Karp ja Rabin tarkastelevat myös universaaliin hajautukseen perustuvia versioita algoritmista:

- Valitaan q *satunnaisesti* lukujoukosta Q.
- Q:n luvut voivat olla pieniäkin, eikä niiden tarvitse olla alkulukuja.
- Väärän täsmäyksen todennäköisyys on erittäin pieni syötteestä riippumatta (universaalius).
- q voidaan myös vaihtaa aina väärän täsmäyksen sattuessa.

1.5 Aho-Corasick-algoritmi

A. Aho & M. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM* 18(6): 333–340, 1975.

- KMP-algoritmin yleistys usealle hahmolle.
- On annettu teksti S, $|S| = n$, ja joukko hahmoja $\mathbb{P} = \{P_1, \ldots, P_k\}$. Merkitään $|\mathbb{P}| = m = \sum_{i=1}^{k} |P_i|$.
- On etsittävä S:stä kaikkien $P_i \in \mathbb{P}$ kaikki esiintymät.

Ratkaisuja

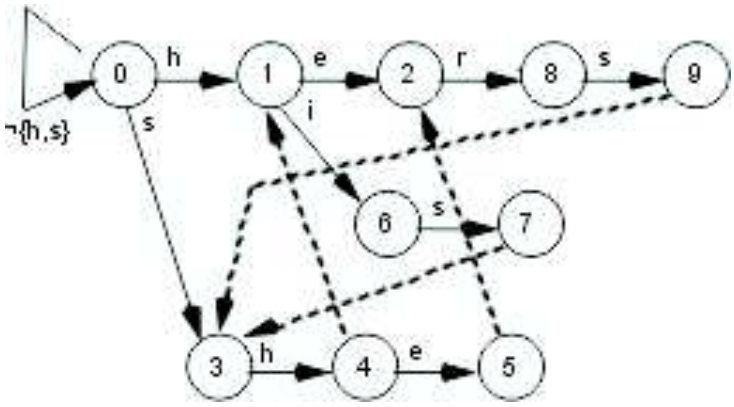
1. Sovelletaan esim. KMP-algoritmia vuorotellen kullakin hahmolla P_i. Aikavaatimus pahimmassa tapauksessa $\mathcal{O}(m + kn)$.
2. Aho-Corasick-algoritmi. Teksti S selataan vain kerran. Aikavaatimus on pahimmassa tapauksessa $\mathcal{O}(m + n)$ (tietyin edellytyksin).
3. Myös BM-algoritmille ja sen variaatioille on vastaavia yleistyksiä.

Aho-Corasick

Muodostetaan $\mathbb{P}$:lle jononsovitus-automaatti $M_{\mathbb{P}}$. $M_{\mathbb{P}}$:n osat:

- Varsinaiset siirtymät: $goto[j, a]$ — tilasta j siirrytään merkillä a tilaan $goto[j, a]$. Jos tilasta j ei ole siirtymää a:lla, $goto[j, a] = \perp$.
- Korjaussiirtymät: $fail[j]$ — jos $goto[j, a] = \perp$ vuorossa olevalla tekstimerkillä a (ts. mikään varsinainen siirtymä ei sovi), siirrytään tilaan $fail[j]$.
- Tulostusfunktio: $output[j]$ — sisältää hahmot, jotka tulostetaan, kun tullaan tilaan j (ts. näiden hahmojen esiintymät on nyt löydetty).

Esimerkki 1.5.1 *Aho-Corasick jononsovitus-automaatti $M_{\mathbb{P}}$ hahmojoukolle* $\mathbb{P} = \{$`he, she, his, hers`$\}$.



Varsinaiset siirtymät muodostavat $\mathbb{P}$:stä ns. trie-rakenteen. Varsinainen siirtymä (nuoli yhtenäisellä viivalla) lukee tekstistä yhden merkin, esim. $goto[1, i] = 6$. Korjaussiirtymää (nuoli katkoviivalla) seurataan, jos varsinainen ei sovi, esim. $fail[9] = 3$; jokaisesta tilasta ($\neq 0$) lähtee korjaussiirtymä (alkutilaan 0 johtavat korjaussiirtymät on jätetty merkitsemättä).

