Exact string matching

Pattern search in a sequence

- $T[1..n]$ sequence (text, string)
- $P[1..m]$ pattern
- *Problem*: locate all *exact* occurrences of P in T (*variants*: check if P occurs in T, count the number of occurrences)

Two frameworks

- *Static pattern*: preprocess pattern – scan text(s)
  - Knuth-Morris-Pratt algorithm
  - Boyer-Moore algorithm
  - Karp-Rabin algorithm
  - … and many others

- *Static text*: preprocess text – lookup for pattern(s)
  - suffix tree
  - … and several others (suffix array, DAWG, position heap, BWT-index, …)

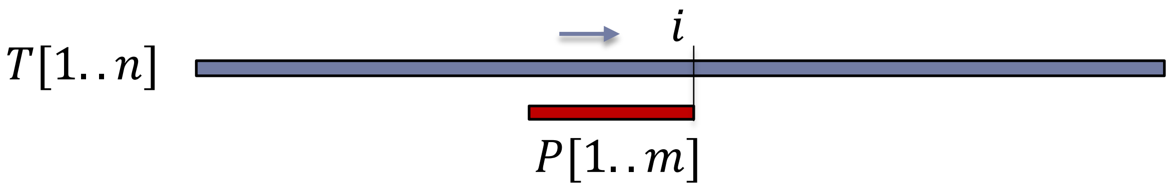
Static pattern: naïve algorithm

Naïve algorithm: $O(n \cdot m)$

T=aaaaaa...aa=a^{n}

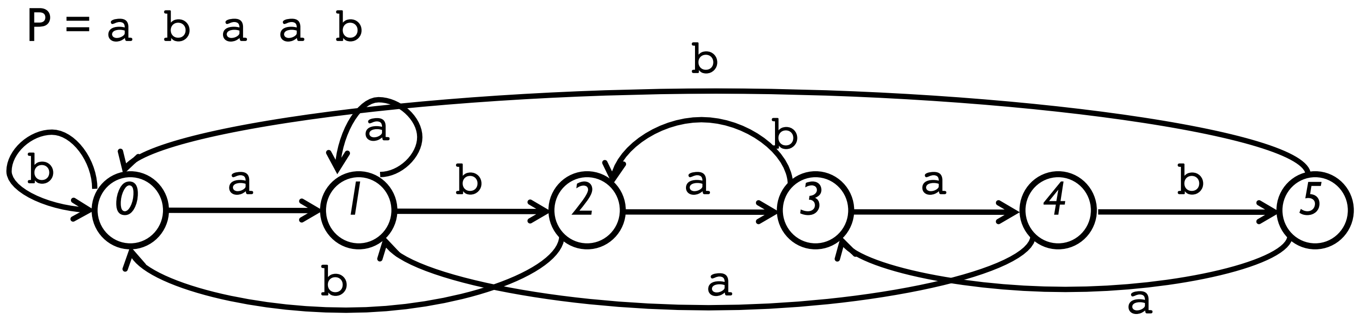
P=aa...ab=a$^{m\text{-}1}$b

String matching by text scan



- Assume we scan the text left-to-right trying to locate P
- All we have to "keep track of" depend on the m last characters in the text
- The search can be done by a finite automaton that depends only on m and alphabet size!

Pattern automaton

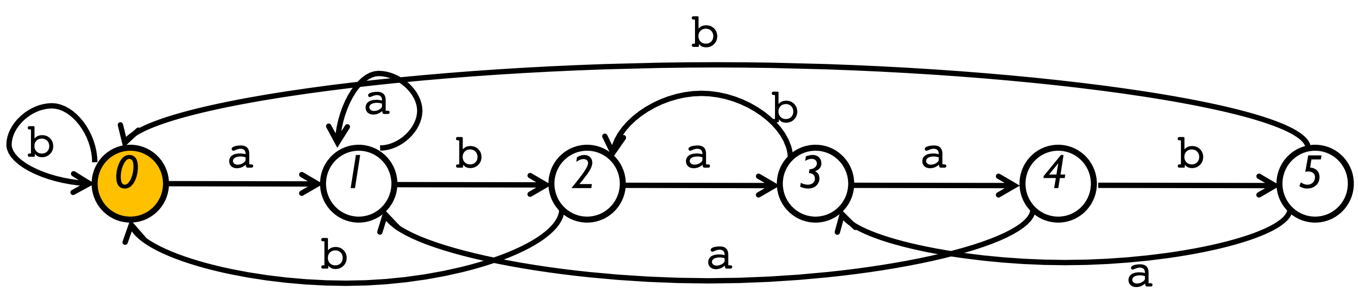


- Each state corresponds to a position in P
- State q corresponds to prefix $P[1..q]$
- *Invariant*: when in state q, $P[1..q]$ is the longest prefix of P which is a suffix of the prefix of T read so far

Pattern automaton

T = a b a b a a a b a b a a b a …

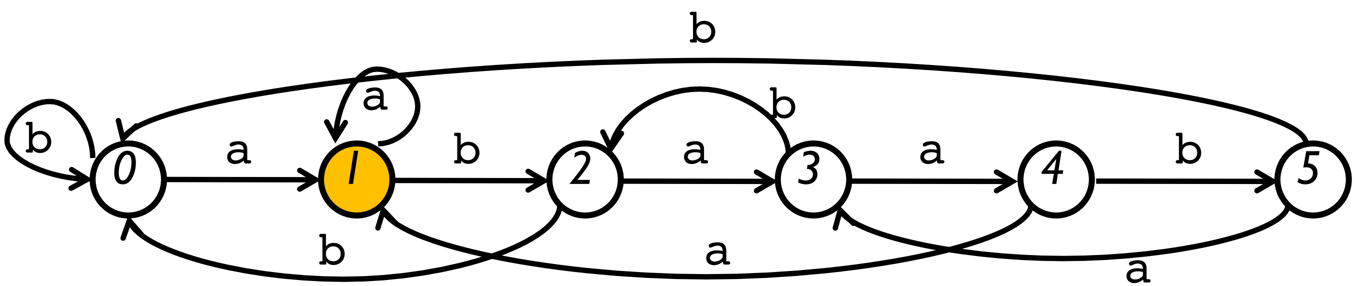
P = a b a a b



Pattern automaton

T = a b a b a a a b a b a a b a ...

