Algorithmen auf Sequenzen

Kapitel 1
Einfache Textsuche

Webseite zur Vorlesung
http://ls11-www.cs.tu-dortmund.de/people/rahmann/teaching/ss2008/AlgorithmenAufSequenzen

Sprechstunde
Mo 16-17 in OH14, R214

Kapitel 1: Einfache Textsuche

Das Ur-Problem: Finden eines Strings in einem anderen String

- **Vergleichsbasierte Verfahren**
- Die naïve Methode
- **Präfixbaiserter Ansatz**
- Verwendung von endlichen Automaten
- Der Knuth-Morris-Pratt-Algorithmus
- KMP als DFA-Simulation
- Bit-parallele DFA-Simulation: Shift-And, Shift-Or
- **Suffixbasierter Ansatz**
- Varianten des Boyer-Moore-Algorithmus (Horspool, Sunday)
- **Teilstringbasierter Ansatz**
- Backward (Nondeterministic) DAWG matching (BNDM)
- Backward Oracle Matching (BOM) mit dem Teilstring-Orakel

- **Arithmetische Verfahren**
- Q-gram Codes
- Rabin-Karp (Fingerprinting)

Problemstellung

Gegeben

- Alphabet *A*
- Sequenz *T* (Text) der Länge *n*
- Sequenz *P* (Muster, pattern) der Länge *m*
- Häufig: $m \ll n$

Fragen

- Kommt *P* in *T* vor?
- Wie oft kommt *P* in *T* vor?
- An welchen Positionen beginnt *P* in *T*?

Algorithmen, die eine dieser Fragen beantworten,
lassen sich oft auf einfache Weise so modifizieren,
dass sie auch die anderen beiden Fragen beantworten.

Wir konzentrieren uns auf die 3. Frage.

Naiver Algorithmus

Algorithmus

for each position *i* in *T*
  if lcp(*P*,1,*T*,*i*)==*m* **then** report *i*

Funktion lcp(*s*,*j*,*t*,*i*) vergleicht *s* und *t*, beginnend an Position *j* in s und *i* in *t*, und gibt die Länge des längsten übereinstimmenden Präfixes zurück.

Laufzeit des Algorithmus: O(*mn*)

- Die äußere Schleife braucht O(*n*) Durchläufe
- Die Überprüfung für jede Position *i* besteht aus bis zu *m* Zeichenvergleichen

Erwartete Laufzeit

Annahmen:

- Zeichen in T sind unabhängig identisch verteilt.
- An jeder Position steht jeder Buchstabe mit Wahrscheinlichkeit 1/|A|

Analyse:

- Die äußere Schleife braucht O(*n*) Durchläufe.
- Die Anzahl der Zeichenvergleiche bis zum ersten Fehlschlag ist geometrisch verteilt mit Parameter (|*A*|-1)/|*A*|; maximal *m*.
- Der Erwartungswert ist |*A*|/(|*A*|-1) = 1 + 1/(|*A*|-1); bzw. *m* wenn |*A*|=1.
- Erwartete Laufzeit: **O(*n* (1 + 1/(|*A*|-1)))**, unabh. von *m;* bzw. **O(*mn*)** wenn |*A*|=1.
- Diskussion?

Verbesserung des naiven Algorithmus

Problem des naiven Algorithmus

Gut auf Zufallssequenzen aus großen Alphabeten,
aber keine gute Laufzeitgarantie!

Zeichen des Textes werden u.U. mehrfach verglichen.
Das sollte nicht nötig sein.

Wunschliste

- Laufzeitgarantie $O(n+m)$ [Linearzeit]
- Weniger als n Vergleiche im Erwartungswert [(schwache) Sublinearität]

Mustersuche mit endlichen Automaten

Idee

- Schiebe ein Fenster der Länge m über den Text, an Position $i = 1 ..n-m+1$ (wie naiv)
- Bestimme jeweils lcp(T,i,P,1) (wie naiver Algorithmus)
- Nutze bereits gelesene Zeichen von T, um den neuen lcp-Wert schneller zu finden.
- Speichere die Information der letzten Zeichen von T in einem endlichen Automaten.

Definition

Ein **deterministischer endlicher Automat** (deterministic finite automaton, DFA) ist ein 5-Tupel (Q, q_0, F, A, δ) mit

- endlicher Zustandsmenge Q
- Startzustand q_0
- akzeptierenden Zuständen F
- Eingabealphabet A
- Übergangsfunktion $\delta: Q \times A \rightarrow Q$

Erweiterung der Übergangsfunktion auf Strings: $\delta: Q \times A^* \rightarrow Q$
Definiere $\delta(q,\epsilon) := q$
Definiere $\delta(q,xa) := \delta(\delta(q,x),a)$ für x aus A^*, a aus A

Definiere state(x) := $\delta(q_0,x)$, Zustand des DFA nach „Lesen“ von x.

Mustersuche mit endlichen Automaten

Anwenden eines DFA auf einen Text $T \in A^*$:

Setze i := 0; starte in Zustand $q := q_0$

Wiederhole [Schritt von i zu i+1]
  Wenn q akzeptierend, **dann** gib $i-m+1$ aus
  Lies das nächste Zeichen a aus T (**wenn** nicht möglich, **dann stop**)
  Vollziehe Zustandsübergang $q := \delta(q,a)$, erhöhe i um 1
  [Jetzt ist q = state(T[1..i])]

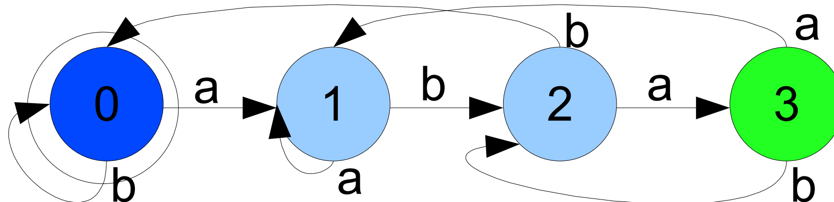
Design-Ziel des Automaten

state(T[1..i]) ist akzeptierend genau dann wenn $T[i-m+1 .. i] = P$
q ist maximal mit dieser Eigenschaft.

Design-Idee

Zustände modellieren die Länge des längsten Suffix von T[1..i], das Präfix von P ist.
Q := {0,...,m}; der Wert q soll bedeuten:

- Die letzten q gelesenen Zeichen von T sind ein Präfix von P.
- q ist maximal mit dieser Eigenschaft.



DFA für P = aba

Mustersuche mit endlichen Automaten

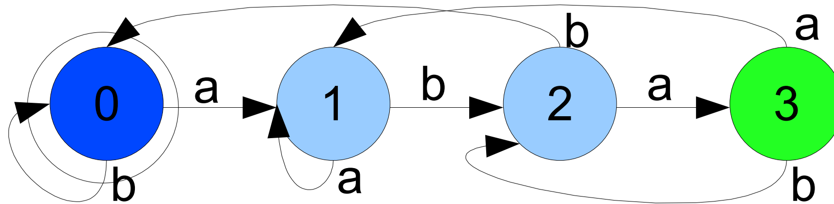
Design-Idee

Q := {0,...,*m*}; der Wert *q* soll bedeuten:

- Die letzten *q* gelesenen Zeichen von *T* sind ein Präfix von *P.*
- *q* ist maximal mit dieser Eigenschaft.

Definition (longest prefix-suffix)

lps(x) := max {*k* : P[1..*k*] ist Suffix von *x*}



DFA für *P* = aba

Spezifikation des Automaten

- *A* ist das gemeinsame Alphabet von *T* und *P*
- $Q := \{0,\ldots,m\}$; $q_0 := 0$
- $Z = \{m\}$

$\delta(q,a)$:= lps(*P*[1..*q*]*a*)

Behauptung Mit dieser Definition der Übergangsfunktion gilt:
Für jeden String *x* ist state(*x*) = lps(x).

Mustersuche mit endlichen Automaten

Konstruktion der Übergangsfunktion (naiv)

Für jedes der $|Q||A|$ Argumente der Übergangsfunktion kann der Wert von lps($P[1..q]a$) durch naiven Vergleich in $O(m^2)$ Zeit ermittelt werden.

```
for q := 0 .. m
    for each character a in A
        for k := min {m, q+1} downto 0
            if P[1..k] is a suffix of P[1..q]a then break
        δ(q,a) := k
```

Analyse

- Laufzeit der Konstruktion: $O(|A|m^3)$
- Speicherbedarf der Übergangsfunktion: $O(|A|m)$
- Laufzeit der Suche: $O(n)$ [bei konstanter Zugriffszeit auf $\delta(q,a)$]

Verbesserung auf $O(|A|m)$ Zeit möglich durch besseren Konstruktionsalgorithmus.
Fundamentaler Nachteil: Faktor $|A|$ in der Spezifikation der Übergangsfunktion.

