Fundamental Algorithms

Chapter 7: String- and Patternmatching

Christian Scheideler

WS 2017

Overview

- Basic notation
- A naive algorithm
- Rabin-Karp algorithm
- Knuth-Morris-Pratt algorithm
- Boyer-Moore algorithm
- Aho-Corasick algorithm
- Suffix trees

Basic Notation

- Alphabet Σ: finite set of symbols
  $|\Sigma|$: cardinality of Σ
- String s: finite sequence of symbols over Σ
  $|s|$: length of s
- ε: empty string, i.e., $|\varepsilon|=0$
- Σ^n: set of all strings over Σ of length n
  $\Sigma^0=\{\varepsilon\}$
- $\Sigma^*=\bigcup_{i\geq 0} \Sigma^i$: set of all strings over Σ
- $\Sigma^+=\bigcup_{i\geq 1} \Sigma^i$: set of all strings over Σ except ε

Basic Notation

Definition 7.1: Let $s=s_1\ldots s_n$ and $s'=s'_1\ldots s'_m$ be strings over Σ.

- s' is called a substring of s if there is an $i\geq 1$ with $s'=s_i s_{i+1}\ldots s_{i+m-1}$
- s' is called a prefix of s if $s'=s_1 s_2\ldots s_m$
- s' is called a suffix of s if $s'=s_{n-m+1} s_{n-m+2}\ldots s_n$

There are two variants for the exact string matching problem. Given two strings s (the search string) and t (the text),

1. Determine if s is a substring of t, or
2. Determine all positions at which s is a substring of t

Basic Notation

Sample problem: find avoctdfytvv in

kvjlixapejrbxeenpphkhthbkwyrwamnugzhppfxiyjyanhapfwbghx
mshrlyujfjhrsovkvveylnbxnawavgizyvmfohigeabgksfnbkmffxjdf
ffqbualeytqrphyrbjqdjqavctgxjifqgfgydhoiwhrvwqbxgrixydzdfss
bpajnhopvlamhhfavoctdfytvvggikngkwzixgjtlxkozjlefilbrboiegwf
gnbzsudssvqymnapbpqvlubdoyxkkwhcoudvtkmikansgsutdjyth
apawlvliygjkmxorzeoafeoffbfxuhkzukeftnrfmocylculksedgrdsfe
lvayjpgkrtedehwhrvvbbltdkctq

In general, |t|>>|s| (Google web search)

Basic Notation

Many applications:

- word processors
- virus scanning
- text information retrieval
- digital libraries
- computational biology
- web search engines

A naive Algorithm

Input: text t, search string s ($|t|=n$, $|s|=m$)

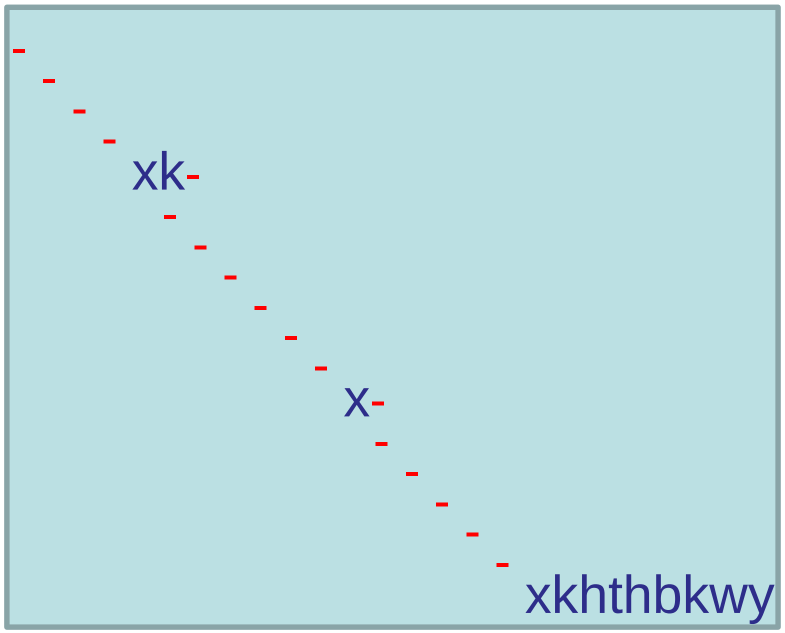
Algorithm SimpleSearch:

```
for i:=1 to n-m+1 do
    j:=1
    while j≤m and s[j]=t[i+j-1] do
        j:=j+1
    if j>m then output i
```

A naive Algorithm

Search string s: xkhthbkwy

Text t: kvavixkpejrbxeenppxkhthbkwy



Is SimpleSearch always good?

Number of compared characters: n+3

A naive Algorithm

Search string s: 000000001

Text t: 00000000000000000000000001

```
00000000-
 00000000-
  00000000-
   00000000-
    00000000-
     00000000-
      ...
```

In the worst case, SimpleSearch has a bad runtime!

Number of compared characters: $n \cdot m$

Karp-Rabin Algorithm

- Σ: alphabet of size $q-1$
- U: set of all q-ary numbers
- $f:\Sigma^*\rightarrow U$ arithmetization of strings over $\Sigma=\{c_1,\ldots,c_{q-1}\}$ with the property that
  - $f(\varepsilon) = 0$
  - $f(c_i) = i$ for all $i\in\{1,\ldots,q-1\}$
  - $f(s) = \Sigma_{i=0}^{n-1} f(s_i)\cdot q^i$ for all strings $s=s_0\ldots s_{n-1}$

For every $x\in U$ there is at most one string s with $f(s)=x$, so f is injective.

Karp-Rabin Algorithm

Idea: use hashing

x: arithmetization of some string

Example:

- use hash function h(x) = x mod 97
- search for 59265 in 31415926535897932384626433

- hash value of search string: h(59265) = 95
- Text hashes:

  31415926535897932384626433

  31415 = 84 (mod 97)
  14159 = 94 (mod 97)
  41592 = 76 (mod 97)
  15926 = 18 (mod 97)
  59265 = 95 (mod 97) $\rightarrow$ match!

Problem: hash uses m characters, so still running time $n \cdot m$!

Karp-Rabin Algorithm

Additional idea: use hash of previous position to compute new hash

$$14159 = (31415 - 30000) \cdot 10 + 9$$

$$14159 \bmod 97 = (31415 \bmod 97 - 30000 \bmod 97) \cdot 10 + 9 \pmod{97}$$

$$= (84 - 3 \cdot 9) \cdot 10 + 9 \pmod{97}$$

$$= 579 \bmod 97 = 94$$

precompute $9 = 10000 \pmod{97}$

Example:

- hash value of search string: 59265 mod 97 = 95
- Text hashes:

  31415926535897932384626433

  31415 mod 97 = 84

  14159 mod 97 = $(84 - 3 \cdot 9) \cdot 10 + 9 \pmod{97} = 94$

  41592 mod 97 = $(94 - 1 \cdot 9) \cdot 10 + 2 \pmod{97} = 76$

  15926 mod 97 = $(76 - 4 \cdot 9) \cdot 10 + 6 \pmod{97} = 18$

Karp-Rabin Algorithm

In general:

- consider a search string s of length m over some alphabet Σ of size $q-1$
- let $h(x) = x \bmod p$ for some prime $p>q$
- compare $h(f(s))$ with $h(f(t_i \ldots t_{m+i-1}))$ by computing $y_i=h(f(t_i \ldots t_{m+i-1}))$ in the following way:

$$y_1 = f(t_1 \ldots t_m) \bmod p$$
$$y_{i+1} = (y_i - f(t_i) \cdot d) \cdot q + f(t_{i+m}) \pmod p \quad \text{for all } i \geq m$$

  where $d=q^{|s|-1} \bmod p$
- whenever $y_i = h(f(s))$, output i

Problem: It can happen that $h(f(s))=h(f(t_i \ldots t_{m+i-1}))$ but $s \neq t_i \ldots t_{m+i-1}$. We call this a wrong matching.

Solution: As we will see, this is unlikely to happen if p is sufficiently large.

Karp-Rabin Algorithm

Karp-Rabin Algorithm:

$q:=|\Sigma|+1$; $m:=|s|$; $n:=|t|$; $d:=1$

$x:=0$ // for $f(s) \bmod p$

$y:=0$ // for $f(t_i \ldots t_{m+i}) \bmod p$

for $i:=1$ to $m-1$ do

  $d:=q \cdot d \bmod p$

for $i:=1$ to m do

  $x:=q \cdot x+f(s_i) \bmod p$

  $y:=q \cdot y+f(t_i) \bmod p$

for $i:=1$ to $n-m+1$ do

  if $x=y$ then

    if $s=(t_i \ldots t_{m+i-1})$ then output i

  if $i \leq n-m$ then

    $y:=(y - f(t_i) \cdot d) \cdot q + f(t_{i+m}) \bmod p$

to be on the safe side

Karp-Rabin Algorithm

Analysis of the Karp-Rabin Algorithm:

Definition 7.2: For some natural number x let $\pi(x)$ be the number of prime numbers that are at most x.

Lemma 7.3 (Prime Number Theorem): For any $x \geq 29$, $0.922 \cdot x/(\ln x) \leq \pi(x) \leq 1.105 \cdot x/(\ln x)$.

Lemma 7.4: For $x \geq 29$, the product of all prime numbers that are at most x is larger than 2^x.

Karp-Rabin Algorithm

Corollary 7.5: If $x \geq 29$ and $y \leq 2^x$, then y has less than $\pi(x)$ different prime divisors.

Proof:

- Suppose that y has $k \geq \pi(x)$ many different prime divisors $q_1,\ldots,q_k$. Then
$$2^x \geq y \geq q_1 \cdot q_2 \cdot \ldots \cdot q_k.$$
- But $q_1 \cdot q_2 \cdot \ldots \cdot q_k$ is at least as large as the product of the first k primes, which is at least as large as the product of the first $\pi(x)$ primes.
- Hence, Lemma 7.4 leads to a contradiction.

Karp-Rabin Algorithm

Lemma 7.6: Let s and t be strings over an alphabet of size $q-1$ with $m \cdot \log q \geq 29$, where $|s|=m$ and $|\Sigma|=q-1$. Let P be an natural number. If p is a random prime number $\leq P$, then the probability of a wrong matching of the hashes of s and $t_i \ldots t_{m+i-1}$ for some fixed i is at most $\pi(m \cdot \log q)/\pi(P)$.

Proof:

- Consider some fixed i with $f(s) \neq f(t_i \ldots t_{m+i-1})$.
- Certainly, $|f(s)-f(t_i \ldots t_{m+i-1})| \leq q^m = 2^{m \cdot \log q}$.
- Hence, Corollary 7.5 implies that $|f(s)-f(t_i \ldots t_{m+i-1})|$ can have at most $\pi(m \cdot \log q)$ prime divisors.

Karp-Rabin Algorithm

Proof (continued):

- Since $f(s) \bmod p = f(t_i \ldots t_{m+i-1}) \bmod p$, p divides $|f(s)-f(t_i \ldots t_{m+i-1})|$.
- Hence, p is a prime divisor of this product.
- If p admits a wrong matching, then p must be one of at most $\pi(m \cdot \log q)$ many prime divisors.
- Since p is randomly chosen out of $\pi(P)$, the probabiliy that p admits a wrong matching is at most $\pi(m \cdot \log q)/\pi(P)$.

Karp-Rabin Algorithm

Theorem 7.7: Let s and t be strings with $m \cdot \log q \geq 29$ and let $P = m^2 \cdot \log q$, where $|t| = n$, $|s| = m$, and $|\Sigma| = q-1$. If s is contained k times in t, then the expected runtime of Karp-Rabin is $O(n+k \cdot m)$.

Proof:

- R: set of positions in t at which s does not start.
- For each position $i \in R$ we define a binary random variable X_i to be 1 if and only if there is a wrong matching at position i.
- Let $N = m \cdot \log q$. From Lemma 7.3 and Lemma 7.6 we know that

$$E[X_i] \leq \frac{\pi(N)}{\pi(P)} \leq \frac{1.105\ N/\ln(N)}{0.922\ N \cdot m/\ln(N \cdot m)} \leq \frac{1.2}{m} \frac{\ln(N \cdot m)}{\ln(N)} \leq \frac{2}{m}$$

- Let $X = \Sigma_{i \in R} X_i$. Due to the linearity of expectation,

$$E[X] = \Sigma_{i \in R} E[X_i] \leq 2|R|/m$$

- Since a wrong matching consumes $O(m)$ time and otherwise we just need time $O(1)$ for a position $i \in R$, the expected total runtime is $O(n)$ for R.
- For the k positions of t that contain s, a total runtime of $O(k \cdot m)$ is needed.
- Combining the runtimes results in the theorem.

Knuth-Morris-Pratt Algorithm

Observation: on mismatch at the i-th symbol in the search string, we know the previous i-1 symbols in the text.

Idea: precompute what to do on a mismatch

Example:

- search string s: ababcab
- text: ababa….

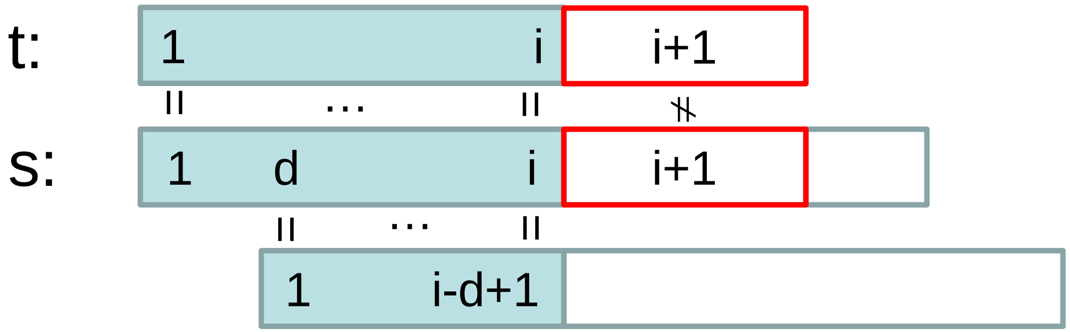
ababcab

ababcab (shift s by two for next possible match and continue scanning at current position a in the text)

Knuth-Morris-Pratt Algorithm

In general:

- Suppose that $(s_1 \ldots s_i)=(t_1 \ldots t_i)$ but $s_{i+1} \neq t_{i+1}$.
- Then move to the first position d in t so that $(s_1 \ldots s_{i-d+1}) = (t_d \ldots t_i)$ and continue with scanning the text at t_{i+1}.
- In this case, it certainly holds that $(s_1 \ldots s_{i-d+1}) = (s_d \ldots s_i)$.
- We want to determine these jumps for all i in a preprocessing.



Knuth-Morris-Pratt Algorithm

In general:

- Suppose that $(s_1 \dots s_i) = (t_1 \dots t_i)$ but $s_{i+1} \neq t_{i+1}$.
- Then move to the first position d in t so that $(s_1 \dots s_{i-d+1}) = (t_d \dots t_i)$ and continue with scanning the text at t_{i+1}.
- In this case, it certainly holds that $(s_1 \dots s_{i-d+1}) = (s_d \dots s_i)$.
- We want to determine these jumps for all i in a preprocessing.

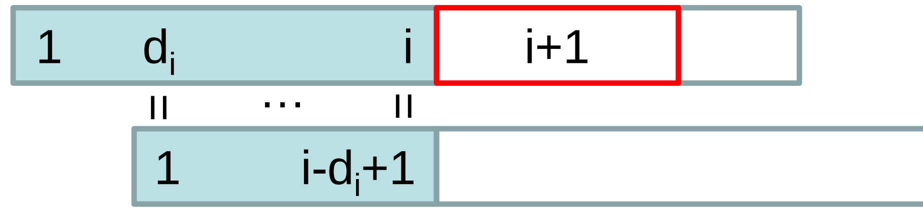
Goal of the preprocessing:

- For every position i in s, find the minimal $d>1$ so that $(s_1 \dots s_{i-d+1}) = (s_d \dots s_i)$. If there is no such d, we set it to $i+1$.
- Let the resulting d for that i be denoted d_i.
- The d_i´s will be stored in an array so that they are quickly accessible to the KMP algorithm.

Knuth-Morris-Pratt Algorithm

Preprocessing:
For each i, find minimial d_i so that

s:



Lemma 7.8: For every $i \in \{1,\ldots,m-1\}$, $d_i \le d_{i+1}$.

Proof:

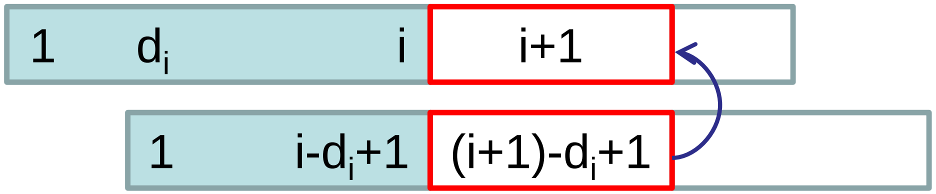
- Consider an arbitrary i.
- There is no $1<d<d_i$ with $(s_d \ldots s_i) = (s_1 \ldots s_{i-d+1})$.
- Hence, there cannot be a $1<d<d_i$ with $(s_d \ldots s_{i+1}) = (s_1 \ldots s_{i-d+2})$, which implies that $d_i \le d_{i+1}$.