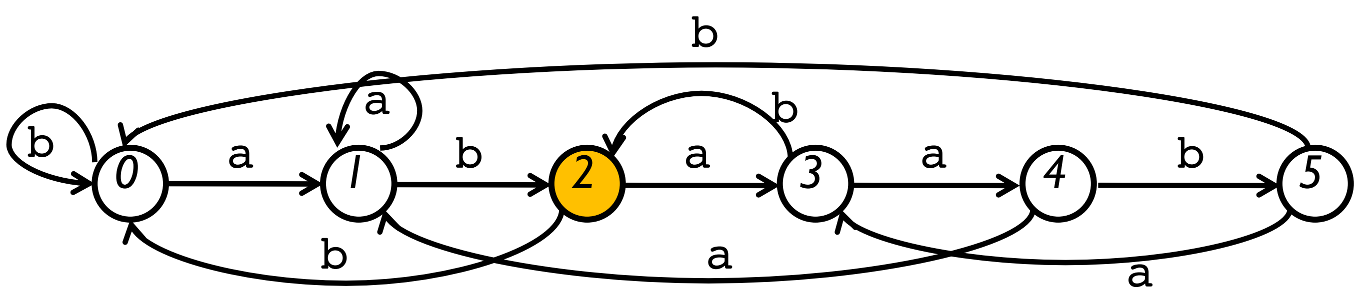
P = a b a a b



Pattern automaton

T = a b a b a a a b a b a a b a ...