Zusammenfassung / Verbesserung

Wichtige Punkte

- Der Zustand q (als Zahl) ist stets die Länge des längsten Präfixes von P mit $P[1..q]$ = T[i-q+1 .. i].
- Beim Übergang i-1 $\rightarrow$ i gibt es zwei Fälle
  - Erfolgsfall T[i] = P[q+1]: q erhöht sich um 1
  - Misserfolgsfall: q wird auf das längstmögliche Präfixlänge reduziert, so dass dann $T[i]$ angehängt werden kann (oder es wird q=0).

Beobachtung

Der Automat speichert explizit zu jedem q und Buchstaben die Präfixlänge für den Misserfolgsfall. Das ist nicht notwendig!

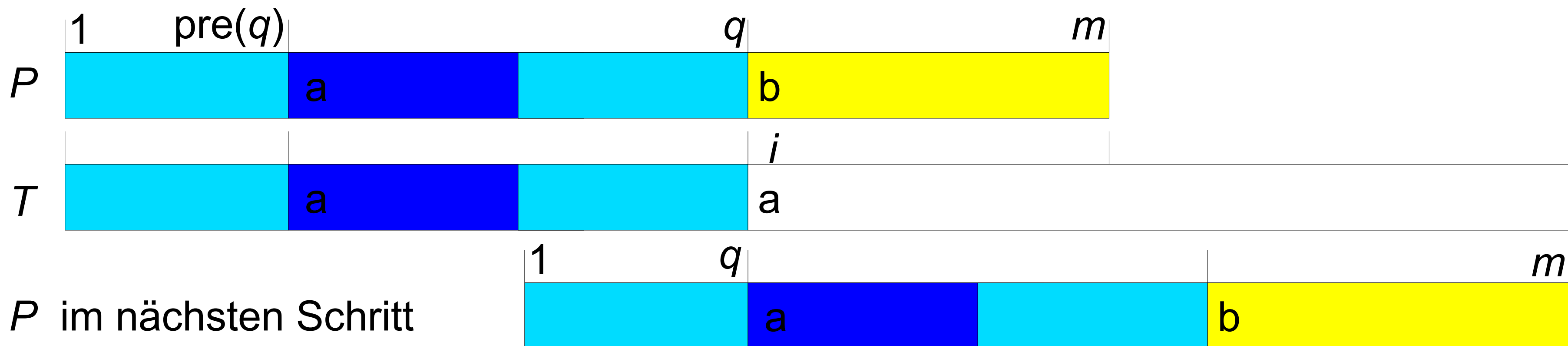
Statt dessen:
Man nimmt immer das nächste längstmögliche Präfix
und prüft dann, ob das nächste Zeichen passt.
Wenn nicht, reduziert man die Präfixlänge weiter.

Das ist die Grundidee hinter dem Knuth-Morris-Pratt Algorithmus,

Knuth-Morris-Pratt Algorithmus

Hauptidee und Definition

Man muss nicht die ganze Übergangsfunktion vorberechnen.
Es genügt zu notieren, welche Präfixe mit welchen Teilstrings von P übereinstimmen.
Sei $\text{pre}(q) := \max \{ k < q : P[1..k] \text{ ist ein echtes Suffix von } P[1..q] \}$ für $q = 1 .. m$



Wenn in Schritt i das nächste Zeichen $T[i]$ nicht passt (ungleich $P[q+1]$),
geht man zum längsten Präfix von P über, das eine Chance hat zu passen.

Nach wie vor misst q, das längste Prefix von P, das an Position i in T endet.
Man kann sich vorstellen: KMP simuliert Übergänge des Automaten.

Knuth-Morris-Pratt Algorithmus

KMP-Matching-Algorithmus

q := 0
for i := 1 .. n
  while (q>0 and P[q+1] != T[i]) **do** q := pre(q)
  if (P[q+1] = T[i]) **then** q := q + 1
  if (q = m) **then** { report position i-m+1; q := pre(q) }

Effiziente Berechnung von pre

pre(q) := max { k < q : P[1..k] ist ein echtes Suffix von P[1..q] } für q = 1 .. m
Derselbe Algorithmus, der P mit T vergleicht, berechnet pre, während P wächst!
Zur Berechnung von pre(q) aus pre(1) ... pre(q-1)=k wird

- wenn möglich (P[k+1]=P[q]) das bekannte k auf k+1 verlängert
- ansonsten vorher auf die längstmögliche Präfixlänge gestutzt.

KMP-pre-Algorithmus

k := 0
pre(1) := k
for q := 2 .. m
  while (k>0 and P[k+1] != P[q]) **do** k := pre(k)
  if (P[k+1] = P[q]) **then** k := k + 1
  pre(q) := k

Knuth-Morris-Pratt Algorithmus

Laufzeitanalyse des pre-Algorithmus [Beispiel für Potentialanalyse]
Kernfrage: Wie oft wird die Bedingung der while-Schleife geprüft?

- Es ist stets pre(k) < k, also wird k in der while-Schleife erniedrigt, aber nicht unter 0.
- Höchstens einmal kann k pro for-Schleifeniteration erhöht werden.
- Also kann die while-Bedingung höchstens O(m) mal insgesamt geprüft werden.
- Gesamtlaufzeit ist O(m), Speicherbedarf ebenfalls O(m)!

Laufzeitanalyse des Matching-Algorithmus
- Dasselbe Argument liefert Laufzeit O(n)
- Gesamtlaufzeit (worst-case) damit: **O(m+n)**

Erwartete Laufzeit
entspricht asymptotisch der worst-case Laufzeit.

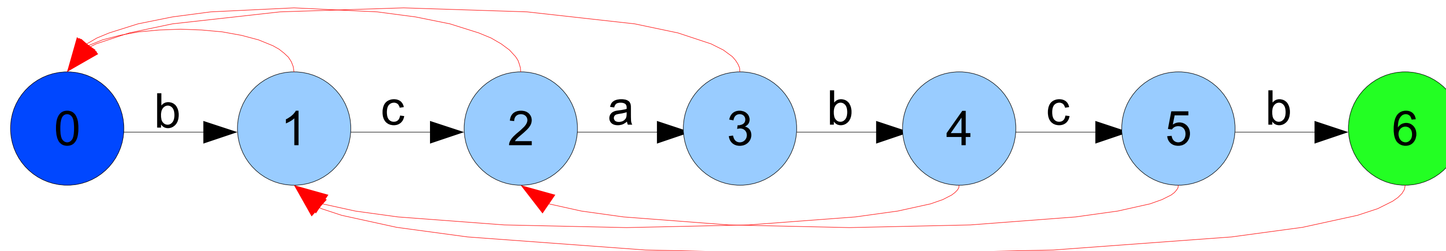
Knuth-Morris-Pratt als DFA-Simulation

Beispiel A = {a,b,c}, P = bcabcb

Der lineare Teil des Automaten („Erfolgsfälle“) wird wie gehabt konstruiert.
Die Präfixreduktionen über sogeannte failure- oder prefix-Links.
Speichern muss man nur die Ziele der prefix-Links: die pre-Funktion.

Sei T= bbcabcabcbc. Dann ist die Zustandsfolge:
(0,b:1,b:01, c:2, a:3, b:4, c:5, a:23, b:4, c:5, b:6!1, c:2)

Zur Erinnerung: Nach einem Match (6!) wird immer der Präfix-Link verfolgt.



Verwendung von **Nicht**deterministischen Automaten

Definition

Ein **nichtdeterministischer endlicher Automat** (NFA)
ist ein 5-Tupel (Q, Q_0, F, A, δ) mit
- endlicher Zustandsmenge Q
- Startzustandsmenge Q_0
- akzeptierenden Zuständen F
- Eingabealphabet A
- Übergangsfunktion δ: $Q \times A \to 2^Q$

[Unterschied zu DFA: Mehrere Übergänge beim Lesen eines Zeichen möglich,
alle Möglichkeiten werden „gleichzeitig“ genommen.]