But how can we compute exact values of d_i?

Knuth-Morris-Pratt Algorithm

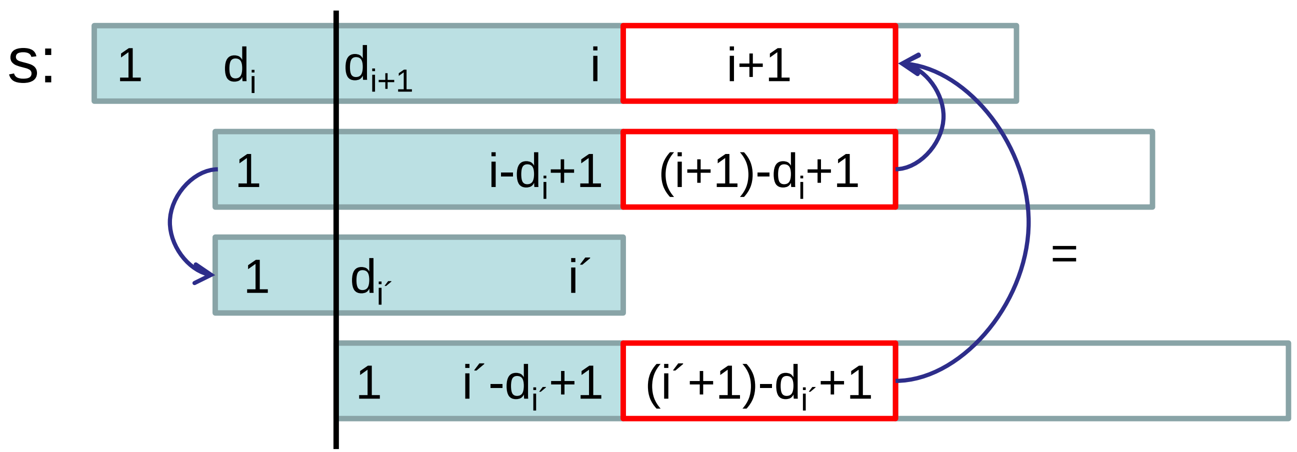
- Suppose that we have already computed $d_1, \dots, d_i$ and we want to compute d_{i+1}. The first candidate according to Lemma 7.8 would be d_i. For d_i it holds that $(s_1 \dots s_{i-d_i+1}) = (s_{d_i} \dots s_i)$. If also $s_{(i+1)-d_i+1} = s_{i+1}$, then $(s_1 \dots s_{(i+1)-d_i+1}) = (s_{d_i} \dots s_{i+1})$ and we can set $d_{i+1} = d_i$.

s:



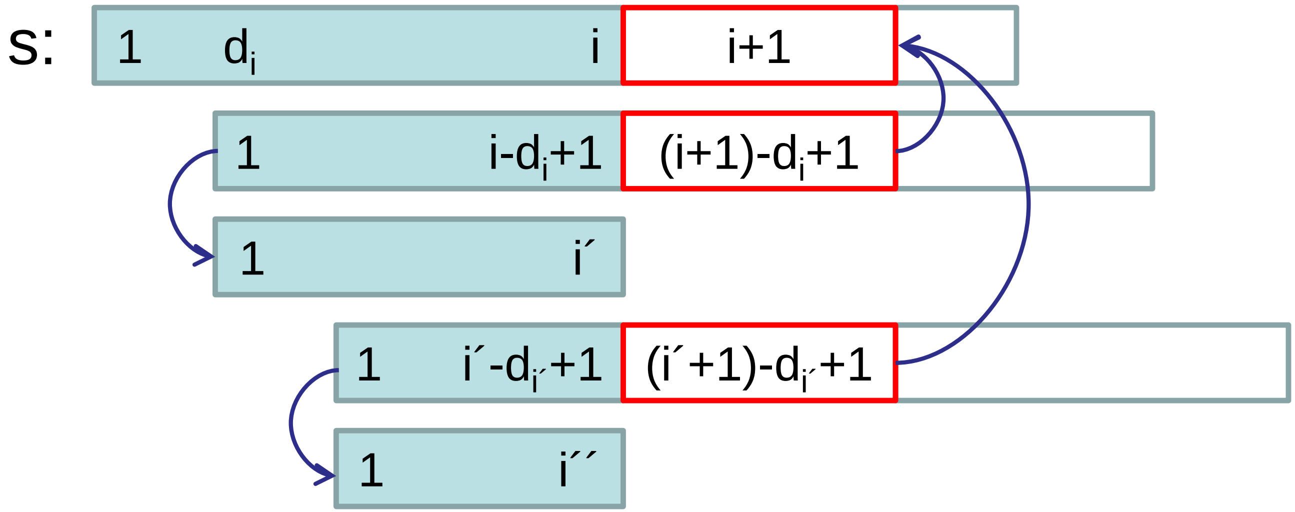
Knuth-Morris-Pratt Algorithm

- If $s_{(i+1)-d_i+1} \neq s_{i+1}$, then we have not yet found a matching for s_{i+1}. Let $i'=i-d_i+1$. Then we have to find for $(s_1\ldots s_{i'})$ the first d with $(s_1\ldots s_{i'-d+1}) = (s_d\ldots s_{i'})$. The first candidate for that is $d_{i'}$ since $(s_1\ldots s_{i'-d_{i'}+1}) = (s_{d_{i'}}\ldots s_{i'})$. If also $s_{(i'+1)-d_{i'}+1}=s_{i+1}$, then we can set $d_{i+1}=d_i+(d_{i'}-1)$.



Knuth-Morris-Pratt Algorithm

- If $s_{(i'+1)-d_{i'}+1} \neq s_{i+1}$, then we set $i'' = i' - d_{i'} + 1$ and we continue our search as for i'.



Knuth-Morris-Pratt Algorithm

From these rules we can construct an efficient algorithm for computing the d_i-values:

Algorithm KMP-Preprocessing:

d_0:=2; d_1:=2 // movement of s by 1
δ:=d_1 // δ: current candidate of d_i
for i:=2 to m do
while $\delta \leq i$ and $s_i \neq s_{i-\delta+1}$ do
// $(s_1 \ldots s_{i-\delta})=(s_\delta \ldots s_{i-1})$ but $s_{i-\delta+1} \neq s_i$
$\delta:=\delta+(d_{i-\delta}-1)$
$d_i:=\delta$

Example: s=ababaca

i	0	1	2	3	4	5	6	7
d_i	2	2	3	3	3	3	7	7

Knuth-Morris-Pratt Algorithm

Algorithm KMP-Preprocessing:

```
d_0:=2; d_1:=2  // movement of s by 1
δ:=d_1          // δ: current candidate of d_i
for i:=2 to m do
   while δ≤i and s_i≠s_{i-δ+1} do
      // (s_1…s_{i-δ})=(s_δ…s_{i-1}) but s_{i-δ+1}≠s_i
      δ:=δ+(d_{i-δ} -1)
   d_i:=δ
```

Theorem 7.9: The runtime of the KMP-Preprocessing is $O(m)$.

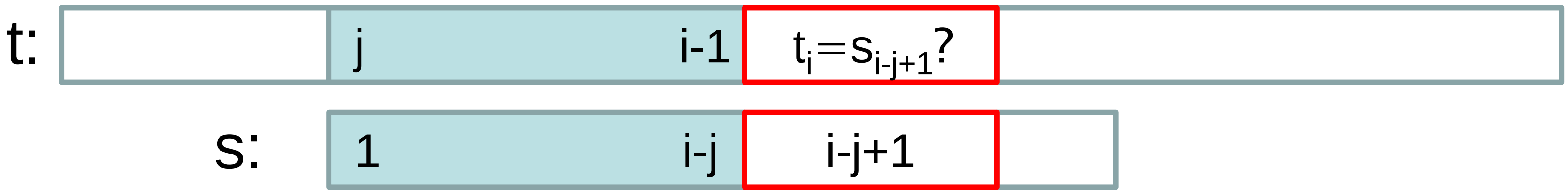
Proof:

- Since all $d_i \geq 2$, δ will be increased in each while loop.
- Since the condition of the while loop cannot be satisfied again once $\delta > m$, the while loop is executed at most m times over all iterations of the for-loop.
- The for-loop is executed at most m times as well.

Knuth-Morris-Pratt Algorithm

Algorithm KMP:

```
execute KMP-Preprocessing
i:=1  // current position in t
j:=1  // current starting position of s in t
while i≤n do
  if j≤i and t_i≠s_{i-j+1} then
    j:=j+d_{i-j}-1
  else
    if i-j+1=m then  // match found
      output j
      j:=j+d_m -1
    i:=i+1
```



Knuth-Morris-Pratt Algorithm

Theorem 7.10: The runtime of the KMP algorithm is $O(n)$.

Proof:

- In each while-loop, i or j is increased.
- Since i and j are bounded above by n, the theorem follows.

Can we be faster than linear time?

Knuth-Morris-Pratt Algorithm

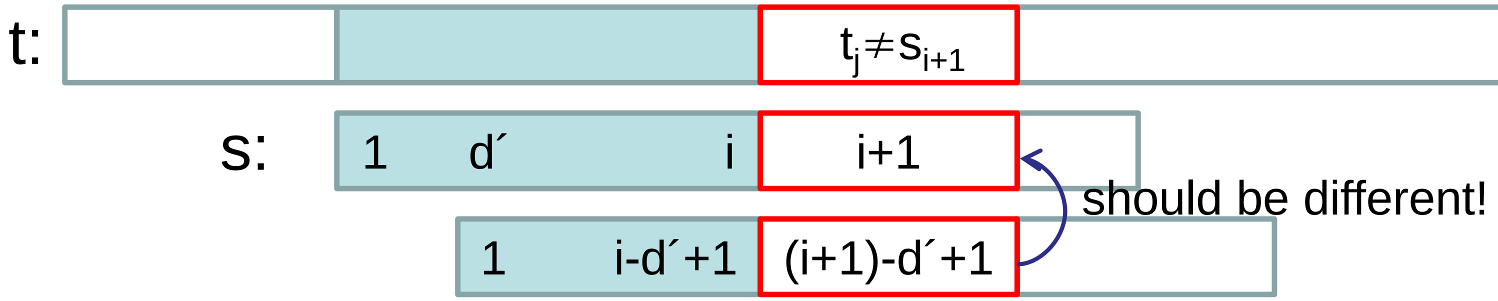
Further improvement of KMP-Preprocessing:

Original goal of the preprocessing:

- For every position i in s, find the minimal $d>1$ so that $(s_1 \ldots s_{i-d+1}) = (s_d \ldots s_i)$. If there is no such d, we set it to $i+1$.

Improved goal of the preprocessing:

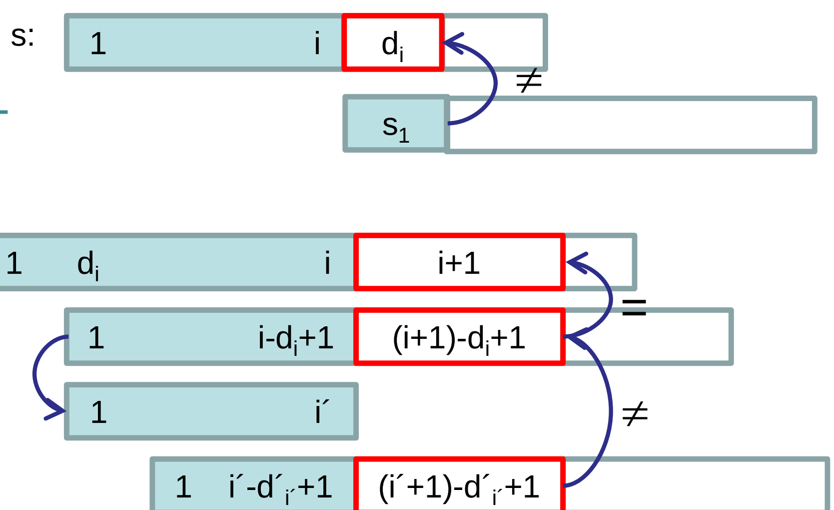
- For every position i in s, find the minimal $d'>1$ so that $(s_1 \ldots s_{i-d'+1}) = (s_{d'} \ldots s_i)$ and $s_{i-d'+2} \neq s_{i+1}$. If there is no such d', we set it to $i+2$.



Knuth-Morris-Pratt Algorithm

Algorithm KMP-Preprocessing2:

d_0:=2; d_1:=2 // movement of s by 1
δ:=d_1 // current shifting position of s
for i:=2 to m do
  while $\delta \leq i$ and $s_i \neq s_{i-\delta+1}$ do
    // $(s_1 \ldots s_{i-\delta})=(s_\delta \ldots s_{i-1})$ but $s_{i-\delta+1} \neq s_i$
    δ:=$\delta + d_{i-\delta}$ -1
  d_i:=δ
// computation of d´-values
$d´_0$:=2
for i:=1 to m-1 do
  if d_i>i then // no matching parts
    if $s_1 \neq s_{i+1}$ then $d´_i$:=d_i else $d´_i$:=d_i+1
  else
    if d_{i+1}>d_i then // mismatch at i+1
      $d´_i$:=d_i
    else
      i´:=i - d_i + 1
      $d´_i$:=d_i + $d´_{i´}$ - 1
$d´_m$:=d_m // all symbols are matching



Knuth-Morris-Pratt Algorithm

Example: s=ababaca

KMP-Preprocessing:

i	0	1	2	3	4	5	6	7
d_i	2	2	3	3	3	3	7	7

KMP-Preprocessing2:

i	0	1	2	3	4	5	6	7
d'_i	2	2	4	4	6	3	8	7

Better, but still not faster than linear time.

Boyer-Moore Algorithm

Idea: compare search string s with a text t from right to left.

Example: s=OHO, t=ALCOHOLIC

ALCOHOLIC

OHO ← mismatch at first letter, no C in OHO

+3 OHO ← match

+2 OHO ← mismatch at first letter, no I in OHO, so we are done

A runtime of O(n/m) is possible.

Boyer-Moore Algorithm

Algorithm Naive Boyer-Moore:

i:=1

while $i \leq n-m+1$ do

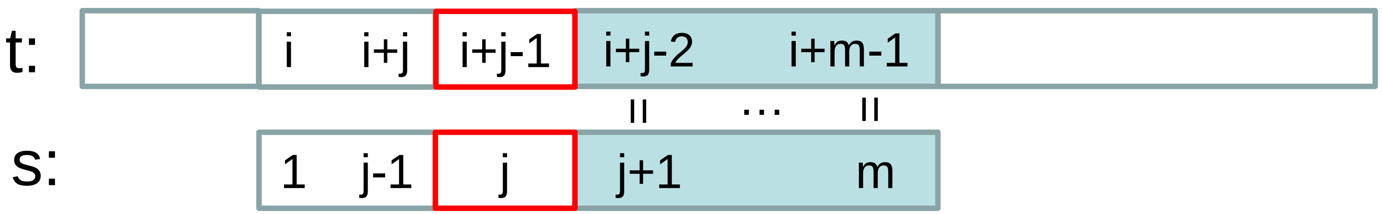
  j:=m // $(s_1 \ldots s_m)=(t_i \ldots t_{i+m-1})$?

  while $j \geq 1$ and $s_j = t_{i+j-1}$ do

    j:=j-1

  if j=0 then output i // match found

  i:=i+1



Naive Boyer-Moore Algorithm doesn´t jump forward quickly enough, but there are various ways to accelerate that.

Boyer-Moore Algorithm

Occurance shift preprocessing:

- For every $c \in \Sigma$, compute
  $last[c] := \max\{ j \in \{1,\ldots,m\} \mid s_j = c \}$
  If there is no c in s, set $last[c] := 0$.

Can certainly be done in $O(m)$ time.