Tulostusfunktio:

j	$output[j]$
2	{`he`}
5	{`she, he`}
7	{`his`}
9	{`hers`}

Seuraava algoritmi esittää $M_{\mathbb{P}}$:n toiminnan, kun se selaa S:n:

Algoritmi 1.5.2 *Aho-Corasick-selaus*

Syöte: S ja $M_{\mathbb{P}}$ (=funktiot *goto*, $fail$ ja *output* hahmojoukolle $\mathbb{P}$)
Tuloste: Kaikki $\mathbb{P}$:n hahmojen esiintymät (= esiintymien loppukohdat) S:ssä
(1) $state := 0$
(2) **for** $j := 0$ **to** $n - 1$ **do**
(3) **while** $goto[state, S[j]] = \perp$ **do** $state := fail[state]$
(4) $state := goto[state, S[j]]$
(5) **if** $output[state] \neq \emptyset$ **then** $print(j, output[state])$

Goto-funktion konstruktio

Lisätään hahmot P_i yksitellen trie-rakenteeseen, joka esittää *goto*-funktiota. Kun P_i on lisätty, päivitetään samalla *output*-funktiota. ($fail$-funktion konstruktio täydentää *output*-funktion).

Algoritmi 1.5.3 *Aho-Corasick-esiprosessointi I*

Syöte: $\mathbb{P} = \{P_1, \ldots, P_k\}$
Tuloste: *goto*-funktio ja osittain laskettu *output*-funktio
Alustus: $output[s] = \emptyset$ kaikilla s;
$goto[s, a] = \perp$ kaikilla s ja a.
(1) $newstate := 0$
(2) **for** $i := 1$ **to** k **do**
(3) $state := 0; j := 0$
(4) **while** $goto[state, P_i[j]] \neq \perp$ **do**
(5) $state := goto[state, P_i[j]]$
(6) $j := j + 1$
(7) **for** $h := j$ **to** $|P_i| - 1$ **do**
(8) $newstate := newstate + 1$
(9) $goto[state, P_i[h]] := newstate$
(10) $state := newstate$
(11) $output[state] := \{P_i\}$
(12) **for** $a \in \Sigma$ **do if** $goto[0, a] = \perp$ **then** $goto[0, a] := 0$

Fail-funktion konstruktio

Laskenta etenee tasoittain alkaen *goto*-trien juuresta. Tasolla d tilojen $fail$-funktion laskemiseksi käydään tason $d-1$ tilat r läpi seuraavasti:

1. Jokaisella a siten, että $goto[r,a] = s$, $s \neq \perp$:
   a) $state := fail[r]$
   b) Toista $state := fail[state]$ kunnes saavutetaan $state$, jolla $goto[state,a] \neq \perp$. (Sellainen löytyy aina, koska $goto[0,a] \neq \perp$.)
   c) Aseta $fail[s] := goto[state,a]$

Algoritmi 1.5.4 *Aho-Corasick-esiprosessointi II*

Syöte: *goto*-funktio ja alustava *output*-funktio Algoritmista 1.5.3
Tuloste: $fail$-funktio ja lopullinen *output*-funktio

```
(1)  queue := ∅
(2)  for a ∈ Σ do if goto[0,a] ≠ 0 then
(3)      s := goto[0,a]
(4)      push_back(queue, s)
(5)      fail[s] := 0
(6)  while queue ≠ ∅ do
(7)      r := pop_front(queue)
(8)      for a ∈ Σ do if goto[r,a] ≠ ⊥ then
(9)          s := goto[r,a]
(10)         push_back(queue, s)
(11)         state := fail[r]
(12)         while goto[state,a] = ⊥ do state := fail[state]
(13)         fail[s] := goto[state,a]
(14)         output[s] := output[s] ∪ output[fail[s]]
```

AC-algoritmin oikeellisuus

Jos tilasta 0 tilaan s johtava *goto*-reitti selaa jonon u, kutsutaan u:ta s-jonoksi. Kaikkien s-jonojen, $s \in M_{\mathbb{P}}$, joukkoa kutsutaan $M_{\mathbb{P}}$-jonoiksi. Jono u on $M_{\mathbb{P}}$-jono, joss se on jonkin hahmon $P_i \in \mathbb{P}$ alkuosa.

Lemma 1.5.5 *Oletetaan, että jono u on s-jono ja v on t-jono automaatissa $M_{\mathbb{P}}$. Silloin $fail[s] = t$, jos ja vain jos v on u:n pisin aito loppuosa, joka on $M_{\mathbb{P}}$-jono.*

Todistus. Käytetään induktiota $|u|$:n (s:n syvyyden) suhteen.