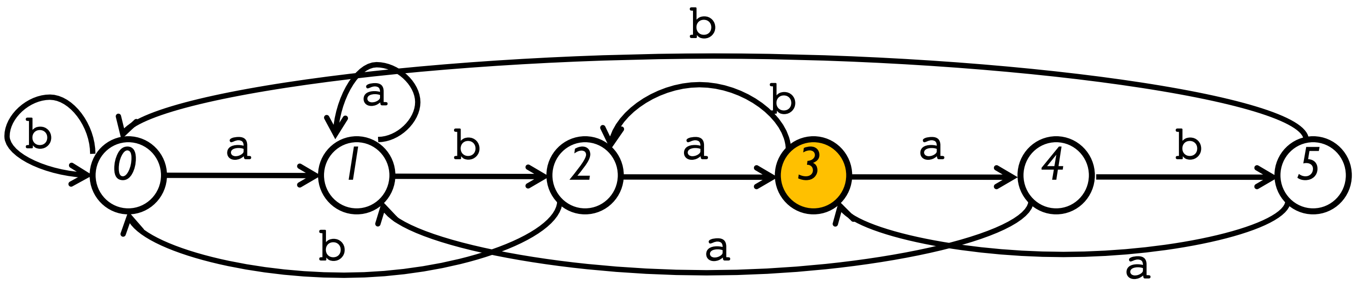
P = a b a a b



Pattern automaton

T = a b a b a a a b a b a a b a …

P = a b a a b

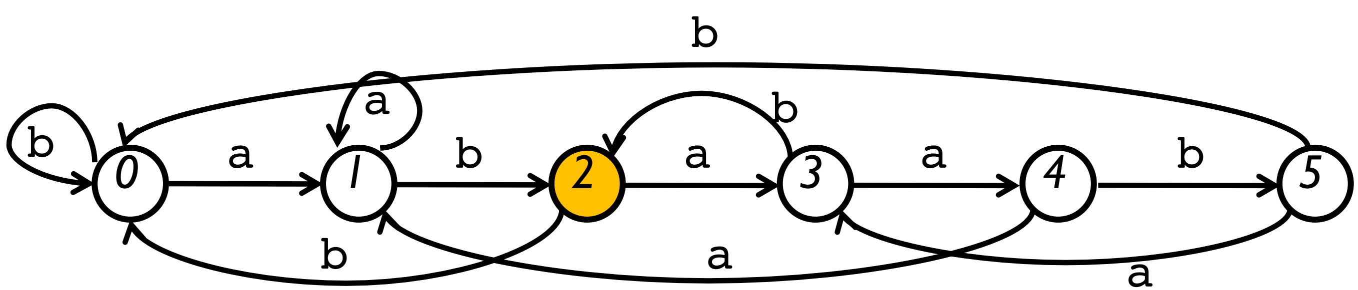


Pattern automaton

T = a b a b a a a b a b a a b a …

P = a b a a b

a b a a b

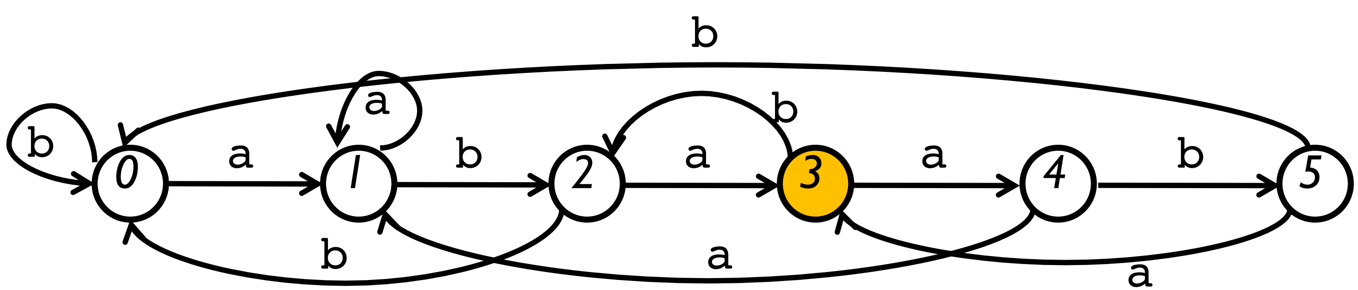


Pattern automaton

T = a b a b a a a b a b a a b a …

P = a b a a b

a b a a b

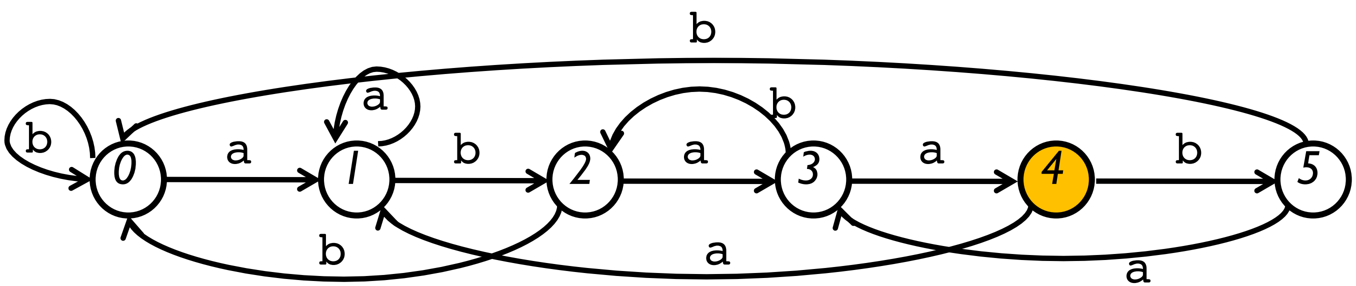


Pattern automaton

T = a b a b a a a b a b a a b a ...

P = a b a a b

a b a a b