Boyer-Moore algorithm with occurance shift:

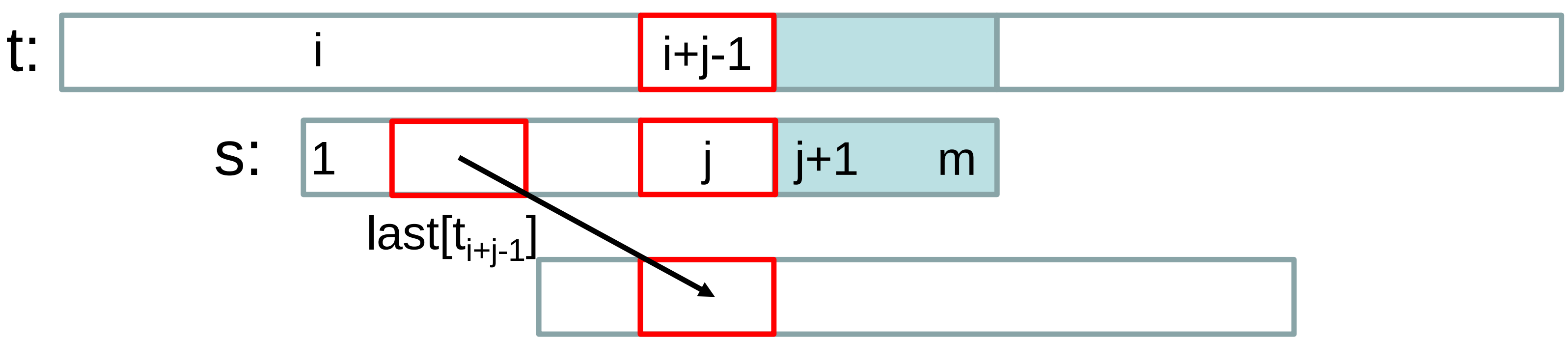
```
i:=1
while i≤n-m+1 do
    j:=m  // (s_1…s_m)=(t_i…t_{i+m-1})?
    while j≥1 and s_j=t_{i+j-1} do
        j:=j-1
    if j=0 then output i ; i:=i+1   // match found
            else i:=i+max{1,j-last[t_{i+j-1}]}
```

Boyer-Moore Algorithm

Boyer-Moore algorithm with occurance shift:

i:=1
while i≤n-m+1 do
    j:=m // $(s_1 \dots s_m)=(t_i \dots t_{i+m-1})$?
    while j≥1 and $s_j=t_{i+j-1}$ do
        j:=j-1
    if j=0 then output i ; i:=i+1   // match found
            else i:=i+max{1,j-last[t_{i+j-1}]}

Or better: i:=i+(d_m-1)



Boyer-Moore Algorithm

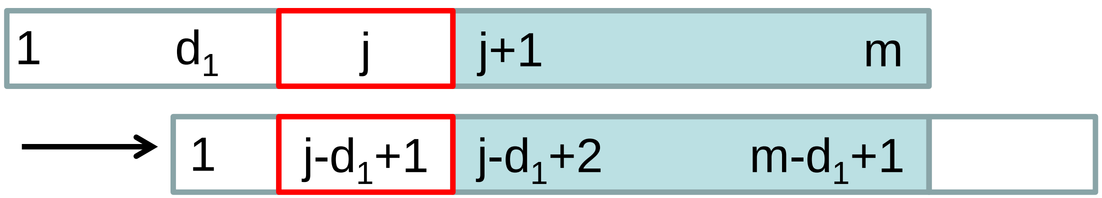
Boyer-Moore algorithm with occurance shift:

```
i:=1
while i≤n-m+1 do
    j:=m  // (s1…sm)=(ti…ti+m-1)?
    while j≥1 and sj=ti+j-1 do
        j:=j-1
    if j=0 then output i ; i:=i+1   // match found
           else i:=i+max{1,j-last[ti+j-1]}
```

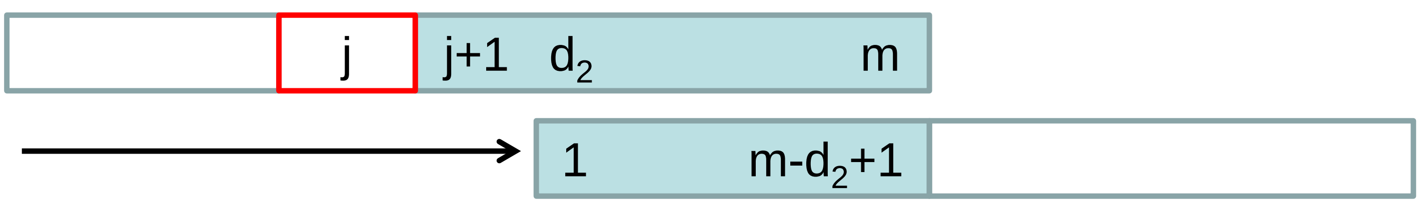
In practice, this is already much faster, but we can do better with the following suffix rule.

Boyer-Moore Algorithm

1. Compute the minimal $d_1 \in \{1,\ldots,j\}$ with $s_{j-d_1+1} \neq s_j$ (BM2) and $(s_{j-d_1+2}\ldots s_{m-d_1+1})=(s_{j+1}\ldots s_m)$ (BM1). If there is no such d_1, we set d_1 to $m+1$.



2. Compute the minimal $d_2 \in \{j+1,\ldots m\}$ with $(s_1\ldots s_{m-d_2+1}) = (s_{d_2}\ldots s_m)$. If there is no such d_2, we set d_2 to $m+1$.



Boyer-Moore Algorithm

The suffix rule allows us to increase i by $d=\min(d_1,d_2)$ without missing a matching. For all $0 \leq j \leq m$ let $D_j=d$ for the d above. With these D_j-values we can run the improved Boyer-Moore Algorithm.

Algorithm Boyer-Moore:

```
execute BM-Preprocessing to obtain D
i:=1
while i≤n-m+1 do
    j:=m
    while j≥1 and s_j=t_{i+j-1} do
        j:=j-1
    if j=0 then output j   // match found
    i:=i+D_j-1    // only change compared to naive BM
```

Boyer-Moore Algorithm

Example: s=abaababaabaab

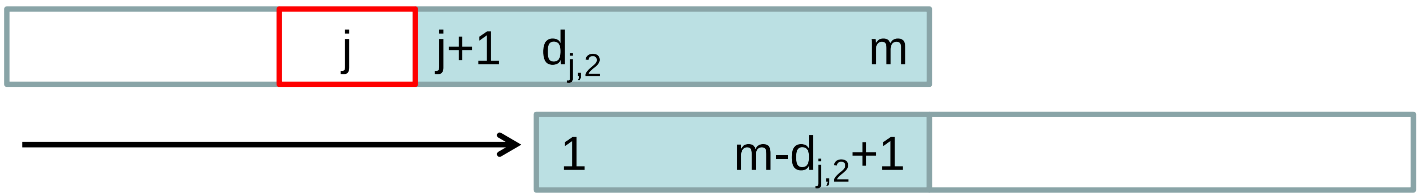
j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
s_j		a	b	a	a	b	a	b	a	a	b	a	a	b
D_j	8	8	8	8	8	8	8	8	3	11	11	6	13	1

It is not so easy to compute that efficiently…

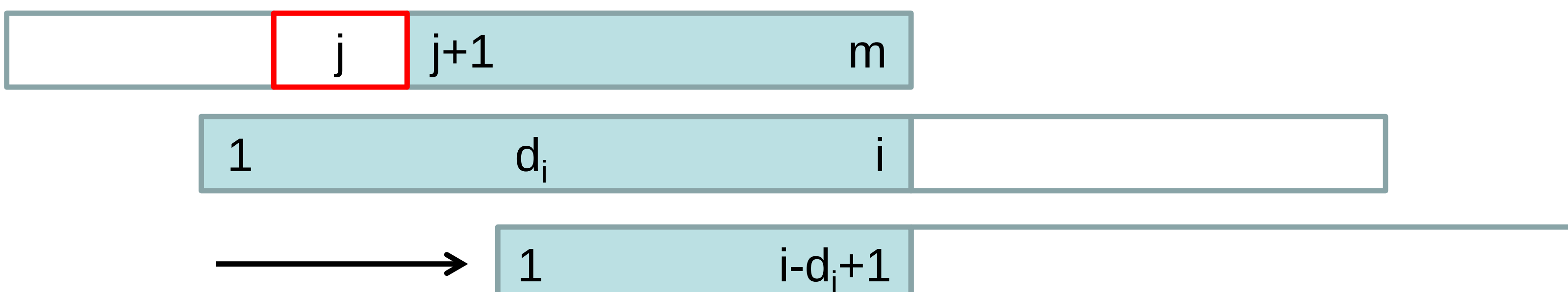
Boyer-Moore Algorithm

First, we consider the problem of implementing rule 2 of the suffix rule:

- Compute the minimal $d_2 \in \{j+1,\ldots,m\}$ with $(s_1\ldots s_{m-d_2+1}) = (s_{d_2}\ldots s_m)$. If there is no such d_2, we set d_2 to $m+1$. Let us call this d_2 $d_{j,2}$.

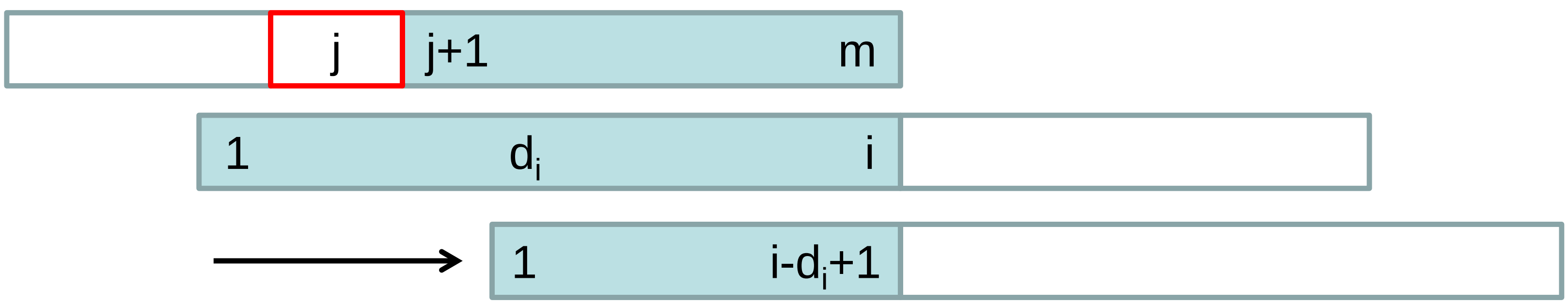


- Let $d_0,\ldots,d_m$ be the values from the KMP preprocessing.
- It is easy to see that $d_{0,2}=d_m$.
- For $j>0$, we keep shifting s until $d_{j,2}>j$.



Boyer-Moore Algorithm

- Let $d_0,\dots,d_m$ be the values from the KMP preprocessing.
- It is easy to see that $d_{0,2}=d_m$. For $j>0$, we keep shifting s until $d_{j,2}>j$.



$d_{0,2}:=d_m$
$\delta:=d_m$; $i:=m-\delta+1$ // δ: shift candidate for $d_{j,2}$
for $j:=1$ to m do
  if $j\geq\delta$ then // j too large: one more shift
    $\delta:=\delta+(d_i-1)$
    $i:=i-d_i+1$
  $d_{j,2}:=\delta$

Boyer-Moore Algorithm

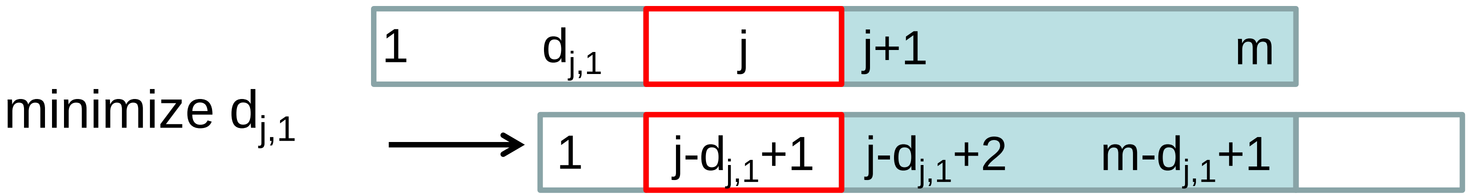
$d_{0,2}$:=d_m
δ:=d_m; i:=m-δ+1 // δ: shift candidate for $d_{j,2}$
for j:=1 to m do
  if j≥δ then // j too large: one more shift
    δ:=δ+(d_i-1)
    i:=i-d_i+1
  $d_{j,2}$:=δ

Example: s=ababaca

i / j	0	1	2	3	4	5	6	7
d_i	2	2	3	3	3	3	7	7
$d_{j,2}$	7	7	7	7	7	7	7	8

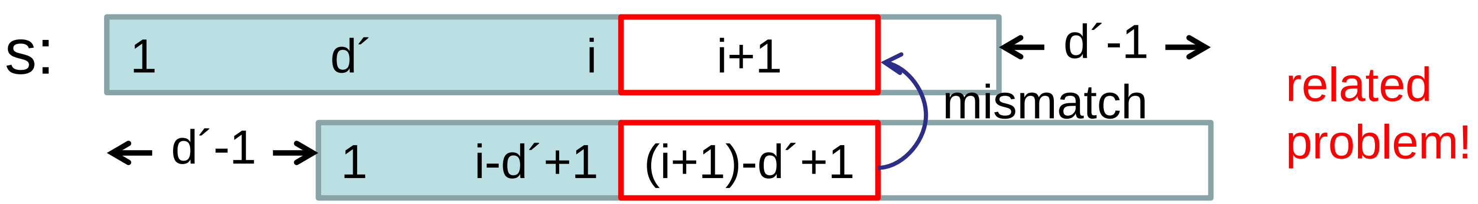
Boyer-Moore Algorithm

Next, we want to implement rule 1 of the suffix rule:



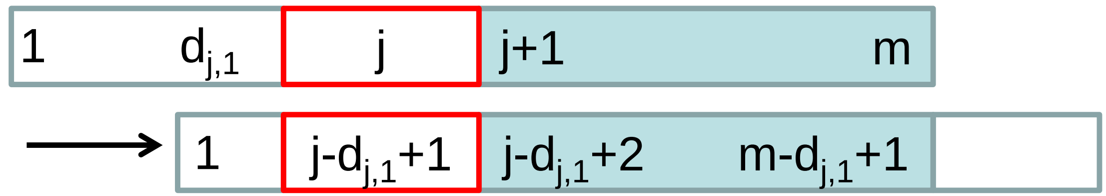
Remember improved KMP-preprocessing:

- For every position i in s, find the minimal $d'>1$ so that $(s_1 \ldots s_{i-d'+1}) = (s_{d'} \ldots s_i)$ and $s_{i-d'+2} \neq s_{i+1}$. If there is no such d', we set it to $i+2$.



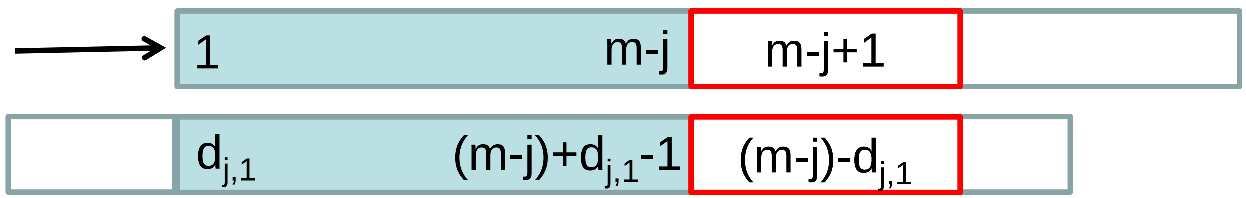
Boyer-Moore Algorithm

minimize $d_{j,1}$



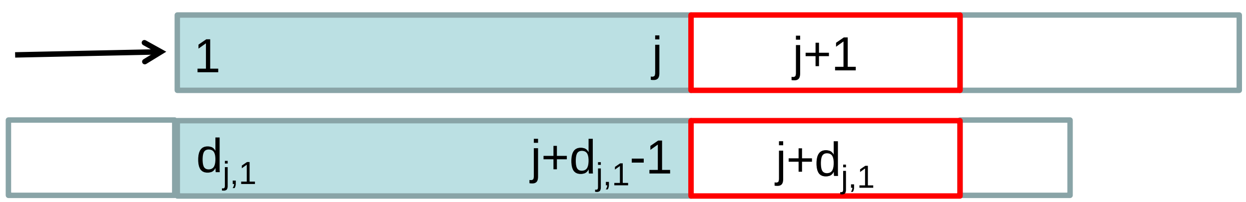
- Let s´ be the reverse s. Then we obtain the following equivalent problem for s´:

minimize $d_{j,1}$



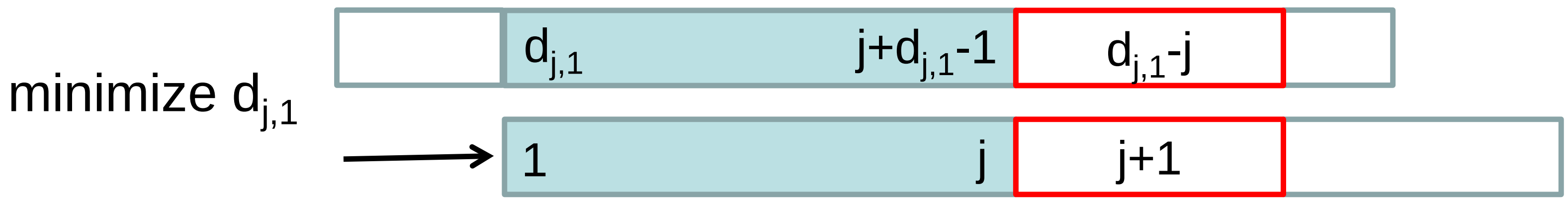
- Substituting j by m-j and re-defining $d_{j,1} := d_{m-j,1}$ gives us:

minimize $d_{j,1}$

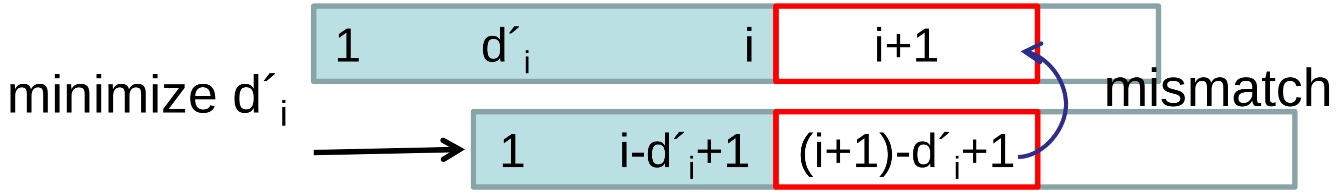


Boyer-Moore Algorithm

- So we have:



In the KMP-Preprocessing2 we solve:



- So for each j we can set $d_{j,1}:=\min\{ d'_i \mid i\in\{1,\ldots,m-1\}, i-d'_i+1=j \}$. For all other j´s there is no solution, so we use the default value given in rule 1.