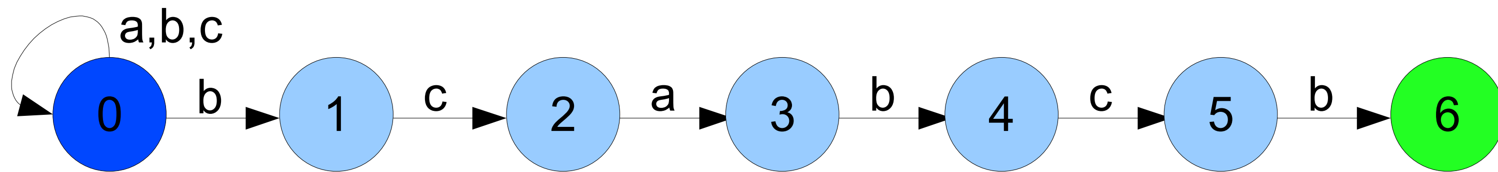
Erweiterung der Übergangsfunktion auf Zustandsmengen und Strings:
δ: $2^Q \times A^* \to 2^Q$
Definiere $\delta(Q,x) := \bigcup_{q \in Q} \delta(q,x)$ für alle strings x
Definiere $\delta(q,\epsilon) := q$
Definiere $\delta(q,xa) := \delta(\delta(q,x),a)$ für x aus A^*, a aus A

Definiere state(x) := $\delta(Q_0,x)$, Zustandsmenge des NFA nach „Lesen“ von x.
NFA akzeptiert x, wenn state(x) $\cap F \neq \emptyset$.

NFAs für die Textsuche

Der NFA zum Erkennen eines Strings P (der alle Strings der Form A^*P akzeptiert) hat eine sehr einfache Struktur:
eine Schleife (ganzes Alphabet) im Zustand 0, das Muster von links nach rechts.

Beispiel *P* = bcabcb



Zustand *q* ist genau dann aktiv,
wenn die letzten *q* gelesenen Zeichen in T gleich P[1..*q*] sind.

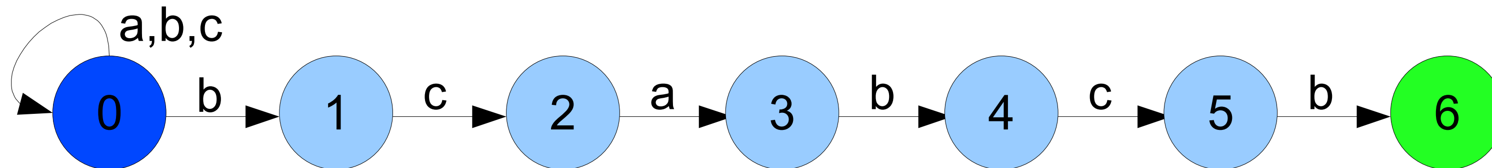
Statt eines einzelnen Zustands ist i.d.R. eine Zustandsmenge aktiv,
z.B. {0,2,5} nach Lesen von bcabc.
[Erinnerung: Umwandlung von NFAs in DFAs mittels Teilmengenkonstruktion.]

Beobachtung
Aktive Zustände können in einem Feld von |*P*|+1 bits gespeichert werden
(ein einziger integer wenn |*P*|<32).

Effiziente NFA-basierte Algorithmen zur Textsuche

Grundidee

- Aktive Zustände in einem (oder wenigen) integer-Wert(en) *z* speichern
- Update aktiver Zustandsmenge (state(*x*) nach state(*xa*))
  durch bit-parellele Operationen (shift, &, |): schneller als mit for-Schleifen o.ä.
- Die einfache links-rechts-lineare Struktur des NFA macht dies möglich.



Shift-And Algorithmus

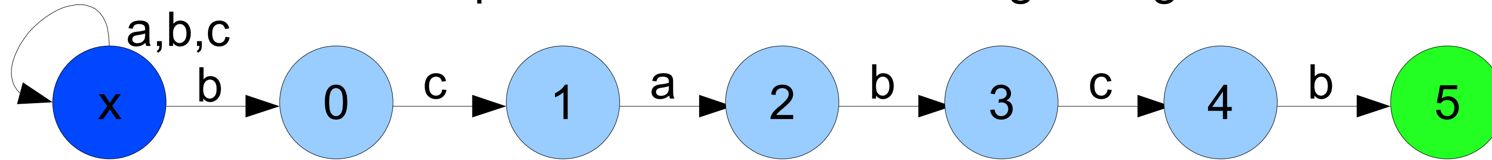
- Bit *q* von *z* ist 1 genau dann wenn Zustand *q* aktiv ist
- Initialisierung: *z*=1 (nur Bit 0 ist 1)
- Update beim Lesen von T_i: $z := (z \ll 1 \ \& \ \text{mask}[T_i]) \mid 1$
- Match gefunden, wenn (z & 2^m) != 0 [*m* war die Länge von *P*]

Das **shift-left** sorgt für das „Weiterschieben" der aktiven Zustände,
das **and** mit der Maske des aktuellen Buchstaben dafür,
dass die bits nur gesetzt bleiben, wenn das richtige Zeichen gelesen wurde.
Definition von mask[*a*]: Bit *q* (*q*=1..*m*) ist genau dann gesetzt, wenn $P_q = a$.

Das **or** mit 2^0=1 sorgt dafür, dass Zustand 0 wieder aktiviert wird.

Shift-And: Bemerkungen

Wie eben beschrieben, benötigt man m+1 Bits; das 0-te Bit ist immer gesetzt.
Man kann das Bit einsparen und die Bits wie folgt belegen:



Folgende Modifikationen sind nötig:

- Definition von mask[a]: Bit q (q=0..m-1) ist genau dann gesetzt, wenn $P_{q+1}=a$.
- Initialisierung: z=0 (keiner der numerierten Zustände ist aktiv)
- Update beim Lesen von T_i: z := ((z << 1) | 1) & mask[T_i]
- Match gefunden, genau dann wenn (z & 2^{m-1}) != 0

Kommentare

- Technische Probleme / Zeitverlust, wenn mehr als 1 integer notwendig.
- Sehr einfach zu implementieren
- Am effektivsten bei kurzem pattern und kleinem Alphabet

Shift-And: Laufzeitanalyse

Laufzeitanalyse

- Eigentlich O(mn), aber w (Registergröße) Operationen bit-parallel
- Laufzeit daher häufig als O(nm/w) oder O(n) angegeben.
- Genaue Aussagen abhängig vom Kostenmodell. Welche Operationen sind O(1)?
- Standardmodell: Elementare Operation auf bis zu O(log m) bits in O(1) Zeit, führt auf O(nm/log m) Zeit
- Erstellen der $|A|$ Masken: O($m|A|$) Zeit und Platz

Shift-Or

Kleine Laufzeitverbesserung

Beim **shift-and** update z := ((z << 1) | 1) & mask[T_i]

Ist einer von drei Schritten nur dazu da, Bit 0 auf Eins zu setzen.

Das kann man sich sparen, wenn man die Bits komplementär codiert.

Dies führt auf den **shift-or** Algorithmus (Details in der Übung).

Zusammenfassung Präfix-basierte Methoden

Die DFA / KMP – Ideen basieren darauf,
sich vorab Informationen über die Struktur des Musters P zu beschaffen,
insbesondere, wo Präfixe von P wieder als Teilstrings von P erscheinen.

Damit kann man in (asymptotisch) konstanter Zeit für jedes Textzeichen ausrechnen,
wie lang das längste Präfix von P ist, das an der aktuellen Position i im Text endet.
(Ist diese Länge gleich m, hat man P an Positionen $i-m+1$ .. i gefunden.)

Mit der pre-Funktion von KMP wird die Übergangsfunktion des Automaten simuliert.

Ein DFA kann effizient durch einen NFA simuliert werden.
Zum Speichern der Menge der aktiven NFA-Zustände verwendet man ein bit-Feld.
Da der Weg zum Startzustand zum akzeptierenden Zustand linear ist,
kann man effizient mit bit-parallelen Operationen arbeiten (shift, &, |).

Gemeinsamkeit
Man aktualisiert Informationen über das Präfix des Musters: Präfix-basierter Ansatz
Nachteil: Jedes Zeichen des Textes muss mindestens einmal angeschaut werden.

Klassifikation von pattern matching Algorithmen

Text vorher nicht bekannt, Vorverarbeitung des Musters *P* möglich
Ein Fenster der Länge |*P*| wird von links nach rechts über den Text verschoben und sein Inhalt mit *P* verglichen.

- **Präfix-basiert**: Aktualisiere Informationen über passende Präfixe von *P*;
  z.B. naiv, DFA (real-time), KMP (online), shift-and, shift-or (online)
  Nachteil: Jedes Zeichen im Text muss mindestens einmal betrachtet werden.
- **Suffix-basiert**: Vergleiche innerhalb des Fensters von rechts nach links, aktualisiere Informationen über passende Suffixe von *P*;
  z.B. naiv rückwärts, Boyer-Moore (Horspool, Sunday)
- **Teilstring-basiert**: Vergleiche von rechts nach links, Informationen über Teilstrings
  z.B. ...