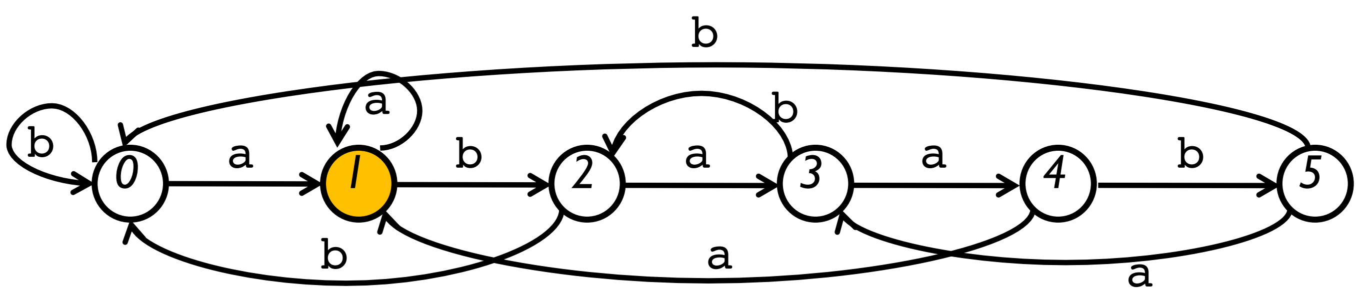
Pattern automaton

T = a b a b a a a b a b a a b a …

P = a b a a b

a b a a b

a b a a b



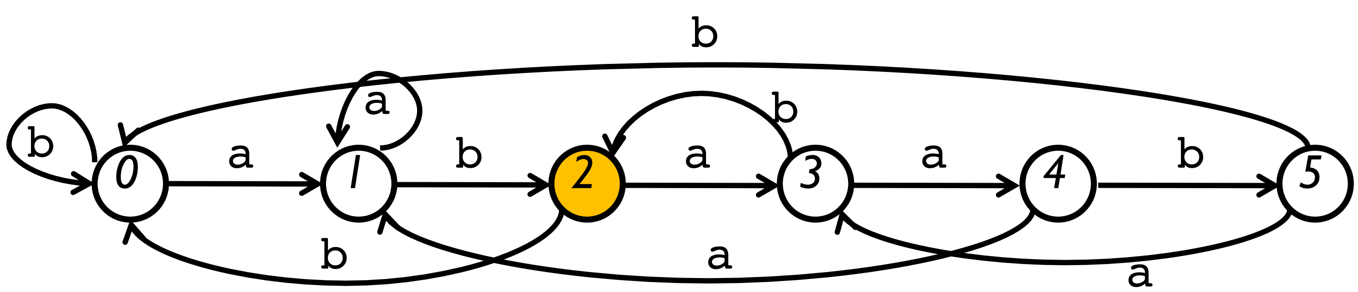
Pattern automaton

T = a b a b a a a b a b a a b a …

P = a b a a b

a b a a b

a b a a b



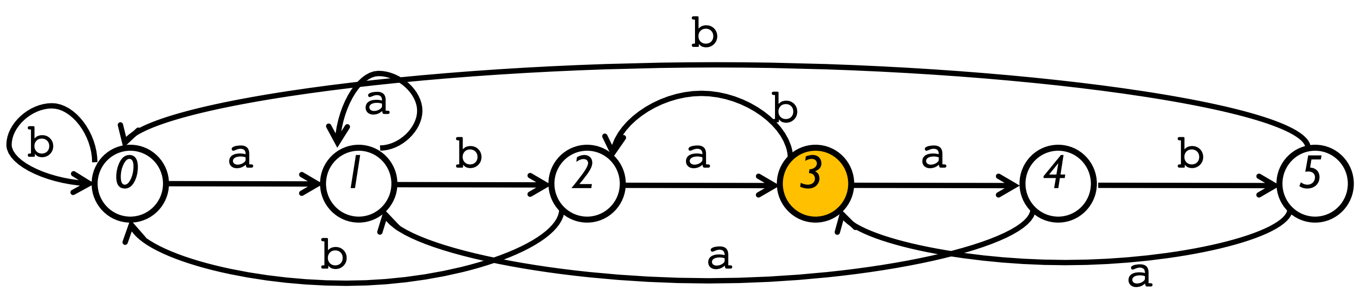
Pattern automaton

T = a b a b a a a b a b a a b a …

P = a b a a b

a b a a b

a b a a b



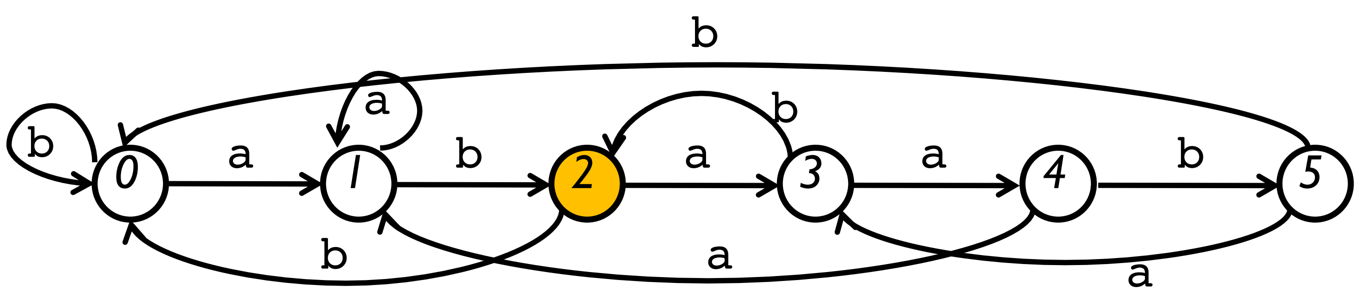
Pattern automaton

T = a b a b a a a b a b a a b a ...