- Kun $|u| = 1$, Alg. 1.5.4 asettaa $fail[s] := 0$. Koska 0-jono on tyhjä jono, joka on u:n ainoa aito loppuosa, väite pätee.
- Oletetaan, että $|u| = j + 1$, $j > 0$, ja että väite on tosi kaikille tiloille, joiden syvyys on enintään j. Olkoon $u[0 \ldots j)$ r_0-jono, eli $goto[r_0, u[j]] = s$. Olkoot $r_1, \ldots, r_p$ tiloja siten, että
  1. $r_{i+1} = fail[r_i]$, $0 \leq i < p$,
  2. $goto[r_i, u[j]] = \perp$, $1 \leq i < p$, ja
  3. $goto[r_p, u[j]] = t \neq \perp$.
- Siis $r_1, \ldots, r_p$ ovat ne tilat "state", joiden kautta Alg. 1.5.4:n sisin silmukka kulkee. Sen jälkeen algoritmi asettaa: $fail[s] := t$.
- Olkoon v_i r_i-jono, $0 \leq i \leq p$. Induktio-oletuksesta seuraa, että v_1 on $v_0 = u[0 \ldots j)$:n pisin aito loppuosa, joka on $M_{\mathbb{P}}$-jono. Vastaavasti v_2 on v_1:n pisin aito loppuosa, joka on $M_{\mathbb{P}}$-jono, jne.
- v_0:n jokainen aito loppuosa on myös v_1:n loppuosa. Siten v_2:n täytyy olla v_0:n toiseksi pisin aito loppuosa, joka on $M_{\mathbb{P}}$-jono. Vastaavasti v_3 on v_0:n kolmanneksi pisin aito loppuosa, joka on $M_{\mathbb{P}}$-jono, jne.
- Siten v_p on v_0:n pisin aito loppuosa, jolla $v_p u[j] = v$ on $M_{\mathbb{P}}$-jono.
- Siis v on u:n pisin aito loppuosa, joka on $M_{\mathbb{P}}$-jono.

□

Lemma 1.5.6 *Tulosjoukko $output[s]$ sisältää jonon y, jos ja vain jos y on jokin $\mathbb{P}$:n hahmo, joka on s-jonon loppuosa.*

Todistus. Kuten Lemma 1.5.5. □

Lemma 1.5.7 *Kun $M_{\mathbb{P}}$ on selannut (Alg. 1.5.2) $j + 1$ merkkiä S:n alusta, se on tilassa s, jos ja vain jos s-jono on $S[0 \ldots j]$:n pisin loppuosa joka on $M_{\mathbb{P}}$-jono.*

Todistus. Kuten Lemma 1.5.5. □

Lause 1.5.8 *Algoritmi 1.5.2 tulostaa täsmälleen kaikki $\mathbb{P}$:n esiintymät S:stä.*

Todistus. Seuraavat väitteet ovat yhtäpitäviä:

- j on P_i:n esiintymän päättymiskohta S:ssä.
- P_i on $S[0 \ldots j]$:n loppuosa.
- s on tila, jossa algoritmi on $S[0 \ldots j]$:n selauksen jälkeen, ja P_i on s-jonon loppuosa. (Lemma 1.5.7).
- s on tila, jossa algoritmi on $S[0 \ldots j]$:n selauksen jälkeen, ja $output[s]$ sisältää P_i:n (Lemma 1.5.6).
- Algoritmi tulostaa, että j on P_i:n esiintymän loppukohta S:ssä.

□

AC-algoritmin aikavaatimus

Lause 1.5.9 *Automaatti $M_{\mathbb{P}}$ (Alg. 1.5.2) tekee vähemmän kuin $2n$ tilasiirtymää selatessaan tekstin S, $|S| = n$.*

Todistus. *Fail*-siirtymiä voi olla korkeintaan *goto*-siirtymien määrä -1, koska jokainen $fail$-siirtymä johtaa aidosti lähemmäksi tilaa 0 (eli syvyys vähenee). Koska *goto*-siirtymiä on n kpl, $fail$-siirtymiä voi siis olla korkeintaan $n - 1$ kpl, joten siirtymiä on yhteensä vähemmän kuin $2n$ kpl. □

Lauseessa 1.5.9 Algoritmille 1.5.2 saatu vertailujen määrän $\mathcal{O}(n)$ raja ei välttämättä todista, että todellinen aikavaatimus olisi myös $\mathcal{O}(n)$. Aikavaatimukseen vaikuttavat myös seuraavat operaatiot:

1. $goto[q, a]$:n evaluointi	$\mathcal{O}(1)$ tai $\mathcal{O}(\log \min(k, \sigma))$
2. $fail[q]$:n evaluointi	$\mathcal{O}(1)$
3. testaus: $output[q] = \emptyset$	$\mathcal{O}(1)$
4. $output[q]$:n tulostaminen	?

Jos kaikki funktiot esitetään taulukkoina, Alg. 1.5.2:n aikavaatimus on $\mathcal{O}(n + r)$, missä r on esiintymien kokonaismäärä. Jos Σ:n alkiot eivät ole kokonaislukuja (tai σ on suuri), on taulukoiden sijasta käytettävä esim. listoja tai hakupuita, jotka antavat aikavaatimuksen $\mathcal{O}(n \log(\min(k, \sigma)) + r)$.