Text vorher bekannt, **Index-basiert**
Vorverarbeitung: Erstellen einer Index-Datenstruktur des Textes
Suchen: Suche *P* im Index statt im Text

Erklärungen
online: Der Text wird Zeichen für Zeichen gelesen,
man muss nicht zu einem früher gelesenen Zeichen zurückkehren (Datenstrom)
real-time: online und O(1) Zeit zur Verarbeitung <u>jedes</u> Zeichens

Suffixbasierte Algorithmen

Grundidee (Best-case Szenario)
Schiebe (wie bei präfixbasierten Algorithmen) ein Fenster W über T.
Betrachte zuerst das letzte Zeichen im Textfenster *W*.
Kommt dieses Zeichen in *P* gar nicht vor, kann man *P* um *m* Schritte verschieben.

Damit erhält man eine best-case Laufzeit von $O(n/m)$.

Schneller kann es nicht (korrekt) gehen, denn:
Gibt es einen Teilstring aus >= *m* Zeichen in *T*, die man alle nicht betrachtet,
kann man ein Vorkommen von *P* übersehen.

Horspool-Algorithmus (1980)

Schiebe Fenster *W* über den Text, bis der Text erschöpft ist:
- Betrachte das letzte Zeichen *c* in *W*.
- Kommt *c* in *P* nicht vor, verschiebe *W* um *m* Positionen.
- Kommt *c* in P zuletzt an Position *j*<*m* vor, verschiebe *W* um *m-j* Positionen.
- Kommt *c* in P an Position *m* vor, verifiziere (in beliebiger Reihenfolge) ob *P*=*W*.
  - Ist *P*=*W*, gib die aktuelle Startposition von *W* aus.
  - Verschiebe *W* um *m-j* Positionen; hier *j* = letzte Position von *c* in W links von *m*

Vorverarbeitung (Berechnung der Verschiebung für jedes Zeichen):

```
for each character c:  shift[c] := m
for j := 1 .. m-1:  shift[P[j]] := m-j //  letztes Zeichen nicht betrachten!
PLast := P[m];
```

Matching:

```
for (i:=0; i <= n-m; i+=shift[c]):
   if ((c:=T[i+m]) == PLast):
      compare T[i+1 .. i+m-1] against P[1 .. m-1] // (1)
      if comparison successful then report i+1
```

Bemerkung zu (1): Z.B. beginnend mit den seltensten Zeichen in P, ...
oder mit hardwareunterstützter memcmp-Instruktion von links nach rechts

Sunday-Algorithmus (1990)

Sunday schlug folgende Variation des Horspool-Algorithmus vor:
Statt das letzte Zeichen in *W* zu betrachten, betrachte das darauf folgende Zeichen.
Dieses wird vor dem nächsten Vergleich mit P in Übereinstimmung gebracht.

```
for each character c:  shiftSunday[c] := m+1 // nicht vorkommende Zeichen
for j := 1 .. m:  shiftSunday[P[j]] := m-j+1 // bis m statt m-1, shifts +1
```

Schiebe Fenster *W* über den Text, bis der Text erschöpft ist:
  Verifiziere in beliebiger Reihenfolge, ob *P*=*W*.
  Ist *P*=*W*, gib die aktuelle Startposition von *W* aus.
  Sei *c* das nächste Zeichen, das im Text auf *W* folgt.
  Verschiebe *W* um shiftSunday[*c*] Positionen.

```
for (i:=0; i <= n-m; i+=shiftSunday[T[i+m+1]):
   compare T[i+1 .. i+m] against P[1 .. m] // (2)
   if comparison successful then report i+1
```

Es wird zur Vereinfachung angenommen, dass T[n+1] einen sentinel enthält.
Bemerkung (2): Es kann wieder in beliebiger Reihenfolge verglichen werden.

Analyse von Horspool und Sunday

Vorverarbeitung

jeweils $O(|A|+m)$ Zeit, $O(|A|)$ Platzbedarf

Best case

Horspool: n/m Textzugriffe
Sunday: $2n/(m+1)$ Textzugriffe

Worst case

Horspool: mn Textzugriffe
Sunday: $(m+1)n$ Textzugriffe

Average case (P, T zufällig): Ein schwieriges Thema für sich – gleich mehr dazu.

Vergleich Horspool - Sunday:

Horspool: Erwartete Anzahl Textzugriffe pro Verifikation: $z \le |A|/(|A|-1) = O(1)$
Horspool: Erwartete Shiftlänge pro Verifikation: s
Sunday Textzugriffe: $z+1$
Sunday Shiftlänge: $s+1 - x$ (letztes Zeichen in P hat kürzeren Shift)

Fall eines großen Alphabets

Sunday liefert pro Verifikationsphase eine Shiftlänge +1,
braucht pro Verifikationsphase +1 Textzugriff -> langsamer als Horspool

Analyse von Horspool und Sunday

Fall $|A|$=Theta(m)

Eine Verifikationsphase dauert O(1) erwartete Zeit, unabhängig von m (wie naiver Alg.)
Ein shift erreicht erwartete Länge Theta(m).
Erwartete Laufzeit damit O(n/m).

Fall $|A|$=O(1)

Eine Verifikationsphase dauert O(1) erwartete Zeit, unabhängig von m (wie naiver Alg.)
Ein shift erreicht erwartete Länge Theta(1).
Erwartete Laufzeit damit O(n).

Bemerkung

Horspool und Sunday verfahren fast wie der naive Algorithmus (links-rechts Verifikation)
und benutzen nur Informationen über das letzte (folgende) Zeichen,
um längere shifts zu erreichen.
Die worst-case Laufzeit ist O(mn).

Average-Case Analyse des Sunday-Algorithmus

Warnung! Die folgende Analyse ist nicht mathematisch rigoros.
Sie soll lediglich die wichtigsten Ideen vermitteln!

- Wir analysieren den Algorithmus für ein festes gegebenes pattern P, daher nehmen wir an, dass wir ShiftSunday[ ] kennen.

- Betrachte die einzelnen Schritte des Sunday-Algorithmus. Ein Schritt besteht aus Vergleichsphase und Shift. Wir analysieren beides getrennt.

- Analyse der Anzahl der Vergleiche V_t in der Vergleichsphase in Schritt t: Die Verteilung hängt nicht von t ab. V_t ist geometrisch verteilt mit Parameter $(|A|-1)/|A|$ (abgeschnitten bei m); der Erwartungswert ist $E[V_t] <= |A|/(|A|-1)$; dies ist analog zum naiven Algorithmus.

- Analyse der Shiftlänge L_t in Schritt t: Alle L_t sind identisch verteilt und unabhängig. Die Verteilung der Shiftlänge ergibt sich direkt aus dem ShiftSunday-Array: Bei Gleichverteilung über A ist $P(L_t = k) = |\{c\text{: ShiftSunday}[c]=k\}| / |A|$

Average-Case Analyse des Sunday-Algorithmus

- Sei X_t die Position des Suchfensters am Ende von Schritt *t* (nach Shift).
  Damit ist $X_0 = 0$ und $X_t = X_{t-1} + L_t = \sum^t_{\tau=1} L_\tau$
  Die Verteilung von X_t kann man explizit ausrechnen.
  (Summe von unabhängigen identisch verteilten Zufallsvariablen mit bekannter Vert.)

- Alternativ kann man den zentralen Grenzwertsatz (ZGWS) anwenden:
  X_t hat näherungsweise eine Normalverteilung mit Erwartungswert *t**E[*L*]
  und Varianz *t**Var[*L*]. Dabei kann man E[*L*] und Var[*L*] elementar ausrechnen.
  Aus der Stochastik bekannte Abschätzungen liefern dann:
  P(X_t >*n*) ~

- Uns interessiert: Wie viele Schritte werden bis zum Ende des Textes benötigt?
  Sei also W_n := min { *t* : X_t > *n*}.

- Wichtigste Beobachtung:
  Das Ereignis {W_n <= *t* } ist identisch mit dem Ereignis {X_t >*n*}.
  Daher auch P(W_n <= *t*) = P(X_t >*n*) ~
  Alternativ kann man dies wieder exakt ausrechnen;
  insbesondere kann man den Erwartungswert E[W_n] bestimmen.

Average-Case Analyse des Sunday-Algorithmus

- Zum Schluss müssen wir beides zusammenbringen.
  Die Gesamtzahl der Vergleiche des Sunday-Algorithmus ist $Z = \sum_{t=1}^{W_n} V_t$.
  eine Summe mit einer zufälligen Anzahl (W_n) von Summanden.
  Die Anzahl der Textzugriffe ist um W_n-1 höher als die Anzahl der Vergleiche.

- Nun setzen wir voraus, dass E[*Z*] = E[W_n] * E[*V*] gilt.
  Beide Faktoren sind jetzt bekannt.
  (An dieser Stelle sind wir extrem unrigoros
  – wir müssten uns über Abhängigkeiten von W_n und *V* sorgfältig Gedanken machen.)

