Contents

Section outline

- String search
- Brute force approach

String search

- Given a pattern string p, find first match in text t
- N: # characters in text
- M: # characters in pattern
- Length of pattern is small compared to the length of the text ($N \gg M$)
- Pattern can be pre-preprocessed
- Text cannot be pre-processed

Example

Search Text, $N = 21$

n	n	e	e	n	l	e	d	e	n	e	e	n	e	e	d	l	e	n	l	d

Search Pattern, $M = 6$

n	e	e	d	l	e

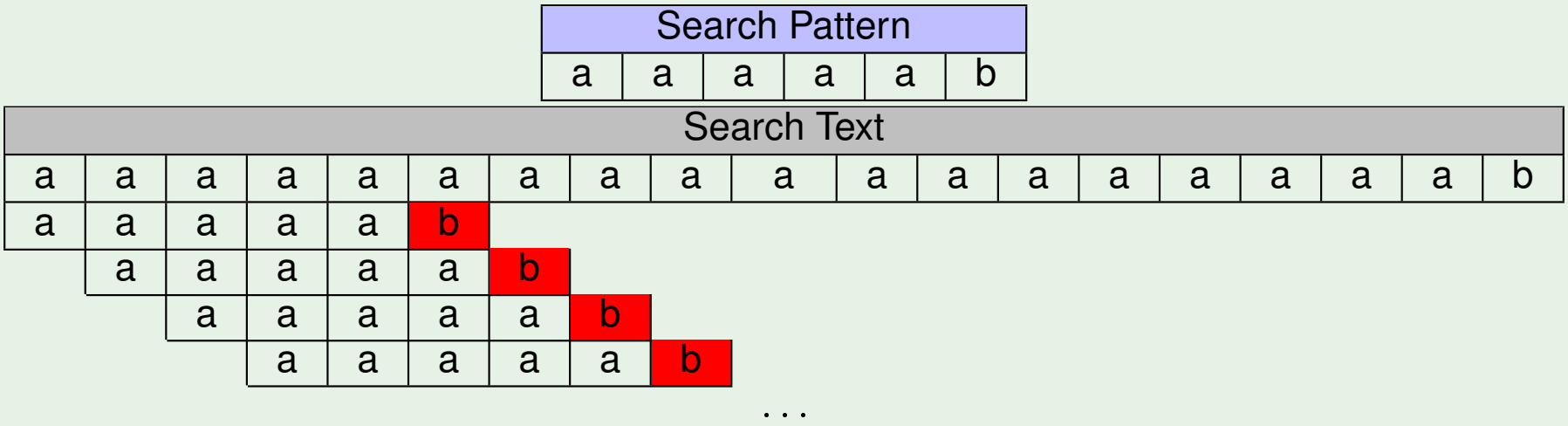
Successful search

n	n	e	e	n	l	e	d	e	n	e	e	n	e	e	d	l	e	n	l	d

Brute force approach

- Check for pattern starting at every text position
- Running time depends on pattern and text
- Worst case: *MN* comparisons, in practice almost linear
- Slow if M and N are large, and have lots of repetition

Example (Worst case of brute force approach)



Section outline

2 **Karp-Rabin fingerprint algorithm**

- String matching using hashing
- Efficient hash computation

String matching using hashing

Example

Search pattern				
5	9	2	6	5

59265 % 97 = 95

Search Text																	
3	1	4	1	5	9	2	6	5	3	5	8	9	7	3	3	4	6

31415 % 97 = 84

14159 % 97 = 94

41592 % 97 = 76

15926 % 97 = 18

. . .

- Match not possible unless computed hash of text substring under consideration matches hash of search string
- In case of match, actual comparison is needed to confirm match

Efficient hash computation

Example

- Pre-compute: 10000 % 97 = 9
- First hash: 31415 % 97 = 84
- . . .
- Previous hash: 41592 % 97 = 76
- Efficient next hash computation

  of 15926 (% 97):

$$
\begin{aligned}
&= (41592 - (4 \times 10000)) \times 10 + 6 \\
&= (76 - (4 \times 9)) \times 10 + 6 \\
&= 406 \\
&= 18
\end{aligned}
$$

Efficient hash computation

Example

- Pre-compute: 10000 % 97 = 9
- First hash: 31415 % 97 = 84
- ...
- Previous hash: 41592 % 97 = 76
- Efficient next hash computation

  of 15926 (% 97):

$$
\begin{aligned}
&= (41592 - (4 \times 10000)) \times 10 + 6 \\
&= (76 - (4 \times 9)) \times 10 + 6 \\
&= 406 \\
&= 18
\end{aligned}
$$

- Choose modulus to be a large prime (q)
- Each window of M expected to be uniformly distributed in $[0, q-1]$
- Expected running time is $O\left(N + M\left(\frac{1}{q}(N-M)\right)\right)$
- Worst case: $\Theta(MN)$ – when?

Efficient hash computation

Example

- Pre-compute: 10000 % 97 = 9
- First hash: 31415 % 97 = 84
- ...
- Previous hash: 41592 % 97 = 76
- Efficient next hash computation

  of 15926 (% 97):

$$
\begin{aligned}
&= (41592 - (4 \times 10000)) \times 10 + 6 \\
&= (76 - (4 \times 9)) \times 10 + 6 \\
&= 406 \\
&= 18
\end{aligned}
$$

- Choose modulus to be a large prime (q)
- Each window of M expected to be uniformly distributed in $[0, q-1]$
- Expected running time is $O\left(N + M\left(\frac{1}{q}(N-M)\right)\right)$
- Worst case: $\Theta(MN)$ – when?
- Possible if the computed hash matches every time and confirmational matching is needed
- Published in 1987 as *Efficient randomized pattern-matching algorithms*

Section outline

3 **Knuth-Morris-Pratt algorithm**

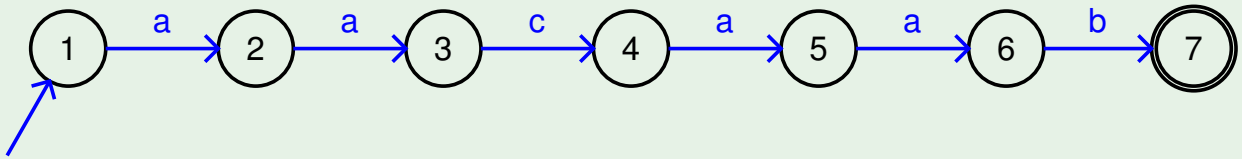
- Optimised pattern matching with KMP
- KMP algorithm
- KMP failure function
- KMP failure function computation
- KMP failure function algorithm
- Overall complexity of KMP
- Optimised failure function computation