Keskitytään jatkossa taulukkototeutukseen.

Lause 1.5.10 *Lause Alg. 1.5.3 laskee goto-funktion ja alustaa output-funktion ajassa $\mathcal{O}(\sigma m)$.*

Todistus. Helppo. □

Lause 1.5.11 *Lause Alg. 1.5.4 laskee fail-funktion ja viimeistelee output-funktion ajassa $\mathcal{O}(\sigma m)$.*

Todistus. Väite pätee helposti muille paitsi riveille (12) ja (14)

- Rivin (14) *output*-funktioiden yhdiste voidaan toteuttaa listojen katenaationa ajassa $\mathcal{O}(1)$.
- Tarkastellaan juuresta alkavaa polkua $s_0 = 0, s_1, \ldots s_p$.
- Kun lasketaan s_i:n $fail$-siirtymää, ensimmäinen askel rivin (12) silmukassa on $fail[s_{i-1}]$. Tällöin siis ohitetaan tasot, jotka käytiin läpi s_{i-1}:n $fail$-siirtymää laskettaessa.
- Siten rivin (12) silmukka suoritetaan $\mathcal{O}(p)$ kertaa tilojen $s_1, \ldots s_p$ $fail$-siirtymiä laskettaessa.
- Yhteensä rivin (12) silmukka suoritetaan $\mathcal{O}(m)$ kertaa.

□

Seuraus 1.5.12 *AC-algoritmin kokonaisaikavaatimus on $\mathcal{O}(n + \sigma m + r)$.*

Jos aakkoston koko on vakio ja tulosteen koko oletetaan pieneksi, aikavaatimus on $\mathcal{O}(n + m)$.

Koetulos: $n = 10^7$, etsintäaika tunteina

	$k = 15$	$k = 24$
triv.ratk.	0.79	1.27
AC	0.18	0.21

1.6 Hahmonsovitus äärellisen automaatin avulla

K. Thompson: Regular expression search algorithm.
Communications of the ACM 11:419–422, 1968.

Edellä on tarkasteltu tapauksia: yksi hahmo ja joukko hahmoja. Seuraavassa tarkastellaan yleisempää tapausta, jossa etsittävien hahmojen joukko on ilmaistu säännöllisenä lausekkeena. Aluksi hieman kertausta:

Määritelmä 1.6.1 *Säännölliset lausekkeet ja kunkin lausekkeen A esittämä merkkijonojoukko $L(A)$:*

1. *$\emptyset$ on säännöllinen lauseke; $L(\emptyset) = \emptyset$*
2. *ϵ on säännöllinen lauseke; $L(\epsilon) = \{\epsilon\}$, ϵ = tyhjä jono.*
3. *Kaikilla $a \in \Sigma$, a on säännöllinen lauseke; $L(a) = \{a\}$.*
4. *Jos A ja B ovat säännöllisiä lausekkeita, seuraavat lausekkeet ovat säännöllisiä lausekkeita:*

$$\begin{aligned} (A \cup B)\ \textit{yhdiste} &\ ;\ L(A \cup B) = L(A) \cup L(B) \\ (AB)\ \textit{katenaatio} &\ ;\ L(AB) = L(A)L(B) \\ (A^*)\ \textit{sulkeuma} &\ ;\ L(A^*) = L(A)^* \end{aligned}$$

Ongelma 1.6.2 *On annettu säännöllinen lauseke A (= hahmojoukon esitys) ja teksti S.*

O1. Onko $S \in L(A)$?

O2. Anna kaikki S:n alkuosat, jotka kuuluvat joukkoon $L(A)$.

O3. Etsi kaikki $L(A)$:n jonojen esiintymät S:stä.

- O1:n ratkaisu: Muodostetaan epädeterministinen äärellinen automaatti M_A siten, että sen hyväksymä kieli on $L(M_A) = L(A)$. Selataan S M_A:lla. Tutkitaan lopussa, onko M_A lopputilassa.
- O2:n ratkaisu: Kuten O1, mutta tutkitaan jokaisen merkin jälkeen, onko M_A lopputilassa.
- O3:n ratkaisu: Ratkaistaan O2 lausekkeella $\Sigma^* A$ (missä Σ on kaikkien aakkoston merkkien yhdiste $a \cup b \cup \ldots$).

Tarkastellaan jatkossa lähinnä ongelmaa O1.

