fhwedel
UNIVERSITY OF APPLIED SCIENCES

# The Boyer-Moore algorithm

Funktionale Programmierung, WS2010

Tobias Lüders

Pascal Rhode

# Inhaltliche Übersicht

1. Einführung
2. Boyer-Moore
   - 2.1 Lösungsansatz
   - 2.2 Basisalgorithmus
   - 2.3 Optimierung

# Einführung

- String matching: Finden eines nichtleeren Strings (pattern) in einem anderen String (text)
- Algorithmus:
  - pattern wird linksbündig unter text geschrieben
  - pattern wird dann von rechts nach links mit text verglichen
  - Tritt mismatch auf, berechnen zwei Heuristiken, wie weit das pattern verschoben wird
    - Bad-Charakter-Heuristik
    - Good-Suffix-Heuristik

# Einführung

- Bad-Character Strategie:

```
Text :    a b b a b a b a c b a
          b a b a c
              b a b a c
```

- Good-Suffix Strategie:

```
Text :    a b a a b a b a c b a
          c a b a b
              c a b a b
```

# Einführung

```
matches    :: Eq a => [a] -> [a] -> [Int]
machtes ws = map length . filter (endswith sw) . inits
```

- Beispiel:
  - matches "abcab" "ababcabcab"
  - Inits "ababcabcab"
    - ["","a","ab","aba","abab","ababc","ababca", "ababcab","ababcabc","ababcabca","ababcabcab"]
  - filter(endswith "abcab").inits "ababcabcab"
    - ["ababcab", "ababcabcab"]
  - map length.filter(endswith "abcab").inits "ababcabcab"
    - [7,10]
- Problem: Laufzeit von matches ws xs: O(mn)

# Boyer-Moore
## Lösungsansatz

- Anwendung von Gesetzen zur Steigerung der Effizienz durch Ausdrucksumformungen:
  - `map(foldl op e) . inits = scanl op e`
  - `map f . filter p = map fst . filter snd . map(fork(f,p))`
  - `fork(foldl op1 e1,foldl op2 e2) = foldl op (e1,e2)`
- `matches ws = map fst.filter snd.scanl step(0,e)`

# Boyer-Moore
## Lösungsansatz

- Definition von endswith:
  - `endswith ws xs = reverse ws # reverse xs`
- `#` : „ist Prefix von“

```
[] # vs          = True
(u:us) # []      = False
(u:us) # (v:vs) = (u == v && us # vs)
```

- Pattern-Vergleich von hinten nach vorne
- ws ist genau dann Suffix von xs, wenn reverse von ws ein Prefix von reverse xs ist

# Boyer-Moore
## Basisalgorithmus

- Anpassung von endswith durch Komposition:
  - endswith ws = (reverse ws #).reverse

```
matches ws =
map fst.filter((sw #).snd) . map(fork(length,reverse)).inits
where sw = reverse ws
```

# Boyer-Moore
## Basisalgorithmus

- Erneute Anwendung des Tupling Law von foldl, gefolgt vom scan lemma führt zu:
  - matches ws =

    map fst . filter((sw #) . snd) . scanl step(0,[])
    **where** sw = reverse ws
  - step(n, sx) x = (n + 1, x : sx)

# Boyer-Moore
## Basisalgorithmus

- Beispiel:
  - matches “abcab“ “ababcabcab“
  - scanl step(0,[])
    - [(0,""),(1,"a"),(2,"ba"),(3,"aba"),(4,"baba"),
      (5,"cbaba"),(6,"acbaba"),(7,"bacbaba"),(8,"cbacbaba"),
      (9,"acbacbaba"),(10,"bacbacbaba")]
  - filter((sw #) . snd) . scanl step(0,[])
    - [(7,"bacbaba"),(10,"bacbacbaba")]
  - map fst . filter((sw #) . snd) . scanl step(0,[])
    - [7,10]

# Boyer-Moore
## Basisalgorithmus

- Jedes Fenster enthält die umgedrehten initialen Segmente des Textes
- Jedes folgende Fenster unterscheidet sich nur in einer Position vom vorherigen („Shift“ der Länge eins zwischen Fenstern)
- Problem: Algorithmus benötigt im schlechtesten Fall immer noch O(mn), denn (sw #) kann in dem Fall m Schritte benötigen

