Pattern matching

Given a word w (the text) and a regular expression r (the pattern), determine the smallest k' such that some $[k,k']$-factor of w belongs to $L(r)$.

Pattern-Matching-NFA(w, r)
Input: word $w = a_1 \dots a_n \in \Sigma^+$, regular expression r
Output: the first occurrence k of r in w, or $\perp$ if no such occurrence exists.

$A \leftarrow \mathit{RegtoNFA}(\Sigma^* r)$
$S \leftarrow \{q_0\}$
for all $i = 0$ to $n - 1$ **do**
  if $S \cap F \neq \emptyset$ **then return** i
  $S \leftarrow \delta(S, a_i)$
return $\perp$

- *RegtoNFA*(r) takes $\mathcal{O}(m)$ time. Let k be the number of states of A.
- The loop is executed at most n times; each line of the body takes at most $\mathcal{O}(k^2)$ time.

Since *RegtoNFA*(r) takes $\mathcal{O}(m)$ time, we have $k \in \mathcal{O}(m)$, and so the loop runs in $\mathcal{O}(nm^2)$ time. The overall runtime is thus $\mathcal{O}(m + nm^2) = \mathcal{O}(nm^2)$.

Pattern-Matching-DFA(w, r)
Input: word $w = a_1 \ldots a_n \in \Sigma^+$, regular expression r
Output: the first occurrence i of r in w, or $\perp$ if no such occurrence exists.

$A \leftarrow \mathit{NFAtoDFA}(\mathit{RegtoNFA}(r))$
$q \leftarrow q_0$
for all $i = 0$ to $n - 1$ **do**
  if $q \in F$ **then return** i
  $q \leftarrow \delta(q, a_i)$
return $\perp$

- **RegtoNFA**(r) takes $\mathcal{O}(m)$ time, and so the call to *NFAtoDFA* (see Table 2.3.1) takes $2^{\mathcal{O}(m)}$ time and space.

- The loop is executed at most n times; each line of the body takes constant time.

The overall runtime is thus $\mathcal{O}(n) + 2^{\mathcal{O}(m)}$.

- 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

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

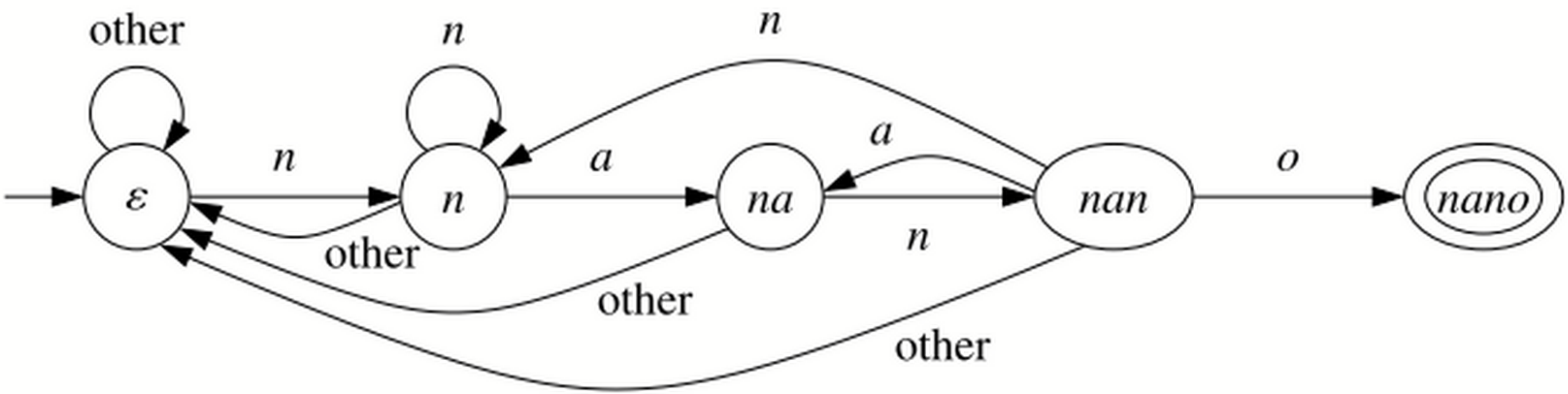
  - We construct a DFA with exactly one state for each prefix, which is therefore the minimal DFA

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 8.2 *Let $w \in \Sigma^*$ be a word and let $p \in \Sigma^*$ be a pattern. We denote by overl(w) the longest suffix of w that is a prefix of p. In other words, overl(w) is the unique longest word of the set*

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

Definition 8.3 *Let $p \in \Sigma^*$ be a pattern. The DFA* **eagerDFA**$(p) = (Q_e, \Sigma, \delta_e, q_{0e}, F_e)$ *is defined as follows:*

