CS 240 – Data Structures and Data Management

Module 9e: String Matching - Enriched

T. Biedl E. Kondratovsky M. Petrick O. Veksler

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2022

version 2022−03−17 08:15

Outline

1. Pattern Matching - details
   - KMP failure function – fast computation
   - KMP failure function – improvement
   - Good suffix array

Outline

1. Pattern Matching - details
   - KMP failure function – fast computation
   - KMP failure function – improvement
   - Good suffix array

KMP failure function – fast computation

$F[j]$ is the length of the longest prefix of P that is a suffix of $P[1..j]$.

- How can we compute this faster?
- Recall property of KMP-automaton of P:
  - If we are in state ℓ, then we have just seen $P[0..\ell-1]$

  $\Leftrightarrow$ $P[0..\ell-1]$ is a suffix of what we have just parsed.

  - Also, KMP is always in the rightmost state where this holds.

  $\Leftrightarrow$ $P[0..\ell-1]$ is the *longest* suffix of what we have just parsed.

  $\Leftrightarrow$ ℓ is the length of the longest prefix of P that is a suffix of what we have just parsed.

KMP failure function – fast computation

$F[j]$ is the length of the longest prefix of P that is a suffix of $P[1..j]$.

- How can we compute this faster?
- Recall property of KMP-automaton of P:
  - If we are in state ℓ, then we have just seen $P[0..\ell-1]$

  $\Leftrightarrow$ $P[0..\ell-1]$ is a suffix of what we have just parsed.

  - Also, KMP is always in the rightmost state where this holds.

  $\Leftrightarrow$ $P[0..\ell-1]$ is the *longest* suffix of what we have just parsed.

  $\Leftrightarrow$ ℓ is the length of the longest prefix of P
  that is a suffix of what we have just parsed.

Combine this with the definition of $F[j]$ to get:

$F[j] = \ell \Leftrightarrow$
we reach state ℓ when parsing $P[1..j]$ on the KMP-automaton for P

KMP failure function – fast computation

$F[j] =$ **the state we reach when parsing** $P[1..j]$

This immediately gives algorithm: For $j = 1, 2, \ldots,$

- parse $P[1..j]$ on the KMP-automaton for P
- Set $F[j] = \ell$ if we reach state ℓ

KMP failure function – fast computation

$F[j]$ = the state we reach when parsing $P[1..j]$

This immediately gives algorithm: For $j = 1, 2, \ldots,$

- parse $P[1..j]$ on the KMP-automaton for P
- Set $F[j] = \ell$ if we reach state ℓ

Observe: We don't need to re-start the parsing from scratch!

- Assume we have computed $F[j]$ already.
- To compute $F[j{+}1]$, parse $P[j{+}1]$ and note reached state.
- So can compute $F[0..m{-}1]$ with *one* parse of $P[1..m{-}1]$

KMP failure function – fast computation

$F[j]$ = **the state we reach when parsing** $P[1..j]$

This immediately gives algorithm: For $j = 1, 2, \ldots,$

- parse $P[1..j]$ on the KMP-automaton for P
- Set $F[j] = \ell$ if we reach state ℓ

Observe: We don't need to re-start the parsing from scratch!

- Assume we have computed $F[j]$ already.
- To compute $F[j+1]$, parse $P[j+1]$ and note reached state.
- So can compute $F[0..m-1]$ with *one* parse of $P[1..m-1]$

But isn't this circular?

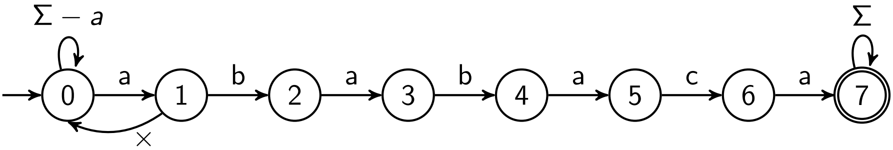
- We need failure-arcs for parsing, but we compute them only now!
- But: To compute $F[j]$, parse $P[1..j-1]$ first ($j-1$ characters)
  $\Rightarrow$ reach state $\leq j$
  $\Rightarrow$ don't need $F[j]$ (= arc from state $j+1$) to parse $P[j]$

Computing the Failure Array

```
KMP::failureArray(P)
P: String of length m (pattern)
    F[0] ← 0
    j ← 1            // index within parsed text
    ℓ ← 0            // reached state
    while j < m do
        if P[j] = P[ℓ]
            ℓ ← ℓ + 1
            F[j] ← ℓ
            j ← j + 1
        else if ℓ > 0
            ℓ ← F[ℓ − 1]
        else
            F[j] ← 0
            j ← j + 1
```

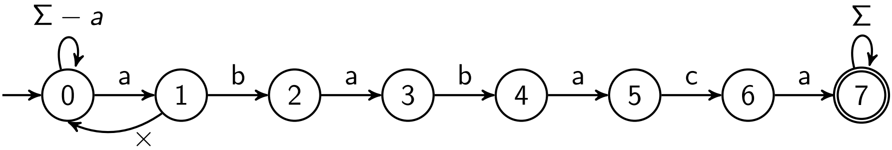
Correctness-idea: $F[j]$ is defined via pattern matching of P in $P[1..j]$. So KMP uses itself! Already-built parts of $F[\cdot]$ are used to expand it.