Optimised pattern matching with KMP

Example

Search pattern					
1	2	3	4	5	6
a	a	c	a	a	b

Search text											
a	a	c	a	a	a	a	b	c	a	a	b



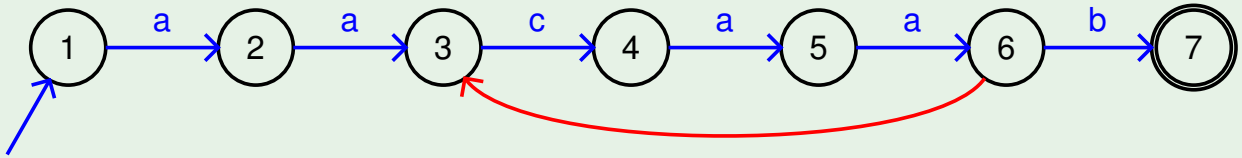
- Suppose aacaa is received; current state will be 6 and b will be expected
- If b is not received, the pattern will have to be moved forward
- Instead of moving forward by one position (brute force approach), better to align the prefix aa with the suffix aa at the point of failure – amounts to resuming comparison at state 3

Optimised pattern matching with KMP

Example

Search pattern					
1	2	3	4	5	6
a	a	c	a	a	b

Search text											
a	a	c	a	a	a	a	b	c	a	a	b



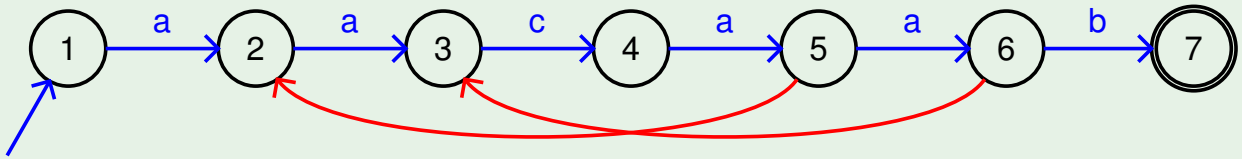
- Suppose aacaa is received; current state will be 6 and b will be expected
- If b is not received, the pattern will have to be moved forward
- Instead of moving forward by one position (brute force approach), better to align the prefix aa with the suffix aa at the point of failure – amounts to resuming comparison at state 3
- We want the *longest prefix* (aa) that is a *suffix at the point of failure* (state 6)

Optimised pattern matching with KMP

Example

Search pattern					
1	2	3	4	5	6
a	a	c	a	a	b

Search text											
a	a	c	a	a	a	a	b	c	a	a	b



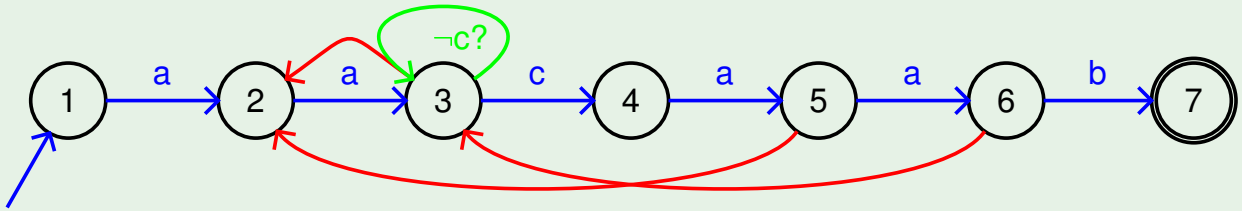
- Suppose aacaa is received; current state will be 6 and b will be expected
- If b is not received, the pattern will have to be moved forward
- Instead of moving forward by one position (brute force approach), better to align the prefix aa with the suffix aa at the point of failure – amounts to resuming comparison at state 3
- We want the *longest prefix* (aa) that is a *suffix at the point of failure* (state 6)
- Similarly, if after receiving aaca another a is not received; failure is at state 5; comparison may be resumed from state 2

Optimised pattern matching with KMP

Example

Search pattern					
1	2	3	4	5	6
a	a	c	a	a	b

Search text											
a	a	c	a	a	a	a	b	c	a	a	b



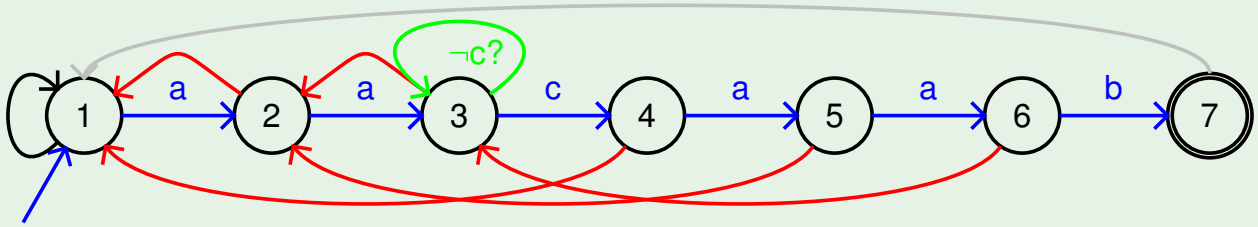
- Suppose aacaa is received; current state will be 6 and b will be expected
- If b is not received, the pattern will have to be moved forward
- Instead of moving forward by one position (brute force approach), better to align the prefix aa with the suffix aa at the point of failure – amounts to resuming comparison at state 3
- We want the *longest prefix* (aa) that is a *suffix at the point of failure* (state 6)
- Similarly, if after receiving aaca another a is not received; failure is at state 5; comparison may be resumed from state 2
- Failure transitions are meant to step back in the pattern string, staying at the same place implies ∞-loop, so on failure at state 3 (on ¬c), matching is resumed at state 2