Boyer-Moore Algorithm

So for the original j we use the rule:

$d_{j,1} := \min\{ d'_i \mid i \in \{1,\dots,m-1\},\ i-d'_i+1=j \}$.

If no such i exists, we set $d_{j,1}:=m+1$.

Algorithm for rule 1:

compute $d'_1,\dots,d'_{m-1}$ for s'
for $j:=0$ to m do
  $d_{j,1}:=m+1$
for $i:=1$ to $m-1$ do
  $j:=m-(i-d'_i+1)$
  if $j \le m$ and $d'_i < d_{j,1}$ then
    $d_{j,1}:=d'_i$

Boyer-Moore Algorithm

// computation of d-values for s´
d_0:=2; d_1:=2 // movement of s by 1
δ:=d_1 // current shift position of s
for i:=2 to m do
  while $\delta \le i$ and $s_i \ne s_{i-\delta+1}$ do
    // $(s_1 \ldots s_{i-\delta}) = (s_\delta \ldots s_{i-1})$ but $s_{i-\delta+1} \ne s_i$
    $\delta := \delta + d_{i-\delta} - 1$
  $d_i := \delta$

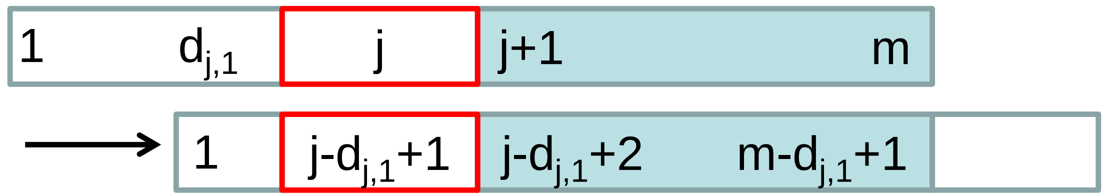
// computation of d´-values for s´
d'_0:=2
for i:=1 to m-1 do
  if $d_i > i$ then // no matching parts
    if $s_1 \ne s_{i+1}$ then $d'_i := d_i$ else $d'_i := d_i + 1$
  else
    if $d_{i+1} > d_i$ then // mismatch at i+1
      $d'_i := d_i$
    else
      $i' := i - d_i + 1$
      $d'_i := d_i + d'_{i'} - 1$
$d'_m := d_m$ // all symbols are matching

Example: s=abababaca, so s´=acababa

i	0	1	2	3	4	5	6	7
d_i	2	2	3	3	5	5	7	7
d'_i	2	2	4	3	6	5	8	7

Boyer-Moore Algorithm

compute $d'_1,\ldots,d'_{m-1}$ for s'
for $j:=0$ to m do
  $d_{j,1}:=m+1$
for $i:=1$ to $m-1$ do
  $j:=m-(i-d'_i+1)$
  if $j\leq m$ and $d'_i<d_{j,1}$ then
    $d_{j,1}:=d'_i$



Example: s=ababaca, so s´=acababa

i / j	0	1	2	3	4	5	6	7
d'_i	2	2	4	3	6	5	8	7
$d_{j,1}$	8	8	8	8	8	8	3	2

Boyer-Moore Algorithm

Example: s=ababaca. Remember that $D_j = \min\{d_{j,1}, d_{j,2}\}$.

j	0	1	2	3	4	5	6	7
$d_{j,1}$	8	8	8	8	8	8	3	2
$d_{j,2}$	7	7	7	7	7	7	7	8
D_j	7	7	7	7	7	7	3	2

Hence, most of the time there are very large jumps.

Boyer-Moore Algorithm

One can show the following result:

Theorem 7.11: Let k be the number of times the search string occurs in the text. Then the Boyer-Moore Algorithm has a runtime of $O(n+k \cdot m)$.

The proof is very complex and omitted here.

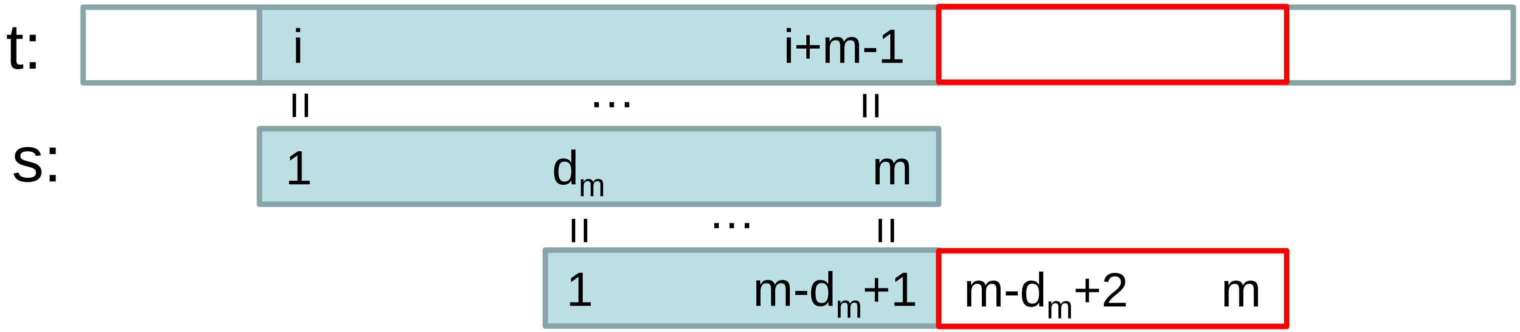
Remarks:

- If (BM2) is dropped, then the runtime increases to $O(n \cdot m)$.
- In practice, the Boyer-Moore Algorithm has a runtime of $O(n/m)$.

Boyer-Moore Algorithm

Remarks:

- To reduce the runtime from $O(n+km)$ to $O(n+m)$, we can use the fact that whenever s has been found in t, we only have to check $s_j=t_{i+j-1}$ for $j\in\{m-d_m+2,\dots,m\}$.



- To further reduce the runtime, we can combine the suffix rule with the occurance shift rule by setting $i:=i+\max\{ D_j-1, j-\text{last}[t_{i+j-1}]\}$.

Aho-Corasick Algorithm

Now we have the following situation: search in a text t for all positions in which a search string in $S=\{s_1,\ldots,s_k\}$ starts.

In the following let $m_i=|s_i|$ and $m = \Sigma_{i=1}^{k} m_i$.

First idea: run the KMP algorithm in parallel for all search strings.

Runtime: $O(m+k\cdot n)$

preprocessing main algorithm

Aho-Corasick Algorithm

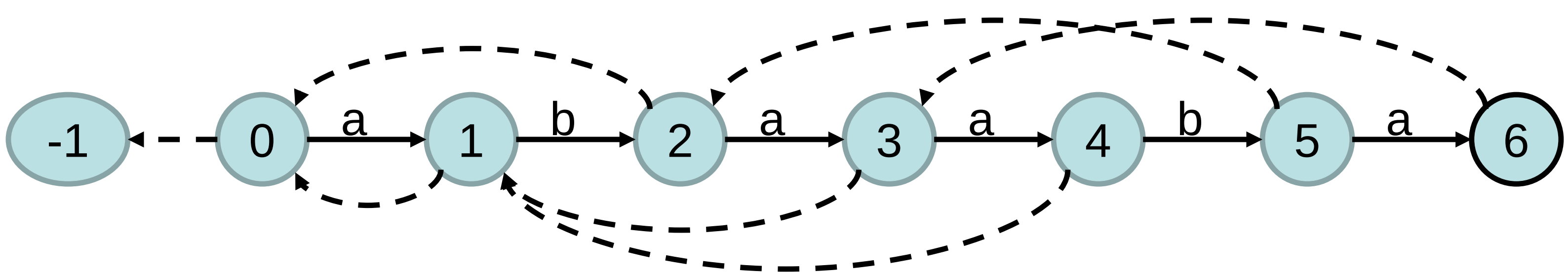
Better idea: instead of tables of d_i-values, use a finite automaton.

Example: let s=abaaba

- Table of d_i-values:

i	0	1	2	3	4	5	6
d_i	2	2	3	3	4	4	4

- Finite automaton:



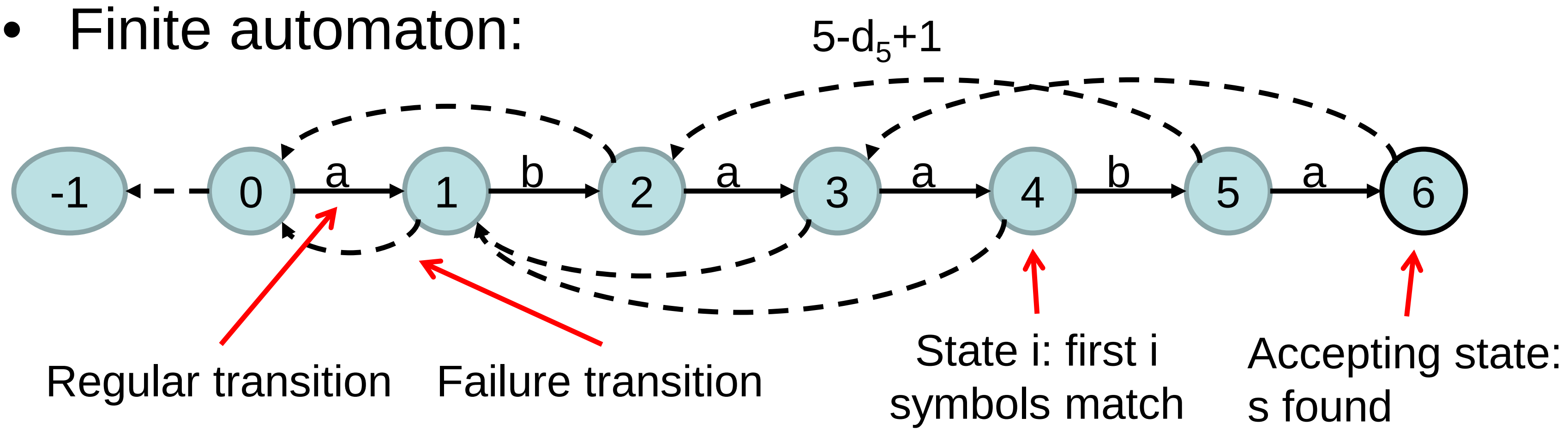
Aho-Corasick Algorithm

Example: let s=abaaba

- Table of d_i-values:

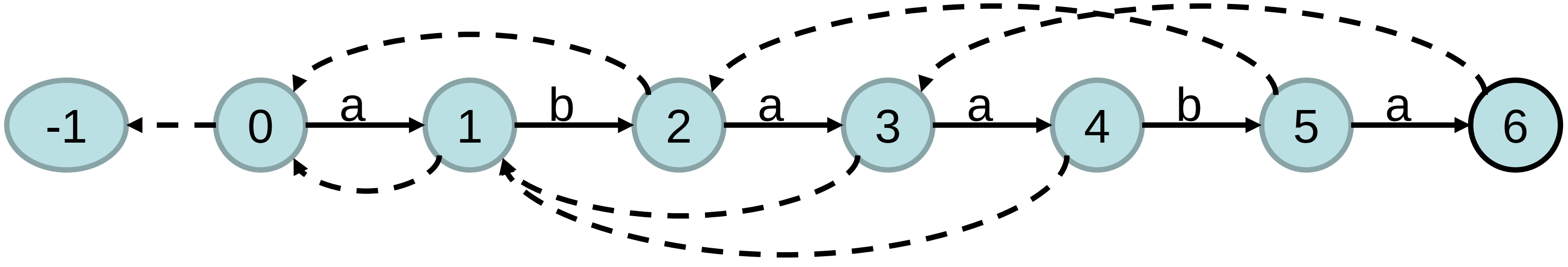
i	0	1	2	3	4	5	6
d_i	2	2	3	3	4	4	4

- Finite automaton:



Aho-Corasick Algorithm

Example: let s=abaaba



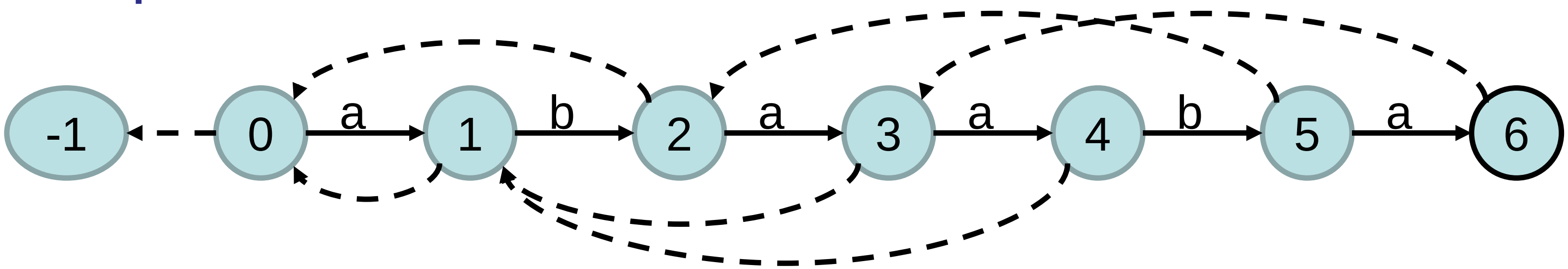
This is called an AC-automaton.

Definition 7.12: An AC-automaton consists of:

- Q: a finite set of states
- $\Gamma=\Sigma\cup\{fail\}$: a finite alphabet (with input alphabet Σ)
- $\delta:Q\times\Gamma\rightarrow Q$: a transition function
- q_0: an initial state and
- $F\subseteq Q$: a set of accepting states

Aho-Corasick Algorithm

Example: let s=abaaba



AC-automaton for $s \in \Sigma^*$ with $|s|=m$:

- $Q=\{-1,0,1\ldots,m\}$, $q_0=0$, and $F=\{m\}$
- $\Gamma=\Sigma\cup\{fail\}$
- For all $i\in\{0,\ldots,m-1\}$, $\delta(i,s_{i+1})=i+1$
- For all $i\in\{0,\ldots,m\}$, $\delta(i,fail)=i-d_i+1$
  The fail-transition is used if a symbol is read that does not have a regular transition.

Aho-Corasick Algorithm

AC preprocessing for a single search string s:

Algorithm AC-Preprocessing:

$d_0:=2$; $d_1:=2$ // movement of s by 1
$\delta:=d_1$ // δ: current candidate of d_i
for $i:=2$ to m do
  while $\delta \leq i$ and $s_i \neq s_{i-\delta+1}$ do
    // $(s_1 \ldots s_{i-\delta})=(s_\delta \ldots s_{i-1})$ but $s_{i-\delta+1} \neq s_i$
    $\delta:=\delta+(d_{i-\delta}-1)$
  $d_i:=\delta$
// compute $f_0,\ldots,f_m$ for fail transitions
for $i:=0$ to m do $f_i:=i-d_i+1$

Lemma 7.13: The AC preprocessing has a runtime of $O(m)$.

Proof: follows from KMP proprocessing.

Aho-Corasick Algorithm

Aho-Corasick Algorithm for one search string:

```
execute AC-Preprocessing
j:=0   // starting position in automaton
for i:=1 to n do
    while (j≠-1 and t_i≠s_{j+1}) do
        j:=f_j
    j:=j+1
    if j=m then output i-m+1
```

Theorem 7.14: The AC algorithm for a single search string is correct and runs in time $O(n)$.