- Man kann noch argumentieren, dass E[W_n] linear mit n wächst; E[W_n] ~ *cn*.
  Damit erhält man E[*Z*] ~ *c* |*A*|/(|*A*|-1) *n*,
  dabei hängt *c* von der ShiftSunday-Funktion des Patterns ab
  und lässt sich wie gezeigt berechnen.

- Eine ähnliche Analyse kann man für den Horspool-Algorithmus durchführen,
  wobei hier die Abhängigkeiten zwischen Shift und Anzahl der Vergleiche stärker sind.

Boyer-Moore Algorithmus

Grundlagen

Suffix-basiert (Vergleiche innerhalb des Fensters von rechts nach links)
Entweder man findet ein Vorkommen von *P*,
oder das erste Zeichen von rechts, das nicht übereinstimmt, ist an Position *q*.

Good suffix heuristic

Für jedes *q*, berechne maximal mögliche sichere Verschiebung.
- Wo kommt das Suffix P[*q*+1 .. *m*] links von *q*+1 wieder als Teilstring von *P* vor?
- Wie lang ist das längste Suffix von P[*q*+1 .. *m*], das auch Präfix von *P* ist?

Bad character heuristic

- Wo kommt das nicht übereinstimmende Textzeichen am weitesten rechts in *P* vor?

Wähle die längere
der beiden Verschiebungen

Boyer-Moore Algorithmus

Bemerkungen

- Die Bad character heuristic kann zu negativen shifts führen.
  Dann wird sie ignoriert, da die Good suffix heuristic einen positiven shift vorschlägt.

- Die Berechnung der Suffix-basierten Verschiebung ist in O(m) Zeit möglich;
  es wird O(m) Speicherplatz benötigt.

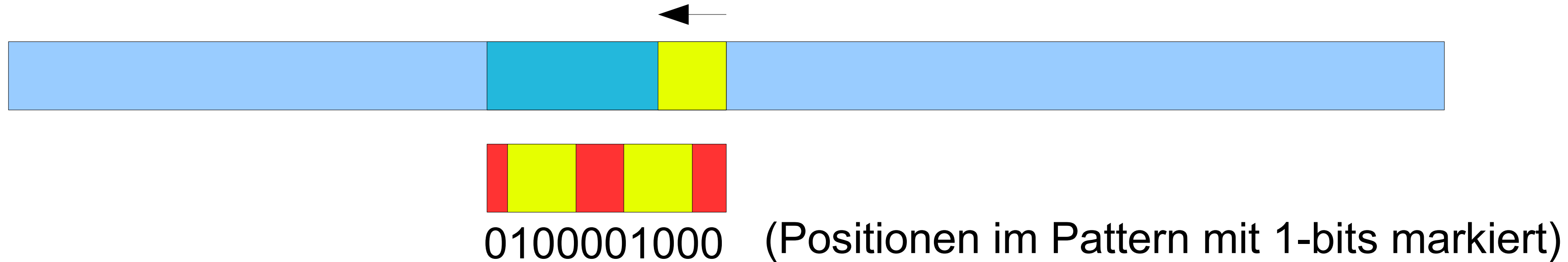
- In der Praxis ist die Horspool-Variante (oder ein anderer Algorithmus) schneller.

- Aus diesem Grund diskutieren wir Boyer-Moore nicht detaillierter.

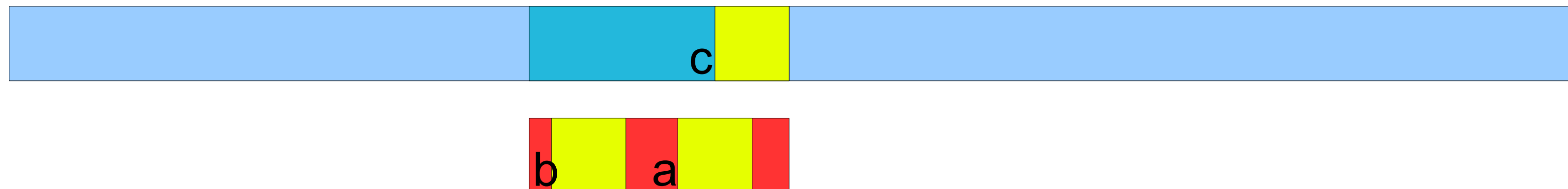
- In der Praxis sind oft einfachere, aber theoretisch „schlechtere“ Algorithmen,
  also die mit schlechterer Laufzeit in O-Notation, die besseren!

Teilstring-Basierte Verfahren

Idee 1: Wie bisher, schiebt man ein Fenster W über den Text T von links nach rechts.
Wie bei Suffix-basierten Verfahren, liest man im Fenster von rechts nach links.
Man merkt sich, wo überall das gelesene Suffix als Teilstring des Patterns vorkommt.



Grundidee dabei:
Wenn das gelesene Suffix nicht mehr als Teilstring des Patterns auftritt
(hier: c + gelber Kasten),
kann man das Pattern auf jeden Fall hinter das zuletzt gelesene Zeichen verschieben
(hier: hinter c).



Diese Idee kann man noch weiter verbessern, wenn man zusätzlich wüsste,
welche gelesenen Suffixe des Fensters auch Präfixe des Patterns sind.

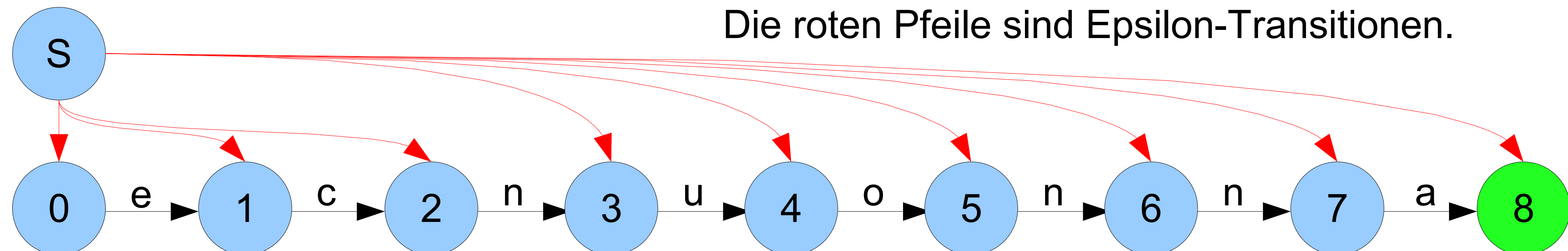
Teilstring-Basierte Verfahren

Idee 2: Dazu brauchen wir einen Automaten,
der die Teilstrings des reversen Musters liest und dessen Suffixe akzeptiert.
Ein nichtdeterministischer Automat mit Epsilon-Transitionen ist leicht zu erstellen.

Epsilon-Transitionen erweitern den NFA-Formalismus.
Sie können jederzeit ausgeführt werden, ohne ein Zeichen zu lesen.

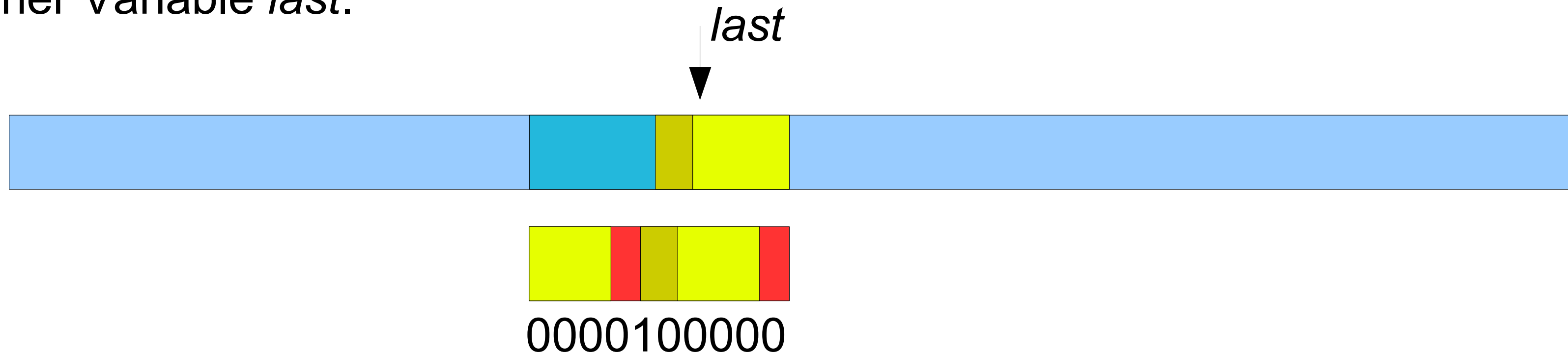
Beispiel: Automat zum reversen String von „announce“

Die roten Pfeile sind Epsilon-Transitionen.



