Pattern matching

Given

a word w (the text)
and a regular expression p (the pattern),

determine

the smallest number k' such that
some [k,k']-factor of w belongs to L(p).

PatternMatchingNFA(t, p)

Input: text $t = a_1 \dots a_n \in \Sigma^+$, pattern $p \in \Sigma^*$

Output: the first occurrence k of p in t, or $\perp$ if no such occurrence exists.

1 $A \leftarrow \mathit{RegtoNFA}(\Sigma^* p)$
2 $S \leftarrow \{q_0\}$
3 **for all** $k = 0$ to $n - 1$ **do**
4   **if** $S \cap F \neq \emptyset$ **then return** k
5     $S \leftarrow \delta(S, a_i)$
6 **return** $\perp$

Line 1 takes O(m) time
At most n loop iterations
One iteration takes O(s^2) time where s number of states of A

Since s=O(m), total runtime is O(m+nm^2)=O(nm^2)

PatternMatchingDFA(t, p)

Input: text $t = a_1 \dots a_n \in \Sigma^+$, pattern p

Output: the first occurrence k of p in t, or $\perp$ if no such occurrence exists.

1 $A \leftarrow NFAtoDFA(RegtoNFA(\Sigma^* p))$

2 $q \leftarrow q_0$

3 **for all** $i = 0$ to $n - 1$ **do**

4   **if** $q \in F$ **then return** k

5     $q \leftarrow \delta(q, a_i)$

6 **return** $\perp$

Line 1 takes 2^O(m) time

At most n loop iterations

One iteration takes constant time

Total runtime is O(n) + 2^O(m)

The word case

- The naive algorithm has O(nm) runtime

- We give an algorithm with O(n + m) runtime, even when the size of the alphabet is not fixed.

- Consider the minimal DFA for Sigma* p (p the pattern)

  - The DFA must contain one state for each prefix of p. (Why ?)

  - We construct a DFA with exactly one state for each prefix, which is therefore the minimal DFA

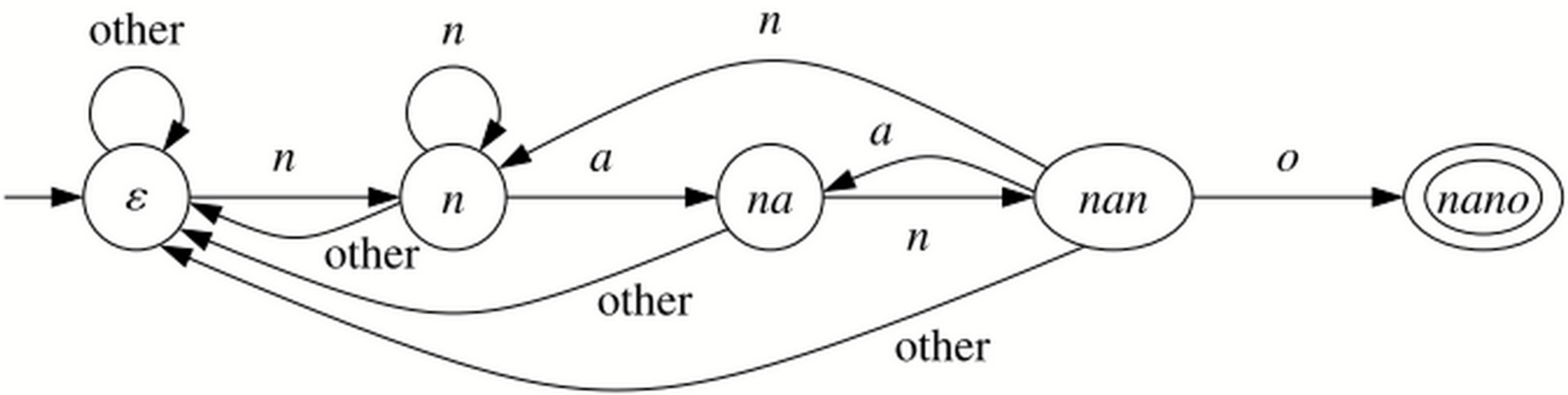
The minimal DFA

Intuition: the DFA keeps track of how close it is to reading the pattern

More precisely: if the DFA is in state p' , then p' is the longest prefix of p that the DFA has just read and has not been yet 'spoilt'.

The general rule is:

> If the DFA is in state $v \in \Sigma^*$ and it reads a letter α, it moves to the largest suffix of $v\alpha$ that is also a prefix of p.