KMP failure function – fast computation



Parse $P[1..m-1]$ = `babaca` while adding failure-arcs:

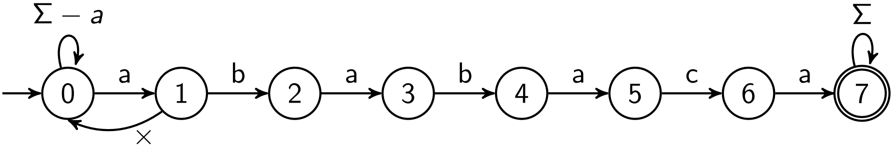
KMP failure function – fast computation



Parse $P[1..m-1]$ = `babaca` while adding failure-arcs:

j	1																
$P[i]$																	
$P[j]$																	
ℓ	0																
$F[j]$																	

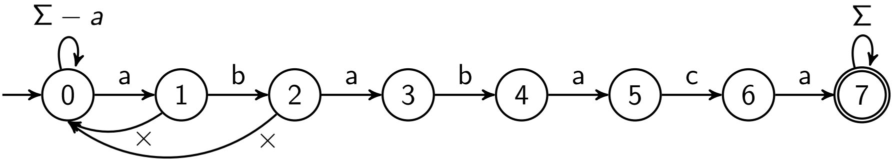
KMP failure function – fast computation



Parse $P[1..m-1]$ = `babaca` while adding failure-arcs:

j	1	1														
$P[i]$		b														
$P[j]$		a														
ℓ	0	$\xrightarrow{b}$														
$F[j]$																

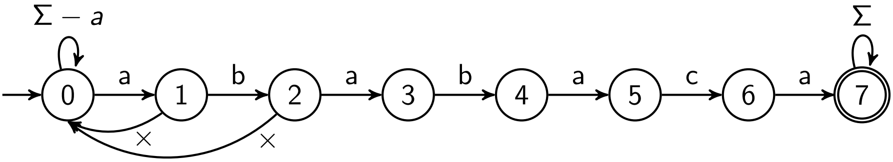
KMP failure function – fast computation



Parse $P[1..m-1]$ = `babaca` while adding failure-arcs:

j	1	1	1														
$P[i]$		b															
$P[j]$		a															
ℓ	0	$\xrightarrow{b}$	0														
$F[j]$			0														

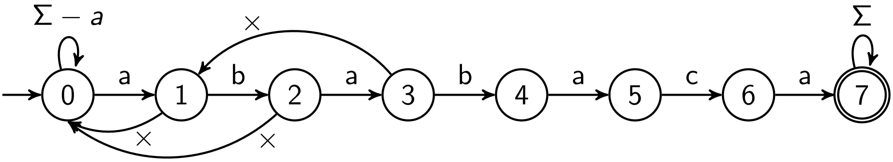
KMP failure function – fast computation



Parse $P[1..m{-}1]$ = `babaca` while adding failure-arcs:

j	1	1	1	2													
$P[i]$		b		a													
$P[j]$		a		a													
ℓ	0	$\stackrel{b}{\to}$	0														
$F[j]$			0														

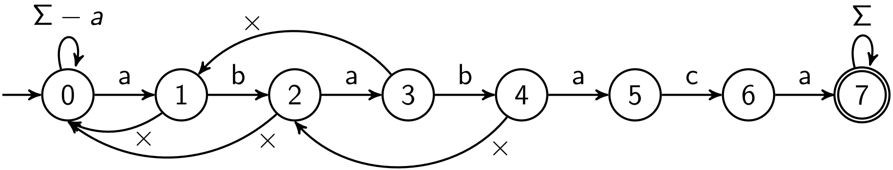
KMP failure function – fast computation



Parse $P[1..m{-}1]$ = `babaca` while adding failure-arcs:

j	1	1	1	2	2												
$P[i]$		b		a													
$P[j]$		a		a													
ℓ	0	$\xrightarrow{b}$	0	$\xrightarrow{a}$	1												
$F[j]$			0		1												

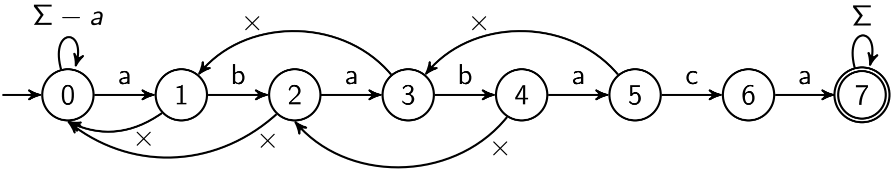
KMP failure function – fast computation



Parse $P[1..m-1] =$ `babaca` while adding failure-arcs:

j	1	1	1	2	2	3	3									
$P[i]$		b		a		b										
$P[j]$		a		a		b										
ℓ	0	$\xrightarrow{b}$	0	$\xrightarrow{a}$	1	$\xrightarrow{b}$	2									
$F[j]$			0		1		2									