Teilstring-Basierte Verfahren

Idee 3: Wann immer man ein Suffix des reversen Patterns erkennt, also ein Präfix des Patterns, merkt man sich dessen Startposition im Fenster in einer Variable *last*.



Behauptung: Im nächsten Schritt kann man das Fenster so weit verschieben, dass es an Position last beginnt.

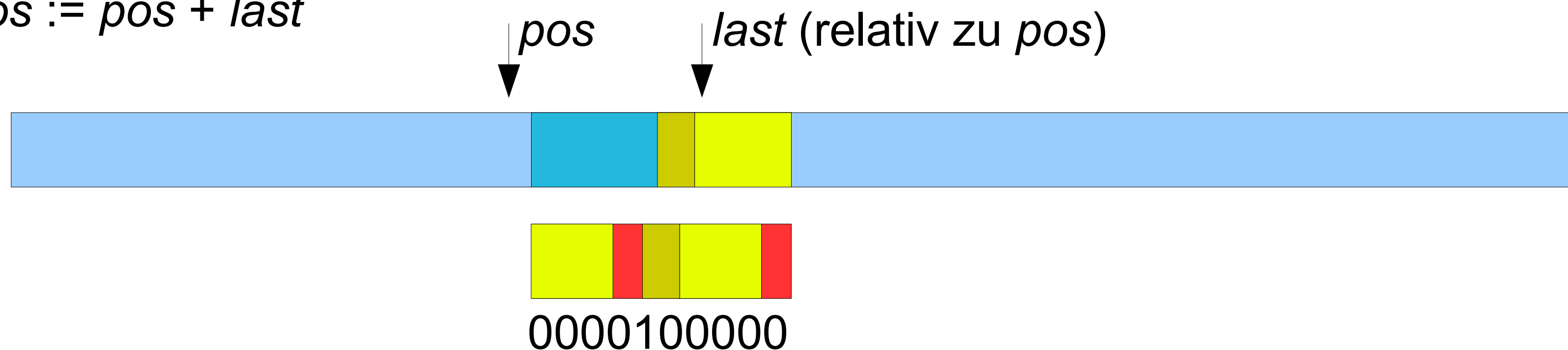
Beweis: Man muss zeigen, dass man kein Vorkommen von P, das zwischen der aktuellen Position des Fensters und *last* beginnt, übersehen kann.
Ein solches Vorkommen entspräche einem Präfix des Patterns im aktuellen Fenster und würde daher vom Automaten erkannt werden, so dass *last* entsprechend gesetzt würde.

BNDM – Bit-parallele Simulation des NFA

Vorverarbeitung:
Berechne Bitmasken mask[c],
so dass Bit q (q = 0 .. m-1) von mask[c] genau dann gesetzt ist, wenn $P_{m-q} = c$.

Suche – äußere Schleife:

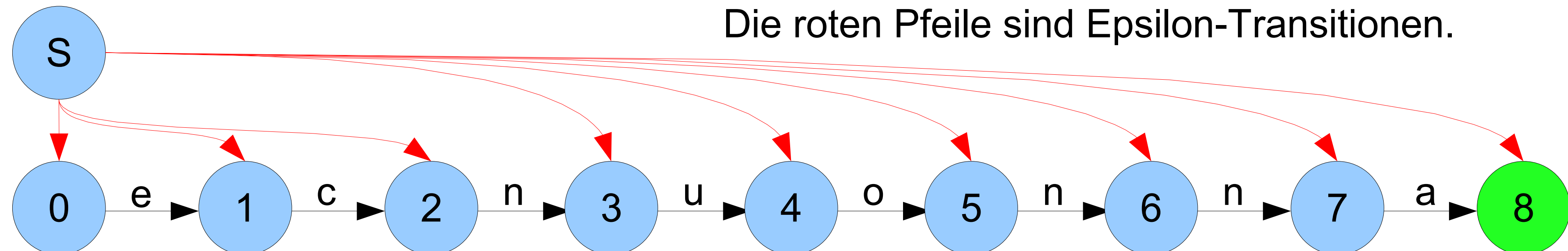
pos := 0 // pos verweist auf die Position vor dem Start des Fensters
while (*pos* <= *n*-*m*)
  j := *m*
  last := *m*
  A := 1^m // Alle Zustände sind zunächst aktiv (Epsilon-Transitionen)
  while (*A* != 0^m)
    Verarbeite das Zeichen an Textposition *pos* + *j*
    Aktualisiere *A* und ggf. *last*, gib an, ob *P* gefunden wurde.
    Dekrementiere *j*
  pos := *pos* + *last*



BNDM – Bit-parallele Simulation des NFA

Suche – Details:

```
pos := 0  // pos verweist auf die Position vor dem Start des Fensters
while (pos <= n-m):
   j := m
   last := m
   A := 1^m // Alle Zustände sind zunächst aktiv (Epsilon-Transitionen)
   while (A != 0^m):
      A := A & mask[T[pos + j]] // Anwendung von & vor dem shift
      j := j – 1
      if (A & 10^(m-1) !=0): // letzter Zustand erreicht?
         if (j>0): last := j
         else: report an occurrence of P at position pos+1
      A := A << 1
   pos := pos + last
```



BNDM – Namensgebung

BNDM heißt „Backward non-deterministic DAWG matching“ (Navarro & Raffinot '00)

- Backward, matching: klar
- non-deterministic: wegen NFA mit Epsilon-Transitionen
- DAWG: directed acyclic word graph

Aus jedem NFA kann man mit Hilfe der Teilmengenkonstruktion einen DFA erstellen.
Dieser erkennt ebenfalls alle Suffixe des reversen Patterns.
Er heißt Suffix-Automat (des reversen Patterns).
Technisch ist es ein gerichteter azyklischer Wort-Graph (DAWG).

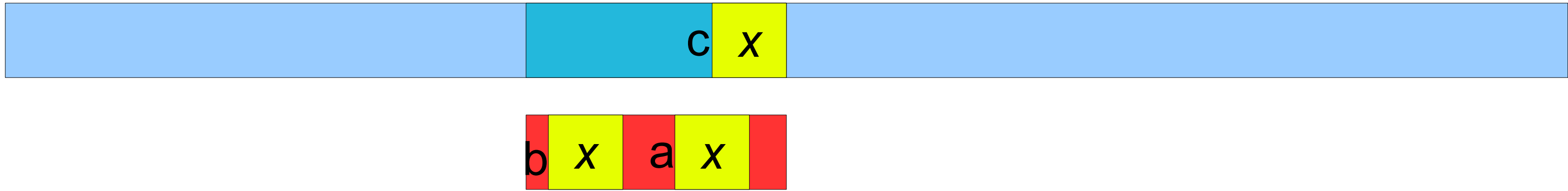
Der Suffix-Automat *S* eines Wortes *s* hat folgende Eigenschaften:

- Ein String *t* ist ein Teilstring von *s* genau dann, wenn es einen Pfad in *S* vom Startzustand aus gibt, der durch *t* beschrieben wird.
- Ein String *t* ist ein Suffix von *s* genau dann, wenn der zugehörige Pfad in einem akzeptierenden Zustand endet.
- Man kann den Automaten in $O(|s|)$ Zeit online erstellen – das ist aber kompliziert.

Verwendung des Suffix-Automaten statt des simulierenden NFA führt auf Algorithmus BDM (Backward DAWG matching; Crochemore et al. '94).
Zeitkomplexität: $O(mn)$ worst case, aber optimale $O(n \log_{|A|} m / m)$ average case.

Andere Teilstring-basierte Idee: Teilstring-Orakel

Bei der Grundidee des Teilstring-basierten Ansatzes war folgendes wichtig:
Wenn das gelesene Suffix nicht mehr als Teilstring des Patterns auftritt,
kann man das Pattern auf jeden Fall hinter das zuletzt gelesene Zeichen verschieben.



Beobachtung: Dazu muss man nicht wissen, dass *x* ein Teilstring war,
sondern nur, dass *cx* keiner ist.

Man kann also statt eines Suffix-Automaten (oder äquivalenten NFA)
einen Automaten verwenden, der mehr (aber nicht viel mehr) als Teilstrings
des reversen Patterns erkennt. Gelten muss:
„Es gibt keinen Pfad, der *y* buchstabiert“ $\Rightarrow$ „*y* ist kein Teilstring des reversen Patterns“.
Äquivalent dazu:
„*y* ist Teilstring des reversen Patterns“ $\Rightarrow$ „Es gibt einen Pfad, der *y* buchstabiert“.
Die Gegenrichtung ist nicht gefordert.
Dies erlaubt, einen einfacher zu konstruierenden Automaten zu verwenden.

