eines Suchwortes $s \in S$ in einem internen Knoten v' des Suchwort-Baumes endet.
- Sei s' die Kantenbeschriftung des Pfades von der Wurzel zum Knoten v'.
- Da $s \in S$, muss ein Knoten v existieren, der ausgehend von der Wurzel über einen einfachen Pfad aus Baumkanten mit Beschriftung s erreicht wird.

Erweiterung des Aho-Corasick-Algorithmus

- Wir überlegen uns zuerst, dass s ein Suffix von s' sein muss.
- Sei $t's$ das Präfix von t, das gelesen wurde, bis ein Suchwort $s \in S$ gefunden wird.
- Gemäß der Vorgehensweise des Algorithmus und der Definition der Failure-Links muss s' ein Suffix von $t's$ sein.
- Ist $|s'| \geq |s|$, dann muss s ein Suffix von s' sein.

  Andernfalls ist $|s'| < |s|$ und somit $level(v') = |s'| < |s|$.
  Wir zeigen jetzt, dass dies nicht möglich sein kann.

Erweiterung des Aho-Corasick-Algorithmus

- Betrachten wir hierzu die Abarbeitung von $t's$.
- Sei $\bar{s}$ das längste Präfix von s, so dass sich der Algorithmus nach der Abarbeitung von $t'\bar{s}$ in einem Knoten w mit $level(w) \geq |\bar{s}|$ befindet.

  Da mindestens ein solches Präfix die Bedingung erfüllt (z.B. für $\bar{s} = \epsilon$, weil dann $|\bar{s}| = 0$), muss es auch ein längstes geben.
- Anschließend muss der Algorithmus dem Failure-Link von w folgen, da ansonsten $\bar{s}$ nicht das Längste gewesen wäre (beide Seiten der Ungleichung wären durch das reguläre Absteigen zu einem Kindknoten um 1 größer geworden).

Erweiterung des Aho-Corasick-Algorithmus

- Sei also $w' = \text{Failure-Link}(w)$.
- Es gilt $level(w') < |\bar{s}|$, sonst wäre $\bar{s}$ nicht das Längste gewesen.
- Dies kann aber nicht sein, denn $s \in S$ ist im Suchwort-Baum enthalten (also von der Wurzel erreichbar) und $\bar{s}$ ist ein Präfix von s.
  Damit muss es zu $\bar{s}$ einen Knoten im Suchwort-Baum auf Level $|\bar{s}|$ geben, wobei die Beschriftung des Pfades von der Wurzel zu diesem Knoten gerade $\bar{s}$ ist.
  Das heißt also: das maximale noch auffindbare Suffix des gelesenenTextteils $t's$ muss mindestens die Länge $\bar{s}$ haben.

Erweiterung des Aho-Corasick-Algorithmus

- Betrachten wir nun den Failure-Link des Knotens v'.
- Dieser kann auf den Knoten v zeigen (da ja s als Suffix auf dem Pfad zu v' auftreten muss).
- Andernfalls kann er nach der vorigen Überlegung nur auf andere Knoten auf einem höheren Level zeigen, wobei die Beschriftung dieses Pfades dann s als Suffix beinhalten muss.
- Eine Wiederholung dieser Argumentation zeigt, dass letztendlich über die Failure-Links der Knoten v besucht wird.
- Daher genügt es, den Failure-Links zu folgen, bis ein Knoten erreicht wird, der einem Wort aus S entspricht.
- In diesem Fall haben wir ein Suchwort im Text gefunden.
- Enden wir andernfalls an der Wurzel, so kann an der betreffenden Stelle in t kein Suchwort aus S enden.

Erweiterung des Aho-Corasick-Algorithmus

Resultierende Erweiterung des Aho-Corasick-Algorithmus

- Wann immer wir einen neuen Knoten über eine Baumkante erreichen, müssen wir testen, ob über eine Folge von Failure-Links (eventuell auch keine) ein Knoten erreichbar ist, der einem Wort aus S entspricht.
- Falls ja, haben wir ein Suchwort gefunden, ansonsten nicht.
- Anstatt nun jedes Mal den Failure-Links zu folgen, können wir dies auch in einem Vorverarbeitungsschritt durchführen.