- $Q_e = \{u \in \Sigma^* \mid \exists v \in \Sigma^* . p = uv\}$ *is the set of prefixes of* p*;*
- *for every* $u \in Q_e$*, for every* $\alpha \in \Sigma$*:* $\delta_e(u, \alpha) = overl(u\alpha)$*;*
- $q_{0e} = \varepsilon$*; and*
- $F_e = \{p\}$

Using this definition, we define *Pattern-Matching-DFA*(w, p) for the pattern matching problem with a pattern p by replacing line 1 in *Pattern-Matching-DFA*(r, p) by

$$A \leftarrow eagerDFA(p)$$

The algorithm stops in state p if and only if the pattern p has been read. For a pattern of length m, *eagerDFA*(p) has $m+1$ states and $m|\Sigma|$ transitions. So, for a fixed alphabet Σ we get a DFA with $\mathcal{O}(m)$ states and transitions.

Variable alphabet size

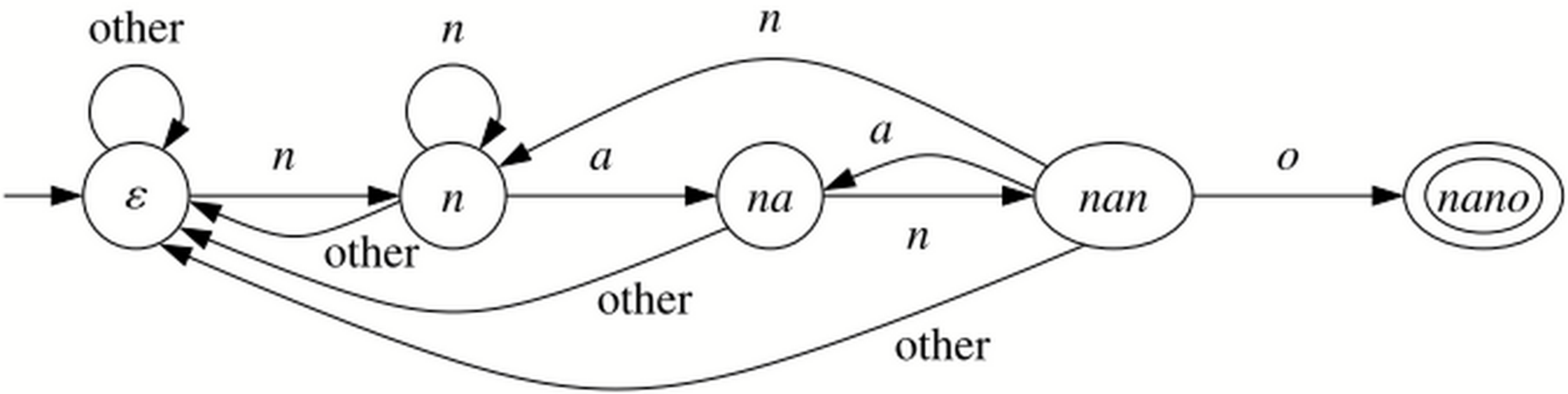
The eager DFA of a pattern of length m has
- m+1 states and
- m |Sigma| transitions

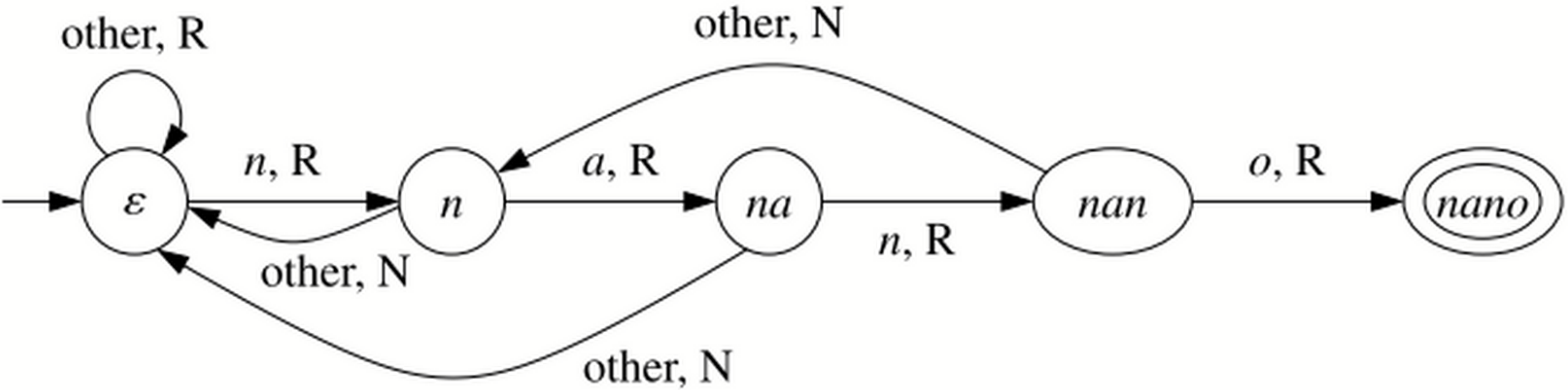
If the alphabet is large, m|Sigma| can also be large!