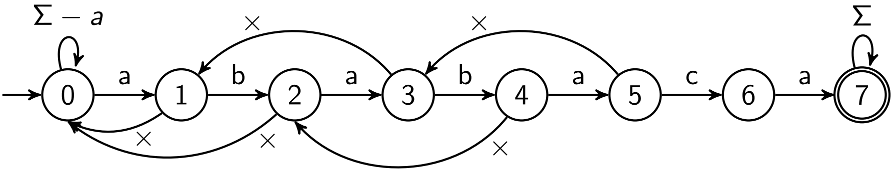
KMP failure function – fast computation



Parse $P[1..m{-}1]$ = `babaca` while adding failure-arcs:

j	1	1	1	2	2	3	3	4	4								
$P[i]$		b		a		b		a									
$P[j]$		a		a		b		a									
ℓ	0	$\xrightarrow{b}$	0	$\xrightarrow{a}$	1	$\xrightarrow{b}$	2	$\xrightarrow{a}$	3								
$F[j]$			0		1		2		3								

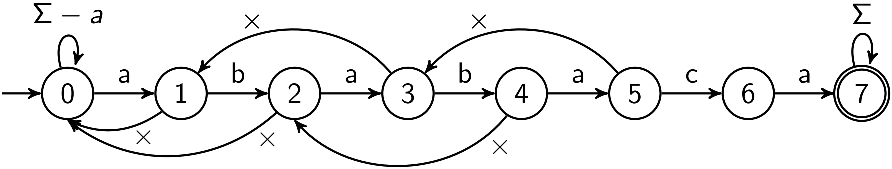
KMP failure function – fast computation



Parse $P[1..m-1]$ = babaca while adding failure-arcs:

j	1	1	1	2	2	3	3	4	4	5						
$P[i]$		b		a		b		a		c						
$P[j]$		a		a		b		a		b						
ℓ	0	$\xrightarrow{b}$	0	$\xrightarrow{a}$	1	$\xrightarrow{b}$	2	$\xrightarrow{a}$	3	$\xrightarrow{\times}$						
$F[j]$			0		1		2		3							

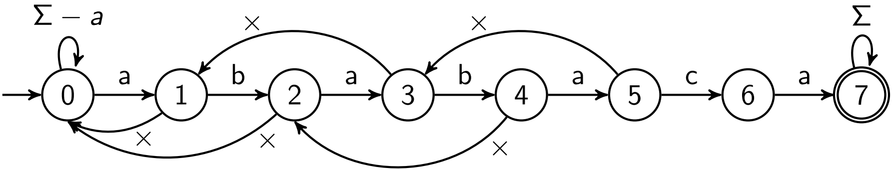
KMP failure function – fast computation



Parse $P[1..m-1]$ = babaca while adding failure-arcs:

j	1	1	1	2	2	3	3	4	4	5	5						
$P[i]$		b		a		b		a		c							
$P[j]$		a		a		b		a		b							
ℓ	0	$\xrightarrow{b}$	0	$\xrightarrow{a}$	1	$\xrightarrow{b}$	2	$\xrightarrow{a}$	3	$\xrightarrow{\times}$	1						
$F[j]$			0		1		2		3								

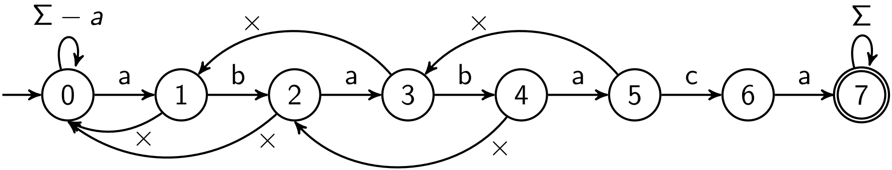
KMP failure function – fast computation



Parse $P[1..m{-}1]$ = `babaca` while adding failure-arcs:

j	1	1	1	2	2	3	3	4	4	5	5	5					
$P[i]$		b		a		b		a		c		c					
$P[j]$		a		a		b		a		b		b					
ℓ	0	$\xrightarrow{b}$	0	$\xrightarrow{a}$	1	$\xrightarrow{b}$	2	$\xrightarrow{a}$	3	$\xrightarrow{\times}$	1	$\xrightarrow{\times}$					
$F[j]$			0		1		2		3								

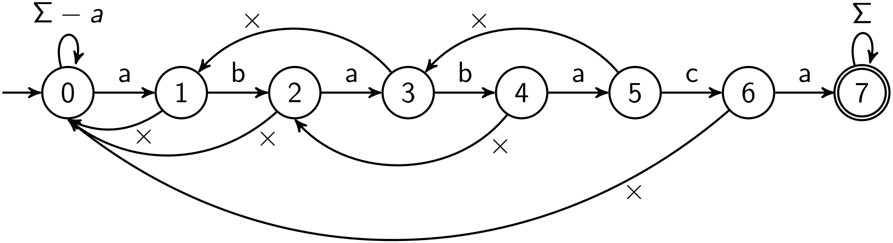
KMP failure function – fast computation



Parse $P[1..m-1]$ = `babaca` while adding failure-arcs:

j	1	1	1	2	2	3	3	4	4	5	5	5	5				
$P[i]$		b		a		b		a		c		c					
$P[j]$		a		a		b		a		b		b					
ℓ	0	$\xrightarrow{b}$	0	$\xrightarrow{a}$	1	$\xrightarrow{b}$	2	$\xrightarrow{a}$	3	$\xrightarrow{\times}$	1	$\xrightarrow{\times}$	0				
$F[j]$			0		1		2		3								