Optimised pattern matching with KMP

Example

Search pattern					
1	2	3	4	5	6
a	a	c	a	a	b
0	1	2	1	2	3
Failure function					

Search text											
a	a	c	a	a	a	a	b	c	a	a	b



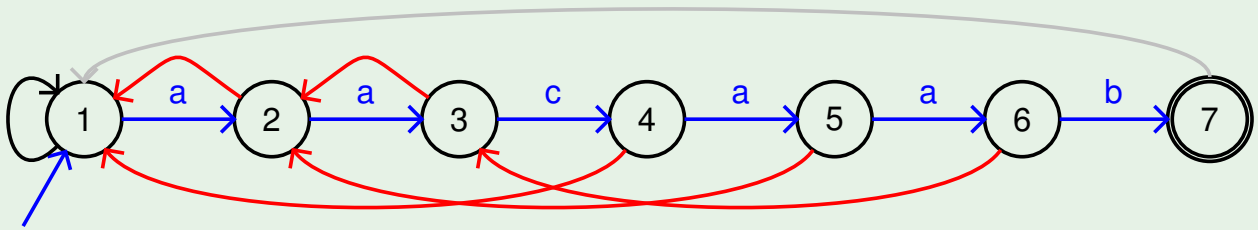
- Suppose aacaa is received; current state will be 6 and b will be expected
- If b is not received, the pattern will have to be moved forward
- Instead of moving forward by one position (brute force approach), better to align the prefix aa with the suffix aa at the point of failure – amounts to resuming comparison at state 3
- We want the *longest prefix* (aa) that is a *suffix at the point of failure* (state 6)
- Similarly, if after receiving aaca another a is not received; failure is at state 5; comparison may be resumed from state 2
- Failure transitions are meant to step back in the pattern string, staying at the same place implies ∞-loop, so on failure at state 3 (on ¬c), matching is resumed at state 2
- The point of resumption for failure at a certain point is the failure function

KMP algorithm

Example

Search pattern					
1	2	3	4	5	6
a	a	c	a	a	b
0	1	2	1	2	3
Failure function					

Search text											
a	a	c	a	a	a	a	b	c	a	a	b



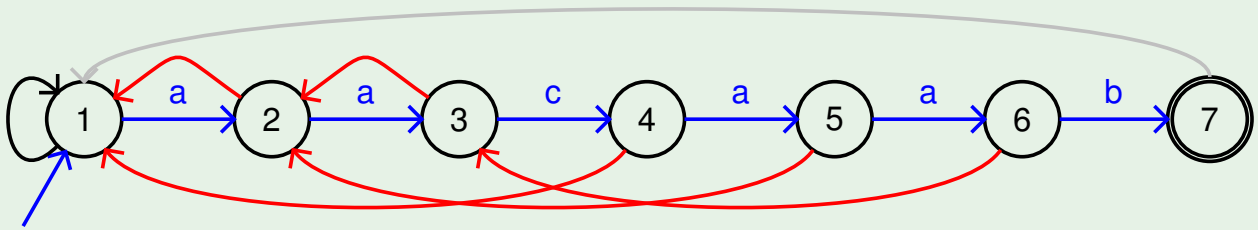
- Use knowledge of search pattern
- Build automaton from pattern
- Run automaton on text
- On failure, go back to the longest proper prefix that is a suffix at the point of the last match – to avoid looping (at state 3, for example)

KMP algorithm

Example

Search pattern					
1	2	3	4	5	6
a	a	c	a	a	b
0	1	2	1	2	3
Failure function					

Search text											
a	a	c	a	a	a	a	b	c	a	a	b



- Use knowledge of search pattern
- Build automaton from pattern
- Run automaton on text
- On failure, go back to the longest proper prefix that is a suffix at the point of the last match – to avoid looping (at state 3, for example)

1. j←1 // start of P
2. for i←1 to N // span through T
3. while j $>$ 0 and T[i]$\neq$P[j]
4. j←fail[j] // fail while no match
5. if j=M return i−M+1
   // terminate (success)
6. j←j+1 // move forward in pattern
7. return NoMatch terminate (failure)

KMP failure function computation

Base case fail[1]=0 – need to start over again ✓

- Need to compute fail[i], assuming fail[j], $j < i$ are available

Inductive cases, starting with $k = 1$ (when $\alpha' = \epsilon$)

- fail$^k[i-1]$ indicates the longest proper prefix (say α) that is a suffix at $\mathbf{P}[i-1]$; $\alpha = \epsilon$ if fail$^k[i-1] = 0$

  Exit case where fail$^k[i-1] = 0$ No prefix, so resume matching at the beginning, fail$[i] = 1$ ✓

  Case $\mathbf{P}[i-1] = \mathbf{P}[\text{fail}^k[i-1]]$

α	a	?	...	α'	α	a		...
	fail$^k[i-1]$					$i-1$	i	

- Thus, $\alpha \cdot \mathbf{P}[i-1]$ is the longest proper prefix that is also a suffix at $\mathbf{P}[i]$, so fail$[i] =$ fail$^k[i-1] + 1$ ✓

KMP failure function computation (contd.)

Inductive cases, starting with $k = 1$ (when $\alpha' = \epsilon$, contd.)

- fail$^k[i-1]$ indicates the longest proper prefix (say α) that is a suffix at $\mathbf{P}[i-1]$; $\alpha = \epsilon$ if fail$^k[i-1] = 0$

  Case $\mathbf{P}[i-1] \neq \mathbf{P}[\text{fail}^k[i-1]]$

α	b	...	α'	α	a		...
	fail$^k[i-1]$				$i-1$	i	

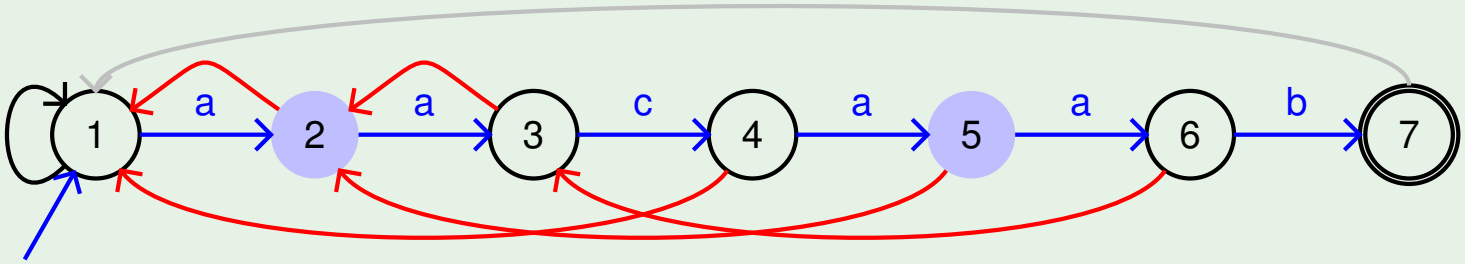
  - Now, $\alpha \cdot \mathbf{P}[i-1]$ is not an admissible suffix at $\mathbf{P}[i]$, as $\mathbf{P}[i-1] \neq \mathbf{P}[\text{fail}^k[i-1]]$
  - But, fail[fail$^k[i-1]] = $ fail$^{k+1}[i-1]$ indicates the longest proper prefix (say β) that is a suffix at $\mathbf{P}[\text{fail}^k[i-1]]$