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 8.4 *Let $p \in \Sigma^*$ be a pattern, and let w be a proper prefix of p.*

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

$$overl_{nano}(nano) = nano, \text{ while } overl_{nano}(nano) = \varepsilon.$$

For nano:

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

For abracadabra:

overlap(abra) = a
overlap(abracadabra) = abra

Definition 8.5 *Let $p \in \Sigma^*$ be a pattern. The lazy DFA* **lazyDFA**$(p) = (Q_l, \Sigma, \delta_l, q_{0l}, F_l)$ *is defined as follows:*

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

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

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

Definition 8.6 *Let lazyDFA(p) = $(Q_l, \Sigma, \delta_l, q_{0l}, F_l)$ be the lazy DFA for a pattern p, and let $u \in Q_l$, $\alpha \in \Sigma$. 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 8.7 *Let $p \in \Sigma^*$ be a pattern, and let lazyDFA(p) $= (Q_l, \Sigma, \delta_l, q_{0l}, F_l)$ and eagerDFA(p) $= (Q_e, \Sigma, \delta_e, q_{0e}, F_e)$. Then $\widehat{\delta_l}(v, \alpha) = \delta_e(v, \alpha)$ for every prefix v of p and every $\alpha \in \Sigma$.*

Proof: If α is a hit, i.e., if $\alpha = h_v$, then we have $\delta_e(v, \alpha) = v\alpha = \widehat{\delta}(v, \alpha)$. If α is a miss, we proceed by induction on $|v|$. If $|v| = 0$, then $v = \varepsilon$ and by the definitions of δ_e and $trans_l$ we have $\delta_e(v, \alpha) = \delta_l(v, \alpha) = \widehat{\delta_l}(v, \alpha)$. If $|v| > 0$, then by the definition of δ_l we have $\delta_l(v, \alpha) = (overlap(v), N)$, and so:

$$
\begin{aligned}
& \widehat{\delta_l}(v, \alpha) \\
= \quad & \{ \ \delta_l(v, \alpha) = (overlap(v), N) \quad \text{and definition of } \widehat{\delta_l} \ \} \\
& \widehat{\delta_l}(overlap(v), \alpha) \\
= \quad & \{ \ |overlap(v)| < |v| \quad \text{and induction hypothesis} \ \} \\
& \delta_e(overlap(v), \alpha)
\end{aligned}
$$

To complete the proof we show $\delta_e(overlap(v), \alpha) = \delta_e(v, \alpha)$. By the definition of δ_e we have $\delta_e(overlap(v), \alpha) = overl(overlap(v)\alpha)$ and $\delta_e(v, \alpha) = overl(v\alpha)$. So we have to prove $overl(overlap(v)\alpha) = overl(v\alpha)$. For convenience we rename $overlap(v)$ as u and show $overl(u, \alpha) = overl(v\alpha)$. Recall that α is a miss.

Constructing the lazy DFA in O(m) time

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

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

Let u = overlap(v):

$$overlap(vh_v) = \begin{cases} uh_v & \text{if } h_u = h_v \\ overlap(uh_v) & \text{if } h_u \neq h_v \end{cases}$$

$$overlap(vh_v) = \begin{cases} uh_v & \text{if } h_u = h_v \\ overlap(uh_v) & \text{if } h_u \neq h_v \end{cases}$$

v := na   h_v:= n   u := overlap(na) = eps   h_u:=n

overlap(nan) = u h_v = eps n = n

v := nan   h_v:=o   u:= overlap(nan) = n   h_u:=a

overlap(nano) = overlap(no) = eps

Overlap(w)
Input: a prefix w of p.
Output: *overlap*(w)

```
if |w| ≤ 1 then return ε
if w = vα and v ≠ ε then
    u ← overlap(v)
    if α = h_u then return uα
    return overlap(uα)
```

Overlap(k)
Input: a number $0 \leq k \leq m$.
Output: the length of *overlap*($p[0] \ldots p[k-1]$).

```
if k ≤ 1 then return 0
if k ≥ 2 then
    u ← overlap(k − 1)
    if p[k] = p[u] then return u + 1
    return overlap(u + 1)
```

OverlapIt(m)
Input: a number $m \geq 0$.
Output: the array *overlap*$[0..m-1]$ with
overlap$[i]$ = *overlap*($p[0] \ldots p[i]$) for every $0 \leq i \leq m-1$.

```
overlap[0] ← 0
overlap[1] ← 0
for all j = 2 to m − 1 do
    u ← overlap[j − 1]
    if p[j] = p[u] then overlap[j] = u + 1
    else overlap[j] ← overlap[u + 1]
```