Määritelmä 1.6.3 *Epädeterministinen äärellinen automaatti (NFA)* $M = (Q, \Sigma, \delta, q_0, F)$ *muodostuu osista:*

1. Q *on äärellinen tilojen joukko.*
2. Σ *on syöttöaakkosto.*
3. δ *on tilasiirtymäfunktio,* $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to Q$*:n osajoukot (*$\delta$ *siis sallii useita siirtymiä samalla aakkosella).*
4. $q_0 \in Q$ *on alkutila.*
5. $F \subseteq Q$*, lopputilojen joukko.*

Nyt voidaan myös määritellä:

- *Siirto*: $(q, aw) \vdash (q', w)$ jos $q' \in \delta(q, a)$, $a \in \Sigma \cup \{\epsilon\}$, $w \in \Sigma^*$.
- *Hyväksytty kieli*: $L(M) = \{w \mid (q_0, w) \vdash^* (q, \epsilon), q \in F\}$, missä "$\vdash^*$" tarkoittaa "0:lla tai useammalla siirrolla".

M_A:n konstruktio A:sta on esitetty esim. kurssilla Laskennan mallit ja kuvassa 1.1.

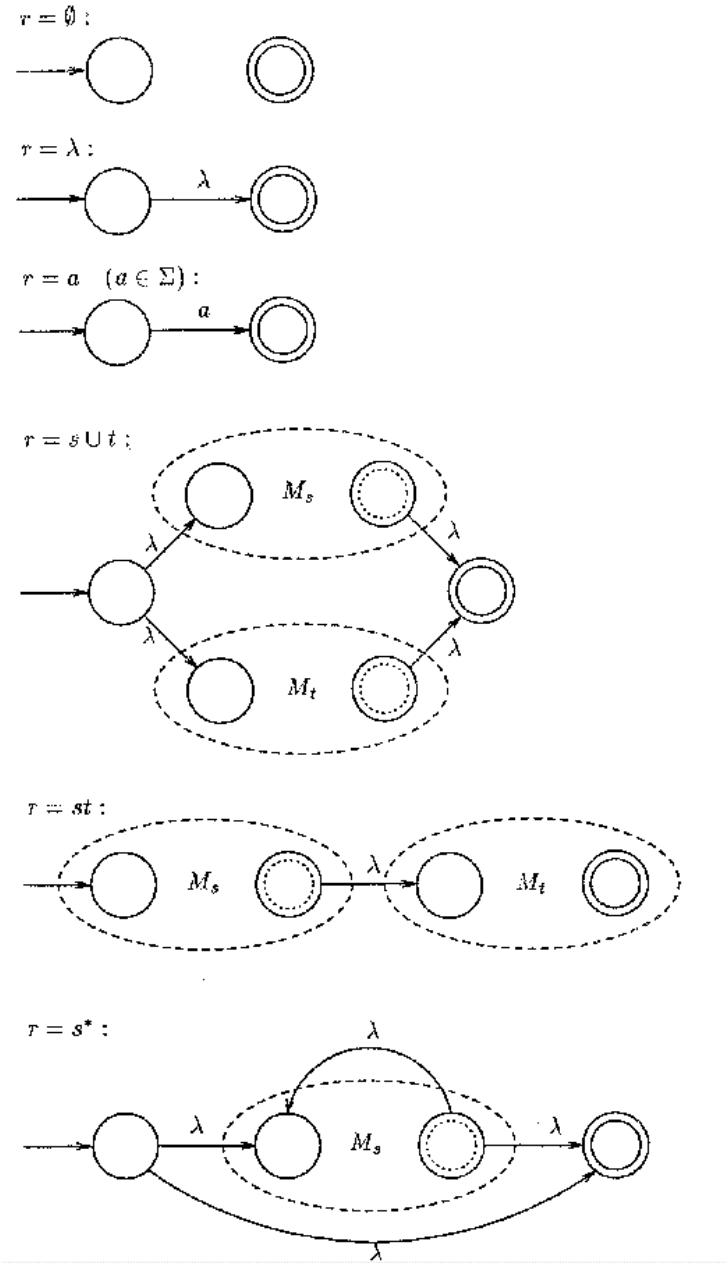
Lause 1.6.4 *Olkoon NFA* $M_A = (Q, \Sigma, \delta, q_0, \{q_f\})$ *konstruoitu* A*:sta em. tekniikalla. Silloin* $L(M_A) = L(A)$ *ja*

(1) $|Q| = 2|A|$ *(*$|A|$ *on* A*:n pituus merkkeinä).*

(2) $\delta(q_f, a) = \emptyset$ *kaikilla* $a \in \Sigma \cup \{\epsilon\}$*.*

(3) Kustakin tilasta $q \in Q$ *lähtee korkeintaan 2 siirtymää.*

Todistus. $L(M_A) = L(A)$: ks. esim. Laskennan mallit -kurssi. Väite (2) pätee selvästi konstruktion nojalla. Väitteet (1) ja (3) osoitetaan induktiolla $|A|$:n suhteen. □



Kuva 1.1: NFA:n konstruktio säännöllisestä lausekkeesta. Kuvan λ on sama kuin tekstin ϵ.