Proof: follows from analysis of KMP algorithm

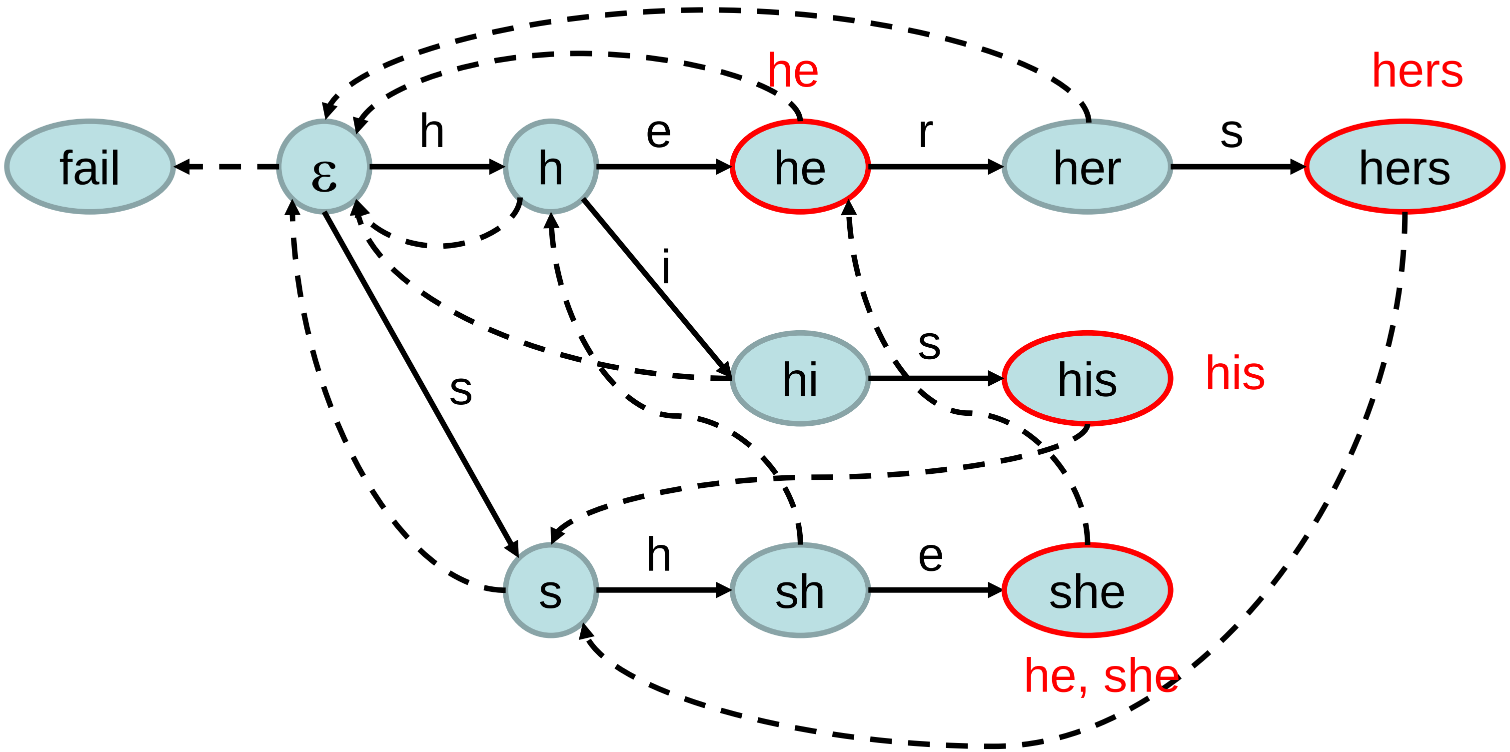
Aho-Corasick Algorithm

AC automaton for a set S of multiple search strings:

- $Q=\{ w\in\Sigma^* \mid w$ is a prefix of an $s\in S\}\cup\{fail\}$ and $q_0=\varepsilon$
- $F=F_1\cup F_2$ where
  - $F_1=S$ and
  - $F_2=\{w\in\Sigma^* \mid \exists s\in S: s$ is a suffix of $w\}$
- For all $w\in Q$ and $a\in\Sigma$ it holds:
  - $\delta(w,a) = w\circ a$ whenever $w\circ a\in Q$, and otherwise
  - $\delta(w,fail)=w'$ for the $w'\in Q$ representing the largest suffix of w. For $w=\varepsilon$, $\delta(w,fail)=fail$ (where „fail“ represents the state that was previously „-1“).

Aho-Corasick Algorithm

Example: S={he,she,his,hers}



Aho-Corasick Algorithm

Aho-Corasick Algorithm for a set S of search strings:

- m: sum of lengths of all $s \in S$
- f_w: state reached by $\delta(w,fail)$
- S_w: set of all $s \in S$ that are a suffix of w

```
execute Extended-AC-Preprocessing
w:=ε   // starting position in AC automaton
for i:=1 to n do
   while (w≠fail and δ(w,t_i) is not defined) do
      w:=f_w
   if w=fail then w:=ε else w:=w∘t_i
   if w∈F then output (i,S_w)
```

Theorem 7.15: The AC algorithm is correct and has a runtime of $O(n+m)$.

Proof: it remains to specify Extended-AC-Preprocessing

Extended-AC-Preprocessing

The AC automaton for S can be constructed in three phases:

Phase I: construct the prefix tree of S with the regular transitions and mark the states belonging to F_1

Phase II: compute the fail transitions in breadth-first-search order starting with state ε

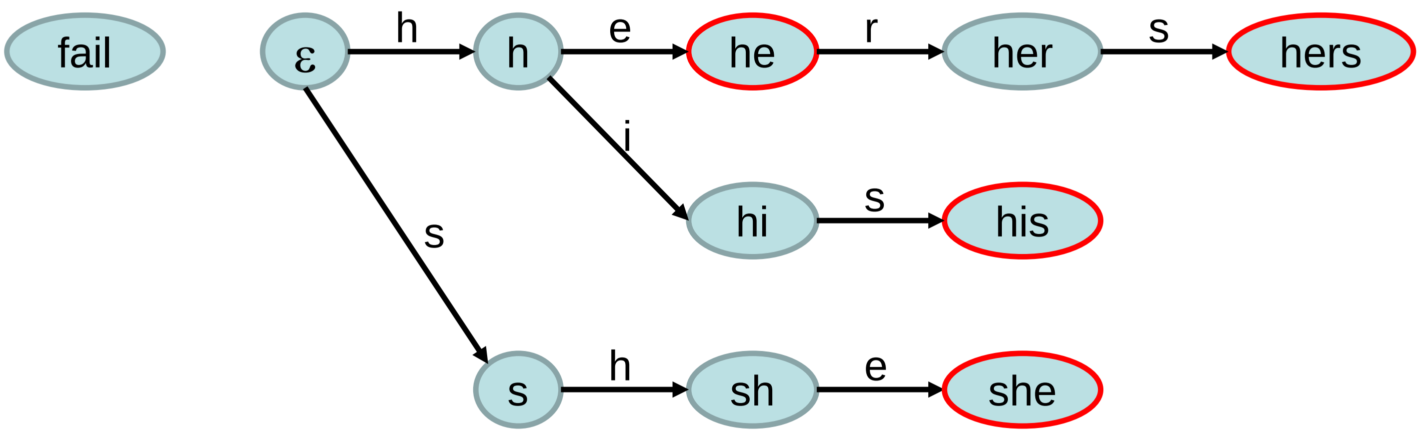
Phase III: compute the states belonging to F_2 and the sets S_w for all $w \in F_1 \cup F_2$ in breadth-first-search order starting with state ε

Extended-AC-Preprocessing

The AC automaton for S can be constructed in three phases:

Phase I: construct the prefix tree of S with the regular transitions and mark the states belonging to F_1

Example: S={he,she,his,hers}



Extended-AC-Preprocessing

The AC automaton for S can be constructed in three phases:

Phase I: construct the prefix tree of S with the regular transitions and mark the states belonging to F_1

Algorithm for Phase I:

Build a trie for S and set F:=S

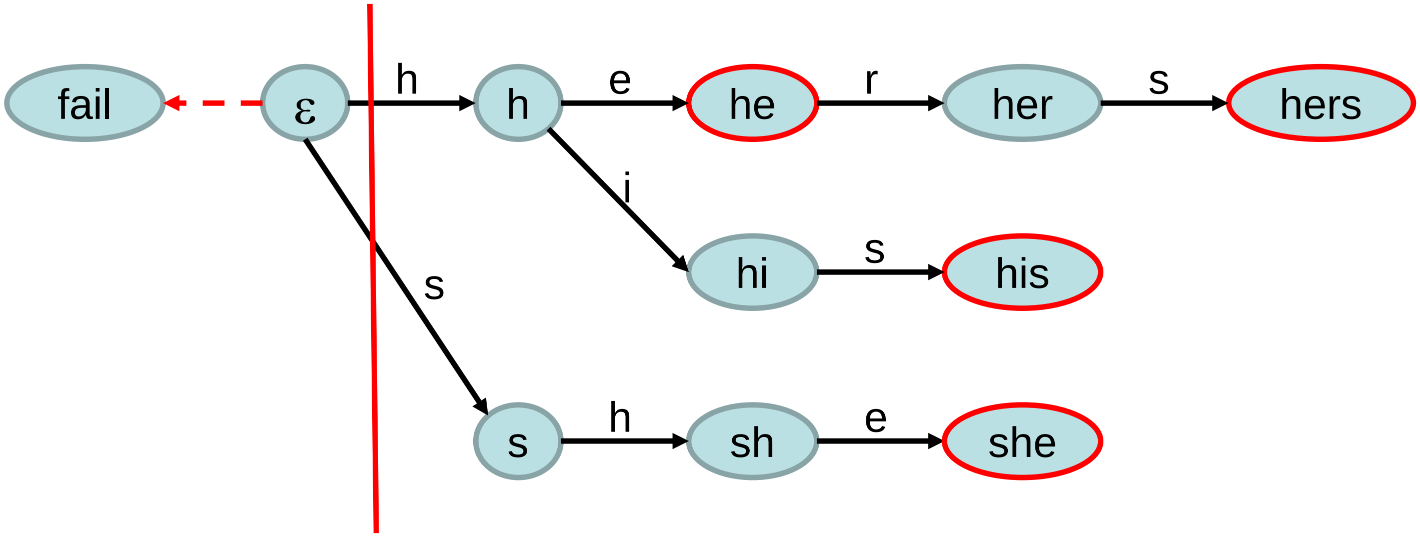
Runtime: O(m)

Extended-AC-Preprocessing

The AC automaton for S can be constructed in three phases:

Phase II: compute the fail transitions in breadth-first-search order starting with state ε

Example: S={he,she,his,hers}

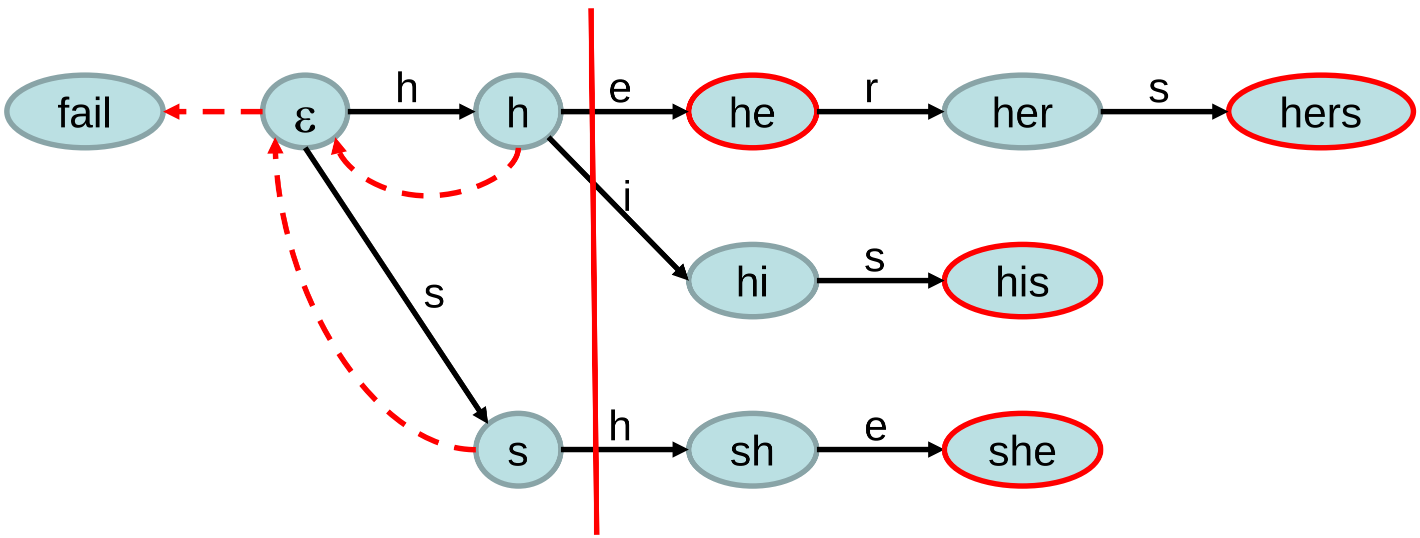


Extended-AC-Preprocessing

The AC automaton for S can be constructed in three phases:

Phase II: compute the fail transitions in breadth-first-search order starting with state ε

Example: S={he,she,his,hers}

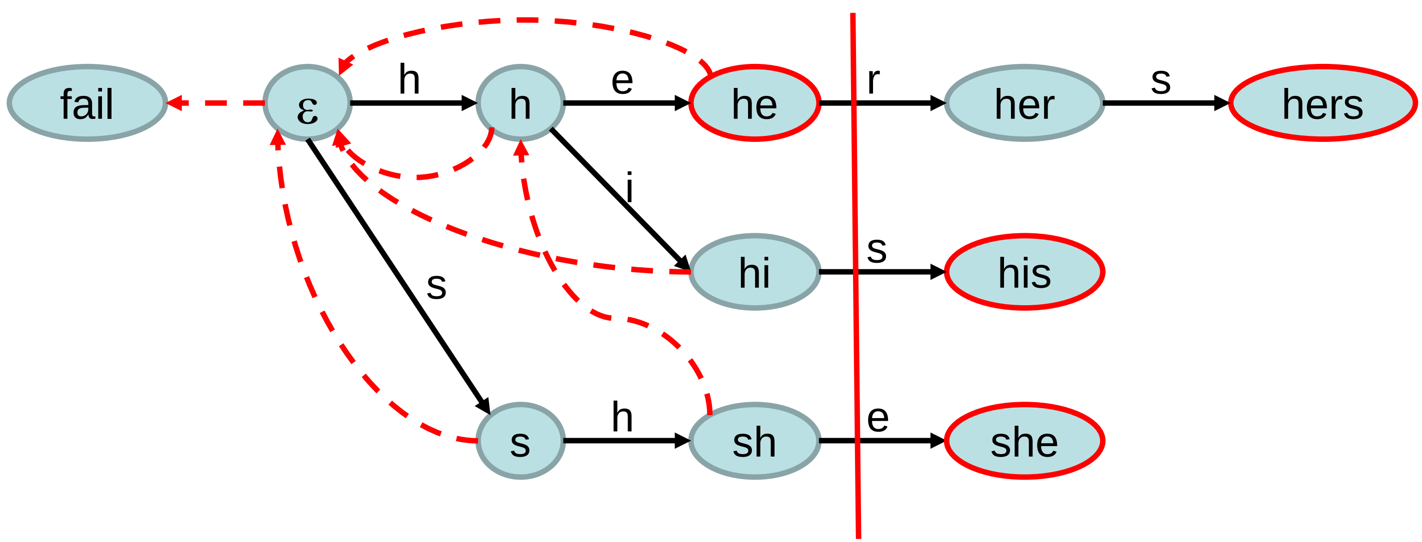


Extended-AC-Preprocessing

The AC automaton for S can be constructed in three phases:

Phase II: compute the fail transitions in breadth-first-search order starting with state ε

Example: S={he,she,his,hers}

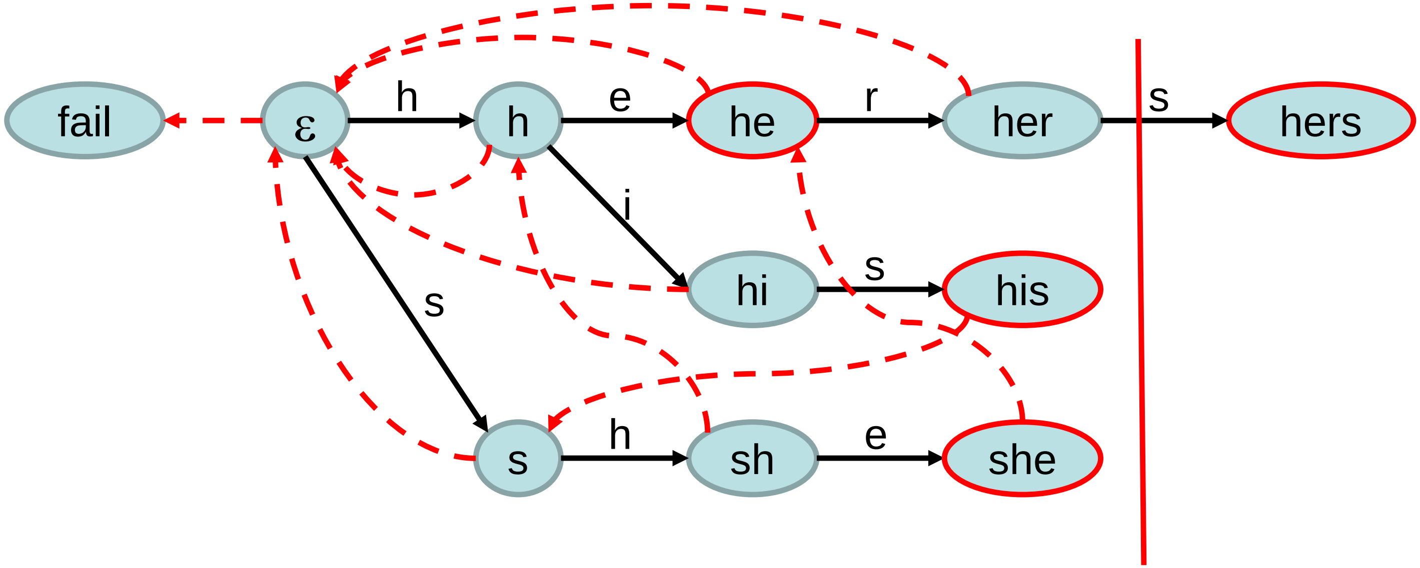


Extended-AC-Preprocessing

The AC automaton for S can be constructed in three phases:

Phase II: compute the fail transitions in breadth-first-search order starting with state ε