Definition 7.2 *We denote by $ol(w)$ the longest suffix of w that is a prefix of p. In other words, $ol(w)$ is the unique longest word of the set*

$$\{u \in \Sigma^* \mid \exists v, v' \in \Sigma^*. w = vu \wedge p = uv'\}$$

Definition 7.3 *The* eager DFA of the pattern p *is the tuple* **eagerDFA**$(p) = (Q_e, \Sigma, \delta_e, q_{0e}, F_e)$, *where :*

- Q_e *is the set of prefixes of* p *(including* ε*);*
- *for every* $u \in Q_e$*, for every* $\alpha \in \Sigma$*:* $\delta_e(u, \alpha) = ol(u\alpha)$*;*
- $q_{0e} = \varepsilon$*; and*
- $F_e = \{p\}$

New pattern-matching algorithm: replace

$$A \leftarrow NFAtoDFA(RegtoNFA(\Sigma^* p))$$

by

$$A \leftarrow eagerDFA(p)$$

Variable alphabet size

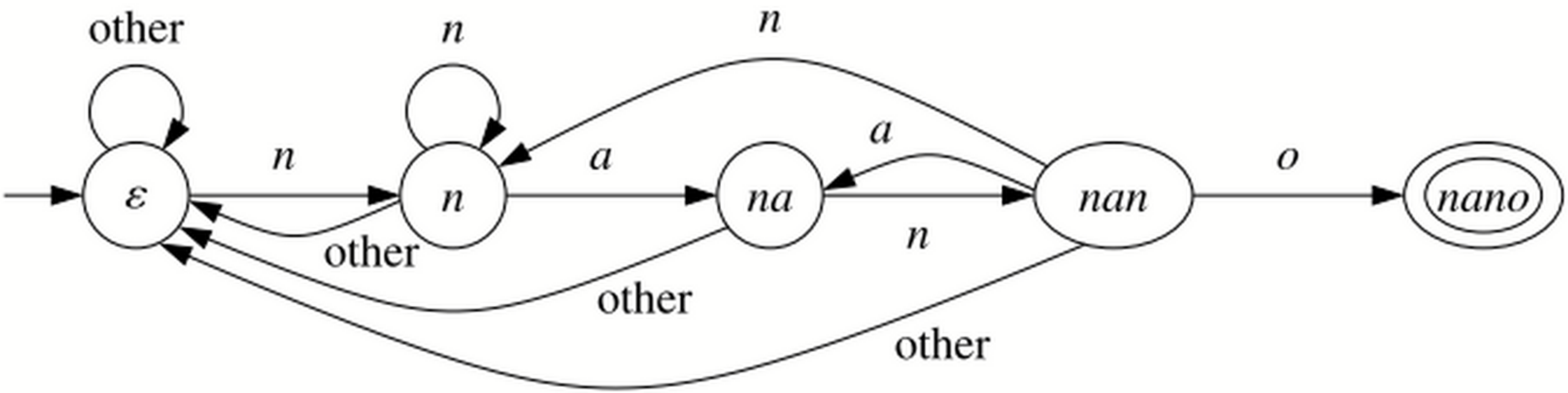
The eager DFA of a pattern of length m has
- m+1 states and
- m |Sigma| transitions

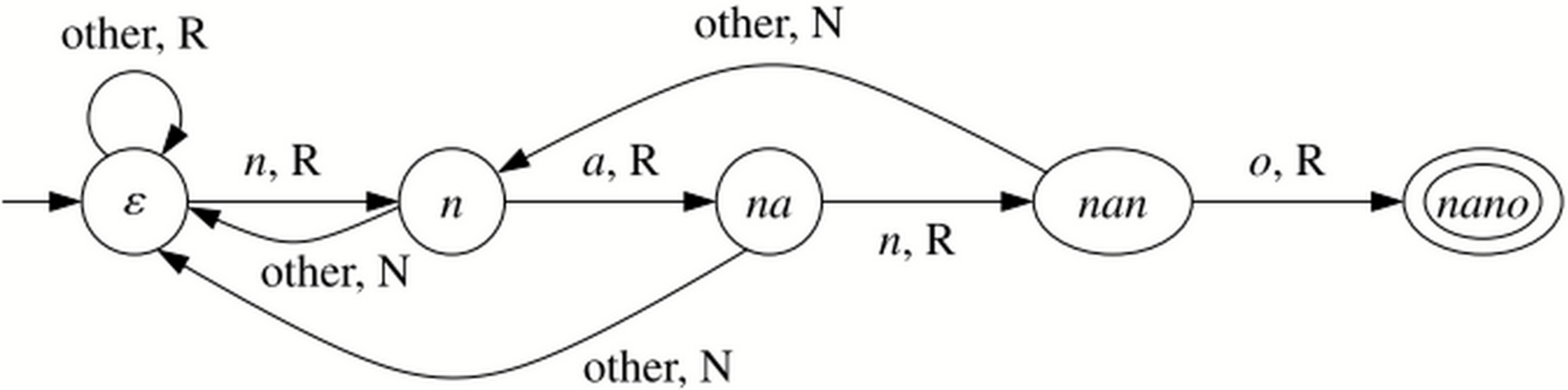
If the alphabet is large, m|Sigma| can also be large!