β	a?	?	...	β	b	...	α'	α	a		...
	fail$^{k+1}[i-1]$				fail$^k[i-1]$				$i-1$	i	

  - Now, β is the longest proper prefix of $\mathbf{P}$ and also a suffix of α
  - Thus, if $\mathbf{P}[\text{fail}^{k+1}[i-1]] = \mathbf{P}[i-1]$, $\beta \cdot \mathbf{P}[i-1]$ is the longest proper prefix at $\mathbf{P}[i]$, so fail$[i] = $ fail$^{k+1}[i-1] + 1$ ✓
  - Continue induction with $k \leftarrow k+1$ ↻

KMP failure function computation example

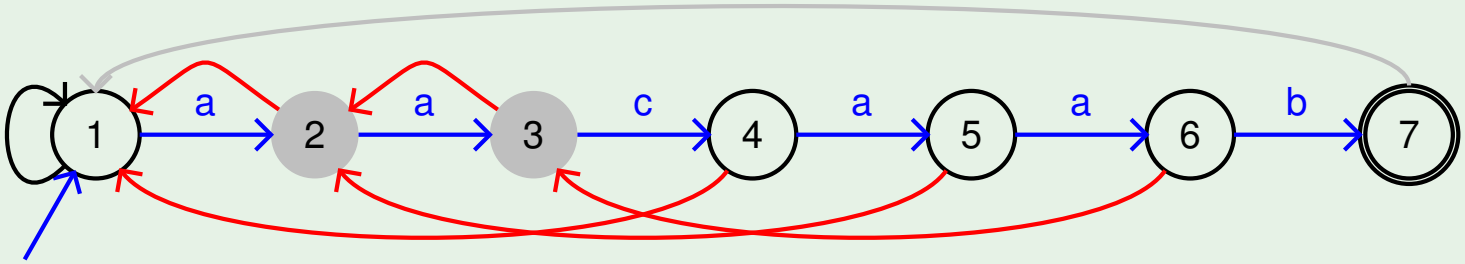
Example (Some steps of failure function computation for "aacaab")



- Consider failure at P[i] (say, P[6]=b)
- We would like to identify the longest proper prefix that is a suffix at P[i]
- The longest proper prefix that is a suffix at P[$i-1$] is denoted by fail[$i-1$]
- So, if P[$i-1$]=P[fail[$i-1$]], then fail[i]=fail[$i-1$]+1
- P[5]=a; P[fail[5]]=P[2]=a; so fail[6]=fail[5]+1=2+1=3

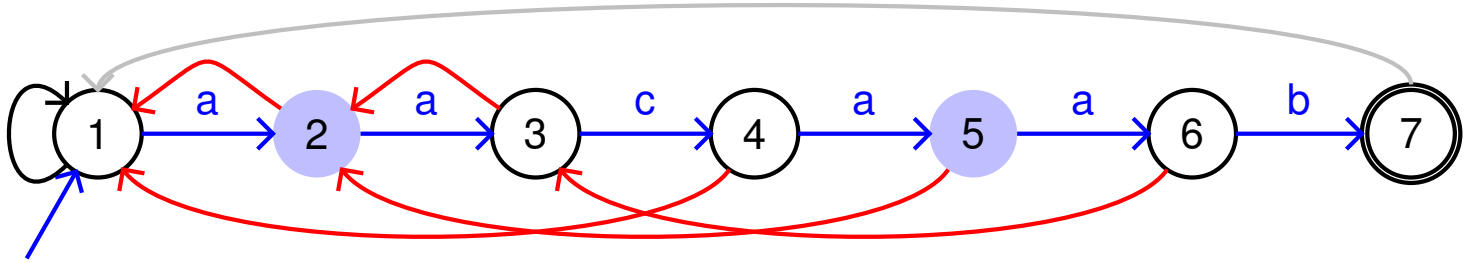
KMP failure function computation example

Example (Some steps of failure function computation for "aacaab")



- Consider failure at P[i] (say, P[6]=b)
- We would like to identify the longest proper prefix that is a suffix at P[i]
- The longest proper prefix that is a suffix at P[$i-1$] is denoted by fail[$i-1$]
- So, if P[$i-1$]=P[fail[$i-1$]], then fail[i]=fail[$i-1$]+1
- P[5]=a; P[fail[5]]=P[2]=a; so fail[6]=fail[5]+1=2+1=3
- If P[$i-1$]$\neq$P[fail[$i-1$]], then continue checking from P[fail[fail[$i-1$]]]=P[fail^2[$i-1$]], and so on, but stopping at P[1]
- While computing fail[4], we find P[3]=c and P[fail[3](=2)]=a; P[3]$\neq$P[fail[3](=2)], so go further back to fail^2[3]=1 and stop there (at P[1]=a)

KMP failure function computation algorithm



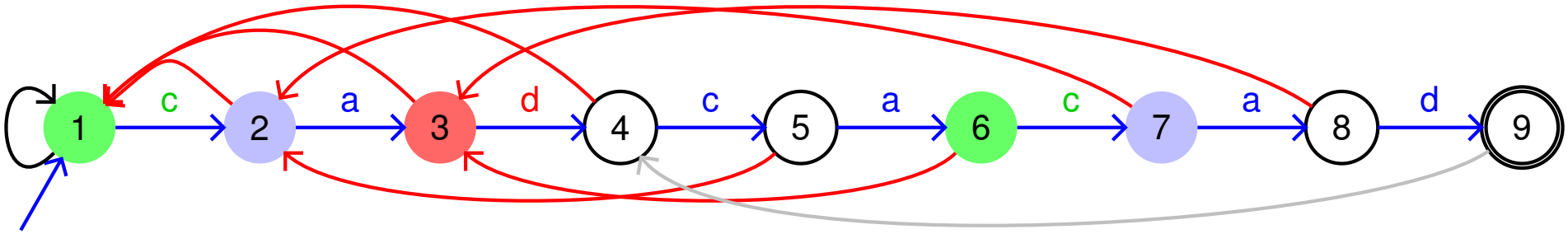
KMPCompFail(P[1..*M*])

1. j←0
2. for i←1 to M // span through M!
3. fail[i]←j
   // next prepare for fail[i+1]
4. while (j>0 and P[i]≠P[j]) do
5. j←fail[j]
6. done
7. j←j+1

Example (FF for "aacaab")

	Search pattern					
i	1	2	3	4	5	6
P[i]	a	a	c	a	a	b
fail[i]	0	1	2	1	2	3
P[j_4]	–	a	a a	a	a	c a a
j_5	–	–	1 0	–	–	2 1 0
j_7	1	2	1	2	3	1

KMP failure function algorithm (contd.)