P = a b a a b

a b a a b

a b a a b



Pattern automaton

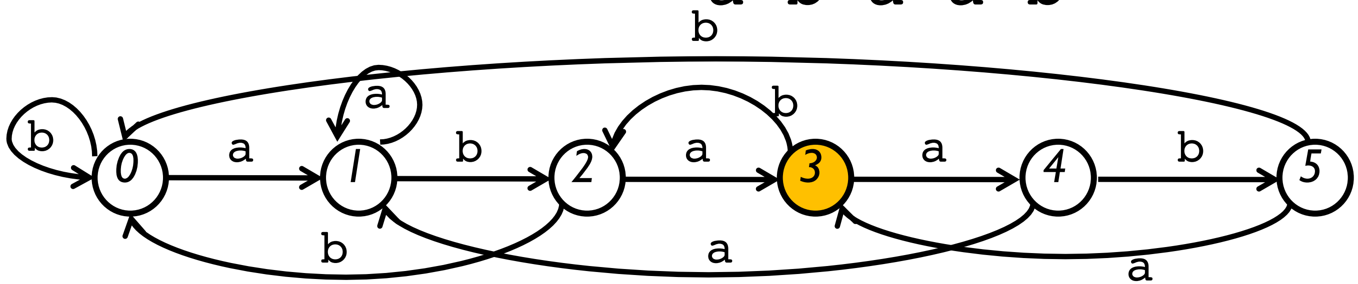
T = a b a b a a a b a b a a b a …

P = a b a a b

a b a a b

a b a a b

a b a a b



Pattern automaton

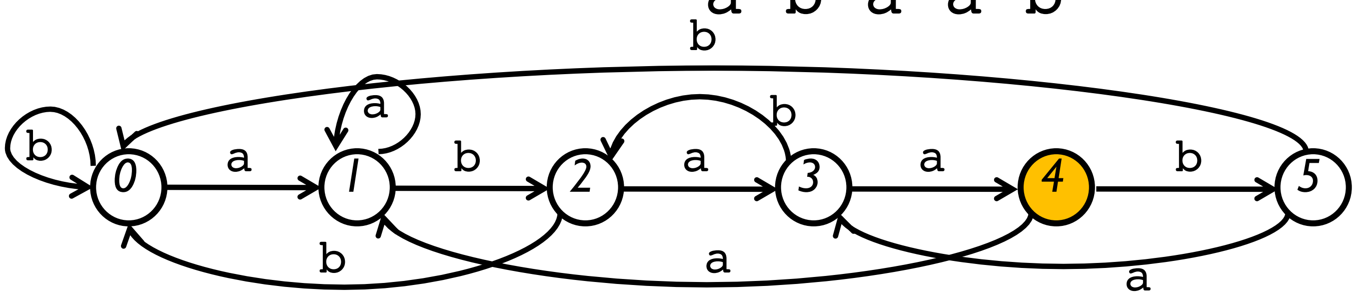
T = a b a b a a a b a b a a b a …

P = a b a a b

a b a a b

a b a a b

a b a a b



Pattern automaton

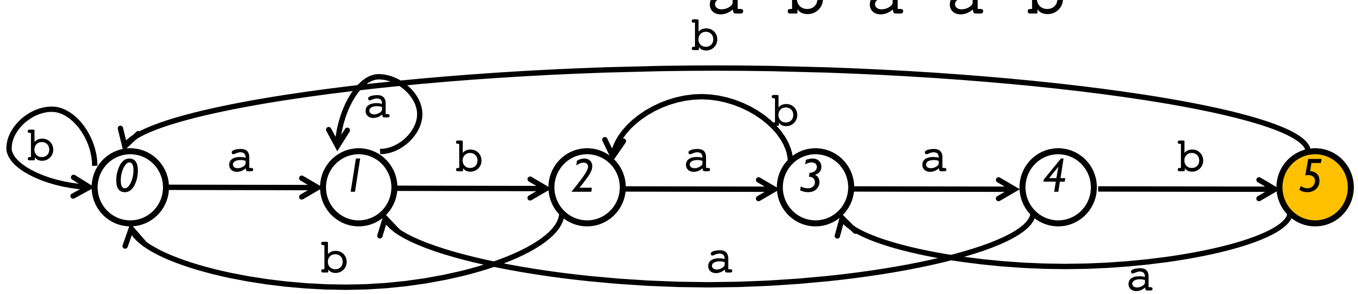
T = a b a b a a a b a b a a b a …

P = a b a a b

a b a a b

a b a a b

a b a a b



Pattern automaton

T = a b a b a a a b a b a a b a …

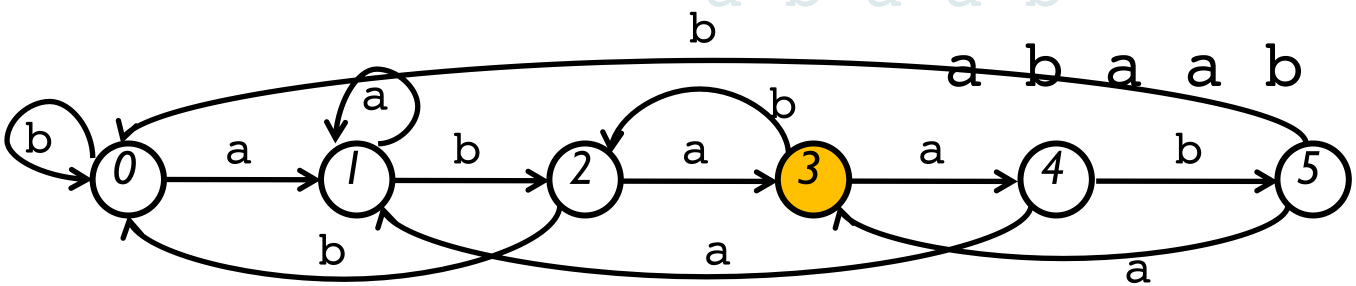
P = a b a a b

a b a a b

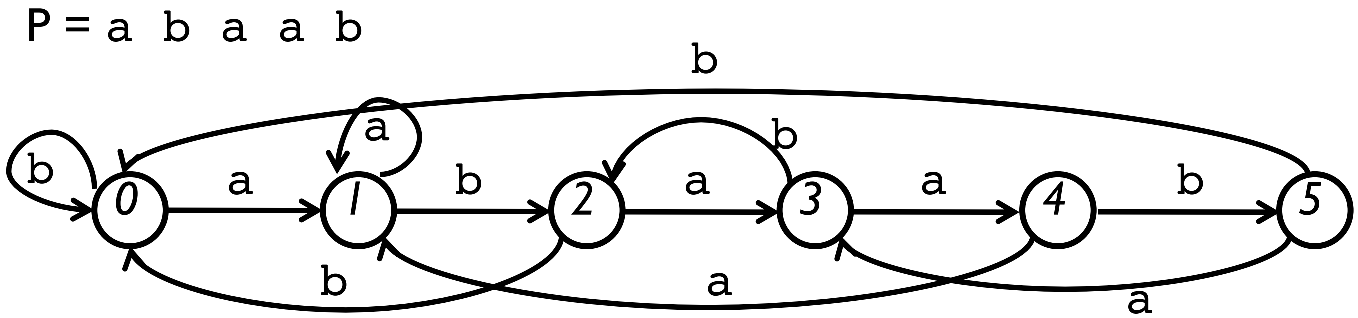
a b a a b

a b a a b

a b a a b



Pattern automaton



- State q corresponds to prefix $P[1..q]$
- *Invariant*: when in state q, $P[1..q]$ is the longest prefix of P which is a suffix of the prefix of T read so far
- If the next letter $T[i] = P[q+1]$, move to state $q+1$, otherwise follow the "failure transition" backwards
- Failure transition $q \underset{a}{\rightarrow} q'$ where $P[1..q']$ is the longest suffix of $P[1..q]a$

Pattern automaton

- Size of the automaton: $O(m \cdot |A|)$
- It can be constructed in time $O(m \cdot |A|)$
- Running time: $O(n \cdot \log |A|)$

Pattern automaton

- Size of the automaton: $O(m \cdot |A|)$
- It can be constructed in time $O(m \cdot |A|)$
- Running time: $O(n \cdot \log |A|)$

- It is (almost) linear, what can we do better?

Pattern automaton

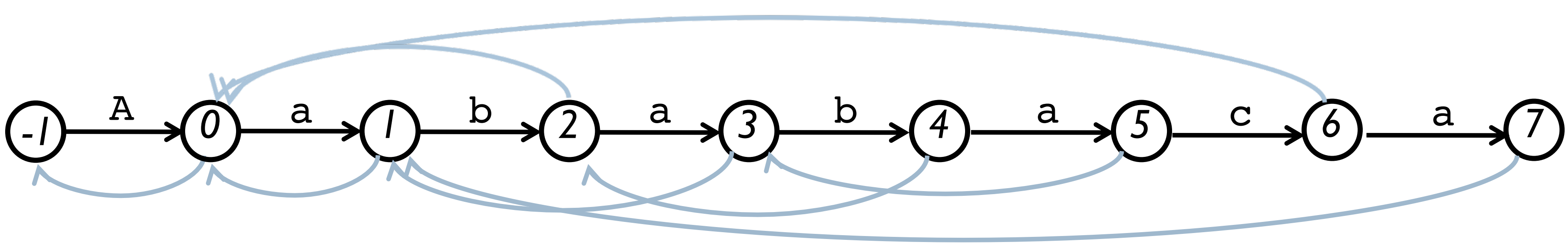
- Size of the automaton: $O(m \cdot |A|)$
- It can be constructed in time $O(m \cdot |A|)$
- Running time: $O(n \cdot \log |A|)$

- It is (almost) linear, what can we do better?
- Knuth-Morris-Pratt: algorithm without dependence on $|A|$!

Knuth-Morris-Pratt algorithm

- Differences with automaton algorithm:
  - failure transition (here called *failure function*) tells us where we go if we have a mismatch ⇒ only one failure transition for each state!
  - Failure function: $q \rightarrow q'$ where $P[1..q']$ is the longest proper suffix of $P[1..q]$
  - if we have a mismatch, we don't advance in the text, but stay at the same position

