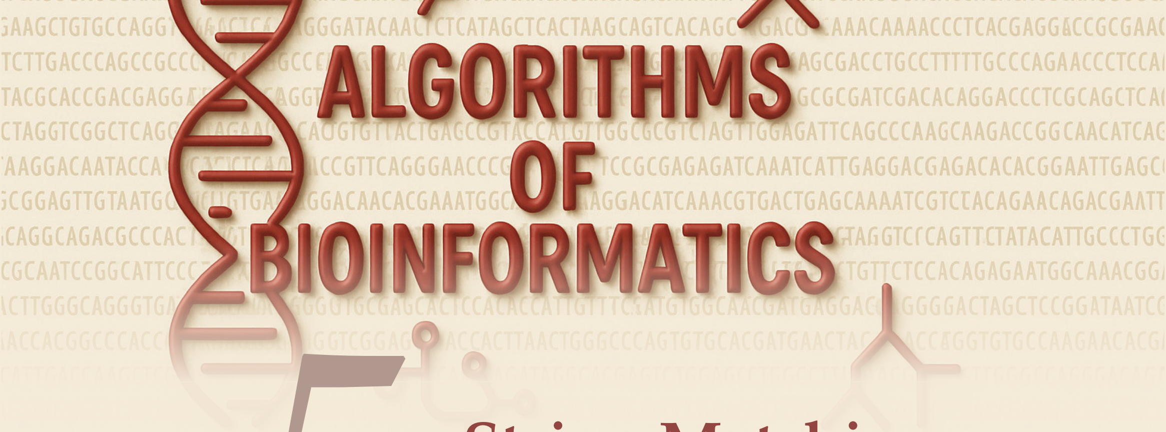


5 String Matching

4 December 2025

Prof. Dr. Sebastian Wild

CS594 (Winter 2025-26)
Philipps-Universität Marburg
version `2025-12-04 13:53`

Outline

5 String Matching

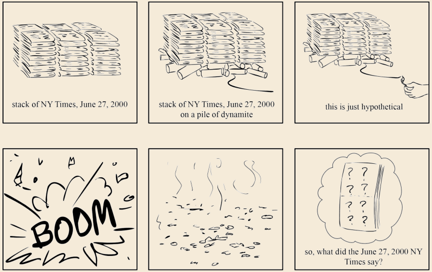
5.1 Read Mapping
5.2 String Matching with Finite Automata
5.3 Constructing String Matching Automata
5.4 The Knuth-Morris-Pratt algorithm
5.5 The Aho-Corasick Algorithm

5.1 Read Mapping

Shotgun Sequencing

Recall:

- *Shotgun sequencing* approach to determine a (human) genome:



Compeau & Pevzner, *Bioinformatics Algorithms*, Fig 3.1
`https://cogniterra.org/lesson/29884/step/2?unit=21982`

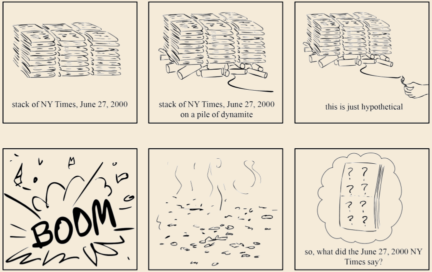
- For a single human genome need 300M reads of 200bp (30x coverage)
  - ⇝ 60 GB of raw data

⇝ genome assembly from those is expensive and error prone

Shotgun Sequencing

Recall:

- *Shotgun sequencing* approach to determine a (human) genome:



Compeau & Pevzner, *Bioinformatics Algorithms*, Fig 3.1
`https://cogniterra.org/lesson/29884/step/2?unit=21982`

- For a single human genome need 300M reads of 200bp (30x coverage)
  - ⇝ 60 GB of raw data

⇝ genome assembly from those is expensive and error prone

We now have carefully assembled reference genomes to compare with!

Shotgun Sequencing in Medicine

Predominant medical use of whole genome sequencing:
detecting known markers (mutations / gene combinations) for diseases

Shotgun Sequencing in Medicine

Predominant medical use of whole genome sequencing:
detecting known markers (mutations / gene combinations) for diseases

Example: (more details in Compeau & Pevzner 2015)

- *Ohdo syndrome* (form of mental retardation, "mask-like" face)
- known to be indicated by single protein-truncating mutation

Often even a **single base replacement** w.r.t. a human reference genome,
a ***SNP (single nucleotide polymorphism)***

Shotgun Sequencing in Medicine

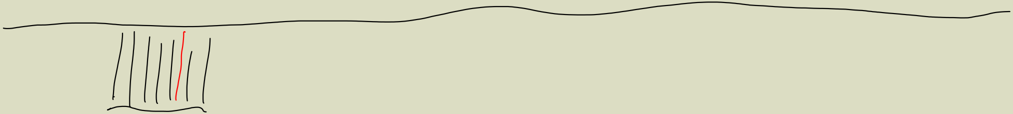
Predominant medical use of whole genome sequencing:
detecting known markers (mutations / gene combinations) for diseases

Example: (more details in Compeau & Pevzner 2015)

- *Ohdo syndrome* (form of mental retardation, "mask-like" face)
- known to be indicated by single protein-truncating mutation

Often even a **single base replacement** w.r.t. a human reference genome,
a ***SNP (single nucleotide polymorphism)***