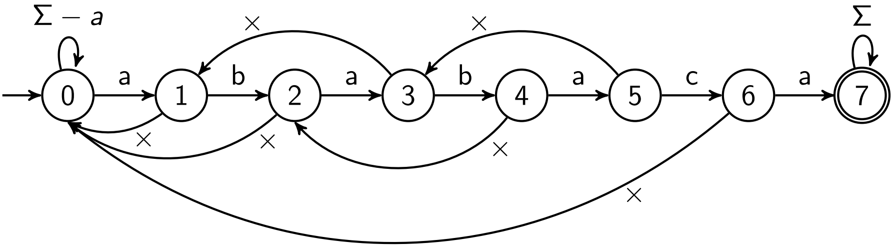
KMP failure function – fast computation



Parse $P[1..m{-}1]$ = `babaca` while adding failure-arcs:

j	1	1	1	2	2	3	3	4	4	5	5	5	5	5	5		
$P[i]$		b		a		b		a		c		c		c			
$P[j]$		a		a		b		a		b		b		a			
ℓ	0	$\xrightarrow{b}$	0	$\xrightarrow{a}$	1	$\xrightarrow{b}$	2	$\xrightarrow{a}$	3	$\xrightarrow{\times}$	1	$\xrightarrow{\times}$	0	$\xrightarrow{c}$	0		
$F[j]$			0		1		2		3						0		

KMP failure function – fast computation



Parse $P[1..m-1]$ = `babaca` while adding failure-arcs:

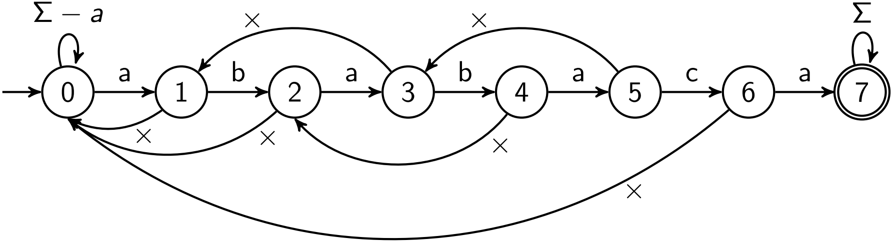
j	1	1	1	2	2	3	3	4	4	5	5	5	5	5	5	6	6
$P[i]$		b		a		b		a		c		c		c		a	
$P[j]$		a		a		b		a		b		b		a		a	
ℓ	0	$\xrightarrow{b}$	0	$\xrightarrow{a}$	1	$\xrightarrow{b}$	2	$\xrightarrow{a}$	3	$\xrightarrow{\times}$	1	$\xrightarrow{\times}$	0	$\xrightarrow{c}$	0	$\xrightarrow{a}$	1
$F[j]$			0		1		2		3						0		1

Outline

1. Pattern Matching - details
   - KMP failure function – fast computation
   - KMP failure function – improvement
   - Good suffix array

KMP failure function – improvement

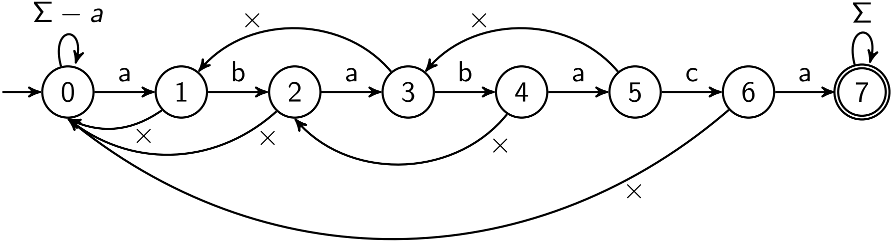
We can define an even better failure-function:



Consider failure-arc from state 4:

KMP failure function – improvement

We can define an even better failure-function:

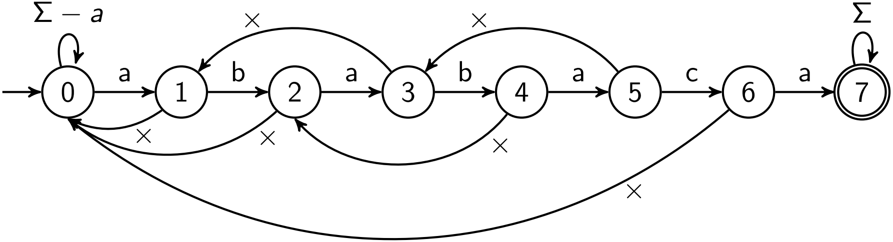


Consider failure-arc from state 4:

- This will be used if $T[i] \neq a = P[4]$ and leads to state 2.

KMP failure function – improvement

We can define an even better failure-function:

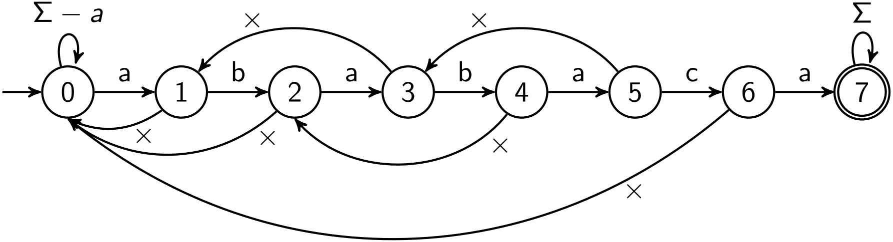


Consider failure-arc from state 4:

- This will be used if $T[i] \neq a = P[4]$ and leads to state 2.
- The next check will again compare $T[i]$ to $a = P[2]$.