Tekstin S selaaminen automaatilla M_A

Tarkastellaan, miten NFA M_A:n vaihtoehtoisia polkuja voidaan seurata samanaikaisesti. Vaihtoehtoinen ratkaisutapa olisi muodostaa ensin M_A:sta vastaava deterministinen äärellinen automaatti M'_A. M'_A voi olla hyvin suuri: $|M'_A| = \mathcal{O}(2^t)$, missä t on M_A:n tilojen lukumäärä, joten M'_A :n muodostaminen ei aina kannata, varsinkaan jos S on lyhyt. M'_A:n etuna on, että S voidaan selata ajassa $\mathcal{O}(|S|)$.

Merkitään Q_j:llä niiden tilojen joukkoa, joissa M_A voi olla luettuaan jonon $S[0 \ldots j) : Q_j = \{q \mid (q_0, S[0 \ldots j)) \vdash^* (q, \epsilon)\}$. Silloin: $q \in Q_j$, jos ja vain jos on olemassa $q' \in Q_{j-1}$ siten, että tilasta q' siirrytään tilaan q_1 merkillä $S[j-1]$ ja tilasta q_1 siirrytään 0:lla tai useammalla ϵ-siirrolla tilaan q.

Algoritmi 1.6.5 *NFA:n simulointi*

Syöte: NFA $M_A = (Q, \Sigma, \delta, q_0, \{q_f\})$, Teksti $S = S[0 \ldots n)$.
Tuloste: Tilajoukkojono $Q_0, Q_1, \ldots, Q_n$.
Huom. $S \in L(M_A)$, jos ja vain jos $q_f \in Q_n$.
Jono *queue* ja joukot $Q_0, Q_1, \ldots, Q_n$ ovat aluksi tyhjiä.

```
(1)  for j := 0 to n do
(2)      merkitse kukin q ∈ Q saavuttamattomaksi
(3)      if j = 0 then Q_0 := {q_0}; queue := {q_0}; merkitse q_0 saavutetuksi
(4)      else
(5)          for q ∈ Q_{j-1} do for p ∈ δ(q, S[j-1]) do
(6)              if p on saavuttamaton then
(7)                  Q_j := Q_j ∪ {p}
(8)                  push_back(queue, p)
(9)                  merkitse p saavutetuksi
(10)     while queue ≠ ∅ do
(11)         q := pop_front(queue)
(12)         for p ∈ δ(q, ε) do
(13)             if p on saavuttamaton then
(14)                 Q_j := Q_j ∪ {p}
(15)                 push_back(queue, p)
(16)                 merkitse p saavutetuksi
```

Riveillä 10-16 seurataan ϵ-siirtymiä ja se osa voitaisiin jättää pois, jos automaatissa ei olisi ϵ-siirtymiä. Niiden poisto saattaisi kuitenkin kasvattaa siirtymien määräksi $\mathcal{O}(|Q|^2)$.
Merkitään: $||M_A||$ = tilojen lkm + siirtymien lkm.

Lause 1.6.6 *NFA M_A:n simulointi Algoritmilla 1.6.5, kun syöte on $S = S[0 \ldots n)$, vie ajan $\mathcal{O}(||M_A|| \cdot n)$.*

Todistus.

- Rivi 2 vie ajan $\mathcal{O}(|Q|)$.
- Siirtymäfunktion δ evaluointi voidaan toteuttaa ajassa $\mathcal{O}(1)$, koska kustakin tilasta on enintään kaksi siirtymää. Kaikki muutkin operaatiot toimivat vakioajassa.
- Riveillä 3–16 viedään vain saavuttamattomia tiloja jonoon ja merkitään ne samalla saavutetuiksi. Sitten seurataan saavutetuista tiloista alkavia siirtymiä, joten kutakin siirtymää seurataan korkeintaan kerran. Yhden siirtymän käsittely vie $\mathcal{O}(1)$ ajan, joten aikaa kuluu $\mathcal{O}$(siirtymien lkm).
- Rivit 2–16 toistetaan n kertaa. Siis kokonaisaika on $\mathcal{O}(||M_A|| \cdot n)$.