Erweiterung des Aho-Corasick-Algorithmus

- Zuerst markieren wir alle Knoten als Treffer, die einem Wort aus S entsprechen.
- Die Wurzel wird als Nicht-Treffer und alle internen Knoten als unbekannt markiert.
- Dann durchlaufen wir alle als unbekannt markierten Knoten des Baumes. Von jedem solchen Knoten aus folgen wir solange den Failure-Links bis wir auf einen mit Treffer oder Nicht-Treffer markierten Knoten treffen.
- Anschließend markieren wir alle Knoten auf diesem Pfad genau so wie den gefundenen Knoten.
- Sobald wir nun im normalen Algorithmus von Aho-Corasick auf einen Knoten treffen, der mit Treffer markiert ist, haben wir ein Suchwort im Text gefunden, ansonsten nicht.
- Die Vorverarbeitung lässt sich in Zeit $O(m)$ implementieren, so dass die Gesamtlaufzeit bei $O(m + n)$ bleibt.

2. Erweiterung des Aho-Corasick-Algorithmus

- Wollen wir zusätzlich auch noch die Endpositionen aller Treffer ausgeben, so müssen wir den Algorithmus noch weiter modifizieren.
- Wir hängen zusätzlich an die Knoten mit Treffer noch eine Liste mit den Längen der Suchworte an, die an dieser Position enden können.
- Zu Beginn erhält jeder Trefferknoten eine einelementige Liste, die die Länge des zugehörigen Suchwortes beinhaltet.
- Alle anderen Knoten bekommen eine leere Liste.
- Finden wir nun einen Knoten, der als Treffer markiert ist, so wird dessen Liste an alle Listen der Knoten auf dem Pfad dorthin als Kopie angefügt.

2. Erweiterung des Aho-Corasick-Algorithmus

- Treffen wir nun beim Algorithmus von Aho-Corasick auf einen als Treffer markierten Knoten, so müssen jetzt mehrere Antworten ausgegeben werden (die Anzahl entspricht der Elemente der Liste).

- Somit ergibt sich für die Laufzeit $O(m + n + k)$, wobei
  - m die Anzahl der Zeichen in den Suchwörtern,
  - n die Länge des Textes t und
  - k die Anzahl der Vorkommen von Suchwörtern aus S in t

  ist.

2. Erweiterung des Aho-Corasick-Algorithmus

Theorem

Sei $S \subseteq \Sigma^+$ eine Menge von Suchwörtern.

Dann findet der Algorithmus von Aho-Corasick alle Vorkommen von $s \in S$ in $t \in \Sigma^*$ in der Zeit $O(n + m + k)$, wobei

- *$n = |t|$, und*
- *$m = \Sigma_{s \in S} |s|$ und*
- *k die Anzahl der Vorkommen von Suchwörtern aus s in t*

ist.

Zweiter naiver Ansatz

- weiterer Ansatzpunkt zum Suchen in Texten:
  das gesuchte Wort mit einem Textstück nicht mehr von links nach rechts, sondern von rechts nach links vergleichen

- Vorteil:
  In Alphabeten mit vielen verschiedenen Symbolen kann man bei einem Mismatch an der Position $i + j$ in t, i auf $i + j + 1$ setzen, falls das im Text t vorgefundene Zeichen t_{i+j} gar nicht im gesuchten Wort s vorkommt.

- D.h. also, in so einem Fall kann man das Suchwort gleich um $j + 1$ Positionen verschieben (im günstigsten Fall m Stellen).