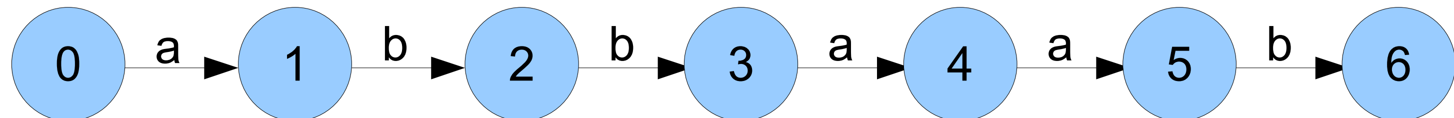
Das Teilstring-Orakel (factor oracle)

Das Teilstring-Orakel von P ist ein azyklischer Automat mit folgenden Eigenschaften:
- Es erkennt alle Teilstrings (Faktoren) von P [und nicht viel mehr]
- Es erkennt genau einen String der Länge m, nämlich P
- Es hat möglichst wenig Zustände, nämlich m+1
- Es hat höchstens eine lineare Anzahl an Übergängen (zwischen m und 2m-1).

Genauer ist das Teilstring-Orakel durch den folgenden Algorithmus definiert:
1. Erzeuge die m+1 Zustände 0 .. m (alle akzeptierend, Startzustand ist 0)
2. Erzeuge die m Transitionen $i \to i+1$, annotiert jeweils mit $P[i+1]$ (linearer Teil)
3. Für i = 0 .. m-1 (nichtlinearer Teil):
   Sei u ein (das!) Wort minimaler Länge, mit dem Zustand i erreicht werden kann.
   Für alle Buchstaben $a \neq P[i+1]$:
   Wenn ua ein Teilstring von $P[i-|u|+1 .. m]$ ist:
      Sei j die erste Endposition von ua in $P[i-|u|+1 .. m]$
      Erzeuge Transition $i \to j$, annotiert mit a

Nach Schritt 1.+2. werden alle Präfixe erkannt.
Nach Schritt 3. werden vom Startzustand 0 aus auch die anderen Teilstrings erkannt, ohne dass zusätzliche Zustände erzeugt werden!

Beispiel: abbaab
Schritte 1.+2.



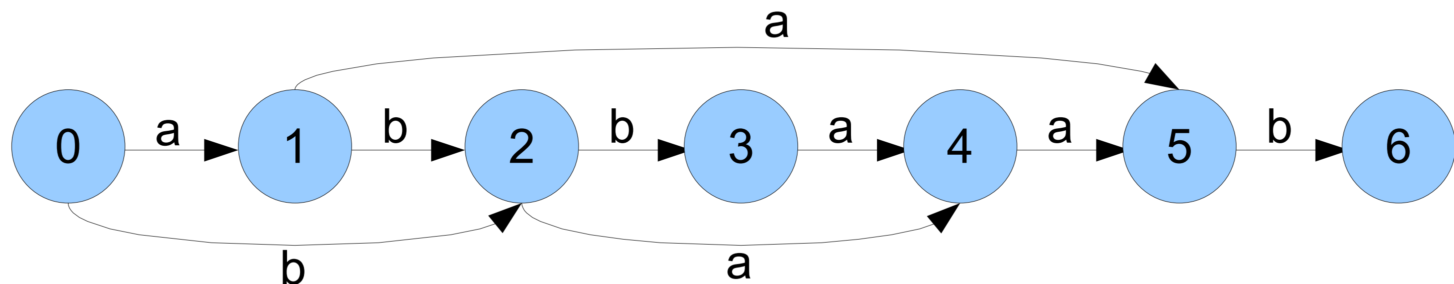
Das Teilstring-Orakel (factor oracle)

3. Für $i = 0 .. m-1$:
   Sei u ein (das!) Wort minimaler Länge, mit dem Zustand i erreicht werden kann.
   Für alle Buchstaben $a \neq P[i+1]$:
   Wenn ua ein Teilstring von $P[i-|u|+1 .. m]$ ist:
      Sei j die erste Endposition von ua in $P[i-|u|+1 .. m]$
      Erzeuge Transition $i \rightarrow j$, annotiert mit a

i=0: u ist leer. Betrachte b in P[1 .. m], erstes Vorkommen von b bei P[2 .. 2].
i=1: u = a. Betrachte aa in P[1 .. m], erstes Vorkommen von aa bei P[4 .. 5].
i=2: u = b. Betrachte ba in P[2 .. m], erstes Vorkommen von ba bei P[3 .. 4].
i=3: u = bb. Betrachte bbb in P[2 .. m], kommt nicht vor.
i=4: u = ba. Betrachte bab in P[3 .. m], kommt nicht vor.
i=5: u = aa. Betrachte aaa in P[4 .. m], kommt nicht vor.

Der Automat erkennt aba (Zustände 0,1,2,4), obwohl es kein Teilstring ist.

Beispiel: abbaab
Schritt 3.



Effiziente On-Line Konstruktion des Teilstring-Orakels

Am Ende von Iteration i ist $S(i)$ der Zustand, der das längste Suffix von $P[1 .. i]$, das auch Teilstring von $P[1 .. i\text{-}1]$ ist, erkennt.

Erzeuge Zustand 0
Setze S(0) := undef
Für $i = 1 .. m$:
    Erzeuge neuen Zustand i
    Erzeuge Transition $i - 1 \rightarrow i$, annotiert mit $P[i]$ (linearer Teil)
    Sei k := S(i-1)
    Solange $k \neq$ undef und es keine $P[i]$-Transition von k aus gibt:
        Erzeuge $P[i]$-Transition $k \rightarrow i$
        Setze k := S(k)
    Wenn k = undef, dann: $S(i)$:= 0; sonst: $S(i)$:= Ziel der $P[i]$-Transition von k aus

i=1: Sofort k=undef, damit $S(1)$=0

i=2: k=S(1)=0, eine b-Transition $0 \rightarrow 2$ wird erzeugt, dann k=undef, S(2)=0

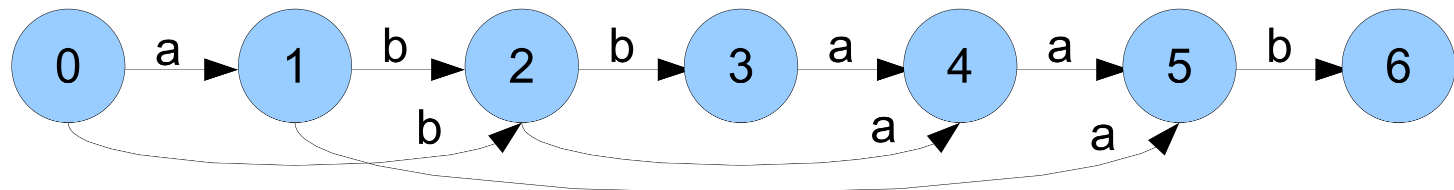
i=3: k=S(2)=0, es gibt eine b-Transition $0 \rightarrow 2$, S(3)=2

i=4: k=S(3)=2, eine a-Transition $2 \rightarrow 4$ wird erzeugt, k=0, S(4)=1

i=5: k=S(4)=1, eine a-Transition $1 \rightarrow 5$ wird erzeugt, k=0, S(5)=1

i=6: k=S(5)=1, es gibt eine b-Transition $1 \rightarrow 2$, S(6)=2

Beispiel: abbaab



Aussagen zum Teilstring-Orakel

Das Teilstring-Orakel von *P* ist ein azyklischer Automat mit folgenden Eigenschaften:
- Es erkennt mindestens alle Teilstrings (Faktoren) von *P.*
- Es erkennt genau einen String der Länge *m*, nämlich *P.*
- Es hat möglichst wenig Zustände, nämlich $m+1$.
- Es hat höchstens eine lineare Anzahl an Übergängen (zwischen *m* und $2m-1$).
- Es kann on-line in linearer Zeit konstruiert werden, mit Hilfe der Funktion
  $S(i)$:= Zustand, der das längste Suffix von $P[1..i]$,
  das auch Teilstring von $P[1..i-1]$ ist, erkennt.
- Wenn das erste Vorkommen eines Teilstrings *u* von *P* an Position *i* endet, wird *u* in einem Zustand $j \leq i$ erkannt.
- Sei $M(i)$ das kürzeste Wort, das in Zustand *i* erkannt wird.
  Dann ist $M(i)$ ein Teilstring von *P* und *i* die Endposition seines ersten Vorkommens.

Suche mit dem Teilstring-Orakel

- Vermutung (empirisch belegt): Average-case Laufzeit ist optimal $O(n \log_{|A|} m / m)$
- Worst-case Laufzeit ist $O(mn)$.
- Vor allem effizient für lange Patterns, kleine bis mittlere Alphabetgröße.