⇝ *To find SNPs, we don't need a new patient's genome fully assembled!*



Read Mapping

- We thus work towards solving the ***read mapping problem***
  - **Given:** genome/text $T[0..n)$, reads/patterns $P[0..p)$, $P[r] = P_r[0..m_r)$
  - **Goal:** locations i_r of "*best match*" for P_r in T for $r \in [0..p)$.
- "best match" can be interpreted in several ways, leading to different problems:

(a) best fitting **semilocal alignment** of P_r to T (gold standard, usually too expensive)

(b) match with **fewest mismatches**

(c) match with $\le d$ **mismatches** or `NO_MATCH` if no such exists
for SNPs can even set $d = 1$

(d) **exact match** or `NO_MATCH` if no such exists

Read Mapping

- We thus work towards solving the ***read mapping problem***
  - **Given:** genome/text $T[0..n)$, reads/patterns $P[0..p)$, $P[r] = P_r[0..m_r)$
  - **Goal:** locations i_r of "*best match*" for P_r in T for $r \in [0..p)$.
- "best match" can be interpreted in several ways, leading to different problems:

  (a) best **semilocal alignment** of P_r to T (gold standard, usually too expensive)

  (b) match with **fewest mismatches**

  (c) match with $\le d$ **mismatches** or `NO_MATCH` if no such exists
  for SNPs can even set $d = 1$

  (d) **exact match** or `NO_MATCH` if no such exists

*We will first focus on **exact matches**.*

- simplifies the problem (to get started)
- the SNPs variants can be reduced to it (using postprocessing)

Part I

Exact matches

Notation

$\Sigma = [0..\sigma)$

- *alphabet* Σ: finite set of allowed **characters**; $\sigma = |\Sigma|$ *"a string over alphabet Σ"*
  - focus on nucleotides $\{A, C, G, T\}$ and amino acids
  - but try to keep methods generic
  - letters (Latin, Greek, Arabic, Cyrillic, Asian scripts, ...) Unicode characters

comprehensive standard character set
including emoji and all known symbols

Notation

- *alphabet* Σ: finite set of allowed **characters**; $\sigma = |\Sigma|$ *"a string over alphabet Σ"*
  - focus on nucleotides $\{A, C, G, T\}$ and amino acids
  - but try to keep methods generic
  - letters (Latin, Greek, Arabic, Cyrillic, Asian scripts, ...) Unicode characters
    comprehensive standard character set
    including emoji and all known symbols
- $\Sigma^n = \Sigma \times \cdots \times \Sigma$: strings of **length** $n \in \mathbb{N}_0$ (n-tuples)
  - $\Sigma^\star = \bigcup_{n \ge 0} \Sigma^n$: set of **all** (finite) strings over Σ, $\Sigma^+ = \bigcup_{n \ge 1} \Sigma^n$
  - $\varepsilon \in \Sigma^0$: the *empty* string (same for all alphabets)

Notation

- *alphabet* Σ: finite set of allowed **characters**; $\sigma = |\Sigma|$ *"a string over alphabet Σ"*
  - focus on nucleotides $\{A, C, G, T\}$ and amino acids
  - but try to keep methods generic
  - letters (Latin, Greek, Arabic, Cyrillic, Asian scripts, ...) Unicode characters

    comprehensive standard character set including emoji and all known symbols
- $\Sigma^n = \Sigma \times \cdots \times \Sigma$: strings of **length** $n \in \mathbb{N}_0$ (n-tuples)
  - $\Sigma^\star = \bigcup_{n \ge 0} \Sigma^n$: set of **all** (finite) strings over Σ, $\Sigma^+ = \bigcup_{n \ge 1} \Sigma^n$
  - $\varepsilon \in \Sigma^0$: the *empty* string (same for all alphabets)

zero-based (like arrays)!

- for $S \in \Sigma^n$, write $S = S[0..n)$, so $S[i]$ (other sources: S_i) for i**th** character $(0 \le i < n)$
- for $S, T \in \Sigma^\star$, write $ST = S \cdot T$ for **concatenation** of S and T
- for $S \in \Sigma^n$, write $S[i..j)$ for the **substring** $S[i] \cdot S[i+1] \cdots S[j-1]$ $(0 \le i \le j \le n)$
  - $S[i..i) = \varepsilon$
  - $S[0..j)$ is a **prefix** of S; $S[i..n)$ is a **suffix** of S

String matching – Definition

Search for a string (pattern) in a large body of text

- **Input:**
  - $T \in \Sigma^n$: The *text* being searched within
  - $P \in \Sigma^m$: The *pattern* being searched for; typically $n \gg m$
- **Output:**
  - the *first occurrence (match)* of P in T: $\min\{i \in [0..n-m) : T[i..i+m) = P\}$
  - or `NO_MATCH` if there is no such i ("P does not occur in T")
  - sometimes also: find **all** occurrences of P in T.

String matching – Definition

Search for a string (pattern) in a large body of text

- **Input:**
  - $T \in \Sigma^n$: The *text* being searched within
  - $P \in \Sigma^m$: The *pattern* being searched for; typically $n \gg m$
- **Output:**
  - the *first occurrence (match)* of P in T: $\min\{i \in [0..n-m) : T[i..i+m) = P\}$
  - or `NO_MATCH` if there is no such i ("P does not occur in T")
  - sometimes also: find **all** occurrences of P in T.

- trivially solvable with $\underbrace{(n-m+1)\cdot \overset{\text{try all starting positions}}{m} \sim nm}$ character comparisons

⇝ too slow for read mapping!