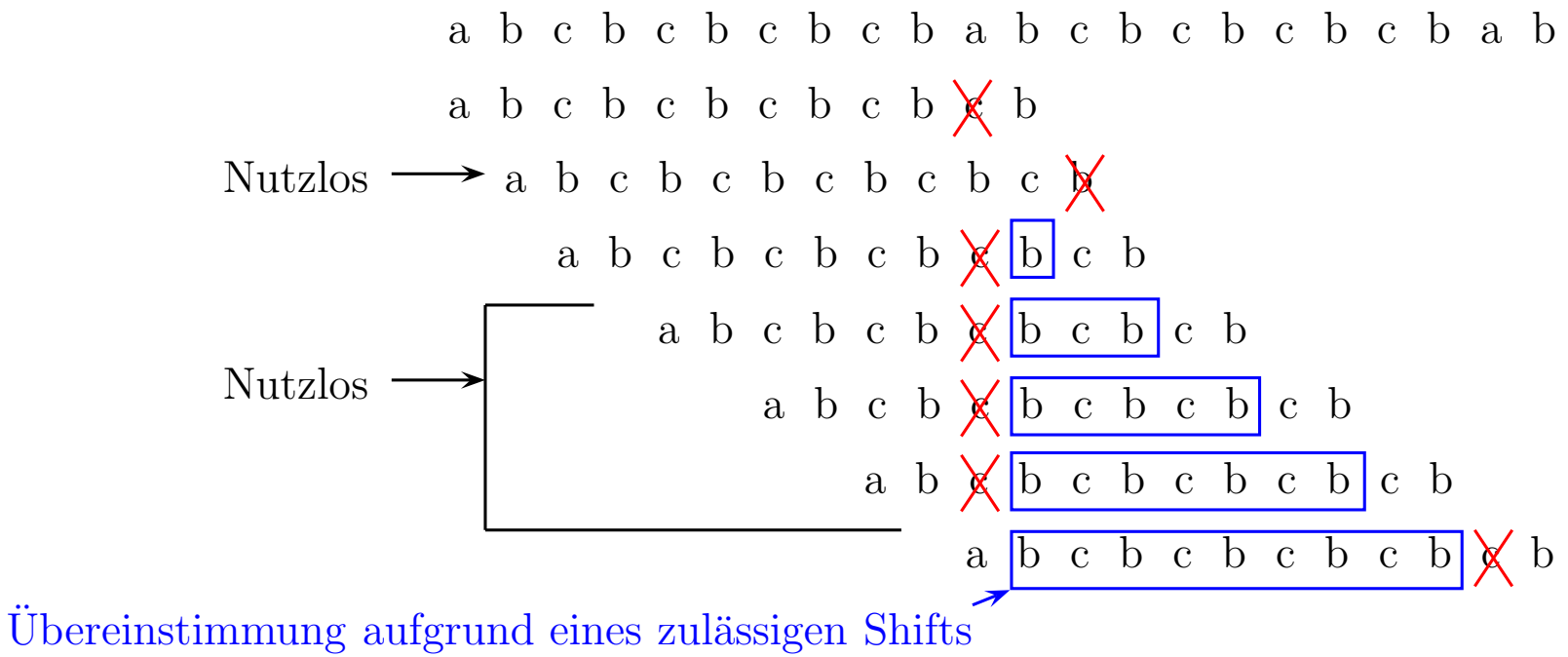
Zweiter naiver Ansatz

Algorithmus 6 : bool Naiv2(char $t[]$, int n, char $s[]$, int m)

```
int i := 0;
int j := m − 1;
while i ≤ n − m do
    while t[i + j] = s[j] do
        if j = 0 then return TRUE;
        j−−;
    i++;
    j := m − 1;
return FALSE;
```

Naive Methode mit rechts-nach-links Vergleichen
(ohne größere Shifts)