KMPCompFail(P[1..*M*])

1. j←0
2. for i←1 to M // span through M!
3. fail[i]←j
   // next prepare for fail[i+1]
4. while (j>0 and P[i]≠P[j]) do
5. j←fail[j]
6. done
7. j←j+1
8. endfor

Example (FF for "aacaab")

	Search pattern							
i	1	2	3	4	5	6	7	8
P$[i]$	c	a	d	c	a	c	a	d
fail$[i]$	0	1	1	1	2	3	2	3
P$[j_4]$	–	c	c	c	a	d c	a	a
j_5	–	0	0	–	–	1 –	–	–
j_7	1	1	1	2	3	2	3	4

Complexity of computing fail[i] and running KMP

KMPCompFail(P[1..M])

1. j←0
2. for i←1 to M // span through M!
3. fail[i]←j
4. while (j>0 and P[i]≠P[j])
5. j←fail[j]
6. done
7. j←j+1
8. endfor

- Note that fail[j]<j
- In L5 j is decreased by at least 1
- Overall j can go back in L5 only as much as it has progressed in L7
- L7 is executed M times
- Complexity of KMPCompFail is $O(M)$ (from $2M$))

Example (FF for "aacaab")

Search pattern					
1	2	3	4	5	6
a	a	c	a	a	b
0	1	2	1	2	3
Failure function					

Complexity of computing fail[i] and running KMP

KMPCompFail(P[1..M])

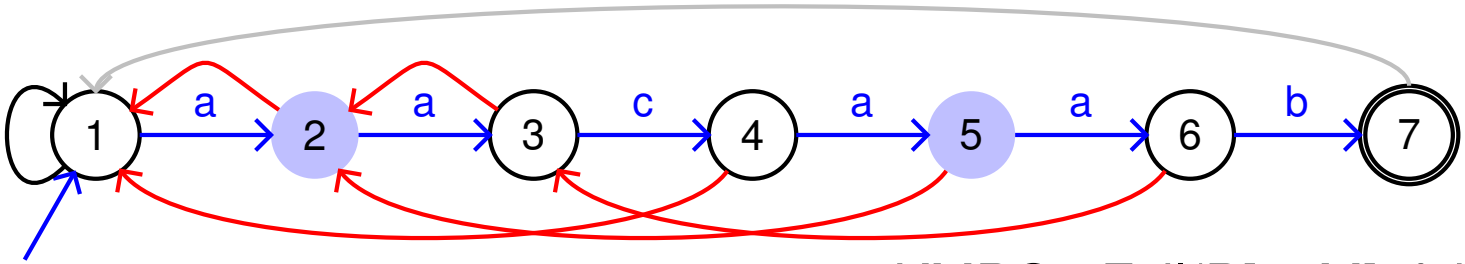
1. j←0
2. for i←1 to M // span through M!
3. fail[i]←j
4. while (j>0 and P[i]≠P[j])
5. j←fail[j]
6. done
7. j←j+1
8. endfor

Example (FF for "aacaab")

Search pattern					
1	2	3	4	5	6
a	a	c	a	a	b
0	1	2	1	2	3
Failure function					

- Note that fail[j]<j
- In L5 j is decreased by at least 1
- Overall j can go back in L5 only as much as it has progressed in L7
- L7 is executed M times
- Complexity of KMPCompFail is $O(M)$ (from $2M$))
- Using similar reasoning complexity of the KMP algorithm is $O(N)$ (from $2N$)
- Overall complexity is $O(M+N)$
- Publication: Fast pattern matching in strings, D E Knuth, J H Morris, V R Pratt, SIAM JoC, v6, n2, June 1997

Optimised failure function computation



- Consider the failure at P[5]=a; fail[5]=2
- But P[2]=a, so after failing to match a at P[5], failure is guaranteed at P[2]
- This definite failure could be remedied by going all the way back to $fail^3[5]=0$
- Function KMPOptFail does the required post-processisng – employing dynamic programming

KMPOptFail(P[1..M], fail[1..M])

1. for i←2 to M // bottom-up DP
2. if P[i]=P[fail[i]] // definite failure
3. fail[i]←fail[fail[i]] // fail all
4. endfor // way back via DP

Example (Opt FF for "aacaab")

Search pattern					
1	2	3	4	5	6
a	a	c	a	a	b
0	0	2	0	0	3
Optimised failure fn					

Section outline

4 **Boyer-Moore algorithm**
- Key aspects
- Bad character shift rule
- Good suffix shift rule
- GSS computation

Key aspects

The Boyer-Moore algorithm is based on three ideas:

- Scanning the pattern pat from right to left: P[M], P[$M-1$], ...
- The “bad character shift rule”: skips over parts of pattern where there is no possibility of matching the current character in the text
- The “good suffix shift rule”: aligns only matching pattern characters against target characters already successfully matched
- These rules work independently, but are more effective together

Bad character shift rule

Example (Skipping over bad characters)

Text string																			
D	o		n	u	r	t	u	r	e		t	h	e		f	u	t	u	r	e
f	u	t	u	r	e														
Search pattern (r≠e)																			

Bad character shift rule

Example (Skipping over bad characters)

Text string																		
D	o		n	u	r	t	u	r	e		t	h	e		f	u	t	u	r	e

f	u	t	u	r	e	
Search pattern (r≠e)						

Text string																		
D	o		n	u	r	t	u	r	e		t	h	e		f	u	t	u	r	e

	f	u	t	u	r	e
	Search pattern (t≠e)					

Bad character shift rule

Example (Skipping over bad characters)