KMP failure function – improvement

We can define an even better failure-function:

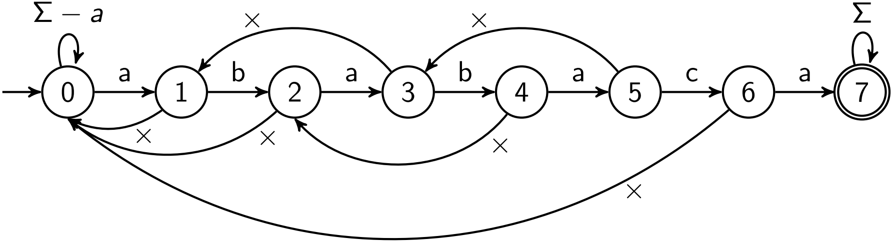


Consider failure-arc from state 4:

- This will be used if $T[i] \neq a = P[4]$ and leads to state 2.
- The next check will again compare $T[i]$ to $a = P[2]$.
- This *must* fail, and the failure-arc will lead to state 0.

KMP failure function – improvement

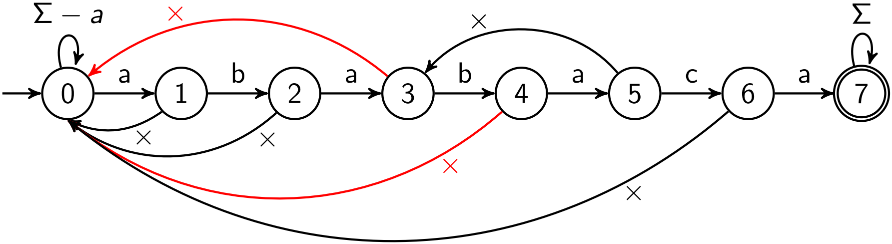
We can define an even better failure-function:



Consider failure-arc from state 4:

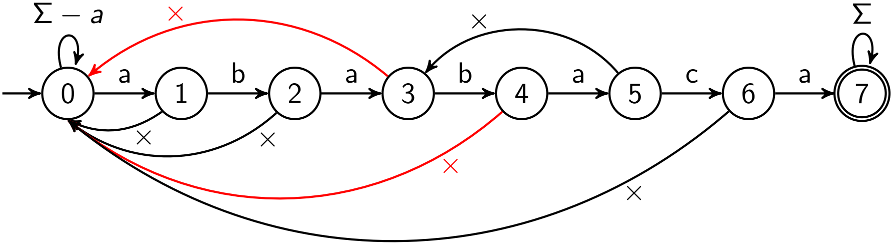
- This will be used if $T[i] \neq a = P[4]$ and leads to state 2.
- The next check will again compare $T[i]$ to $a = P[2]$.
- This *must* fail, and the failure-arc will lead to state 0.
- We might as well have gone to state 0 directly.

KMP failure function – improvement



$$F^+[j] = \begin{cases} \text{length } \ell \text{ of the longest prefix of } P \text{ that is a suffix of } P[1..j] \\ \qquad\qquad\qquad\qquad \textbf{and where } \mathbf{P}[\ell] \neq \mathbf{P}[\mathbf{j}+\mathbf{1}]. \\ 0 \text{ if no such } \ell \text{ exists} \end{cases}$$

KMP failure function – improvement



$$F^+[j] = \begin{cases} \text{length } \ell \text{ of the longest prefix of } P \text{ that is a suffix of } P[1..j] \\ \qquad\qquad\qquad\qquad \textbf{and where } \mathbf{P}[\ell] \neq \mathbf{P}[\mathbf{j}+\mathbf{1}]. \\ 0 \text{ if no such } \ell \text{ exists} \end{cases}$$

Easy to compute: $F^+[j] = \begin{cases} F[j] & \text{if } P[j+1] \neq P[F[j]] \text{ or } F[j]=0 \\ F^+[F[j]-1] & \text{otherwise} \end{cases}$

Outline

1. Pattern Matching - details
   - KMP failure function – fast computation
   - KMP failure function – improvement
   - Good suffix array

Good suffix array - example

$P =$onobobo

n	o	o	o	b	o	o	o	o	o	b	n	b	b	o	b	o	...	...
			b	o	b	o												

Good suffix array - example

P =onobobo

n	o	o	o	b	o	o	o	o	o	b	n	b	b	o	b	o	…	…
			b	o	b	o												

- Do smallest shift so that obo fits in the new guess

Good suffix array - example

P =onobobo

n	o	o	o	b	o	o	o	o	o	b	n	b	b	o	b	o	…	…
			b	o	b	o												
				(o)	(b)	(o)												

- Do smallest shift so that obo fits in the new guess

Good suffix array - example

$P =$onobobo

n	o	o	o	b	o	o	o	o	o	b	n	b	b	o	b	o	...	...
			b	o	b	o												
				(o)	(b)	(o)	b	o										

- Do smallest shift so that obo fits in the new guess
- Do smallest shift so that matched suffix fits in the new guess

Good suffix array - example

$P =$onobobo

n	o	o	o	b	o	o	o	o	o	b	n	b	b	o	b	o	...	...
			b	o	b	o												
				(o)	(b)	(o)	b	o										
								(o)		o								

- Do smallest shift so that obo fits in the new guess
- Do smallest shift so that matched suffix fits in the new guess
- No suffix matched $\rightsquigarrow$ shift over by one