Nutzlose Verschiebungen

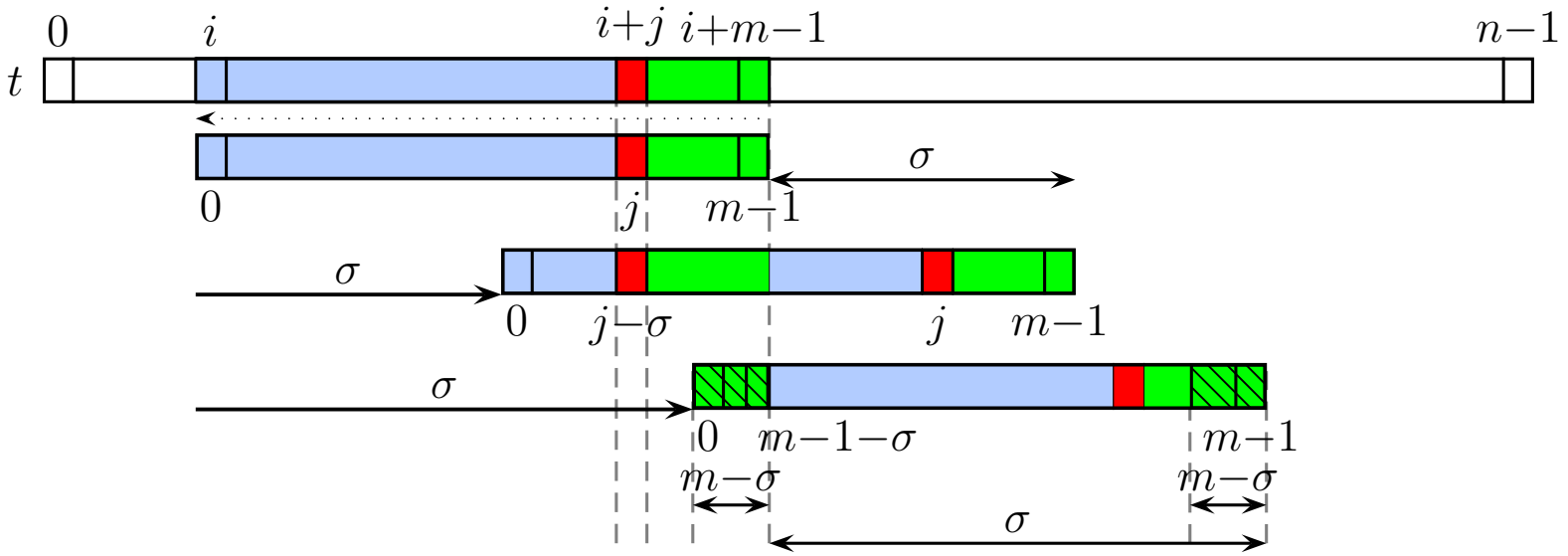


Nutzlose Verschiebungen

- Wie beim KMP-Algorithmus ist es nutzlos (siehe zweiter Versuch von oben), wenn man die Zeichenreihe so verschiebt, dass im Bereich erfolgreicher Vergleiche nach einem Shift keine Übereinstimmung mehr herrscht.

- Es macht ebenfalls keinen Sinn, das Muster so zu verschieben, dass an der Position des Mismatches in t im Muster s wiederum dasselbe Zeichen zum Liegen kommt, das schon vorher den Mismatch ausgelöst hat.
(vierter bis sechster Versuch)

Boyer-Moore-Algorithmus



Skizze: Zulässige Shifts bei Boyer-Moore (Strong-Good-Suffix-Rule)

Boyer-Moore-Algorithmus: Shifts

- Zwei mögliche Arten eines „vernünftigen“ Shifts bei der Variante von Boyer-Moore:
  - Im oberen Teil ist ein „kurzer“ Shift angegeben, bei dem im grünen Bereich die Zeichen nach dem Shift weiterhin übereinstimmen.
    Das rote Zeichen in t (das den Mismatch ausgelöst hat) soll nach dem Shift auf ein anderes Zeichen in s treffen, damit überhaupt die Chance auf Übereinstimmung besteht.
  - Im unteren Teil ist ein „langer“ Shift angegeben, bei dem die Zeichenreihe s soweit verschoben wird, dass an der Position des Mismatches in t gar kein weiterer Vergleich mehr entsteht. Aber auch hier: wieder Übereinstimmung im schraffierten grünen Bereich mit den bereits verglichenen Zeichen aus t

Boyer-Moore-Algorithmus: Good-Suffix-Rule

- Kombination der beiden Regeln: Good-Suffix-Rule
  (da man darauf achtet, die Zeichenreihen so zu verschieben, dass im letzten übereinstimmenden Bereich nach dem Shift wieder Übereinstimmung herrscht)
- Achtet man noch speziell darauf, dass an der Position, in der es zum Mismatch gekommen ist, jetzt in s ein anderes Zeichen liegt als das, welches den Mismatch ausgelöst hat, so spricht man von der Strong-Good-Suffix-Rule (andernfalls Weak-Good-Suffix-Rule).
- Wir betrachten nur die Strong-Good-Suffix-Rule, da ansonsten die worst-case Laufzeit wieder quadratisch werden kann.

Boyer-Moore-Algorithmus

Algorithmus 7 : bool Boyer-Moore(char $t[]$, int n, char $s[]$, int m)

```
int S[m + 1];
compute_shift_table(S, m, s);
int i := 0, j := m − 1;
while i ≤ n − m do
    while t[i + j] = s[j] do
        if j = 0 then return TRUE;
        j−−;
    i := i + S[j];
    j := m − 1;
return FALSE;
```

Wiederholte Vergleiche in übereinstimmenden Bereichen

- Nach dem Shift gibt es einen Bereich, in dem Übereinstimmung von s und t vorliegt.

- Allerdings werden auch in diesem Bereich wieder Vergleiche ausgeführt, da es letztendlich doch zu aufwendig ist, sich diesen Bereich explizit zu merken und bei folgenden Vergleichen von s in t zu überspringen.