- string matching available, e. g., Java in `String.indexOf`, Python in `str.find`
  - not always robust enough for bioinformatics data (small alphabet, long repetitions)

5.2 String Matching with Finite Automata

Theoretical Computer Science to the rescue!

- string matching $=$ deciding whether $T \in \Sigma^\star \cdot P \cdot \Sigma^\star$
- $\Sigma^\star \cdot P \cdot \Sigma^\star$ is *regular* formal language

$\rightsquigarrow$ $\exists$ *deterministic finite automaton* (DFA) to recognize $\Sigma^\star \cdot P \cdot \Sigma^\star$

$\rightsquigarrow$ can check for occurrence of P in $|T| = n$ steps!

Theoretical Computer Science to the rescue!

- string matching $=$ deciding whether $T \in \Sigma^\star \cdot P \cdot \Sigma^\star$
- $\Sigma^\star \cdot P \cdot \Sigma^\star$ is *regular* formal language

$\rightsquigarrow$ $\exists$ *deterministic finite automaton* (DFA) to recognize $\Sigma^\star \cdot P \cdot \Sigma^\star$

$\rightsquigarrow$ can check for occurrence of P in $|T| = n$ steps!

Job done!

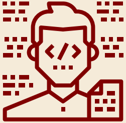
Theoretical Computer Science to the rescue!

- string matching = deciding whether $T \in \Sigma^\star \cdot P \cdot \Sigma^\star$
- $\Sigma^\star \cdot P \cdot \Sigma^\star$ is *regular* formal language

$\rightsquigarrow$ $\exists$ *deterministic finite automaton* (DFA) to recognize $\Sigma^\star \cdot P \cdot \Sigma^\star$

$\rightsquigarrow$ can check for occurrence of P in $|T| = n$ steps!

Job done!



`WTF!?`

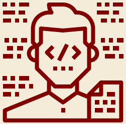
Theoretical Computer Science to the rescue!

- string matching $=$ deciding whether $T \in \Sigma^\star \cdot P \cdot \Sigma^\star$
- $\Sigma^\star \cdot P \cdot \Sigma^\star$ is *regular* formal language

$\rightsquigarrow$ $\exists$ *deterministic finite automaton* (DFA) to recognize $\Sigma^\star \cdot P \cdot \Sigma^\star$

$\rightsquigarrow$ can check for occurrence of P in $|T| = n$ steps!

Job done!



`WTF!?`

We are not quite done yet.

- (Problem 0: programmer might not know automata and formal languages . . .)
- Problem 1: existence alone does not give an algorithm!
- Problem 2: automaton could be very big!

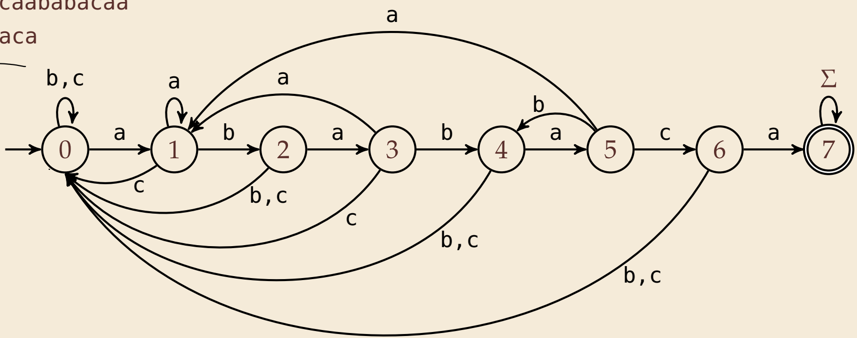
String matching with DFA

- Assume first, we already have a deterministic automaton
- How does string matching work?

Example:

T = aabacaababacaa

P = ababaca



text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	1	1	2	3	0									

String matching with DFA

- Assume first, we already have a deterministic automaton
- How does string matching work?

x b a b a

a b a b a | b a

... | P[0..5) | x |

a b a b a

| a b a b |

| P[0..4) |

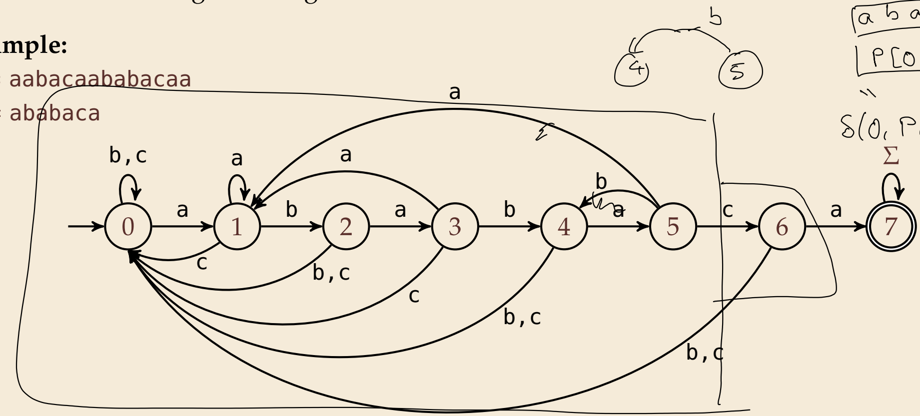
"

δ(0, P[1..5)b)

Example:

T = aabacaababacaa

P = ababaca

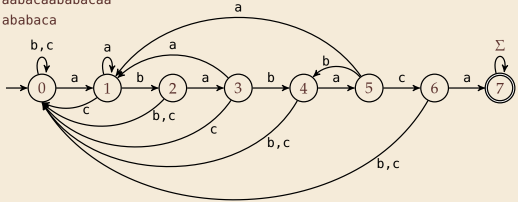


text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	1	1	2	3	0	1	1	2	3	4	5	6	7	7

String matching DFA – Intuition

Why does this work?