Knuth-Morris-Pratt "automaton" for abbabaca

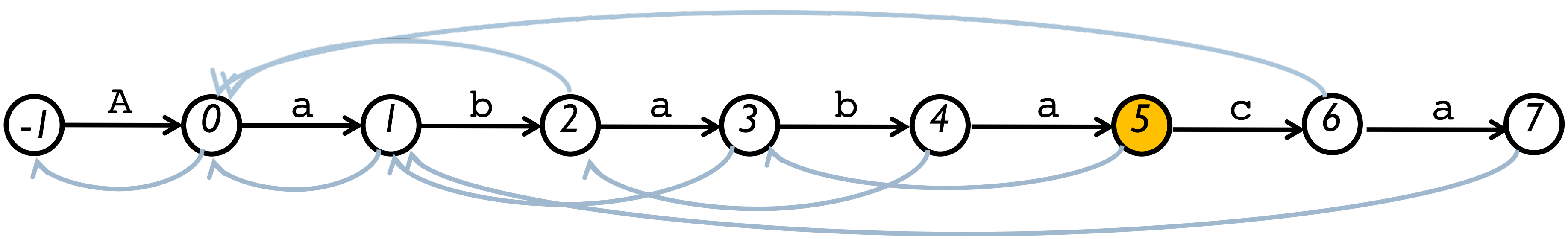


Knuth-Morris-Pratt algorithm: run

≠c

T = * * a b a b a * * * * * * …

P = a b a b a c a

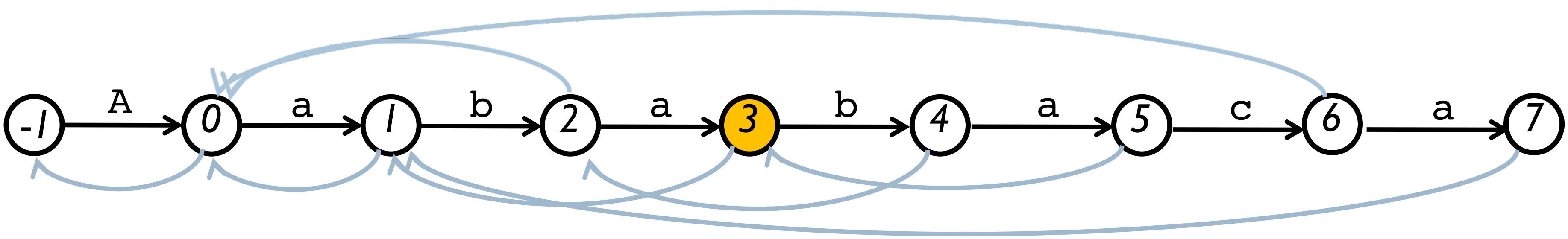


Knuth-Morris-Pratt algorithm: run

≠b

T = * * a b a b a * * * * * * …

P = a b a b a c a

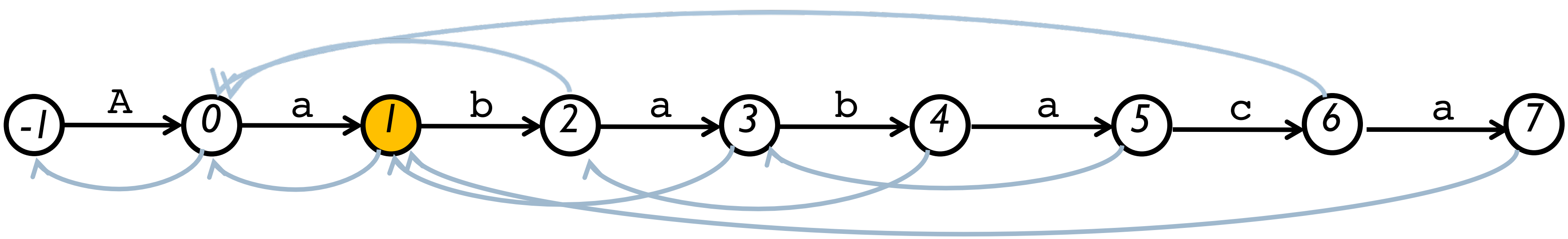


Knuth-Morris-Pratt algorithm: run

≠b

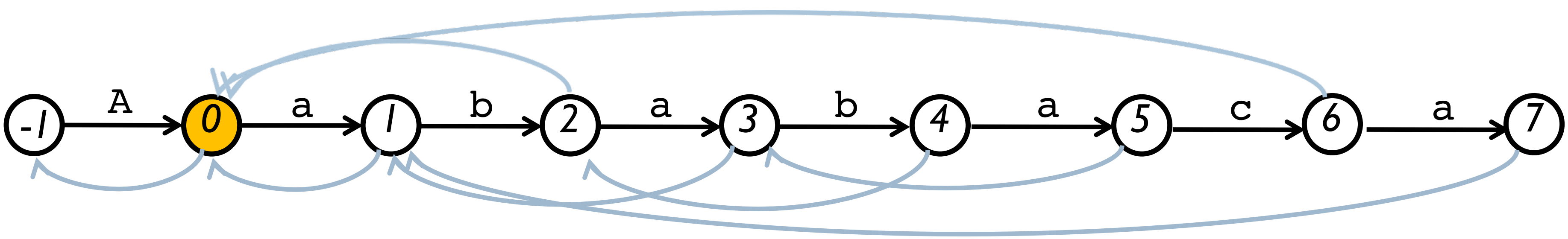
T = * * a b a b a * * * * * * …

P = a b a b a c a



Knuth-Morris-Pratt algorithm: run

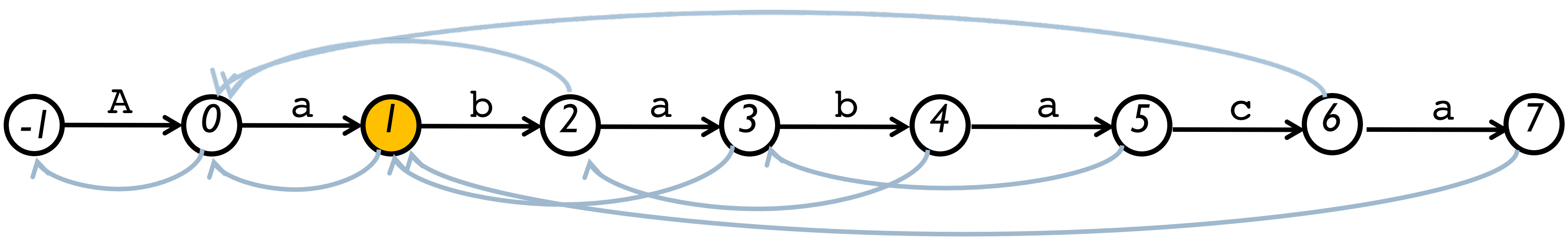
T = * * a b a b a a * * * * * …
P = a b a b a c a



Knuth-Morris-Pratt algorithm: run

T = * * a b a b a a * * * * * …

P = a b a b a c a



Knuth-Morris-Pratt algorithm

$f : [1..m] \to [-1..m-1]$ failure function

KMP($T[1..n]$,f)