□

Seuraus 1.6.7 *Koska $||M_A|| \leq 6 \cdot |A|$ (Lause 1.6.4), voidaan*

- *testata onko $S \in L(A)$ ajassa $\mathcal{O}(|A| \cdot |S|)$ (ongelma O1),*
- *laskea S:n alkuosat, jotka ovat $L(A)$:ssa, ajassa $\mathcal{O}(|A| \cdot |S|)$ (ongelma O2), ja*
- *löytää S:stä $L(A)$:n jonojen esiintymät ajassa $\mathcal{O}((|\Sigma| + |A|) \cdot |S|)$ (ongelma O3).*

Todistus. O2: Ajassa $\mathcal{O}(|A|)$ voidaan tutkia, sisältääkö Q_j lopputilan.
O3: $|\Sigma^* A| = \mathcal{O}(|\Sigma| + |A|)$. □

1.7 Hahmonsovituksen erikoistapauksia

Edellä käsiteltiin tapausta, jossa hahmojen joukko esitettiin säännöllisenä lausekkeena. Luonnollinen kysymys on

- löytyykö sellaisia ei-triviaaleja säännöllisten lausekkeiden aliluokkia, jolla hahmontunnistus voitaisiin toteuttaa oleellisesti nopeammin kuin Algoritmilla 1.6.5?

Tarkastellaan lyhyesti joitakin tällaisia aliluokkia.

1.7.1 Merkkiluokat

Tarkastellaan tapausta, jossa hahmon "merkit" ovat *merkkiluokkia*, jotka voivat täsmätä useampaan eri merkkiin. Toisin sanoen on annettu hahmo $P = p_0 p_1 \cdots p_{m-1}$, jonka merkit ovat aakkoston osajoukkoja: $p_i \subseteq \Sigma$. Hahmo P esiintyy tekstin S kohdassa j, jos $s_{j+i} \in p_i$ kaikilla $i \in [0, m)$.

Esimerkki 1.7.1 *Hahmo $[Aa][Ss][Ss][Ii]$ täsmää sanaan assi, mutta sallii pienten ja isojen kirjainten kaikki yhdistelmät.*

Shift-or-algoritmi voidaan helposti muokata tähän tapaukseen.

- Ainoa muutos koskee vektoreita $T[a]$, $a \in \Sigma$, jotka kertovat mihin hahmon kohtiin kukin merkki täsmää.
- Itse haun suorittava pääsilmukka ei muutu ollenkaan.

Algoritmissa 1.3.1 rivi (2)

(2) **for** $i := 0$ **to** $m - 1$ **do** $T[p[i]] := T[p[i]] - 2^i$

korvataan seuraavilla riveillä:

(2) **for** $i := 0$ **to** $m - 1$ **do**
  for $a \in p[i]$ **do** $T[a] := T[a] - 2^i$

- Selauksen aikavaatimus säilyy samana: $O(\lceil m/w \rceil n)$.
- Alustuksen aikavaatimus on nyt $O(\lceil m/w \rceil \sigma + ||P||)$, missä $||P|| = \sum_{i \in [0,m)} |p_i|$ on hahmon merkkiluokkien yhteiskoko.

Shift-or-algoritmia voidaan yleistää vielä lisää:

- Sallitaan hahmossa erikoismerkit ? ja $*$.
- $p_i?$ tarkoittaa, että merkki tai merkkiluokka p_i on valinnainen.
- p_i* tarkoittaa, että p_i esiintyy 0 kertaa tai useammin.

  Esimerkki 1.7.2 *Hahmo $s?h[ei][rs]*$ täsmää mm. sanoihin he, she, his ja hers.*

- Tuetaan siis kaikkia äärellisten lausekkeiden konstruktioita, mutta sillä rajoituksella, että yhdistettä (merkkiluokka) ja sulkeumaa (*) voidaan soveltaa vain yksittäiseen merkkiin tai merkkiluokkaan, ei mielivaltaiseen säännölliseen lausekkeeseen.

Yleistys on merkkiluokkia monimutkaisempi ja ohitetaan tässä. Lisätietoa löytyy kirjasta:

G. Navarro & M. Raffinot: *Flexible Pattern Matching in Strings*. Cambridge University Press, 2002.

1.7.2 Jokerimerkit

M. Fischer & M. Paterson. String matching and other products. Teoksessa *R. Karp, editor, SIAM AMS Complexity of Computation*, sivut 113–125, 1974.

Merkkiluokkia rajoitetumpi tapaus on sallia hahmossa normaalien merkkien lisäksi jokerimerkkejä #, jotka täsmäävät mihin tahansa yksittäiseen merkkiin.

Esimerkki 1.7.3 *Hahmolle* $P =$ `#oke#i` *löytyy 3 esiintymää tekstissä* $S =$ `sokeripokeritokeni`.

Ratkaisuja

- Triviaalialgoritmi. Aikavaatimus $\mathcal{O}(nm)$.
- Shift-or-algoritmi. Aikavaatimus $\mathcal{O}(n\lceil m/w \rceil)$.
- Fischer-Paterson-algoritmi. Aikavaatimus $\mathcal{O}(n \log^2 m \log\log m)$.