If the alphabet is not fixed: |Sigma| is O(n), and the eager DFA has size O(nm).

We introduce a more compact data structure: the lazy DFA

The lazy DFA





if the current state is not ϵ, then the head *does not move*, and the eager DFA moves to a new state *which depends only on the current state, not on the current letter*.

If the lazy DFA is at state $u \neq \epsilon$, and it reads a miss, what should be the new state?

The state is chosen to guarantee that the lazy DFA "simulates" the eager DFA: a step $u \xrightarrow{\alpha} v$ of the eager DFA is simulated by a sequence of moves

$$u \xrightarrow{(\alpha,N)} u_1 \xrightarrow{(\alpha,N)} v_2 \cdots u_k \xrightarrow{\alpha,R} v$$

of the lazy DFA. For instance, in our example the move $nan \xrightarrow{n} n$ of the eager DFA is simulated in the lazy DFA by the sequence

$$nan \xrightarrow{(n,N)} n \xrightarrow{(n,N)} \epsilon \xrightarrow{(n,R)} n \; .$$

Formal definition of the lazy DFA

Definition 7.4 *Let w be a proper prefix of p.*

- *We denote by h_w the unique letter such that $w\,h_w$ is a prefix of p. We call h_w a* hit *(from state w). Notice that $h_\varepsilon = a_1$.*
- *For $w \neq \varepsilon$ we define $pol(w)$ (short for proper overlap) as the longest* proper *suffix of w that is a prefix of p, that is, $pol(w)$ is the unique longest word of the set*

$$\{u \in \Sigma^* \mid \text{there exists } v \in \Sigma^+, v' \in \Sigma^* \text{ such that } w = vu \text{ and } p = uv'\}$$

Notice: pol(w) is a proper suffix of w
ol(w) is a suffix of w

For p=nana, ol(nana)=nana but pol(nana)=na

For nano:

pol(eps) = eps
pol(n) = eps
pol(na) = eps
pol(nan) = n
pol(nano) = eps

For abracadabra:

pol(abra) = a
pol(abracadabra) = abra

Definition 7.5 *The lazy DFA for p is the tuple* **lazyDFA**$(p) = (Q_l, \Sigma, \delta_l, q_{0l}, F_l)$*, where:*

- Q_l *is the set of prefixes of* p*;*
- *for every* $u \in Q_l, \alpha \in \Sigma$*:*

$$\delta_l(u, \alpha) = \begin{cases} (u\alpha, R) & \text{if } \alpha = h_u & \text{(hit)} \\ (\varepsilon, R) & \text{if } \alpha \neq h_u \text{ and } u = \varepsilon & \text{(miss from } \varepsilon\text{)} \\ (pol(u), N) & \text{if } \alpha \neq h_u \text{ and } u \neq \varepsilon & \text{(miss from other states)} \end{cases}$$

- $q_{0l} = \varepsilon$*; and*
- $F_l = \{p\}$

Definition 7.6 *Let lazyDFA(p)* $= (Q_l, \Sigma, \delta_l, q_{0l}, F_l)$*, let* $u \in Q_l$*, and let* $\alpha \in \Sigma$*. We denote by* $\widehat{\delta_l}(u, \alpha)$ *the unique state* v *such that*

$$u = u_0 \xrightarrow{(\alpha,N)} u_1 \xrightarrow{(\alpha,N)} u_2 \cdots u_k \xrightarrow{(\alpha,R)} v$$

for some $u_1, \ldots, u_k \in Q_l$*,* $k \geq 0$*.*

Proposition 7.8 $\widehat{\delta_l}(v, \alpha) = \delta_e(v, \alpha)$ *for every prefix v of p and every* $\alpha \in \Sigma$.