Good suffix array - example

$P =$onobobo

n	o	o	o	b	o	o	o	o	o	b	n	b	b	o	b	o	...	...
			b	o	b	o												
				(o)	(b)	(o)	b	o										
								(o)		o								
										[b]	o							

- Do smallest shift so that obo fits in the new guess
- Do smallest shift so that matched suffix fits in the new guess
- No suffix matched $\rightsquigarrow$ shift over by one (or by last-char heuristic)

Good suffix array - example

$P =$onobobo

n	o	o	o	b	o	o	o	o	o	b	n	b	b	o	b	o	...	...
			b	o	b	o												
				(o)	(b)	(o)	b	o										
								(o)		o								
										[b]	o							
											[n]	(o)	b	o	b	o		

- Do smallest shift so that obo fits in the new guess
- Do smallest shift so that matched suffix fits in the new guess
- No suffix matched $\rightsquigarrow$ shift over by one (or by last-char heuristic)

Good suffix array - example

P =onobobo

n	o	o	o	b	o	o	o	o	o	b	n	b	b	o	b	o	...	...
			b	o	b	o												
				(o)	(b)	(o)	b	o										
								(o)		o								
										[b]	o							
											[n]	(o)	b	o	b	o		

- Do smallest shift so that obo fits in the new guess
- Do smallest shift so that matched suffix fits in the new guess
- No suffix matched ⇝ shift over by one (or by last-char heuristic)
- What to do if the matched part does not repeat?

Good suffix array - if matched part doesn't repeat

$P =$ nbonnnbo (different from before)

n	b	b	n	n	n	b	o	o	n	n	b	o	b	b	o	b	o	...	...
		o	n	n	n	b	o												

- Cannot match all of nnnbo

Good suffix array - if matched part doesn't repeat

$P =$nbonnnbo (different from before)

n	b	b	n	n	n	b	o	o	n	n	b	o	b	b	o	b	o	...	...
		o	n	n	n	b	o												
					(n)	(b)	(o)												

- Cannot match all of nnnbo
- But nbo fits a prefix of P $\rightsquigarrow$ shift to that guess

Good suffix array - if matched part doesn't repeat

$P =$nbonnnbo (different from before)

n	b	b	n	n	n	b	o	o	n	n	b	o	b	b	o	b	o	...	...
		o	n	n	n	b	o												
					(n)	(b)	(o)	n	n	n	b	o							
										(n)	(b)	(o)							

- Cannot match all of nnnbo
- But nbo fits a prefix of P $\rightsquigarrow$ shift to that guess
- Generally: Re-use longest suffix of matched part that fits a prefix of P

Good suffix array - if matched part doesn't repeat

$P =$nbonnnbo (different from before)

n	b	b	n	n	n	b	o	o	n	n	b	o	b	b	o	b	o	…	…
		o	n	n	n	b	o												
					(n)	(b)	(o)	n	n	n	b	o							
										(n)	(b)	(o)			n	b	o		

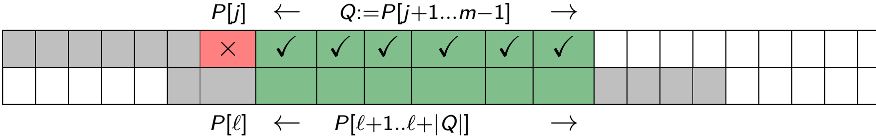
- Cannot match all of nnnbo
- But nbo fits a prefix of $P \rightsquigarrow$ shift to that guess
- Generally: Re-use longest suffix of matched part that fits a prefix of P
- If nothing fits: Shift guess all the way past previous guess.

$P =$nobnnnbo (different from before)

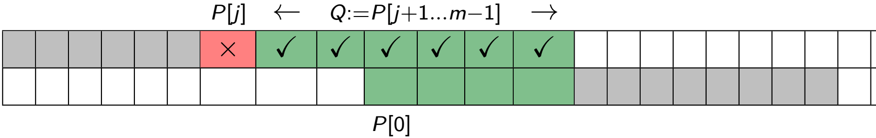
n	b	b	n	n	n	b	o	o	n	n	b	o	b	b	o	b	o	…	…
		o	n	n	n	b	o												

Definition of good suffix array

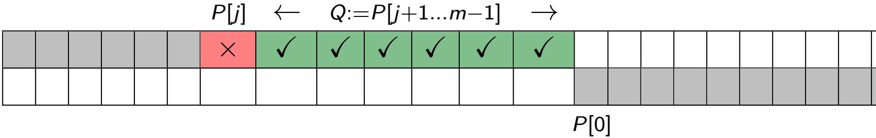
- Assume search failed at $P[j]$, but had matched $P[j+1..m-1] =: Q$
- Case 1: Q appears as substring of P elsewhere



- Case 2: A suffix of Q is a prefix of P.

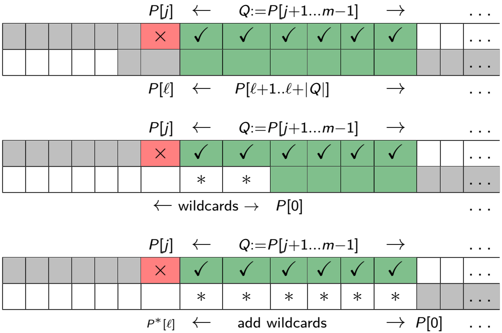


- Case 3: Neither (i.e., only empty suffix fits).