Textsuche mit dem Teilstring-Orakel - BOM

Algorithmus BOM (Backward Oracle Matching)

Erzeuge das Teilstring-Orakel von P^{rev}; sei δ die Übergangsfunktion des Orakels.

```
pos := 0  // pos verweist auf die Position vor dem Start des Fensters
while (pos <= n-m):
    j := m
    state := 0
    while (j>0 and state != undef):
        state := δ(state,T[pos + j])
        j := j – 1
    if (state != undef): report an occurrence of P at position pos+1
    pos := pos + j + 1
```

Komplexität

- Vermutung (empirisch belegt): Average-case Laufzeit ist optimal $O(n \log_{|A|} m / m)$
- Worst-case Laufzeit ist $O(mn)$.
- Vor allem effizient für lange Patterns, kleine bis mittlere Alphabetgröße.

Arithmetische Verfahren

Bisher: Vergleichsbasierte Verfahren, kaum Voraussetzungen über das Alphabet.
Grundlegene Operation: Vergleich von zwei Zeichen (z.B. $P[i]$ gegen $T[j]$)

Voraussetzung jetzt: Alphabet ist Teilmenge von $\{0, 1, ..., K-1\}$.
Erlaubt Codierung von Strings als Zahlen.
Annahme ist realistisch:

- ASCII-Code
- Unicode
- Explizite Abbildung des Alphabets auf $\{0, 1, ..., K-1\}$,
  z.B. DNA $\{A, C, G, T\} \rightarrow \{0, 1, 2, 3\}$

q-gram Codierung

Definition

Ein q-gram ist ein String der Länge q, so dass $|A|^q$ <= #Integers

Ein q-gram über $A = \{0, .., K\text{-}1\}$ kann somit als K-äre Zahl zwischen 0 und K^q-1 codiert werden, wobei das Zeichen an i-ter Stelle (i=1..q) die Wertigkeit K^{q-i} bekommt.

Beispiel (DNA)

Das DNA 4-gram CTGA unter der Annahme A=0, C=1, G=2, T=3 also (1,3,2,0), hätte den Code 1*64 + 3*16 + 2*4 + 0*1 = 118.
Dem Code 120 entspricht (bei einer vorausgesetzten Länge von q=4) CTGG.
Bei einer Länge von q=6 gehört 120 hingegen zu AACTGG.

Satz

Die Zuordnung $\text{code}_q: A^q \to \{0, 1, ..., K^q-1\}$ durch

$$\text{code}_q(a_1,a_2,...,a_q) := \sum_{i=1}^{q} a_i \, K^{q-i}$$

ist bijektiv. [Beweis: elementar, Division mit Rest.]

Bemerkung

Der Name q-gram kommt von griechisch gramma = Buchstabe.

q-gram Matching

Anwendung – als präfixbasierte Methode für ein Pattern der Länge q
Annahme: Der Text T verfügt an Stelle n+1 über ein sentinel-Zeichen (egal welches)

1. Berechne $p := \text{code}_q(P)$
2. Berechne $t := \text{code}_q(T[1..q])$
3. Setze $i := q$
4. Wiederhole

   Wenn $t=p$, melde ein Vorkommen von P beginnend an Position $i-q+1$

   $i := i + 1$

   $t := (t - K^{q-1} * T[i-q]) * K + T[i]$ (Update von t in O(1) Zeit)

   Bis i>n

Beachtenswert:

- Der Code für das nächste Textfenster kann in konstanter Zeit durch eine Subtraktion, eine Multiplikation, und eine Addition aus dem aktuellen Code berechnet werden.
- Zur Berechnung von $K^{q-1} * T[i-q]$ kann man auch eine Hilfs-Tabelle vorberechnen.
- Vertauscht man die Wertigkeiten (linkes Zeichen Wertigkeit K^0, rechtes Zeichen K^{q-1}), erhält man das Update $t := t / K + T[i] * K^{q-1}$.
- Ist K eine Zweierpotenz (z.B. bei DNA), verwendet man statt Multiplikation / Division eher Bit-shift, statt Addition bitweises oder.

Rabin-Karp-Ideen

Probleme bei der q-gram Codierung:
basiert darauf, dass jedes Wort (q-gram) einen eindeutigen Code bekommt;
funktioniert nur für kleine Alphabete und kleine q.

Idee 1: Führe die Berechnungen modulo einer Zahl R durch,
so dass $K*R$ noch in einen Integer passt.

Berechnung von $p := \text{code}_q(P)$ in $m=q$ Schritten:

$p := 0$
for $i := 1 .. m$:
    $p := (K*p + P[i]) \bmod R$

Wichtig hierbei ist, dass das Zwischenergebnis vor 'mod' in einen Integer passt.

Update des Codes des Textfensters beim Weiterschieben:
$t := ((t - T[i-m]*H) * K + T[i]) \bmod R$

Neues Problem:

Der Fall $p \neq t$ bedeutet nach wie vor, dass das Pattern an dieser Stelle nicht vorkommt.
Aber $p = t$ bedeutet nicht mehr, dass ein Vorkommen gefunden wurde!
Dies muss jetzt für das aktuelle Fenster explizit verifiziert werden (kostet $O(m)$ Zeit).

Rabin-Karp-Ideen

Idee 2: Die Zahlentheorie legt nahe, dass Primzahlen als Moduli R die Eigenschaft haben, Texte aus A^* „zufällig“ auf $\{0,1,...,R-1\}$ abzubilden.

Laufzeitanalyse damit:
Sei v die Anzahl der tatsächlichen Vorkommen ($v=O(1)$).
Es wird $O(m+n)$ Zeit für arithmetische Operationen
und $O(m(v+n/R))$ erwartete Zeit für Verifikation benötigt.
Wählt man $R > m$, ist die erwartete Laufzeit $O((m+n) + mv)$.
Die worst-case Laufzeit ist $O(mn)$.

Problem dabei:
Die erwartete Laufzeit gilt gemittelt über alle Texte und Patterns,
aber nicht für alle Kombinationen von Texten und Patterns!
Für jede Wahl von R gibt es einen Text, für den die worst-case Laufzeit $O(mn)$ eintritt.

Rabin-Karp-Ideen

Idee 3: Wähle die Primzahl *R* zufällig.

Laufzeitanalyse: Sei prim(x) die Anzahl der Primzahlen <= x.
Satz (Primzahlsatz): $x/\ln(x) \le \text{prim}(x) \le 1.26 * x/\ln(x)$ [Zahlentheorie].

Zentraler Satz der Analyse
Sei A ein binäres Alphabet, sei $mn \ge 29$.
Sei J eine positive ganze Zahl, R eine zufällig gewählte Primzahl <= J.
Dann ist die Wahrscheinlichkeit, dass jemals der Fall $p=t$ eintritt, obwohl kein Match vorliegt, <= prim(nm)/prim(J).

Anwendung
Wähle $J := nm^2$; damit ist für jeden Text und jedes Pattern die Wahrscheinlichkeit eines falschen Matches beschränkt durch

$$\frac{\text{prim}(nm)}{\text{prim}(nm^2)} \le 1.26 \frac{nm}{nm^2} \frac{\ln(nm^2)}{\ln(nm)} \le 1.26 \frac{1}{m} \cdot 2 = \mathrm{O}(1/m).$$

Zur Wahl einer zufälligen Primzahl existieren effiziente randomisierte Algorithmen.
Wenn einmal ein falscher Match auftritt, wähle eine neue zufällige Primzahl.
Damit: Erwartete Laufzeit O(m+n).

Rabin-Karp-Ideen

Idee 4: Verringerung der Fehlerwahrscheinlichkeit durch k Primzahlen simultan.

Stärkere Version des Zentralen Satzes:
Sei A ein binäres Alphabet, sei $mn >= 29$, J eine positive ganze Zahl.
Bei Verwendung von k unabhängigen Primzahlen $(R_j) <= J$ ist die Wahrscheinlichkeit eines falschen Matches $<= n*[\text{prim}(m)/\text{prim}(J)]^k$.

Beweis:
Es gebe eine Position im Text mit einem falschen Match ($p=t$ bzgl. aller Primzahlen).
Jede der k Primzahlen muss dann die Differenz der (ursprünglichen) Codes teilen:
R_j teilt $y := |\ \text{code}(P) - \text{code}(T_{pos})\ |$ für alle $j=1..k$.
Nun ist $y <= 2^m$ (binäres Alphabet!).

Man benutzt einen Satz aus der Zahlentheorie,
nach dem dann y höchstens prim(m) verschiedene Primfaktoren hat.

Damit ist R_j eine von prim(m) aus prim(J) Primzahlen;
die Wahrscheinlichkeit, dass alle R_j es sind, ist damit $[\text{prim}(m)/\text{prim}(J)]^k$.
Es gibt n mögliche Positionen im Text.