Example: S={he,she,his,hers}

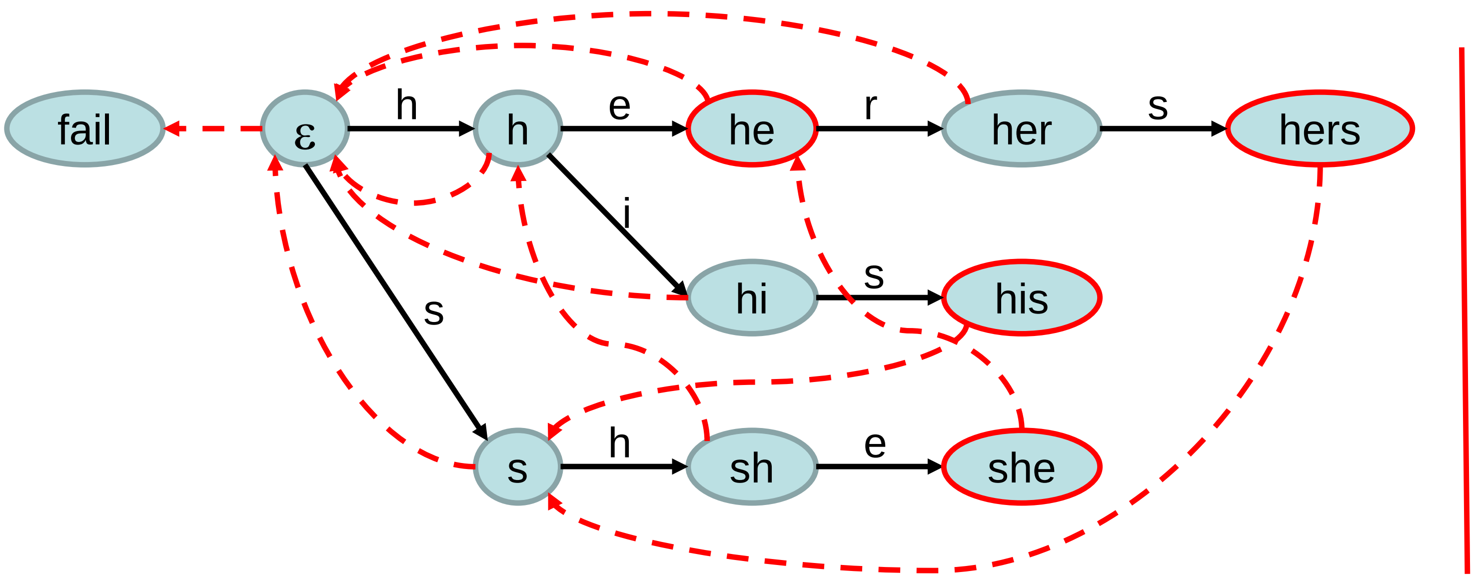


Extended-AC-Preprocessing

The AC automaton for S can be constructed in three phases:

Phase II: compute the fail transitions in breadth-first-search order starting with state ε

Example: S={he,she,his,hers}



Extended-AC-Preprocessing

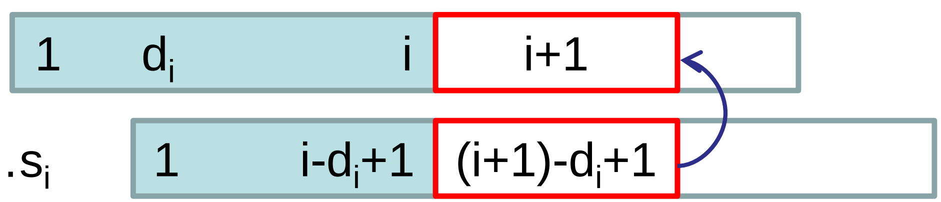
The AC automaton for S can be constructed in three phases:

Phase II: compute the fail transitions in breadth-first-search order starting with state ε

Algorithm for Phase II: similar to KMP preprocessing

- Consider a state of the AC automaton representing $s_1 \ldots s_{i+1}$. Start with fail transition of $s_1 \ldots s_i$ for largest potential suffix for fail transition of $s_1 \ldots s_{i+1}$.

s:



Largest suffix for $s_1 \ldots s_i$ (which might come from some other string s´!)

Extended-AC-Preprocessing

Phase II:

- Initialization:
  - f_ε:=fail
  - $f_a:=\varepsilon$ for all $a\in\Sigma$
- For all $w\in Q\setminus\{\varepsilon\}$ in BFS order:
  - $f_w:=f_{pred(w)}$ // pred(w): w without last symbol
  - while ($f_w\neq$fail and $\delta(f_w,last(w))$ undefined) do
    // last(w): last symbol of w
    $f_w:=f_{f_w}$
  - if $f_w=$fail then $f_a:=\varepsilon$ else $f_w:=\delta(f_w,last(w))$

Lemma 7.16: The Extended-AC-Preprocessing needs at most $O(m)$ time to compute the AC automaton.

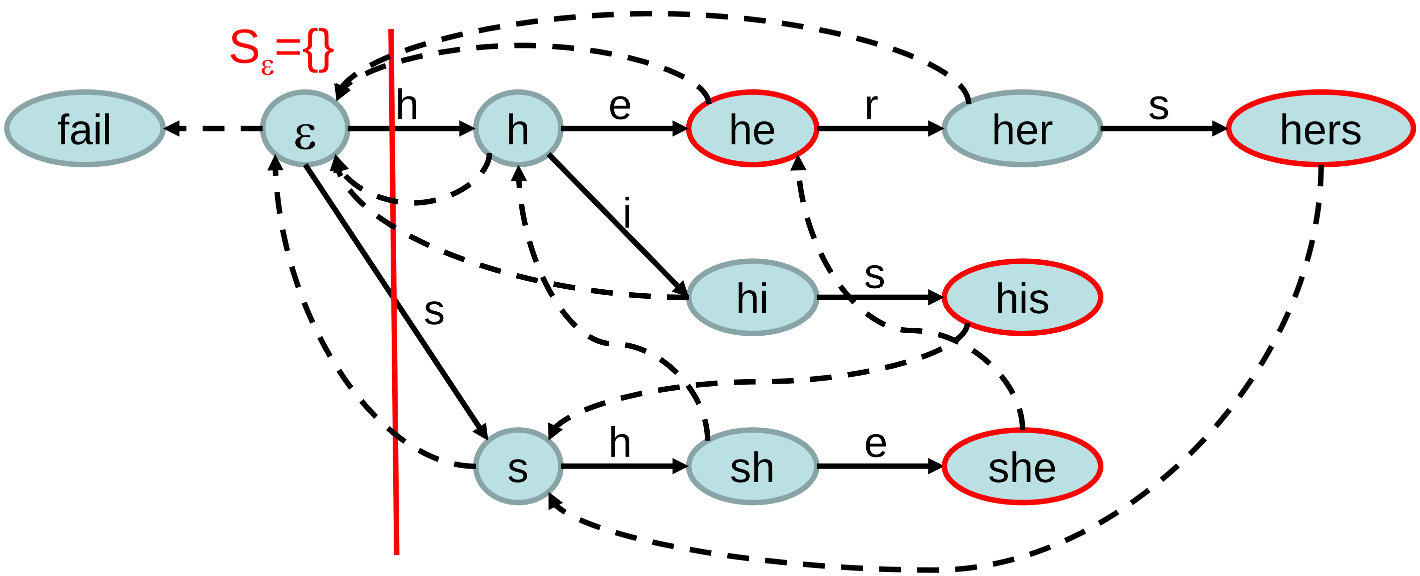
Proof: Exercise.

Extended-AC-Preprocessing

The AC automaton for S can be constructed in three phases:

Phase III: compute the states belonging to F_2 and the sets S_w for all $w \in F_1 \cup F_2$ in breadth-first-search order starting with state ε

Example: S={he,she,his,hers}

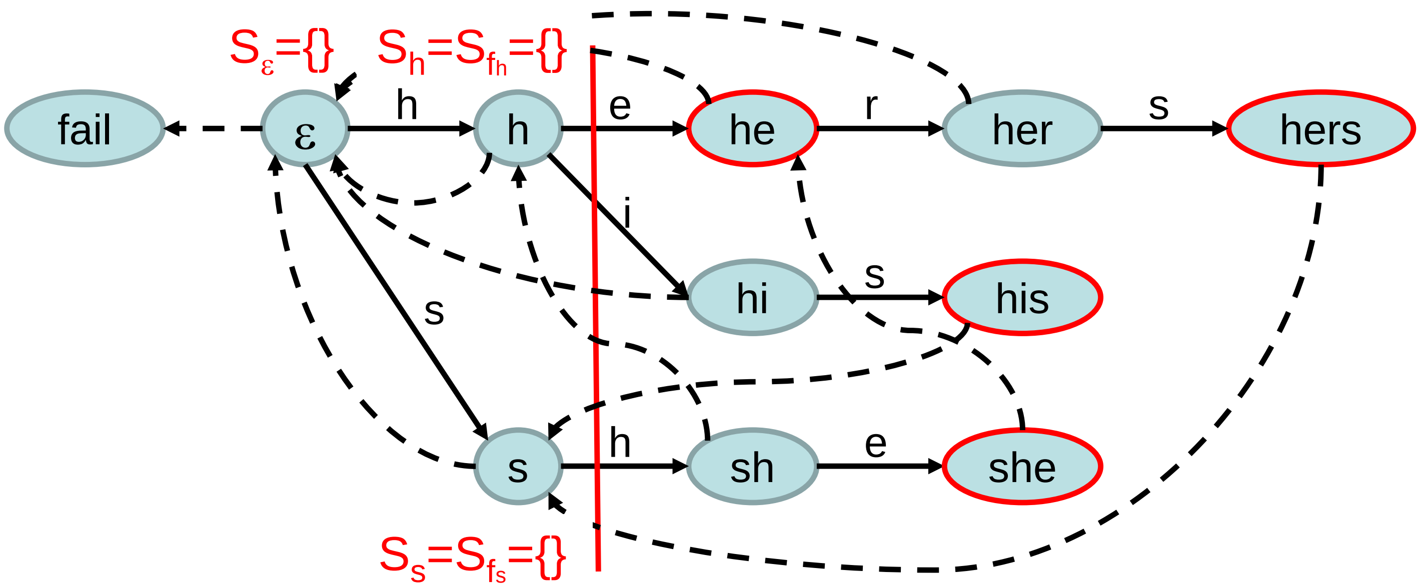


Extended-AC-Preprocessing

The AC automaton for S can be constructed in three phases:

Phase III: compute the states belonging to F_2 and the sets S_w for all $w \in F_1 \cup F_2$ in breadth-first-search order starting with state ε

Example: S={he,she,his,hers}

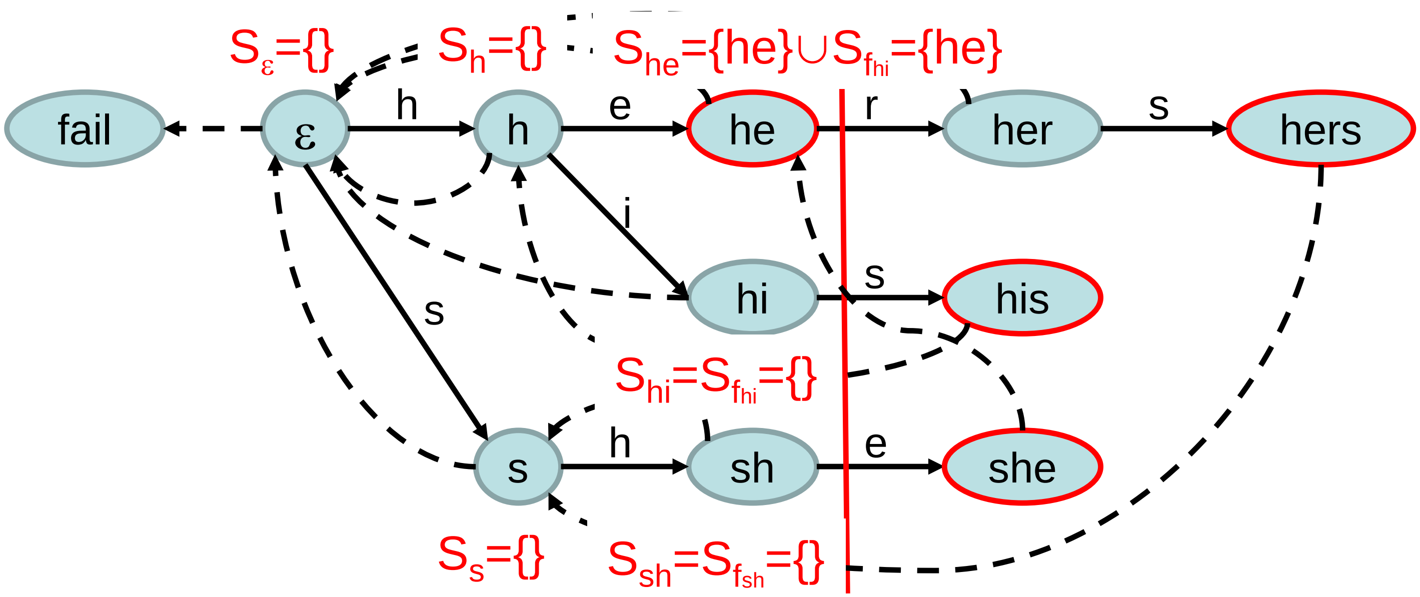


Extended-AC-Preprocessing

The AC automaton for S can be constructed in three phases:

Phase III: compute the states belonging to F_2 and the sets S_w for all $w \in F_1 \cup F_2$ in breadth-first-search order starting with state ε

Example: S={he,she,his,hers}

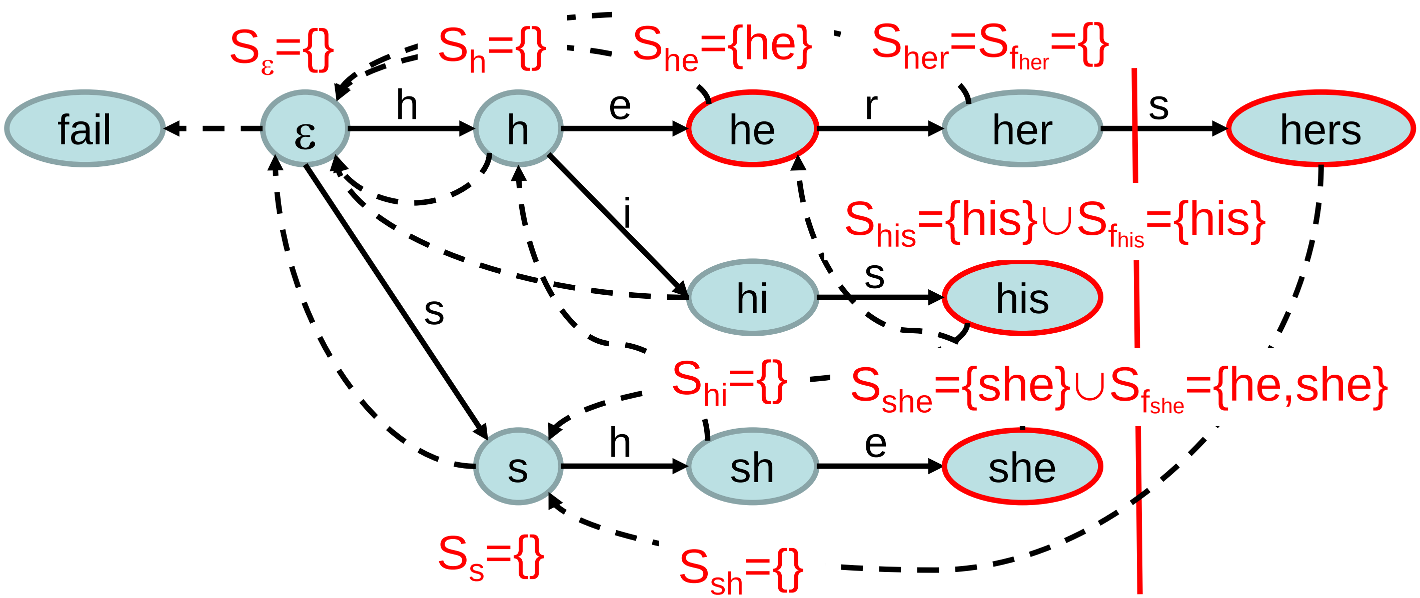


Extended-AC-Preprocessing

The AC automaton for S can be constructed in three phases:

Phase III: compute the states belonging to F_2 and the sets S_w for all $w \in F_1 \cup F_2$ in breadth-first-search order starting with state ε