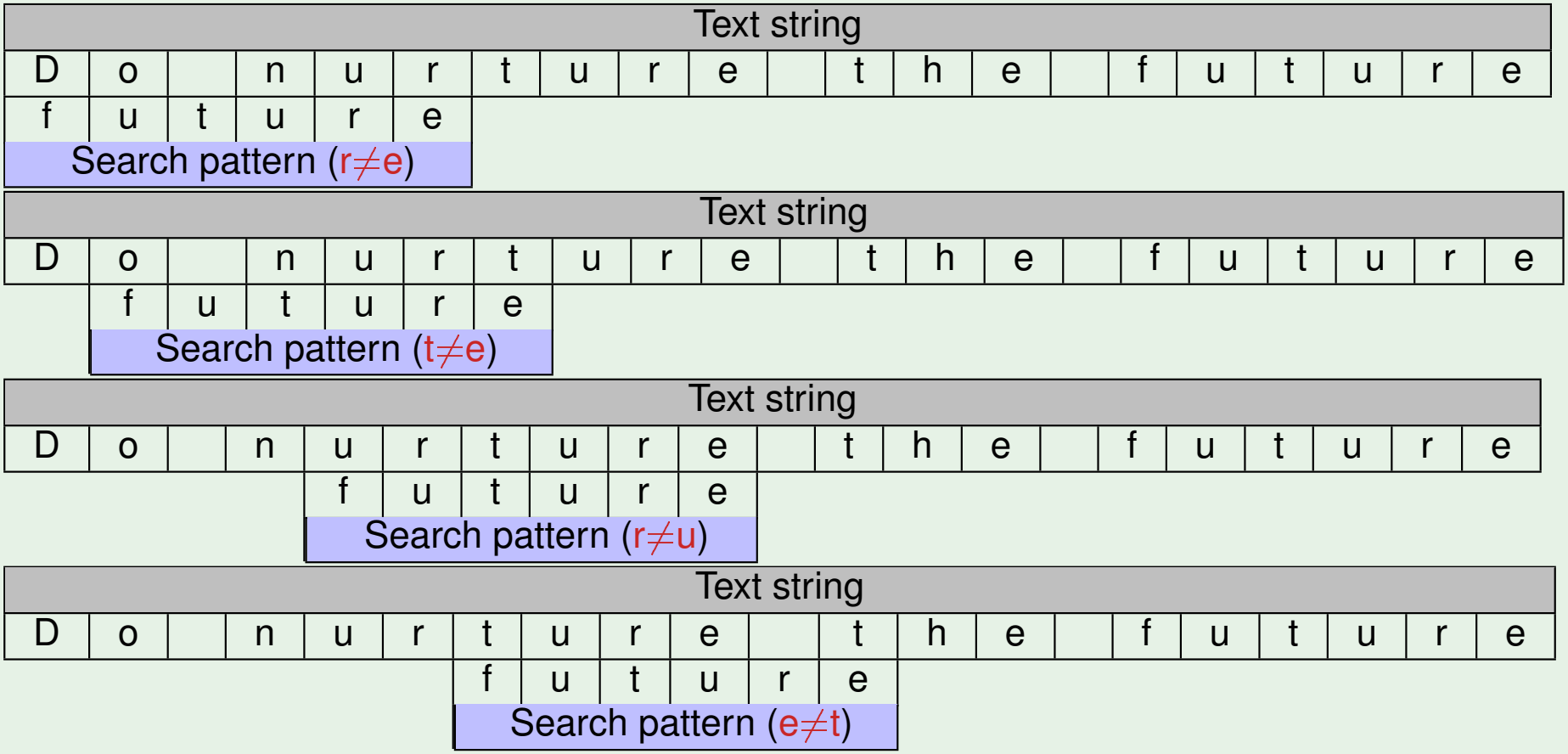
Text string																		
D	o		n	u	r	t	u	r	e		t	h	e		f	u	t	u	r	e
f	u	t	u	r	e													
Search pattern (r≠e)																		

Text string																		
D	o		n	u	r	t	u	r	e		t	h	e		f	u	t	u	r	e
	f	u	t	u	r	e												
	Search pattern (t≠e)																	

Text string																		
D	o		n	u	r	t	u	r	e		t	h	e		f	u	t	u	r	e
				f	u	t	u	r	e									
				Search pattern (r≠u)														