T = aabacaababacaa
P = ababaca



text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	1	1	2	3	0	1	1	2	3	4	5	6	7	7

- Main insight:

  State q means:
  "we have seen $P[0..q)$ until here (but not any longer prefix of P)"

- If the next text character c does not match, we know:
  - (i) text seen so far ends with $P[0...q) \cdot c$
  - (ii) $P[0...q) \cdot c$ is not a prefix of P
  - (iii) without reading c, $P[0..q)$ was the *longest* prefix of P that ends here.

$T = \cdots \boxed{P[0..q)}\boxed{c}$
$\boxed{P[0..q')}$
with $q' < q$

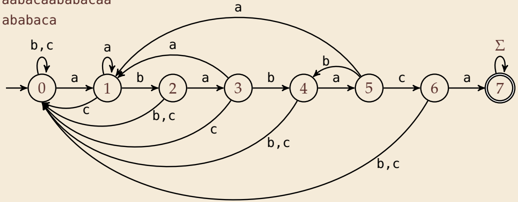
String matching DFA – Intuition

Why does this work?

- Main insight:

> State q means:
> *"we have seen $P[0..q)$ until here*
> *(but not any longer prefix of P)"*

T = aabacaababacaa
P = ababaca



text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	1	1	2	3	0	1	1	2	3	4	5	6	7	7

- If the next text character c does not match, we know:
  - (i) text seen so far ends with $P[0...q) \cdot c$
  - (ii) $P[0...q) \cdot c$ is not a prefix of P
  - (iii) without reading c, $P[0..q)$ was the *longest* prefix of P that ends here.

$T = \cdots \boxed{P[0..q)}\,\boxed{c}$
$\boxed{P[0..q')}$
with $q' < q$

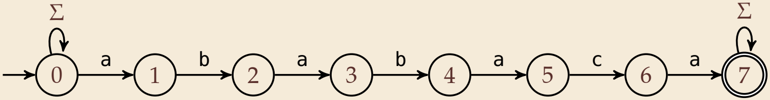
⇝ New longest matched prefix will be (weakly) shorter than q

⇝ All information about the text needed to determine it is contained in $P[0...q) \cdot c$!

5.3 Constructing String Matching Automata

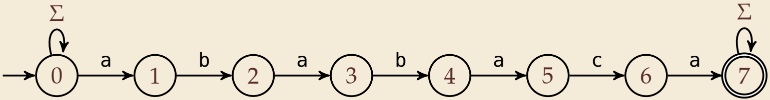
NFA instead of DFA?

It remains to *construct* the DFA.

- trivial part: 

NFA instead of DFA?

It remains to *construct* the DFA.

- trivial part: 

- that actually is a ***nondeterministic*** *finite automaton* (NFA) for $\Sigma^{\star} P\, \Sigma^{\star}$

⇝ We *could* use the NFA directly for string matching:

- at any point in time, we are in a ***set* of states**
- accept when one of them is final state

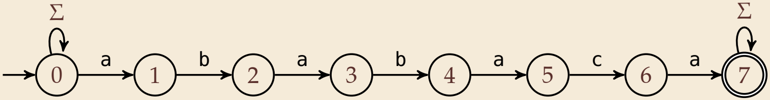
Example:

text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	0,1	0,1	0,2	0,1,3										

But maintaining a whole set makes this slow . . .

NFA instead of DFA?

It remains to *construct* the DFA.

- trivial part: 

- that actually is a ***non**deterministic finite automaton* (NFA) for $\Sigma^\star P\, \Sigma^\star$

⇝ We *could* use the NFA directly for string matching:

- at any point in time, we are in a ***set* of states**
- accept when one of them is final state

Example:

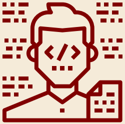
text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	0, 1	0, 1	0, 2	0, 1, 3	0	0, 1	0, 1	0, 2	0, 1, 3	0, 2, 4	0, 1, 3, 5	0, 6	0, 1, 7	0, 1, 7

But maintaining a whole set makes this slow . . .

Computing DFA directly

You have an NFA and want a DFA?
Simply apply the power-set construction
(and maybe DFA minimization)!