Example: S={he,she,his,hers}

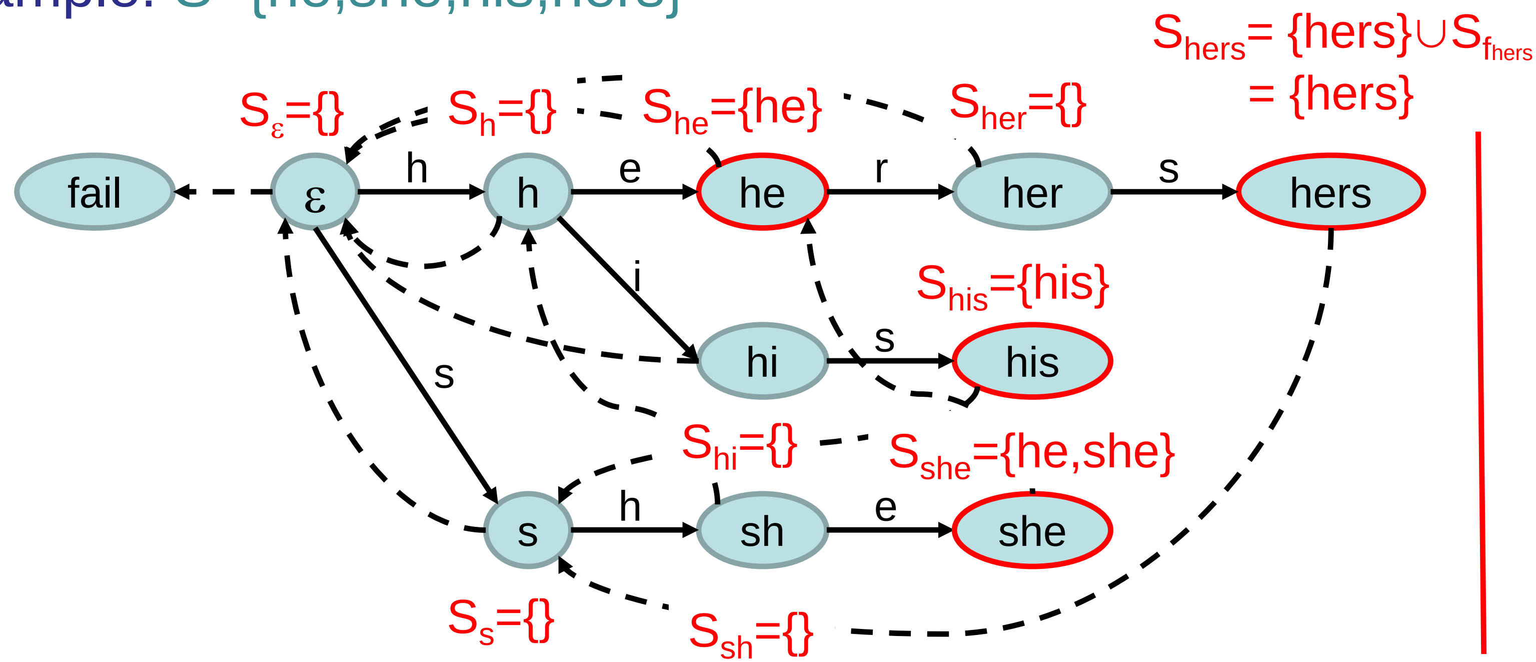


Extended-AC-Preprocessing

The AC automaton for S can be constructed in three phases:

Phase III: compute the states belonging to F_2 and the sets S_w for all $w \in F_1 \cup F_2$ in breadth-first-search order starting with state ε

Example: S={he,she,his,hers}



Extended-AC-Preprocessing

The AC automaton for S can be constructed in three phases:

Phase III: compute the states belonging to F_2 and the sets S_w for all $w \in F_1 \cup F_2$ in breadth-first-search order starting with state ε

Algorithm of Phase III:

- For all $w \in Q \setminus F$ do $S_w := \{\}$ // at this point we still have $F=F_1$
- For all $w \in F$ do $S_w := \{w\}$
- For all $w \in Q \setminus \{\varepsilon\}$ in BFS order:
  - $S_w := S_w \cup S_{f_w}$
  - if $S_w \neq \{\}$ then $F := F \cup \{w\}$

Runtime: $O(m)$ (when storing S_w's implicitly via links)

Aho-Corasick Algorithm

Aho-Corasick Algorithm for regular expressions (basic idea):

- Build non-deterministic finite automaton (NFA) for that regular expression with starting state q_0.
- Add transitions $\delta(q_0,c)=q_0$ for every $c\in\Sigma$ to take into account that the string s matching the regular expression in the given text t could start at any point in t.
- Convert the NFA into a deterministic automaton (DFA) using the power set method, if the state-space of the DFA does not get too large.

Theorem 7.17: With an NFA of size m for the regular expression R, it can be checked in $O(n\cdot m)$ time whether there is a substring s in t with $s\in R$. With a DFA, the runtime can be reduced to $O(n)$, but the time needed to set up the DFA might be around $O(2^m)$.

Suffix Trees

- Given a text t, we now consider the problem of preprocessing t so that we can check for any search string s of length m in $O(m)$ time whether s is a substring of t.

- Solution: suffix tree of t

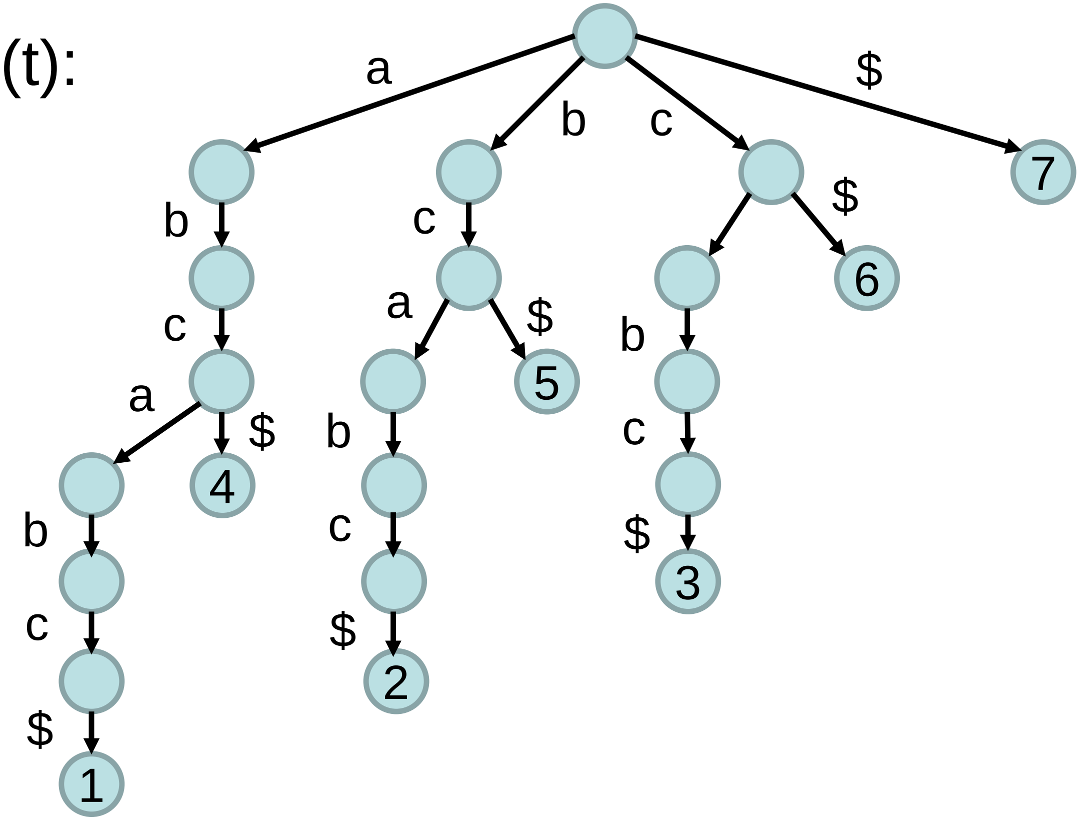
Definition 7.18: Let $t=t_1\ldots t_{n-1}\$$ be a text with special end symbol $\$$.

- $t[i..n]=t_i\ldots t_n$ denotes the suffix of t starting with t_i.
- The suffix trie $ST(t)$ of t is the trie resulting from the strings $t[1..n],t[2..n],\ldots,t[n..n]$ (see Section 3). Every leaf of $ST(t)$ stores i if and only if it represents $t[i..n]$.

Suffix Trees

Example: t=abcabc$.

ST(t):



Suffix Trees

Remarks:

- If we want to check whether s is a substring of t, we simply follow the unique path in ST(t) whose edge labels form s. If this path exists, s is indeed a substring of t, and otherwise this is not the case. Certainly, this checking can be done in O(|s|) time.
- If we additionally want to know all positions at which s starts in t, we need to determine the set of all $i \in \{1,..,n\}$ stored in the leaves reachable from the trie node representing s in ST(t).

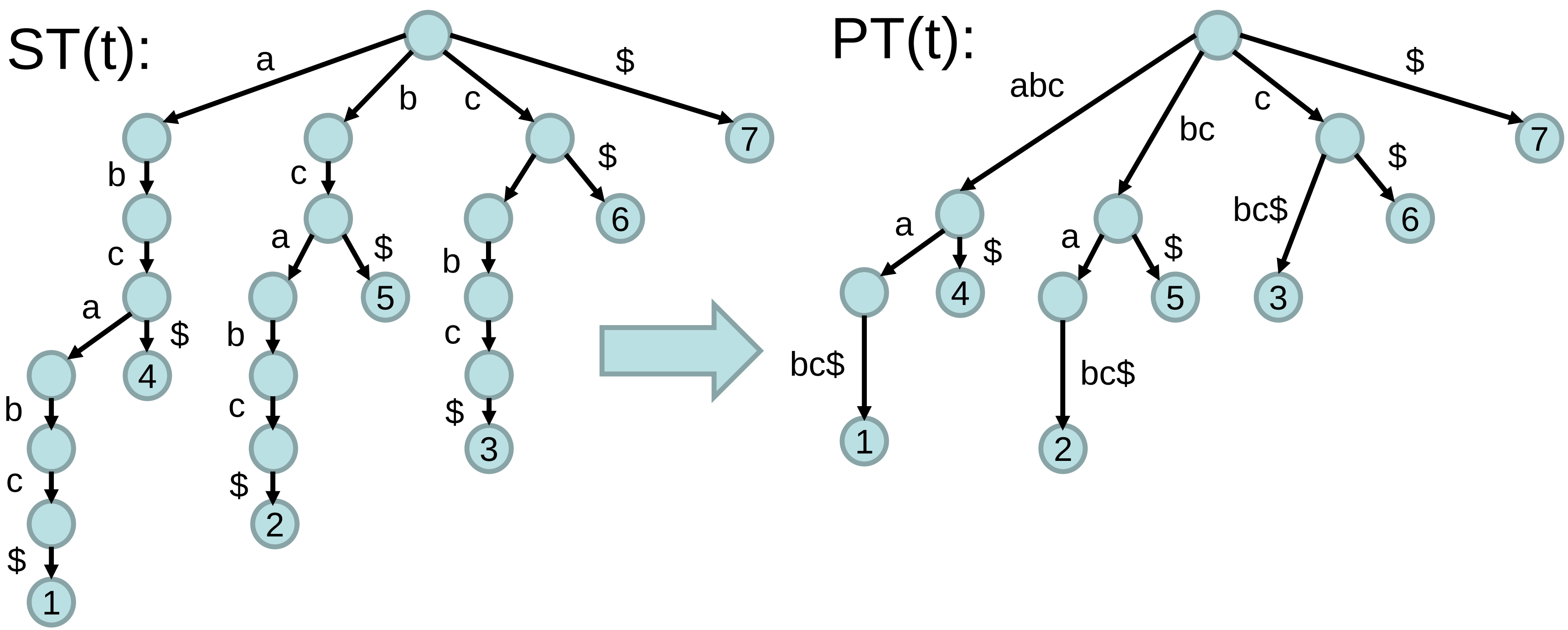
Problem: ST(t) may have $\Theta(n^2)$ many nodes, where n is the length of t. This is the case, for example, for $t=a^m b^m \$$.

Solution: Condense ST(t) to the Patricia trie of ST(t).

Suffix Trees

Definition 7.19: The suffix tree PT(t) of t is the Patricia trie of ST(t).

Example: t=abcabc$.



Suffix Trees

Lemma 7.20: For any text $t=t_1\ldots t_{n-1}\$$, PT(t) consists of just O(n) nodes.
Proof: follows from the properties of Patricia tries.

For every node v in PT(t) define
- count(v): number of leaves below it,
- first(v): minimum index i stored below it, and
- last(v): maximum index i stored below it.

Suppose that every node v in PT(t) stores count(v), first(v), and last(v).

Theorem 7.21: For every search string s, the following queries can be answered in O(|s|) time:
- Find the first occurence of s in t.
- Find the last occurence of s in t.
- Find the number of times s occurs in t.

Suffix Trees

Problem: How to construct PT(t) efficiently?

Naive approach:

T_0:= suffix tree just consisting of the root
for i:=1 to n do
    T_i:=insert(T_{i-1},t[i..n])

Runtime of insert(T_{i-1},t[i..n]):

- Standard approach of traversing the edges of T_{i-1} from the root: time O(n) (since depth of T_{i-1} can be proportional to i and up to n-i characters may have to be checked to find insertion point)
- When using the hashed Patricia trie with msd-nodes and ignoring work for individual character comparisons: runtime is O(log n)

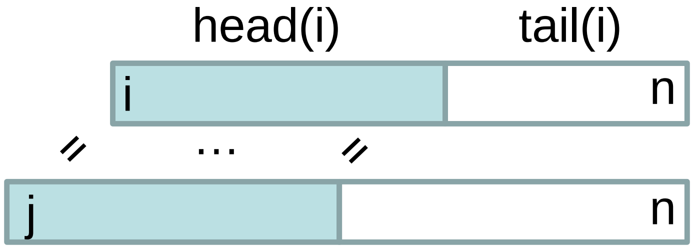
In any case, the best achievable bound seems to be O(n log n) for constructing PT(t).

Suffix Trees

The algorithm of McCreight can construct $PT(t)$ in time $O(n)$ (including the time for character comparisons). To understand that algorithm we need some notation.

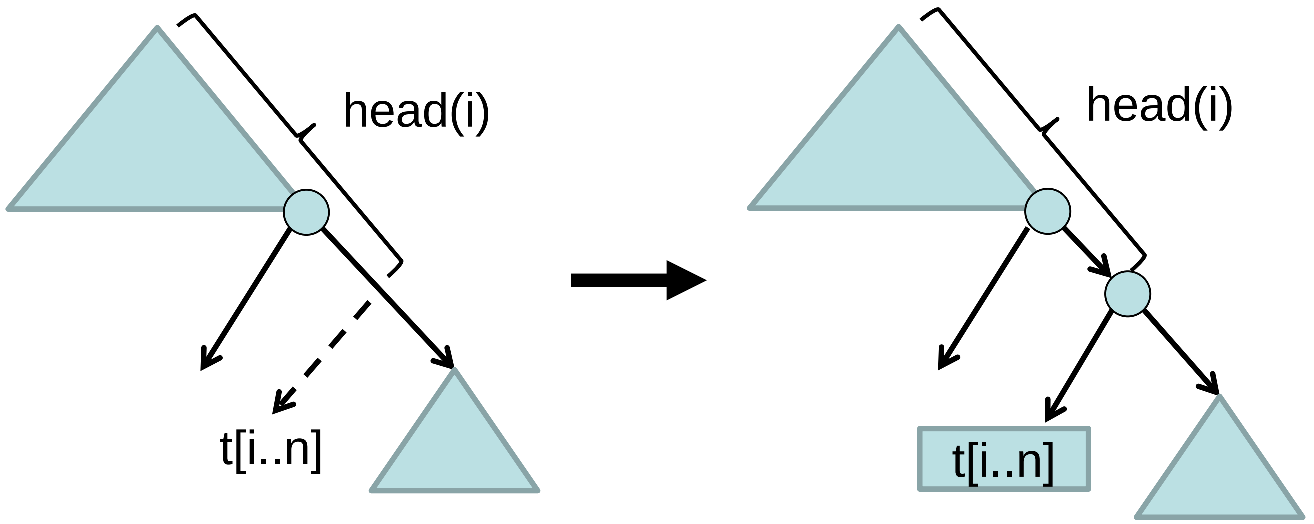
Definition 7.22:

- For any node v in a suffix tree T let $path(v)$ be the concatenation of edge labels from the root of T down to v.
- For any string $\alpha \in \Sigma^*$, we say that $\alpha \in T$ if there is a node v in T with α being a prefix of $path(v)$.
- For any $i \in \{1,\ldots,n\}$, let $head(i)$ be the longest prefix of $t[i..n]$ that is a prefix of some $t[j..n]$ with $j<i$. Let $tail(i)$ be $t[i..n]$ without $head(i)$.



Suffix Trees

Note that head(i) is the place where the new node v with path(v)=t[i..n] needs to be inserted into T_{i-1}.



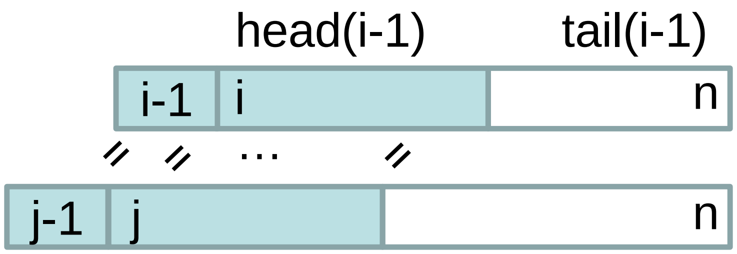
If we can find head(i) efficiently, we can quickly insert t[i..n]. For that we need so-called suffix links.

Suffix Trees

Lemma 7.23: Consider any $a \in \Sigma$ and $\beta \in \Sigma^*$, and let T_i be defined as in the naive suffix tree algorithm. If $head(i-1)=a\beta$ then β is a prefix of $head(i)$.