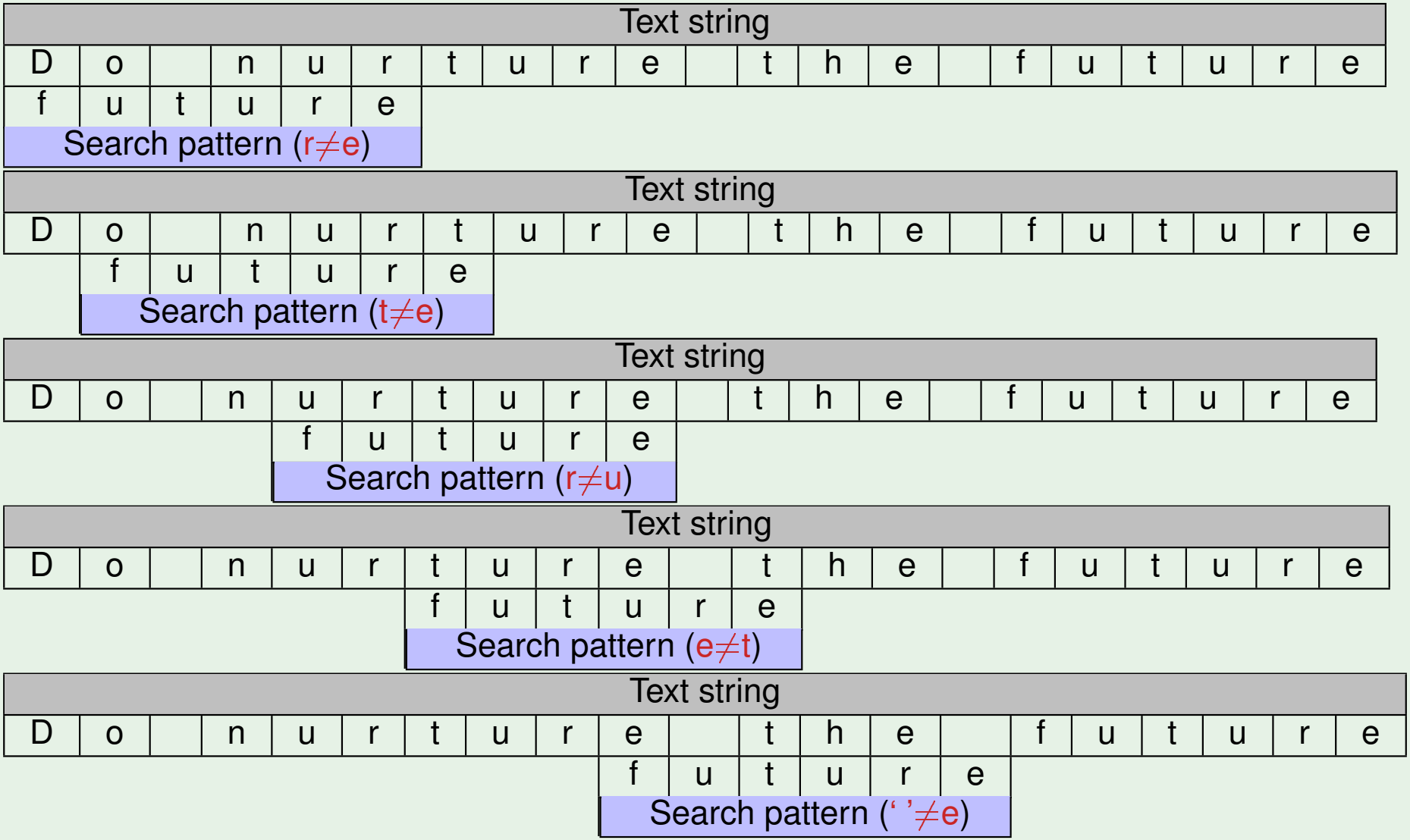
Bad character shift rule

Example (Skipping over bad characters)



Bad character shift rule

Example (Skipping over bad characters)



Bad character shift rule (contd)

- If the current character *c* of the text **T** does not match the corresponding character in the pattern, jump to the right most occurrence of *c* in **P** without shifting backwards
- If *c* does not occur ahead in **P**, just slide **P** all the way back

Bad character shift rule (contd)

- If the current character *c* of the text **T** does not match the corresponding character in the pattern, jump to the right most occurrence of *c* in **P** without shifting backwards
- If *c* does not occur ahead in **P**, just slide **P** all the way back

Definition ($R_i(x)$)

$R_i(x)$ is the position of the rightmost occurrence of character x before position i

Bad character shift rule (contd)

- If the current character c of the text **T** does not match the corresponding character in the pattern, jump to the right most occurrence of c in **P** without shifting backwards
- If c does not occur ahead in **P**, just slide **P** all the way back

Definition ($R_i(x)$)

$R_i(x)$ is the position of the rightmost occurrence of character x before position i

- $\forall c \in \Sigma$, $R_i(c) = \begin{cases} 0 & \text{if } \nexists j < i | \mathbf{P}[j] = c \\ \max\{j < i | \mathrm{P}[j] = c\} & \text{otherwise} \end{cases}$
- $R_i(c)$ is computed in $O(|\Sigma| + M)$ time
- When a mismatch occurs at pattern position i in **P**, shift by $i - R_i(\mathbf{T}[k])$ characters so that the next occurrence of $\mathbf{T}[k]$ in **P** is underneath position k in **T**
- Best case time complexity for BCS is $O(N/M)$ – sublinear!
- $R_i(x)$ may be realised as `R[i,x]`, but inefficient

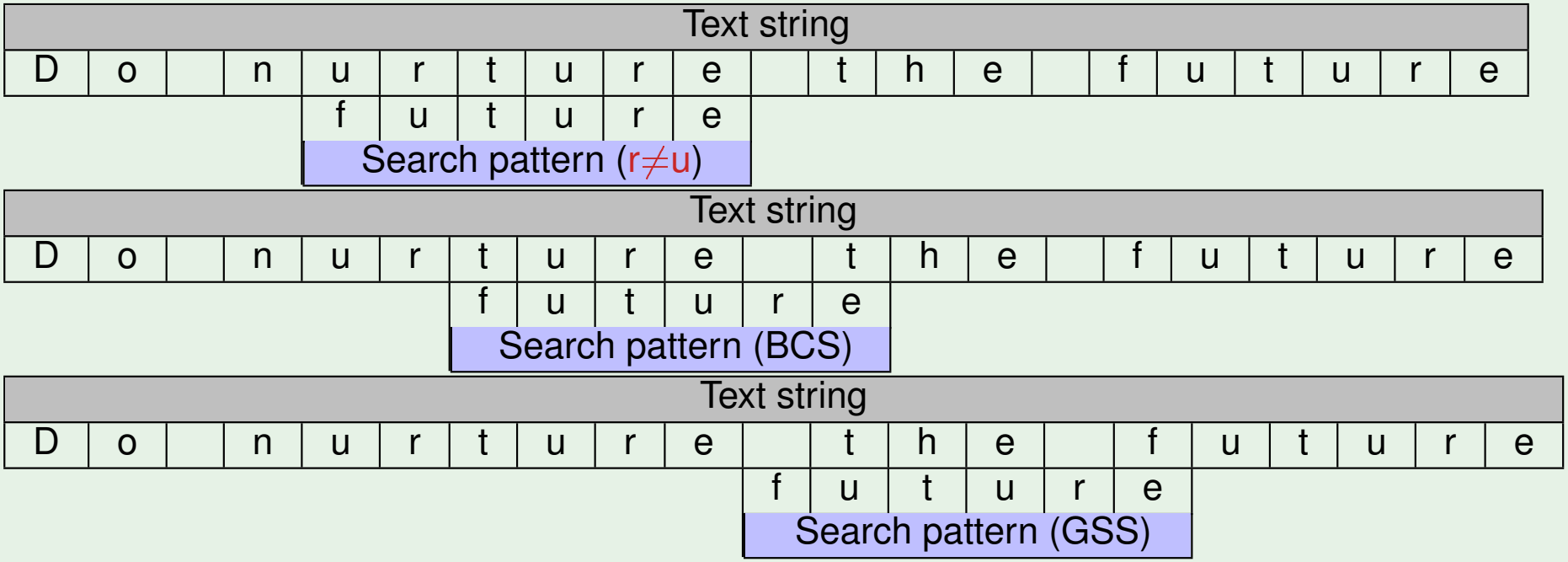
Good suffix shift rule

Example (Comparing shifts by BCS and GSS for $r \neq u_1$)

Text string																		
D	o		n	u	r	t	u	r	e		t	h	e		f	u	t	u	r	e
				f	u	t	u	r	e											
				Search pattern (r≠u)																

Good suffix shift rule

Example (Comparing shifts by BCS and GSS for $r \neq u_1$)



- Let $s = \mathbf{P}[i..M] = \mathbf{T}[i + j..j + M]$ and $\mathbf{P}[i - 1] \neq \mathbf{T}[i + j - 1]$
- The GSS rule aligns the substring s with its rightmost occurrence in **P** (but not as a suffix) that is preceded by a character different from P[$i - 1$]
- Similar to the optimised KMP failure function from the right end