Proof:

α is a hit ($\alpha = h_v$). Then $\delta_e(v, \alpha) = v\alpha = \widehat{\delta}(v, \alpha)$.
α is a miss ($\alpha \neq h_v$). By induction on $|v|$.
$|v| = 0$. Then $v = \varepsilon$ and we have $\delta_e(v, \alpha) = \delta_l(v, \alpha) = \widehat{\delta_l}(v, \alpha)$.
$|v| > 0$. We have :

$$\begin{aligned} & \widehat{\delta_l}(v, \alpha) \\ = \; & \widehat{\delta_l}(pol(v), \alpha) \quad (\text{ because } \delta_l(v, \alpha) = (pol(v), N)) \\ = \; & \delta_e(pol(v), \alpha) \quad (\, |pol(v)| < |v| \text{ and ind. hyp.}) \end{aligned}$$

We show $\delta_e(pol(v), \alpha) = \delta_e(v, \alpha)$.

We show $\delta_e(pol(v), \alpha) = \delta_e(v, \alpha)$.

We have $\delta_e(pol(v), \alpha) = ol(pol(v)\alpha)$ and $\delta_e(v, \alpha) = ol(v\alpha)$.

We prove: if $\alpha \neq h_v$, then $ol(pol(v)\alpha) = ol(v\alpha)$.

1) $ol(pol(v)\,\alpha)$ is a suffix of $ol(v\alpha)$.

$pol(v)$ is suffix of $v \Rightarrow$ every suffix of $pol(v)\alpha$ is suffix of $v\alpha$.

2) $ol(v\alpha)$ is a suffix of $ol(pol(v)\,\alpha)$

Since α is a miss, $ol(v\alpha) = pol(v\alpha)$, so we show $pol(v\alpha)$ is a suffix of $ol(pol(v)\,\alpha)$.

It suffices: every suffix of $v\alpha$ that is prefix of p is also suffix of $pol(v)\,\alpha)$.

Nothing to show for the empty suffix.

$$
\begin{array}{ll}
 & w\alpha \text{ is prefix of } p \text{ and suffix of } v\alpha \\
\Rightarrow & w \text{ is prefix of } p \text{ and suffix of } v \\
\Rightarrow & w \text{ is suffix of } pol(v) \\
\Rightarrow & w\alpha \text{ is suffix of } pol(v)\alpha
\end{array}
$$

Constructing the lazy DFA in O(m) time

Reduces to computing pol(v) for every prefix v of p in O(m) time

Recall: pol(w) is the longest proper suffix of w that is a prefix of p. The following equation holds for every proper prefix v of p

$$pol(v\,h_v) = \begin{cases} \varepsilon & \text{if } v = \varepsilon \\ pol(v)\,h_v & \text{if } v \neq \varepsilon \text{ and } h_{pol(v)} = h_v \\ pol(pol(v)\,h_v) & \text{if } v \neq \varepsilon \text{ and } h_{pol(v)} \neq h_v \end{cases}$$

$$pol(v\,h_v) = \begin{cases} \varepsilon & \text{if } v = \varepsilon \\ pol(v)\,h_v & \text{if } v \neq \varepsilon \text{ and } h_{pol(v)} = h_v \\ pol(pol(v)\,h_v) & \text{if } v \neq \varepsilon \text{ and } h_{pol(v)} \neq h_v \end{cases}$$

For v = na we have h_v= n and h_pol(na)=n

pol(nan) = pol(na) h_v = eps n = n

For v = nan we have h_v=o h_pol(nan)=a

pol(nano) = pol(no) = eps

POL(v, α)
Input: a prefix v of p, a letter $\alpha \in \Sigma$.
Output: $pol(v\alpha)$.

if $|v| = 0$ **then return** ε
else if $v = w\beta$ **then**
    $u \leftarrow$ POL(w, β)
    if $\alpha = h_u$ **then return** $u\alpha$
    else return POL(u, α)

POLnum(v, k)
Input: numbers $0 \leq v, k \leq m$.
Output: the length of $pol(p[1] \dots p[v]p[k])$.

if $v = 0$ **then return** 0
else
    $u \leftarrow$ POLnum($v - 1, v$)
    if $p[k] = p[u + 1]$ **then return** $u + 1$
    else return POLnum(u, k)

POLiterative(m)
Input: a number $1 \leq m$.
Output: the array `pol`[1..m] with
`pol`[i] = length of $pol(p[1] \dots p[i])$ for every $1 \leq i \leq m$.

for all $v = 1$ to m **do**
    `pol`[v] $\leftarrow$ POLnum($v - 1, v$)