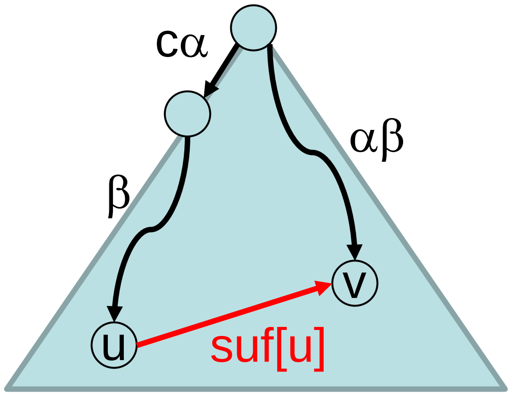
Proof:

- Let $head(i-1)=a\beta$.
- Then there is a $j<i$ with $a\beta$ being a prefix of $t[j-1..n]$.
- Hence, β is a prefix of $t[j..n]$ and $t[i..n]$.
- Therefore, β is a prefix of $head(i)$.



Suffix Trees

Definition 7.24: Let u and v be two inner nodes of a suffix tree T. Then suf[u]=v if and only if there is a $c \in \Sigma$ with $path(u)=c \circ path(v)$. suf[u] is called the suffix link of u.



path(u)=cαβ

path(v)=αβ

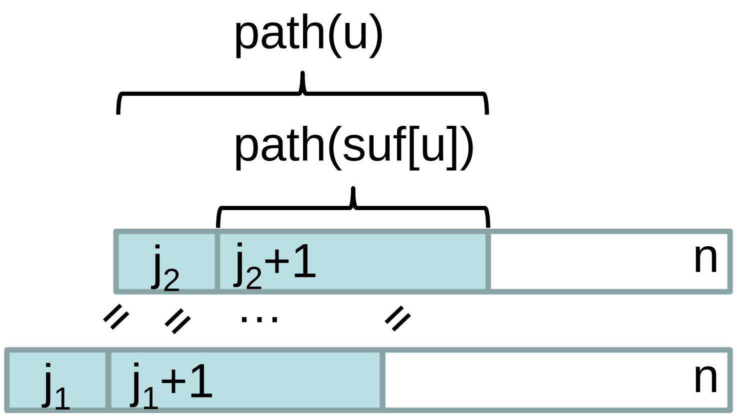
Lemma 7.25: If u is an inner node in T_{i-1} then suf[u] is an inner node in T_i.

Suffix Trees

Lemma 7.25: If u is an inner node in T_{i-1} then suf[u] is an inner node in T_i.

Proof:

- Suppose that u is an inner node in T_{i-1}.
- Then there are $j_1, j_2 < i$ with path(u) being the longest common prefix of $t[j_1..n]$ and $t[j_2..n]$.
- But then path(suf[u]) is the longest common prefix of $t[j_1+1..n]$ and $t[j_2+1..n]$, which implies that suf[u] is an inner node in T_i.



Suffix Trees

Recall the naive algorithm:

T_0:= suffix tree just consisting of the root
for i:=1 to n do
    T_i:=insert(T_{i-1},t[i..n])

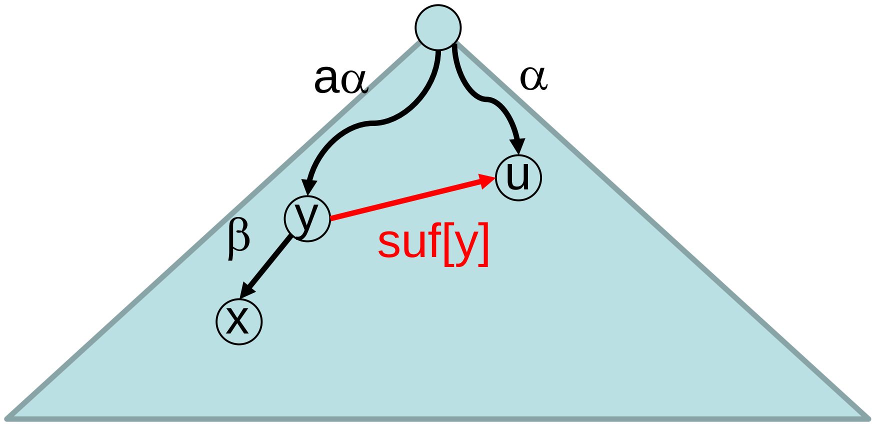
This is also the basic framework for the algorithm of McCreight, but the insertion of t[i..n] into T_{i-1} is performed differently from the standard insert:

- At the beginning of the i-th iteration, we assume that all nodes except for the node v with path(v)=head(i-1) have a suffix link.
- Given that the algorithm knows head(i-1) at the beginning of the i-th iteration, it will make use of the suffix links to efficiently locate head(i), which will allow it to insert t[i..n].
- This strategy is called Up-Link-Down.

Suffix Trees

Up-Link-Down Strategy:

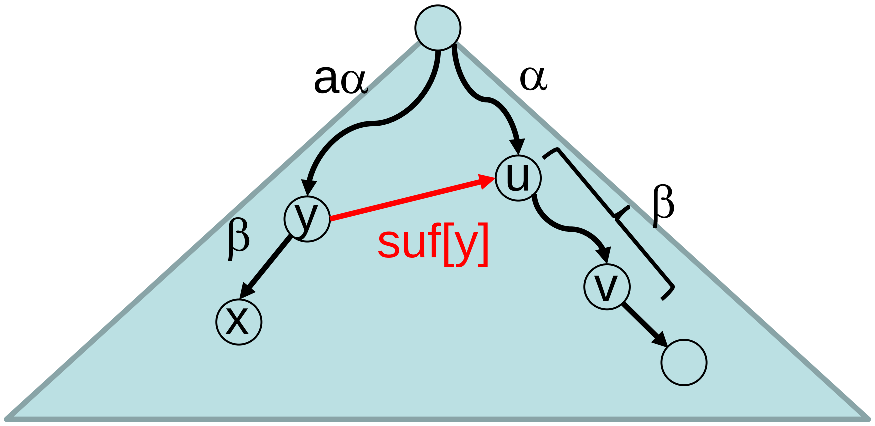
- Let x be the node in T_{i-1} with path(x)=head(i-1) and let y be the father of x. Suppose that head(i-1)=$a\alpha\beta$ with $a\in\Sigma$ and $\alpha,\beta\in\Sigma^*$, as shown in the figure.
- According to Lemma 7.23, we know that $\alpha\beta\in T_{i-1}$ and that head(i)=$\alpha\beta\gamma$ for some $\gamma\in\Sigma^*$.
- Since x does not have a suffix link, we go to y and use the suffix link from there. This leads to a node u with path(u)=α.



Suffix Trees

Up-Link-Down Strategy (continued):

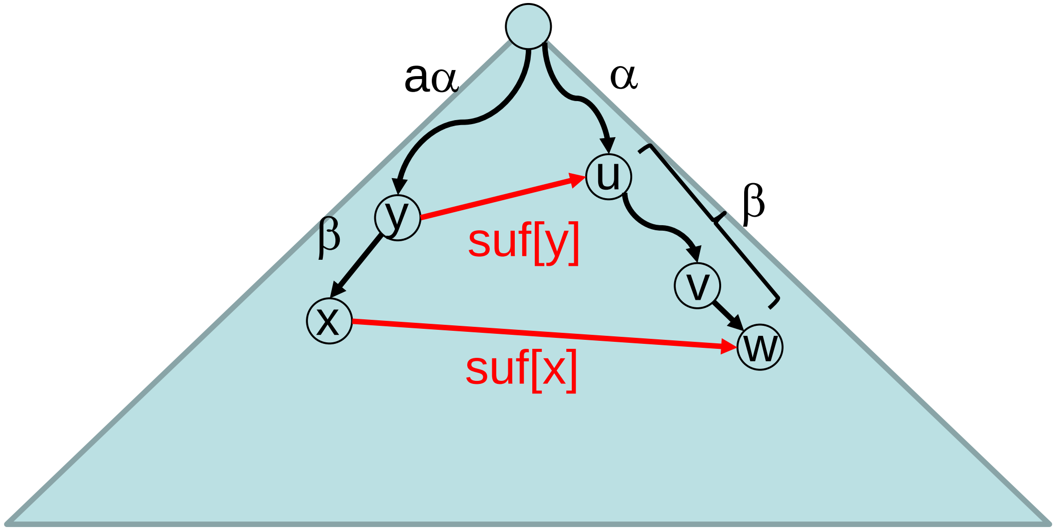
- We follow the links downwards from u till we reach the node v with path(v) being the longest prefix of $\alpha\beta$. Up to that node we only have to look at the first character of each edge (fastfind) since we know that $\alpha\beta \in T_{i-1}$.
- We can find out when we have reached v by looking at the length of the edge labels (if these are stored together with the labels).



Suffix Trees

Up-Link-Down Strategy (continued):

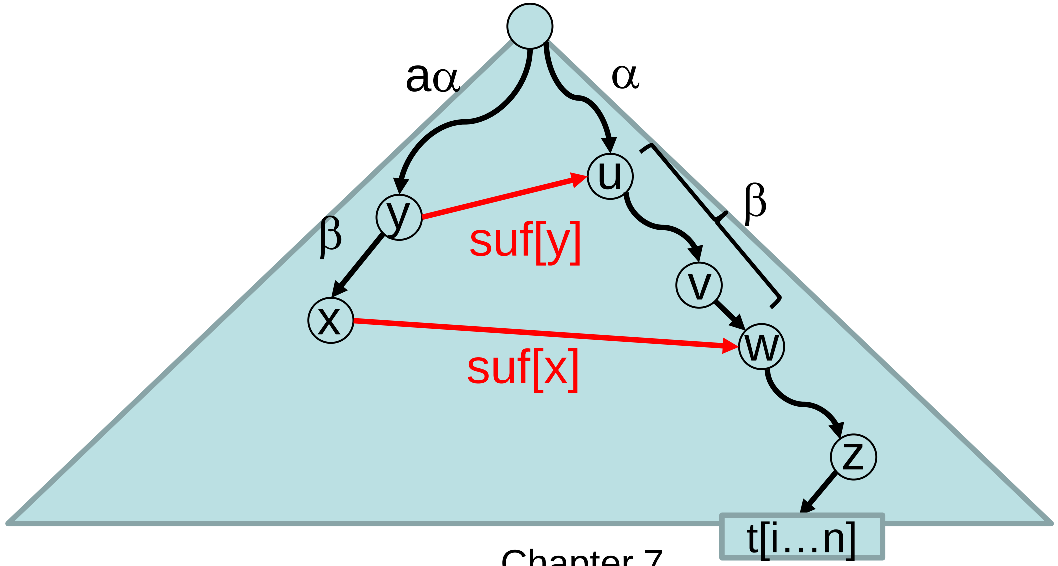
- If there is no node w yet with path(w)=$\alpha\beta$, we create a new node w at that location (by splitting an edge), so in any case we have reached a node w at the end with path(w)=$\alpha\beta$. Lemma 7.25 implies that in this case path(w)=head(i).
- Afterwards, we set suf[x] to w.



Suffix Trees

Up-Link-Down Strategy (continued):

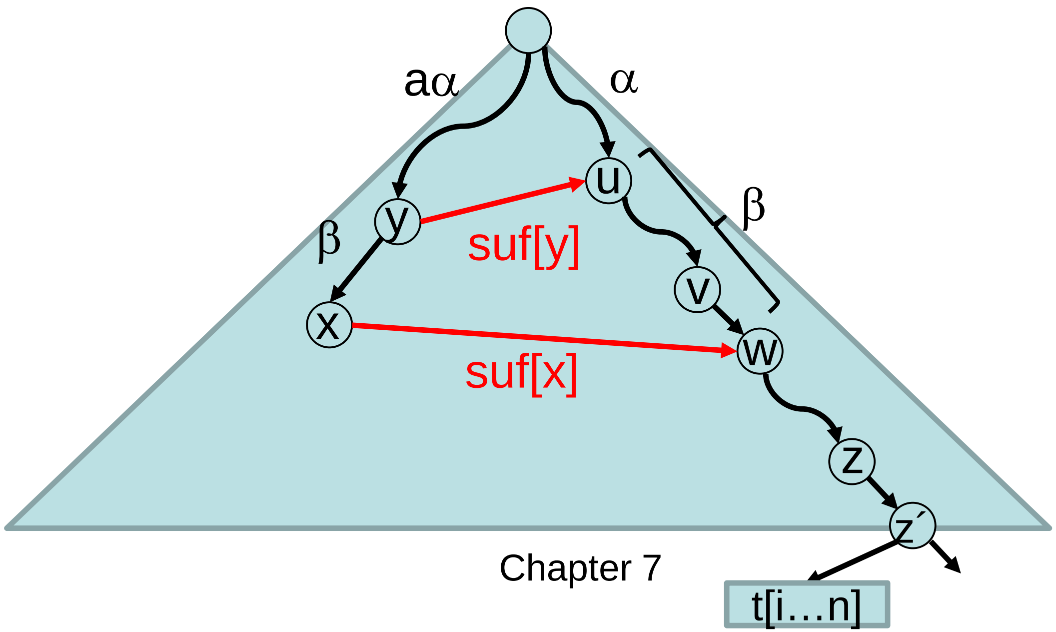
- If w already existed (so maybe path(w)≠head(i)), we follow the links downwards from w till we reach the node z with path(z) being the longest prefix of t[i…n]. Here, we have to look at the full edge labels, which is why we call this phase slowsearch.
- If path(z)=head(i), then we simply insert a new edge with label tail(i) into T_{i-1} leading to a new leaf representing t[i…n].



Suffix Trees

Up-Link-Down Strategy (continued):

- We follow the links downwards from w till we reach the node z with path(z) being the longest prefix of t[i…n].
- Otherwise, we insert a new node z´ with path(z´)=head(i) below z by splitting an edge and insert a new edge leaving z´ with label tail(i) that leads to a new leaf representing t[i…n].



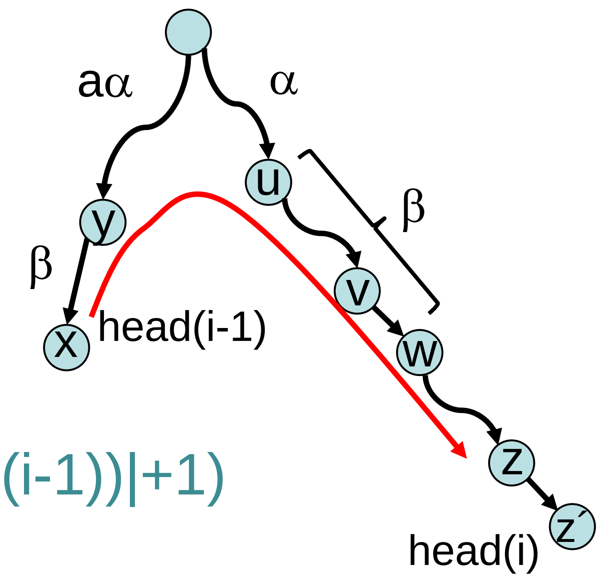
Suffix Trees

Theorem 7.25: The algorithm of McCreight can construct the suffix tree of a text t in time O(|t|).

Proof:

- The dominant parts of the runtime are the times needed for fastfind and slowfind.

Runtime of fastfind:



- The time needed is upper bounded by |father(head(i))|-|father(head(i-1))|+1, where |v| is the length of the path(v).
- Hence, the overall runtime for fastfind is at most

$$\Sigma_{i=1}^{n} (|father(head(i))|-|father(head(i-1))|+1)$$
$$\leq |father(head(n))| + n$$
$$= O(n)$$

Suffix Trees

Theorem 7.25: The algorithm of McCreight can construct the suffix tree of a text t in time O(|t|).

Proof:

- The dominant parts of the runtime are the times needed for fastfind and slowfind.

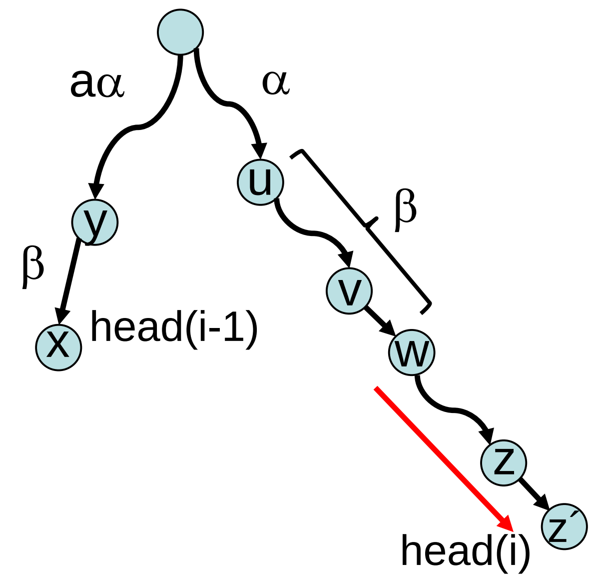
Runtime of slowfind:

- The time needed is proportional to

  |head(i)|-|head(i-1)|+1

- Hence, the overall runtime for slowfind is proportional to

$$\sum_{i=1}^{n} (|head(i)|-|head(i-1)|+1)$$
$$\leq |father(head(n))| + n$$
$$= O(n)$$



Suffix Trees

Remarks:

- Once we have built the suffix tree of t, we can search for any string s in t in time O(|s|).
- We can further accelerate that (in certain cases such as external memory) when transforming t's suffix tree into a hashed Patricia trie, which can be done in O(n) time.
- Then we only need O(log |s|) hash table lookups to find out whether s is a substring of t or not.