GSS computation

Example (P has the form: $\delta \cdot b \cdot \alpha \cdot \gamma \cdot a \cdot \alpha$, mismatch at $a \cdot \alpha$)

Reversed search pattern						
1	2	3	4	5	6	7
a	a	c	a	a	b	–
0	0	2	0	0	3	0
Opt failure fn →						

Search pattern						
0	1	2	3	4	5	6
–	b	a	a	c	a	a
0_7	3_6	0_5	0_4	2_3	0_2	0_1
Opt failure fn from the right						
	–	–	–	1	4	–
← Computed GSS						

- KMP: fail[6]=3, fail@$\mathbf{P}^r[..aab]$, resume@ $\mathbf{P}^r[aac]$
- BM: fail@$\mathbf{P}[6-3+1=4](caa)$, resume@ $\mathbf{P}[6-6+11](baa)$, gss[4]=1

GSS computation

Example (P has the form: $\delta \cdot b \cdot \alpha \cdot \gamma \cdot a \cdot \alpha$, mismatch at $a \cdot \alpha$)

Reversed search pattern						
1	2	3	4	5	6	7
a	a	c	a	a	b	–
0	0	2	0	0	3	0
Opt failure fn →						

Search pattern						
0	1	2	3	4	5	6
–	b	a	a	c	a	a
0_7	3_6	0_5	0_4	2_3	0_2	0_1
Opt failure fn from the right						
	–	–	–	1	4	–
← Computed GSS						

- KMP: fail[6]=3, fail@$\mathbf{P}^r[..aab]$, resume@ $\mathbf{P}^r[aac]$
- BM: fail@$\mathbf{P}[6-3+1=4](caa)$, resume@ $\mathbf{P}[6-6+11](baa)$, gss[4]=1

- KMP: fail[3]=2, fail@$\mathbf{P}^r[..ac]$, resume@ $\mathbf{P}^r[aa]$
- BM: fail@$\mathbf{P}[6-2+1=5](aa)$, resume@ $\mathbf{P}[6-3+1=4](ca)$, gss[5]=4

GSS computation

Example (P has the form: $\delta \cdot b \cdot \alpha \cdot \gamma \cdot a \cdot \alpha$, mismatch at $a \cdot \alpha$)

Reversed search pattern						
1	2	3	4	5	6	7
a	a	c	a	a	b	–
0	0	2	0	0	3	0
Opt failure fn $\rightarrow$						

Search pattern						
0	1	2	3	4	5	6
–	b	a	a	c	a	a
0_7	3_6	0_5	0_4	2_3	0_2	0_1
Opt failure fn from the right						
	–	–	–	1	4	–
$\leftarrow$ Computed GSS						

- KMP: fail[6]=3, fail@$\mathbf{P}^r[..aab]$, resume@ $\mathbf{P}^r[aac]$
- BM: fail@$\mathbf{P}[6-3+1=4](caa)$, resume@ $\mathbf{P}[6-6+11](baa)$, gss[4]=1

- KMP: fail[3]=2, fail@$\mathbf{P}^r[..ac]$, resume@ $\mathbf{P}^r[aa]$
- BM: fail@$\mathbf{P}[6-2+1=5](aa)$, resume@ $\mathbf{P}[6-3+1=4](ca)$, gss[5]=4

- KMP: fail[1]=fail[2]=fail[4]=fail[5]=0
- BM: unfilled cells are marked with '–' indicating start over

GSS computation

Example (P has the form: $\delta \cdot b \cdot \alpha \cdot \gamma \cdot a \cdot \alpha$, mismatch at $a \cdot \alpha$)

Reversed search pattern						
1	2	3	4	5	6	7
a	a	c	a	a	b	–
0	0	2	0	0	3	0
Opt failure fn →						

Search pattern						
0	1	2	3	4	5	6
–	b	a	a	c	a	a
0_7	3_6	0_5	0_4	2_3	0_2	0_1
Opt failure fn from the right						
	–	–	–	1	4	–
← Computed GSS						

- KMP: fail[6]=3, fail@$\mathbf{P}^r[..aab]$, resume@ $\mathbf{P}^r[aac]$
- BM: fail@$\mathbf{P}[6-3+1=4](caa)$, resume@ $\mathbf{P}[6-6+11](baa)$, gss[4]=1

- KMP: fail[3]=2, fail@$\mathbf{P}^r[..ac]$, resume@ $\mathbf{P}^r[aa]$
- BM: fail@$\mathbf{P}[6-2+1=5](aa)$, resume@ $\mathbf{P}[6-3+1=4](ca)$, gss[5]=4

- KMP: fail[1]=fail[2]=fail[4]=fail[5]=0
- BM: unfilled cells are marked with '–' indicating start over

---

- Here, KMP does not help for $\mathbf{P}[M]$, but BCS can, eg $R_6(c) = 4$
- On mismatch on c at $\mathbf{P}[\imath]$, can use max(GSS[$\imath$], $R_\imath(c)$)

GSS computation (contd.)

Example (P also has the form: (longest β) $\beta \cdot \gamma \cdot \beta$, mismatch at $a \cdot \alpha$)

Reversed search pattern										
1	2	3	4	5	6	7	8	9	10	11
l	f	o	s	r	l	f	o	l	f	–
0	1	1	1	1	0	1	1	4	1	3
Optimised failure function →										

- fail[9]=4, so gss[7]=2 (as for earlier form of **P**)
- fail[2,3,4,5,7,8,10]=1,
- $\therefore$ gss[11-1]=gss[10]= $\max\{9, 8, 7, 6, 4, 3, 1\} = 9$

Search pattern ($M = 10$)										
0	1	2	3	4	5	6	7	8	9	10
–	f	l	o	f	l	r	s	o	f	l
3_{11}	1_{10}	4_9	1_8	1_7	0_6	1_5	1_4	1_3	1_2	0_1
Optimised failure function from the right										
	-7	-6	-5	-4	-3	-2	2	0	–	9
← Computed GSS										

- fail[$M + 1$]=3, so gss[8]=0
- gss[1,2,3,4,5,6]=?

- Note the prospect of matching "fl" at the head of **P**, indicated by gss[8]=0
- gss[6]=-2, gss[5]=-3, gss[4]=-4, gss[3]=-5, gss[2]=-6, gss[1]=-7
- For $\imath < \max\{\jmath | \mathbf{P}[\jmath] = 0\} \wedge (\text{fail}[\imath] = 0 \vee \text{fail}[\imath] = 1), \text{gss}[\imath] = \imath - \jmath$