$j = 0$ // pointer in P

for $i = 1$ **to** n **do** {

while $j \geq 0$ **and** $P[j+1] \neq T[i]$ **do** {

$j = f(j)$ }

$j = j + 1$

if $j == m$ **then** {

output(occurrence of P at position $(i - m)$)

$j = f(j)$ }

}

Failure function

- $f(q) = \max\{k \mid k < q \text{ and } P[1..k] \text{ is a suffix of } P[1..q]\}$
- f can be computed in time $O(m)$
- interestingly, the computation is similar to the run of KMP (compute values left-to-right, P slid against itself)

$P =$		a	b	a	b	a	c	a
q	0	1	2	3	4	5	6	7
$f(q)$	−1	0	0	1	2	3	0	1

Computation of the failure function

FF($P[1..m]$)
    $f[0] = -1$
    $f[1] = 0$
    $k = 0$
    for $j = 2$ **to** m **do** {
        while $k \geq 0$ **and** $P[k+1] \neq P[j]$ **do** {
            $k = f(k)$ }
        $k = k + 1$
        $f(j) = k$
    }

Quiz 9

Question 1 **1 pts**

Compute failure function f for a string s="abacaba".

f(0)=

f(1)=

f(2)=

f(3)=

f(4)=

f(5)=

f(6)=

f(7)=

Failure function

- $f(q) = \max\{k | k < q \text{ and } P[1..k] \text{ is a suffix of } P[1..q]\}$
- f can be computed in time $O(m)$
- interestingly, the computation is similar to the run of KMP (compute values left-to-right, P slid against itself)
- An *optimized* failure function can be computed in $O(m)$ as well

P =		a	b	a	b	a	c	a
q	0	1	2	3	4	5	6	7
f(*q*)	−1	0	0	1	2	3	0	1
h(*q*)	−1	0	0	0	0	3	0	1

Knuth-Morris-Pratt: summary

- Preprocessing: computing failure function in $O(m)$ time
- Search: scanning the text in $O(n)$ time (*amortized* $O(1)$ time per character)
- *Note*: worst-case time per character is $O(\log m)$

- *History*: Morris-Pratt (1970), Knuth-Morris-Pratt (1976), Matiyasevich (1971)

- *In practice*: Boyer-Moore(-Horspool) algorithm, $O(\frac{n}{|A|})$ time *on average*

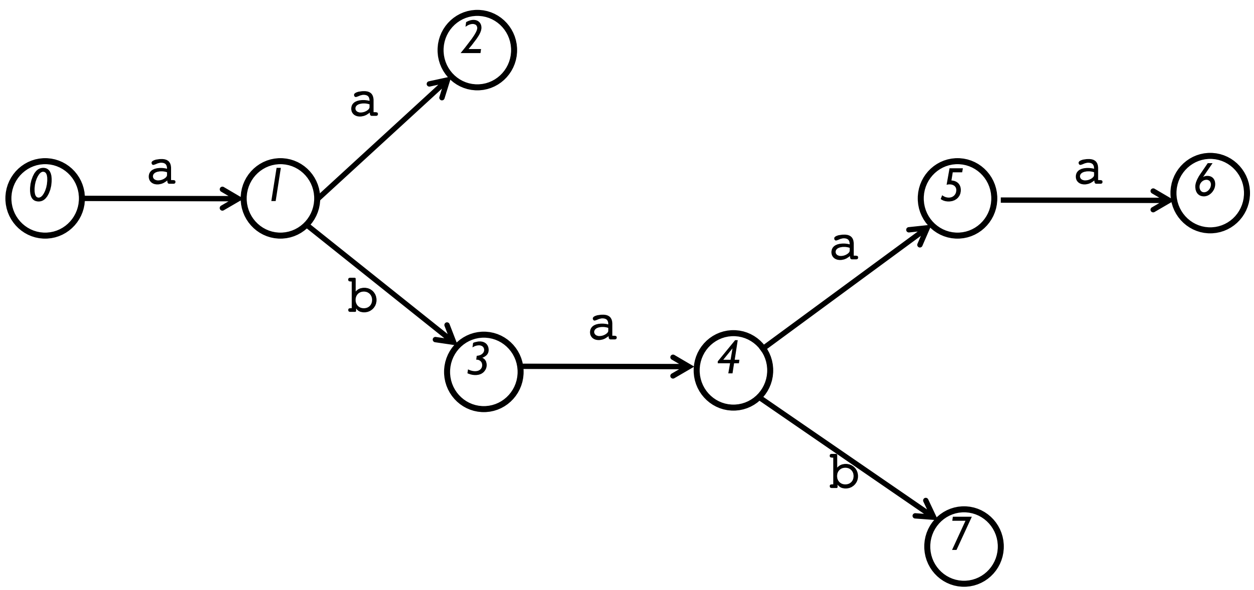
Aho-Corasick algorithm (1984)

- Ideas of the Knuth-Morris-Pratt algorithm can be generalized to several patterns ⇒ Aho-Corasick algorithm (1974)