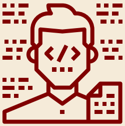
```
The powerset method has exponential state blow up!
I guess I might as well use brute force string matching ...
```



Computing DFA directly

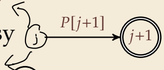
You have an NFA and want a DFA?
Simply apply the power-set construction
(and maybe DFA minimization)!

```
The powerset method has exponential state blow up!
I guess I might as well use brute force string matching ...
```



Ingenious algorithm by Knuth, Morris, and Pratt: construct DFA *inductively*:

Suppose we add character $P[j]$ to automaton A_j for $P[0..j)$ to construct A_{j+1}

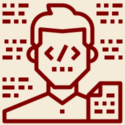
- add new state and matching transition $\rightsquigarrow$ easy 

- for each $c \neq P[j]$, we need $\delta(j, c)$ (transition from j when reading c)

Computing DFA directly

You have an NFA and want a DFA?
Simply apply the power-set construction
(and maybe DFA minimization)!

```
The powerset method has exponential state blow up!
I guess I might as well use brute force string matching ...
```



Ingenious algorithm by Knuth, Morris, and Pratt: construct DFA *inductively*:

Suppose we add character $P[j]$ to automaton A_j for $P[0..j)$ to construct A_{j+1}

- add new state and matching transition $\rightsquigarrow$ easy $\xrightarrow{P[j+1]} (j{+}1)$
- for each $c \neq P[j]$, we need $\delta(j, c)$ (transition from (j) when reading c)
- $\delta(j, c)$ $=$ length of the longest prefix of $P[0..j)c$ that is a suffix of $P[1..j)c$
  $=$ state of automaton after reading $P[1..j)c$
  $\leq$ j $\rightsquigarrow$ can use known automaton A_j for that!

> State q means:
> *"we have seen $P[0..q)$ until here (but not any longer prefix of P)"*

Computing DFA directly

You have an NFA and want a DFA?
Simply apply the power-set construction
(and maybe DFA minimization)!

```
    The powerset method has exponential state blow up!
I guess I might as well use brute force string matching ...
```

Ingenious algorithm by Knuth, Morris, and Pratt: construct DFA *inductively*:

Suppose we add character $P[j]$ to automaton A_j for $P[0..j)$ to construct A_{j+1}

- add new state and matching transition $\rightsquigarrow$ easy $\xrightarrow{P[j+1]} (j{+}1)$
- for each $c \neq P[j]$, we need $\delta(j, c)$ (transition from (j) when reading c)
- $\delta(j, c)$ = length of the longest prefix of $P[0..j)c$ that is a suffix of $P[1..j)c$
  = state of automaton after reading $P[1..j)c$
  $\leq j \rightsquigarrow$ can use known automaton A_j for that!

State q means:
"we have seen $P[0..q)$ until here
(but not any longer prefix of P)"

$\rightsquigarrow$ can directly compute A_{j+1} from A_j!

seems to require simulating automata $m \cdot \sigma$ times

Computing DFA efficiently

- **KMP's second insight:** simulations in one step differ only in last symbol

$\rightsquigarrow$ simply maintain state x, the state after reading $P[1..j)$.

  - copy its transitions
  - update x by following transitions for $P[j]$

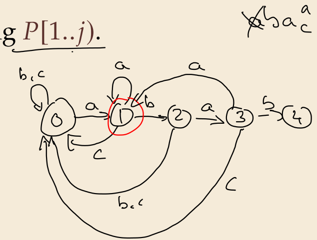
$P[1..j)c$

Computing DFA efficiently

- **KMP's second insight:** simulations in one step differ only in last symbol

⇝ simply maintain state x, the state after reading $P[1..j)$.

- copy its transitions
- update x by following transitions for $P[j]$



```
procedure constructDFA(P[0..m)):
    // δ[q][c] = target state when reading c in state q
    for c ∈ Σ do
        δ[0][c] := 0
    δ[0][P[0]] := 1
    x := 0
    for j = 1, ..., m − 1 do
        for c ∈ Σ do // copy transitions
            δ[j][c] := δ[x][c]
        δ[j][P[j]] := j + 1 // match edge
        x := δ[x][P[j]] // update x
```

Example: $P[0..7)$ = ababaca

$\delta(c,q)$	0	1	2	3	4	5	6
a	1	1	3	1			
b	0	2	0	4			
c	0	0	0	0			

Computing DFA efficiently

- **KMP's second insight:** simulations in one step differ only in last symbol

$\rightsquigarrow$ simply maintain state x, the state after reading $P[1..j)$.

  - copy its transitions
  - update x by following transitions for $P[j]$

procedure constructDFA($P[0..m)$):
    // $\delta[q][c]$ = target state when reading c in state q
    for $c \in \Sigma$ **do**
        $\delta[0][c] := 0$
    $\delta[0][P[0]] := 1$
    $x := 0$
    for $j = 1, \ldots, m-1$ **do**
        for $c \in \Sigma$ **do** *// copy transitions*
            $\delta[j][c] := \delta[x][c]$
        $\delta[j][P[j]] := j+1$ *// match edge*
        $x := \delta[x][P[j]]$ *// update x*

Example: $P[0..7) =$ `ababaca`

$\delta(c,q)$	0	1	2	3	4	5	6
a	1	1	3	1	5	1	7
b	0	2	0	4	0	4	0
c	0	0	0	0	0	6	0

String matching with DFA – Discussion

- **Time:**
  - Matching: n table lookups for DFA transitions
  - building DFA: $\Theta(m\sigma)$ time (constant time per transition edge).

  $\rightsquigarrow$ $\Theta(m\sigma + n)$ time for string matching.

- **Space:**
  - $\Theta(m\sigma)$ space for transition matrix.

👍 **fast matching** time   actually: hard to beat!