# Boyer-Moore
## Optimierung

- Fenster, die keine Kandidaten für das matching sein können, werden übersprungen(shiften)
- Shift hängt davon ab, wieviel Übereinstimmung im aktuellen Fenster vorliegt
- Länge des längsten gemeinsamen Prefix bestimmen:

```
llcp xs []              = 0
llcp [] ys              = 0
llcp (x:xs) (y:ys)      = if x==y then 1+llcp xs ys else 0
```

# Boyer-Moore
## Optimierung

- Wenn m = llcp sw sx, dann gilt sw # sx
- Ausgehend von gegebenem i = llcp sw sx für das aktuelle Fenster (n,sx), muss eine untere Grenze an der Position n + k des nächsten Fensters gefunden werden, wo ein matching auftreten könnte
- Damit keine matches verpasst werden, muss gelten: 0 < k <= m

# Boyer-Moore
## Optimierung

- Das gesuchte Fenster hat dann die Form:
  (n + k, ys ++ sx), wobei k = length ys
- Falls sw # ys ++ sx, dann take k sw = ys und drop k sw # sx
  - llcp sw (drop k sw) = min i (m - k)
  - Die nächsten k-1 Fenster können übersprungen werden, ohne ein match zu verpassen

# Boyer-Moore
## Optimierung

```
matches ws                      = test m . scanl step(0,[])
where
test j []                       = []
test j ((n,sx) : nxs)           | i == m           = n : test k (drop (k–1) nxs)
                                | m-k <= i         = test k (drop (k–1) nxs)
                                | otherwise        = test m (drop (k-1) nxs)
                                  where   i'       = llcp sw (take j sx)
                                          i        = if i' == j then m else i'
                                          k        = shift sw i
                                                   where
                                                   shift sw i =
                                                   head [ k | k <- [ 1 .. m ],
                                                   llcp sw (drop k sw) == min i (m - k)]
(sw,m)    = (reverse ws, length ws)
```

# Boyer-Moore
## Optimierung

- Problem : shifts sw = map (shift sw)[0..m] führt zu einer kubischen Laufzeit des Algorithmus
  - Die Berechnung von shift sw i kann quadratisch sein
  - Es existieren m + 1 Werte für i
- Idee : Berechnung von shifts sw in linearer Zeit und speichern der Resultate in einem array a und Ersetzung von shift sw i durch a!i

# Boyer-Moore
## Optimierung

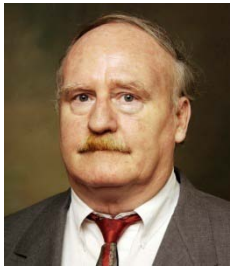
- `a   = accumArray min m (0,m) vks`
- `vks = [(llcp sw (drop k sw),k) | k <- [1..m]]`
- `a!i = minimum([k | k <- [1..m],`
  `llcp sw (drop k sw) == i] ++ [m])`
- Idee ist a zu ersetzen durch:
  - `a = accumArray min m (0, m) (vks ++ vks‘), wobei`
  - `vks‘ = zip [m, m-1 .. 1] (foldr op [] vks)`
  - `op (v,k) ks = if v + k == m then k : ks else head ks : ks`

- `allcp xs = [llcp xs (drop k xs) | k <- [0..length xs - 1]]`
- `allcp‘ xs = tail (allcp xs) ++ [0]`

# Boyer-Moore
## Optimierung

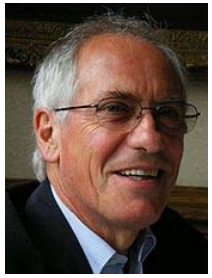
```
matches ws                                  = test m . scanl step(0,[])
where
test j []                                   = []
test j ((n,sx) : nxs) | i == m               = n : test k (drop(k-1)nxs)
                      | m-k <= i             = test k (drop(k-1)nxs)
                      | otherwise            = test m (drop(k-1)nxs)
                      where      i'          = llcp sw (take j sx)
                                 i           = if i' == j then m else i'
                                 k           = accumArray min m (0,m) (vks ++ vks')!i
                                 where       m           = length sw
                                             vks         = zip(allcp' sw) [1..m]
                                             vks‘        = zip[m,m-1..1](foldr op [] vks)
                                             op(v,k) ks = if v+k == m then k:ks else head ks:ks
(sw,m)                                      = (reverse ws, length ws)
```

# Vielen Dank für die Aufmerksamkeit


Boyer


Moore

+


Bird

=

**Fragen ?**