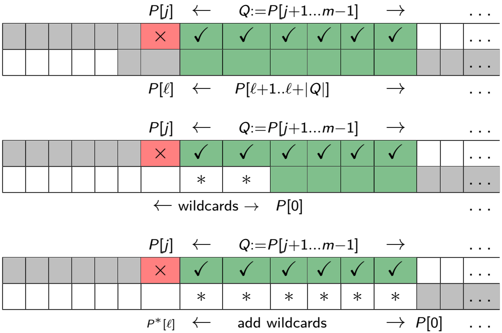
Definition of good suffix array

- Can unify all three cases into one!
- Let P^* be P with m *wildcards* attached in front.



Definition of good suffix array

- Can unify all three cases into one!
- Let P^* be P with m *wildcards* attached in front.

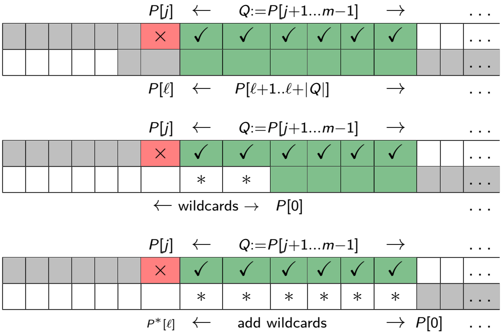


In all cases:

- Q is a substring of P^*

Definition of good suffix array

- Can unify all three cases into one!
- Let P^* be P with m *wildcards* attached in front.

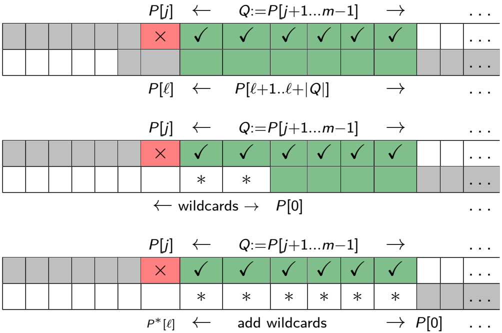


In all cases:

- Q is a substring of P^*
- Align old $P[j]$ with new $P[\ell]$ (then $S[j] \leftarrow \ell$ fits update)
- So Q is prefix of $P^*[\ell{+}1 \ldots m{-}1]$

Definition of good suffix array

- Can unify all three cases into one!
- Let P^* be P with m *wildcards* attached in front.



In all cases:

- Q is a substring of P^*
- Align old $P[j]$ with new $P[\ell]$ (then $S[j] \leftarrow \ell$ fits update)
- So Q is prefix of $P^*[\ell{+}1...m{-}1]$
- Want $\ell \neq j$ so that we actually shift

$$S[j] = \max_{\ell \neq j} \quad P[j{+}1..m{-}1] \textbf{ is a prefix of } P^*[\ell{+}1..m{-}1]$$

Good Suffix Array Computation - human

$$
\begin{aligned}
S[j] &= \max_{\ell \neq j} \quad P[j{+}1..m{-}1] \text{ is a prefix of } P^*[\ell{+}1..m{-}1] \\
&= \max_{\ell} \quad P[j{+}1..m{-}1] \text{ is a prefix of } P^*[\ell{+}1..m{-}2]
\end{aligned}
$$

$P =$ boobobo		$P^*[-m..m-2]$														
		-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5		
j	$P[j{+}1..m{-}1]$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	b	o	o	b	o	b	$\ell+1$	$S[j]$
5	o												o		4	3
4	bo											b	o		3	2
3	obo										o	b	o		2	1
2	bobo						b	o	b	o					-2	-3
1	obobo					o	b	o	b	o					-3	-4
0	oobobo				o	o	b	o	b	o					-4	-5

Easy to compute in polynomial time:

- Write down P^*, omitting rightmost character.
- For each j, write down $P[j{+}1..m{-}1]$
- Find rightmost match $\rightsquigarrow$ gives $\ell+1$ $\rightsquigarrow$ gives $S[j]$

Good Suffix Array Computation - computer

Idea: After reformulations, this resembles the KMP failure function!

$$S[j] = \max_{\ell \neq j}\{\ P[j{+}1..m{-}1] \text{ is a prefix of } P^*[\ell{+}1..m{-}1]\ \}$$

$$= \max_{\ell}\{\ P[j{+}1..m{-}1] \text{ is a prefix of } P^*[\ell{+}1..m{-}2]\ \}$$

Good Suffix Array Computation - computer

Idea: After reformulations, this resembles the KMP failure function!

$$S[j] = \max_{\ell \neq j} \{ \ P[j{+}1..m{-}1] \text{ is a prefix of } P^*[\ell{+}1..m{-}1] \ \}$$

$$= \max_{\ell} \{ \ P[j{+}1..m{-}1] \text{ is a prefix of } P^*[\ell{+}1..m{-}2] \ \}$$

$$= \max_{\ell} \{ \ (P[j{+}1..m{-}1])^{\text{reverse}} \text{ is a suffix of } (P^*[\ell{+}1..m{-}2])^{\text{reverse}} \ \}$$

Good Suffix Array Computation - computer

Idea: After reformulations, this resembles the KMP failure function!