👍 total time asymptotically optimal for small alphabet   (for $\sigma = O(n/m)$)

👎 substantial **space overhead**, in particular for large alphabets

5.4 The Knuth-Morris-Pratt algorithm

Failure Links

- Recall: String matching with DFA is fast,
  but needs table of $m \times \sigma$ transitions.
- in fast DFA construction, we used that all simulations differ only by *last* symbol

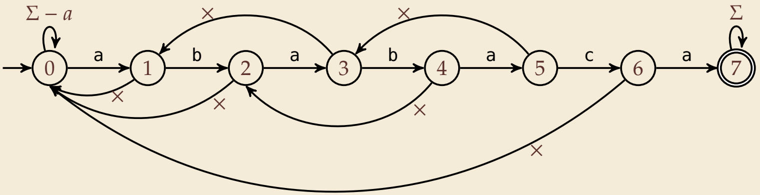
$\rightsquigarrow$ **KMP's third insight:** do this last step of simulation from state x during *matching*!
... but how?

Failure Links

- Recall: String matching with DFA is fast,
  but needs table of $m \times \sigma$ transitions.
- in fast DFA construction, we used that all simulations differ only by *last* symbol

$\rightsquigarrow$ **KMP's third insight:** do this last step of simulation from state x during *matching*!
... but how?

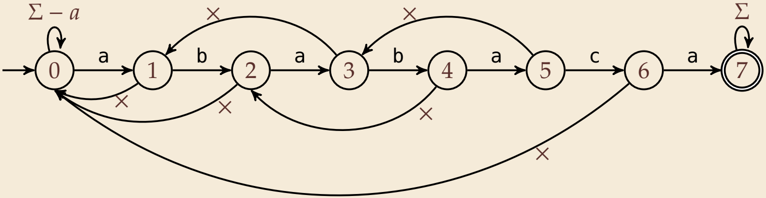
- **Answer:** Use a new type of transition: $\times$, the *failure links*
  - Use this transition (only) if no other one fits.
  - $\times$ *does not consume a character.* $\rightsquigarrow$ might follow several failure links



$\rightsquigarrow$ Computations are deterministic (but automaton is not a classic DFA.)

Failure link automaton – Example

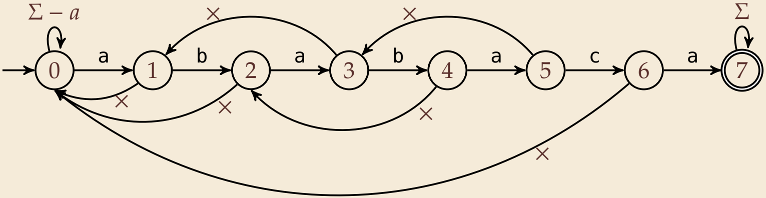
Example: T = ababababaaaca, P = ababaca



T : a b a b a b a a b a b

Failure link automaton – Example

Example: T = `abababaaaca`, P = `ababaca`



T :	a	b	a	b	a	b	a	a	b	a	b	
P :	a	b	a	b	a	×						to state 3
			(a)	(b)	(a)	b	a	×				to state 1
								a	b	a	b	
q :	1	2	3	4	5	3,4	5	3,1,0,1	2	3	4	

(after reading this character)

The Knuth-Morris-Pratt Algorithm

```
procedure KMP(T[0..n), P[0..m)):
    fail[0..m] := failureLinks(P)
    i := 0 // current position in T
    q := 0 // current state of KMP automaton
    while i < n do
        if T[i] == P[q] then
            i := i + 1; q := q + 1
            if q == m then
                return i − q // occurrence found
        else // i.e. T[i] ≠ P[q]
            if q ≥ 1 then
                q := fail[q] // follow one ×
            else
                i := i + 1
    end while
    return NO_MATCH
```

- only need single array *fail* for failure links
- (failureLinks on next slide)

The Knuth-Morris-Pratt Algorithm

```
procedure KMP(T[0..n), P[0..m)):
    fail[0..m] := failureLinks(P)
    i := 0 // current position in T
    q := 0 // current state of KMP automaton
    while i < n do
        if T[i] == P[q] then
            i := i + 1; q := q + 1
            if q == m then
                return i − q // occurrence found
        else // i.e. T[i] ≠ P[q]
            if q ≥ 1 then
                q := fail[q] // follow one ×
            else
                i := i + 1
    end while
    return NO_MATCH
```

- only need single array *fail* for failure links
- (failureLinks on next slide)

Analysis: (matching part)

- always have $fail[j] < j$ for $j \geq 1$

$\rightsquigarrow$ in each iteration

  - either advance position in text ($i := i + 1$)
  - or shift pattern forward (guess $i - q$)

- each can happen at most n times

$\rightsquigarrow$ $\leq 2n$ symbol comparisons!

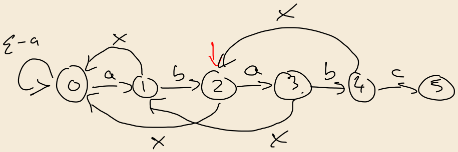
Computing failure links

- failure links point to error state x (from DFA construction)

$\rightsquigarrow$ run same algorithm, but store $fail[j] := x$ instead of copying all transitions

```
procedure failureLinks(P[0..m]):
    fail[0] := 0
    x := 0
    for j := 1, . . . , m − 1 do
        fail[j] := x
        // update failure state using failure links:
        while P[x] ≠ P[j]
            if x == 0 then
                x := −1;  break
            else
                x := fail[x]
        end while
        x := x + 1
    end for
```

a b a b a c a



Computing failure links

- failure links point to error state x (from DFA construction)