Fischer-Paterson-algoritmi palauttaa ongelman kokonaislukujen kertolaskuun, johon on tunnetusti kehitetty useita algoritmeja eri laskennan malleissa.

Lause 1.7.4 *Olkoon jokerimerkkejä sisältävien hahmon ja tekstin pituudet* m *ja* n, $m \leq n$. *Tällöin hahmon esiintymät tekstisssä voidaan löytää ajassa* $IM(n \log m, m \log m)$, *missä* $IM(N, M)$ *on kahden* N *ja* M, $N \geq M$, *bittiä pitkien kokonaislukujen kertolaskuun tarvittava aika.*

Schönhage-Strassen kertolaskualgoritmi käyttää $O(N \log M \log\log M)$ bittioperaatiota, mistä seuraa $O(n \log^2 m \log\log m)$ algoritmi hahmonsovitukselle jokerimerkkien kanssa. Kertolaskussa voidaan käyttää myös tietokoneen omaa kertolasku-operaatiota, mutta tällöin aikavaatimukseksi tulee $O(mn)$, eli sama kuin triviaalialgoritmilla.

Algoritmin kuvaus ohitetaan tässä, mutta se löytyy esim. syksyn 2005 kurssin muistiinpanoista ja kirjasta

M. Crochemore & W. Rytter. *Jewels of Stringology*. World Scientific, 2003.

Luku 2

Likimääräinen hahmonsovitus

Useissa sovelluksissa ei ole mielekästä etsiä vain hahmon P tarkkoja esiintymiä, vaan pyritään löytämään kaikki tekstin osajonot, jotka ovat riittävän lähellä hahmoa P. Jonojen samankaltaisuuden eräs mitta on *editointietäisyys*.

2.1 Editointietäisyys

Olkoon $A = a_1 \cdots a_m \in \Sigma^*$ "lyöntivirheitä sisältävä jono" ja $B = b_1 \cdots b_n \in \Sigma^*$ "virheetön jono".

- Merkkijonojen A ja B välinen editointietäisyys on pienin editointioperaatioiden lukumäärä, joka tarvitaan korjaamaan virheellinen jono A virheettömään muotoon B.

- Editointioperaatiot

  P. *Poisto*: A:n merkkiä a_i ei vastaa mikään B:n merkki, $a_i \to \epsilon$.

  L. *Lisäys*: B:n merkkiä b_j ei vastaa mikään A:n merkki, $\epsilon \to b_j$.

  M. *Muutos*: A:n merkkiä a_i vastaa B:n merkki b_j, $a_i \neq b_j$, $a_i \to b_j$.

- Operaatiot M eivät saa mennä ristiin:

  – Jos $a_i \to b_j$ ja $a_{i'} \to b_{j'}$, niin $i < i'$, jos ja vain jos $j < j'$.

- Usein on myös hyödyllistä määritellä "operaatio":

  T. *Täsmäys*: A:n merkkiä a_i vastaa B:n merkki b_j, $a_i = b_j$, $a_i \to b_j$.

Määritelmä 2.1.1 *Merkkijonojen A ja B välinen editointietäisyys $D(A, B)$ on pienin editointioperaatioiden P, L ja M määrä, joka tarvitaan muuntamaan A B:ksi.*

- Editointietäisyydestä käytetään myös nimitystä *Levenshtein*-etäisyys.
- Jos sallitaan pelkkä operaatio M (muutos), etäisyydestä käytetään nimitystä *Hamming*-etäisyys.
- Voidaan osoittaa, että D on *metriikka*:
  1) $D(A, B) \geq 0$,
  2) $D(A, B) = 0$, jos ja vain jos $A = B$,
  3) $D(A, B) = D(B, A)$,
  4) $D(A, C) \leq D(A, B) + D(B, C)$.

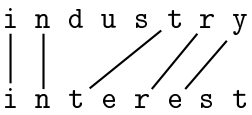
Esimerkki 2.1.2 $A =$ `abba`, $B =$ `bbb`

```
abba                    abba
|:||   3 operaatiota     |::|   2 operaatiota
bb b                    bbb
```

Tarvitaan vähintään yksi poisto, koska $|A| = |B| + 1$, ja vähintään yksi lisäys tai muutos, koska B:ssä on yksi b enemmän kuin A:ssa. Tämän ja ylläolevan perusteella $D(A, B) = 2$.

Editoinnin havainnollistaminen

1. Jäljitys (trace)



- Viivat eivät saa mennä ristiin. (Miksi?)

2. Kohdistus

```
indust-ry--
in---terest
```