$$S[j] = \max_{\ell \neq j}\{ \; P[j{+}1..m{-}1] \text{ is a prefix of } P^*[\ell{+}1..m{-}1] \; \}$$

$$= \max_{\ell}\{ \; P[j{+}1..m{-}1] \text{ is a prefix of } P^*[\ell{+}1..m{-}2] \; \}$$

$$= \max_{\ell}\{ \; (P[j{+}1..m{-}1])^{\text{reverse}} \text{ is a suffix of } (P^*[\ell{+}1..m{-}2])^{\text{reverse}} \; \}$$

(define R to be reverse of P^*: $R[j] = P^*[m{-}1{-}j]$)

$$= \max_{\ell}\{ \; R[0..m{-}j{-}2] \text{ is a suffix of } R[1..m{-}\ell{-}2] \; \}$$

Good Suffix Array Computation - computer

Idea: After reformulations, this resembles the KMP failure function!

$$S[j] = \max_{\ell \neq j}\{\ P[j{+}1..m{-}1] \text{ is a prefix of } P^*[\ell{+}1..m{-}1]\ \}$$

$$= \max_{\ell}\{\ P[j{+}1..m{-}1] \text{ is a prefix of } P^*[\ell{+}1..m{-}2]\ \}$$

$$= \max_{\ell}\{\ (P[j{+}1..m{-}1])^{\text{reverse}} \text{ is a suffix of } (P^*[\ell{+}1..m{-}2])^{\text{reverse}}\ \}$$

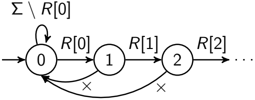
(define R to be reverse of P^*: $R[j] = P^*[m{-}1{-}j]$)

$$= \max_{\ell}\{\ \underbrace{R[0..m{-}j{-}2]}_{\text{prefix of } R} \text{ is a suffix of } R[1..\underbrace{m{-}\ell{-}2}_{k}]\ \}$$

$$= m - 2 - \min_{k}\{\ \text{the } (m{-}j{-}1)^{\text{st}} \text{ prefix of } R \text{ is a suffix of } R[1..k]\ \}$$

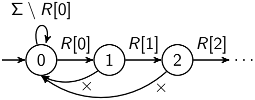
This should remind you of properties of a KMP-automaton.

Good Suffix Array Computation - computer

Recall KMP-automaton for R: 

We reach state q if the q^{th} prefix of R was a suffix of what was parsed.

Good Suffix Array Computation - computer

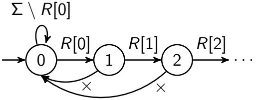
Recall KMP-automaton for R: 

We reach state q if the q^{th} prefix of R was a suffix of what was parsed.

$$S[j] \;=\; m-2-\min_{k}\{\text{ the } (m-j-1)^{\text{st}} \text{ prefix of } R \text{ is a suffix of } R[1..k]\ \}$$

Good Suffix Array Computation - computer

Recall KMP-automaton for R:

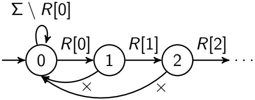


We reach state q if the q^{th} prefix of R was a suffix of what was parsed.

$$S[j] = m-2-\min_k\{\text{ the } (m-j-1)^{\text{st}} \text{ prefix of } R \text{ is a suffix of } R[1..k]\ \}$$

$$= m-2-\min_k\{\text{ state } m-j-1 \text{ is reached when } \underbrace{\text{parsing}}_{\text{on KMP-automaton for } R}\ R[1..k]\ \}$$

Good Suffix Array Computation - computer

Recall KMP-automaton for R: 

We reach state q if the q^{th} prefix of R was a suffix of what was parsed.

$$S[j] \;=\; m{-}2{-}\min_k\{\text{ the } (m{-}j{-}1)^{\text{st}} \text{ prefix of } R \text{ is a suffix of } R[1..k]\ \}$$

$$=\; m{-}2{-}\min_k\{\text{ state } \underbrace{m{-}j{-}1}_{q} \text{ is reached when } \underbrace{\text{parsing}}_{\text{on KMP-automaton for } R}\ R[1..k]\ \}$$

$$S[m{-}q{-}1] \;=\; m{-}2{-}\min_k\{\text{ state } q \text{ reached when parsing } R[1..k]\ \}$$

Good Suffix Array Computation - computer

Final result:

$$S[m-q-1] \quad = \quad m-2-\min_k \left\{ \begin{array}{l} \text{parsing } R[1..k] \text{ on the KMP-automaton for} \\ R = P^{\text{reverse}} *** \ldots * \text{ brings us to state } q \end{array} \right\}$$

Good Suffix Array Computation - computer

Final result:

$$S[m-q-1] \;=\; m-2-\min_k \left\{ \begin{array}{l} \text{parsing } R[1..k] \text{ on the KMP-automaton for} \\ R = P^{\text{reverse}} {*}{*}{*} \ldots {*} \text{ brings us to state } q \end{array} \right\}$$

So to compute $S[\cdot]$:

- Create KMP-automaton $\mathcal{K}_R$ for $R := P^{\text{reverse}} {*}{*}{*} \ldots {*}$
- Parse $R[1..2m-1]$ on $\mathcal{K}_R$
- Whenever we reach a state q
  - Check whether q was visited already
  - If not: $S[m-q-1] \leftarrow m-2-k$, where $R[k]$ is last parsed character.

Run-time: $O(m)$ since R has $O(m)$ characters.