$\rightsquigarrow$ run same algorithm, but store $\mathit{fail}[j] := x$ instead of copying all transitions

```
procedure failureLinks(P[0..m]):
    fail[0] := 0
    x := 0
    for j := 1, ..., m − 1 do
        fail[j] := x
        // update failure state using failure links:
        while P[x] ≠ P[j]
            if x == 0 then
                x := −1;   break
            else
                x := fail[x]
        end while
        x := x + 1
    end for
```

Analysis:

- m iterations of for loop
- while loop always decrements x
- x is incremented only once per iteration of for loop

$\rightsquigarrow$ $\le m$ iterations of while loop *in total*

$\rightsquigarrow$ $\le 2m$ symbol comparisons

Knuth-Morris-Pratt – Discussion

- **Time:**
  - $\leq 2n + 2m = O(n + m)$ character comparisons
  - clearly must at least *read* both T and P in the worst case
  - $\rightsquigarrow$ KMP has optimal worst-case complexity

- **Space:**
  - $\Theta(m)$ space for failure links

👍 total time asymptotically optimal (for any alphabet size)

👍 reasonable extra space

The KMP prefix function

- It turns out that the failure links are useful beyond KMP
- a slight variation is (more?) widely used: (for historic reasons)
  the (KMP) *prefix function* $F : [1..m-1] \to [0..m-1]$:

  $F[j]$ is the length of the longest prefix of $P[0..j]$ that is a suffix of $P[1..j]$.

- Can show: $\mathit{fail}[j] = F[j-1]$ for $j \geq 1$, and hence

$$\boxed{\mathit{fail}[q] = \textbf{length of the longest prefix of } P[0..q) \text{ that is a suffix of } P[1..q).}$$

memorize this!

- EAA Buch: String indices are 1-based, but definition of failure links matches! $\boxed{\Pi_P(q) = \mathit{fail}[q]}$
  $\Pi_P : [1..m] \to [0..m-1]$ with $\Pi_P(q) = \max\{k \in \mathbb{N}_0 : k < q \wedge P[0..k) \sqsupset P[0..q)\} = \mathit{fail}[q]$

5.5 The Aho-Corasick Algorithm

Multiple-Pattern Matching

- So far: process a single pattern P in one pass over T.

$\rightsquigarrow$ Cannot beat time $\Omega(p \cdot n)$

Multiple-Pattern Matching

- So far: process a single pattern P in one pass over T.

$\rightsquigarrow$ Cannot beat time $\Omega(p \cdot n)$

- The essence of KMP can be generalized to deal with several patterns!

$\rightsquigarrow$ The ***Aho-Corasick Algorithm***

Multiple-Pattern Matching

- So far: process a single pattern P in one pass over T.

$\rightsquigarrow$ Cannot beat time $\Omega(p \cdot n)$

- The essence of KMP can be generalized to deal with several patterns!

$\rightsquigarrow$ The ***Aho-Corasick Algorithm***

The Multiple-Pattern Matching Problem

- **Given:** text $T[0..n)$, patterns $P[0..p)$, $P[r] = P_r[0..m_r)$
  - all over $\Sigma = [0..\sigma)$ for **constant** σ
  - total length of patterns: $m := \sum_{0 \le r < p} m_r$
- **Goal:** all matches, i. e., all pairs (i, r) such that $P_r = T[i..i + m_r)$

Multiple-Pattern Matching

- So far: process a single pattern P in one pass over T.

$\rightsquigarrow$ Cannot beat time $\Omega(p \cdot n)$

- The essence of KMP can be generalized to deal with several patterns!

$\rightsquigarrow$ The ***Aho-Corasick Algorithm***

The Multiple-Pattern Matching Problem

- **Given:** text $T[0..n)$, patterns $P[0..p)$, $P[r] = P_r[0..m_r)$
  - all over $\Sigma = [0..\sigma)$ for **constant** σ
  - total length of patterns: $m := \sum_{0 \le r < p} m_r$
- **Goal:** all matches, i. e., all pairs (i, r) such that $P_r = T[i..i + m_r)$

single pass!

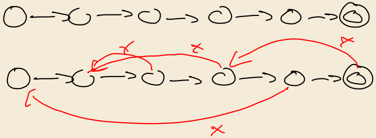
Aho-Corasick can do this with $O(m)$ preprocessing and $O(n + output)$ matching time!
Here *output* is the number of match pairs (i, r).

Aho-Corasick Automaton – Overview

Aho-Corasick Automaton

1. Build trie A from patterns $P[0..p)$.
2. Add *failure links* to A.
3. Add *output links* to A. $\boxed{\text{NEW}}$

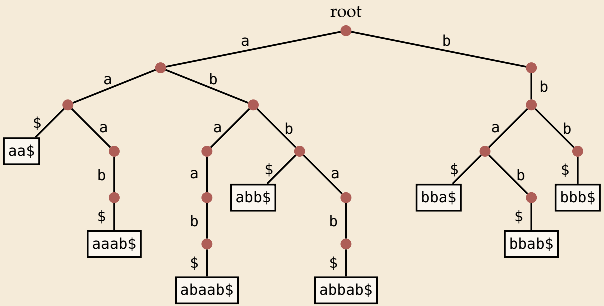
KMP



Recap: Tries

- efficient dictionary data structure for strings
- name from re**trie**val, but pronounced "try"
- tree based on symbol comparisons