Aho-Corasick algorithm

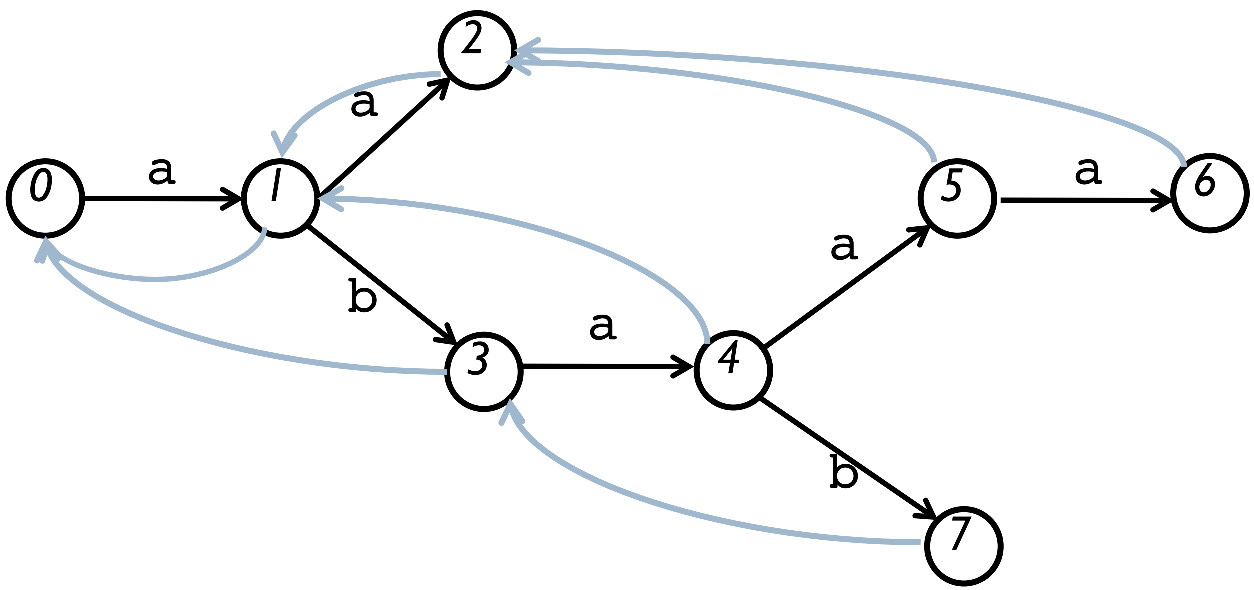
- S={aa, abaaa, abab}
- Construct the *trie* of S

- every string of the set is "spelled" starting from root
- edges outgoing from a node are labeled by different characters
- can be viewed as an automaton recognizing the given set of strings (or all their prefixes)



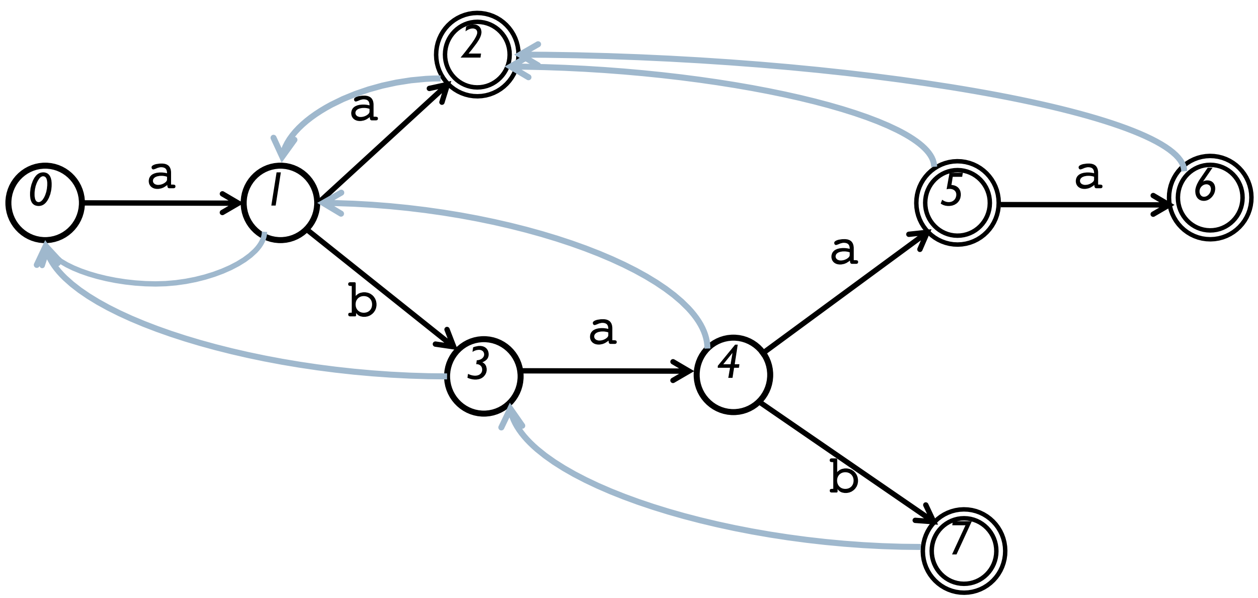
Aho-Corasick algorithm

- S={aa, abaaa, abab}
- Construct the *trie* of S, compute the failure function



Aho-Corasick algorithm

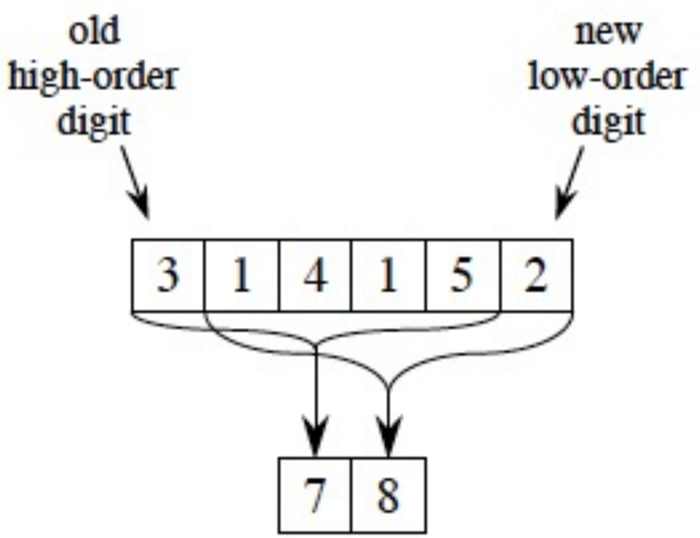
- S={aa, abaaa, abab}
- Construct the *trie* of S, compute the failure function, identify final states



Can be constructed in time $O(m)$, where m is the **total** size of patterns in S

Karp-Rabin algorithm

- computing hash of t_{i+1} from hash of t_i (illustration):



$$
\begin{aligned}
14152 &\equiv (31415 - \overset{\text{old high-order digit}}{3}\cdot 10000)\overset{\text{shift}}{\cdot 10} + \overset{\text{new low-order digit}}{2} \pmod{13} \\
&\equiv (7 - 3\cdot 3)\cdot 10 + 2 \pmod{13} \\
&\equiv 8 \pmod{13}
\end{aligned}
$$

- once a candidate ($T[i..i+m-1]$ with the same hash) is found, we verify it by comparing P and $T[i..i+m-1]$ letter-by-letter