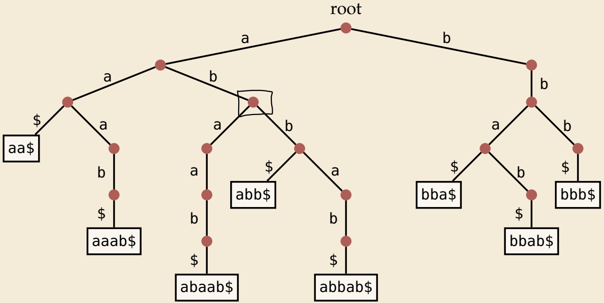
- **Example**:
  {`aa$`, `aaab$`, `abaab$`, `abb$`,
  `abbab$`, `bba$`, `bbab$`, `bbb$`}



Recap: Tries

- efficient dictionary data structure for strings
- name from re**trie**val, but pronounced "try"
- tree based on symbol comparisons

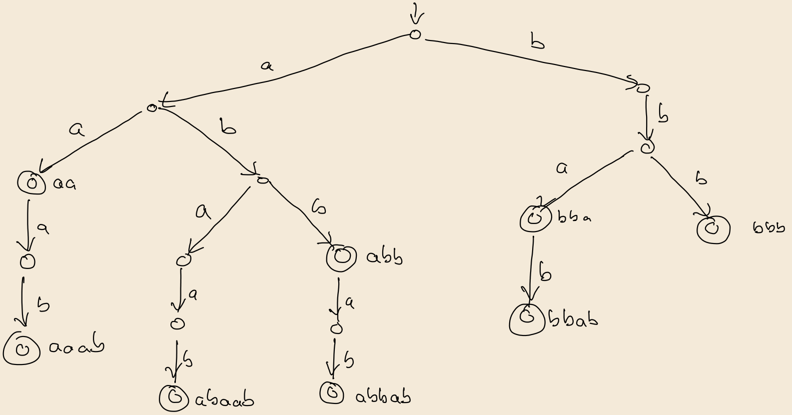
- **Example**:
  `{aa$, aaab$, abaab$, abb$,`
  `abbab$, bba$, bbab$, bbb$}`



When stored string is a strict prefix of another, internal nodes can correspond to strings.

{aa$, aaab$, abaab$, abb$,
abbab$, bba$, bbab$, bbb$}

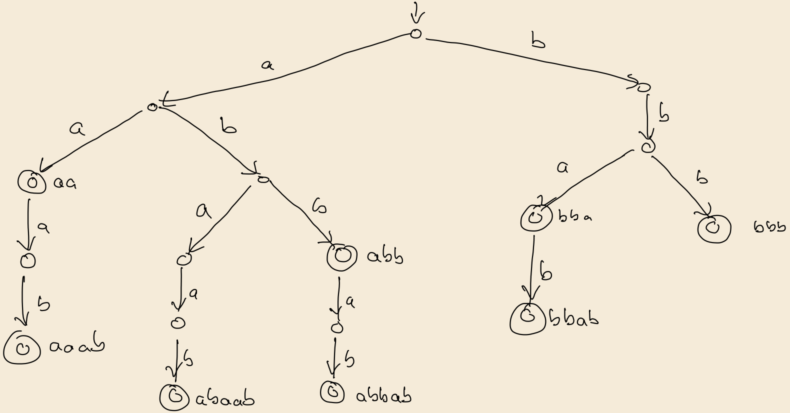
w/o $



Aho-Corasick Automaton – Adding Failure Links

Trie for P_r corresponds to match-edges-only NFA.

⇝ Interpreting the trie as automaton, add ε-edges back to the root.



Aho-Corasick Automaton – Adding Failure Links

Trie for P_r corresponds to match-edges-only NFA.

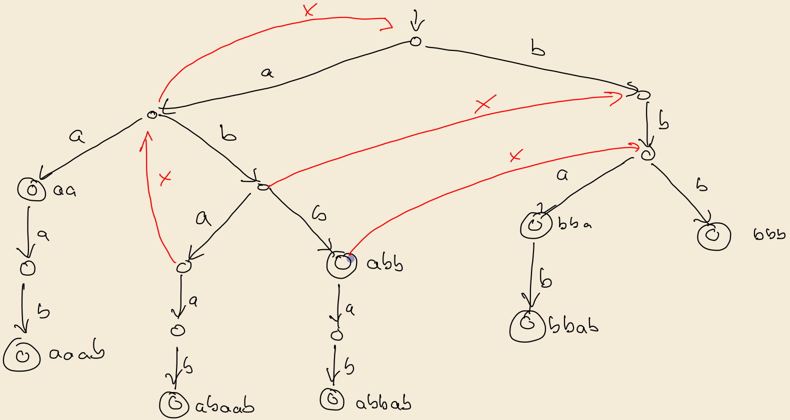
$\rightsquigarrow$ Interpreting the trie as automaton, add ε-edges back to the root.

- ▶ as in KMP, instead of determinizing the automation classically, we again use failure links

$\rightsquigarrow$ construction as for KMP using failure state x, repeated for each word.

~~a~~baab

abb



Aho-Corasick Automaton – Output Links

An automaton state might contain other patterns as suffix $\rightsquigarrow$ *must output match(es)! But we are not in an accepting state, so direct use of automaton so far would miss occurrence!*

- ***output links:*** each state points to longest suffix pattern (if any).
  - During matching, traverse linked list of matches and output each $\rightsquigarrow$ $O(output)$ cost overall
  - Computation of output links similar to failure links! (handle patterns by length)

in
ting