FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Gevorderde algoritmen

Bert De Saffel

Master in de Industriële Wetenschappen: Informatica Academiejaar 2018–2019

Gecompileerd op 26 november 2019

UNIVERSITEIT
GENT

Inhoudsopgave

Deel I

Strings

Hoofdstuk 1

Zoeken in strings

- De gebruikte symbolen:

Symbool	Betekenis
Σ	Het gebruikte alfabet
$\Sigma*$	De verzameling strings van eindige lengte van letters uit Σ
d	Aantal karakters in Σ
P	Patroon (de tekst die gezocht wordt)
p	Lengte van P
T	De hele tekst waarin gezocht wordt
t	lengte van T

- We willen een bepaalde string (het patroon P) in een langere string (de tekst T) lokaliseren.
- We nemen aan dat we alle plaatsen zoeken waar dat patroon voorkoomt.
- We veronderstellen ook dat P en T in het inwendig geheugen opgeslaan zitten.
- In de voorbeelden worden volgende concrete informatie gebruikt:
  - $\Sigma = \{A, C, G, T\}$
  - $d = 4$
  - $P =$ `GCAGAGCAG`
  - $p = 9$
  - $T =$ `GCATCGCAGAGCAGAGTACAGCACG`
  - $t = 25$

1.1 Formele talen

- Een **formele taal** over een alfabet is een verzameling eindige strings over dat alfabet.
- Een formele taal wordt vrij vaag gedefinieerd (maar zien we niet in de cursus).
- Een formele taal kan op twee manieren gedefinieerd worden: via **generatieve grammatica's** of via **reguliere expressies**.

1.1.1 Generatieve grammatica's

- Een **generatieve grammatica** is een methode om een taal te beschrijven.
- Er is een startsymbool dat getransformeerd kan worden tot een zin van de taal met behulp van substitutieregels.
- Buiten de karakters Σ van het alfabet, is er ook nog een verzameling **niet-terminale symbolen**.
- Een niet-terminaal symbool wordt aangeduid als

$$\langle \dots \rangle$$

  waarin $\dots$ vervangen wordt door de naam van het niet-terminale symbool.
- De verzameling alle strings uit Σ vermengd met de niet-terminale symbolen is Ξ, en de daarbijhorende verzameling strings $\Xi*$.
- Een belangrijk geval zijn de **contextvrije grammatica's**.
  - Er is op elk moment een string uit $\Xi*$.
  - Als er geen niet-terminale symbolen meer zijn krijgt men een zin in de taal, anders kan men één niet-terminaal vervangen door een string uit $\Xi*$.
  - De taal is contextvrij omdat de substitutie onafhankelijk is wat voor en achter de betreffende niet-terminaal staat.
  - Een voorbeeld van een contextvrije grammatica:

$$\begin{aligned} \langle \mathbf{S} \rangle &::= \langle \mathbf{AB} \rangle \mid \langle \mathbf{CD} \rangle \\ \langle \mathbf{AB} \rangle &::= a\langle \mathbf{AB} \rangle b \mid \epsilon \\ \langle \mathbf{CD} \rangle &::= c\langle \mathbf{CD} \rangle c \mid \epsilon \end{aligned}$$

    - Hierbij is $\Sigma = \{a, b, c, d\}$ en ϵ de lege string.
    - Deze grammatica definieert als formele taal de verzameling van alle strings ofwel bestaande uit een rij 'a's gevolgd door een even lange rij 'b's ofwel bestaande uit een rij 'c's gevolgd door een even lange rij 'd's.
    - De afleiding van "cccddd":

$$\langle \mathbf{S} \rangle \to \langle \mathbf{CD} \rangle \to c\langle \mathbf{CD} \rangle d \to cc\langle \mathbf{CD} \rangle dd \to ccc\langle \mathbf{CD} \rangle ddd \to cccddd$$

1.1.2 Reguliere uitdrukkingen

- Een **reguliere uitdrukking** is ook een methode om een taal te beschrijven.
- Een reguliere uitdrukking, of *regexp*, is een string over het alfabet $\Sigma = \{\sigma_0, \sigma_1, \cdots, \sigma_{d-1}\}$ aangevuld met de symbolen $\varnothing, \epsilon, *, (,)$ en $\perp$, gedefinieerd door

$$\begin{aligned} \langle \mathbf{Regexp} \rangle &::= \langle \mathbf{basis} \rangle \mid \langle \mathbf{samengesteld} \rangle \\ \langle \mathbf{basis} \rangle &::= \sigma_0 | \cdots | \sigma_{d-1} | \varnothing | \epsilon \\ \langle \mathbf{samengesteld} \rangle &::= \langle \mathbf{plus} \rangle | \langle \mathbf{of} \rangle | \langle \mathbf{ster} \rangle \\ \langle \mathbf{plus} \rangle &::= (\langle \mathbf{Regexp} \rangle \langle \mathbf{Regexp} \rangle) \\ \langle \mathbf{of} \rangle &::= (\langle \mathbf{Regexp} \rangle \perp \langle \mathbf{Regexp} \rangle) \\ \langle \mathbf{ster} \rangle &::= (\langle \mathbf{Regexp} \rangle)* \end{aligned}$$

- Elke regexp R definieert een formele taal, Taal(R).
- Een taal die door een regexp gedefinieerd kan worden heet een reguliere taal.
- De definitie van een regexp en reguliere taal is recursief:
  1. $\varnothing$ is een regexp, met als taal de lege verzameling.
  2. De lege string ϵ is een regexp met als taal Taal(ϵ) = $\{\epsilon\}$.
  3. Voor elke $a \in \Sigma$ is "a" een regexp, met als taal Taal("a") = {"a"}.
- Regexps kunnen gecombineerd worden via drie operaties:

Operatie	Regexp	Operatie op taal/talen
Concatenatie	(RS)	Taal(R) · Taal(S)
Of	(R|S)	Taal(R) ∪ Taal(S)
Kleenesluiting	(R)*	Taal(R)*

- Vaak worden verkorte notaties gebruikt:
  - **Minstens eenmaal herhalen**
    $$rr* \to r+$$
  - **Optionele uitdrukking**
    $$r|\epsilon \to r?$$
  - **Unies van symbolen**
    $$a|b|c \to [abc]$$
    $$a|b|\ldots|z \to [a-z]$$
- Regexps kunnen gelinkt worden met graafproblemen.
- **Stelling 1** Zij G een gerichte multigraaf met verzameling takken Σ. Als a en b twee knopen van G zijn dan is de verzameling $P_G(a,b)$ van paden beginnend in a en eindigend in b een reguliere taal over Σ.
- **Bewijs:**

  Via inductie op het aantal verbindingen m van G.
  - Als $m = 0$ dan
    $$P_G(a,b) = \begin{cases} \varnothing, & \text{als } a \neq b \\ \{\epsilon\}, & \text{als } a = b \end{cases}$$
  - Breidt nu de graaf G uit naar G' door één verbinding toe te voegen.
    - Een verbinding v_{xy} van knoop x naar knoop y, waarbij eventueel $x = y$.
    - Alle paden van a naar b zijn één van de twee volgende vormen:
      1. De paden die v_{xy} niet bevatten. Deze vormen de reguliere taal $P_G(a,b)$.
      2. De paden die v_{xy} wel bevatten. Deze verzameling wordt gegeven door
         $$P_G(a,x) \cdot \{v_{xy}\} \cdot (P_G(y,x) \cdot \{v_{xy}\}) * \cdot P_G(y,b)$$

      Deze is bekomen uit reguliere talen en is dus regulier.
-

1.2 Variabele tekst

1.2.1 Een eenvoudige methode

- We zitten op een bepaalde positie j in T.
- Vanaf j wordt $T[j+i]$ met $P[i]$ vergeleken voor $0 < i \leq p$.
  1. Het eerste geval komt voor wanneer $T[j+i] \neq P[i]$, voor $i \leq p$, en het patroon dus niet gevonden is op positie j in T.
  2. Het tweede geval komt dan voor wanneer het patroon wel gevonden is op positie j in T.
- Voor willekeurige strings zal $T[j]$ vaak verschillen van $P[0]$.
  - Op veel posities j zal de karaktervergelijking na één positie dan stoppen.
- De **gemiddelde uitvoeringstijd** is $O(t)$.
- Het **slechtste geval** is $O(tp)$.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
T	G	C	A	T	C	G	C	A	G	A	G	C	A	G	A	G	T	A	C	A	G	C	A	C	G
	G	C	A	G	A	G	C	A	G																
		G	C	A	G	A	G	C	A	G															
			G	C	A	G	A	C	G	C	A	G													
				G	C	A	G	A	G	G	C	A	G												
					G	C	A	G	A	G	C	A	G	A	G										
						G	C	A	G	A	G	C	A	G	A	G									
							G	C	A	G	A	G	C	A	G	A	G								
								G	C	A	G	A	G	C	A	G	C	A	G						
									G	C	A	G	A	G	C	A	G	C	A	G					
										G	C	A	G	A	G	C	A	G	C	A	G				
											G	C	A	G	A	G	C	A	G	C	A	G			
												G	C	A	G	A	G	C	A	G	C	A	G		
													G	C	A	G	A	G	C	A	G	C	A	G	
														G	C	A	G	A	G	C	A	G	C	A	G
															G	C	A	G	A	G	C	A	G	C	A
																G	C	A	G	A	G	C	A	G	C
																	G	C	A	G	A	G	C	A	G
																		G	C	A	G	A	G	C	A
																			G	C	A	G	A	G	C
																				G	C	A	G	A	G
																					G	C	A	G	A
																						G	C	A	G
																							G	C	A
																								G	C
																									G

1.2.2 Knuth-Morris-Pratt

De prefixfunctie

- Gegeven een string P en index i met $i \leq p$.
- Een string Q kan voor i op P gelegd woorden als $i \geq q$ en als Q overeenkomt met de even lange deelstring van P endigend voor i.
  - De index i wijst naar de plaats *voorbij* de deelstring, niet naar de laatste letter van de deelstring.
- De prefixfunctie $q(i)$ van een string P bepaalt voor elke stringpositie i, $1 \leq i \leq p$, de lengte van de langste prefix van P met lengte kleiner dan i dat we voor i kunnen leggen.
- Volgende eigenschappen gelden:
  - $q(0) = -$ (niet gedefinieerd)
  - $q(1) = 0$
  - $q(i) < i$

- $q(i+1) \leq q(i)+1$

- De waarde van $q(i+1)$ kan bepaald worden als de waarden van de vorige posities gekend zijn.

$$q(i+1) = \begin{cases} q(i)+1 & \text{als } P[q(i)] = P[i] \\ q(q(i))+1 & \text{als } P[q(q(i))] = P[i] \\ q(q(q(i)))+1 & \text{als } P[q(q(q(i)))] = P[i] \\ \dots & \\ 0 & \text{als } q(q(q(....))) = 0 \end{cases}$$

  - De waarden van de prefixfunctie voor $P =$ `ANOANAANOANO` zijn als volgt:
    - Voor $i = 2$ geldt $q(i) = 0$:
      - $P[q(1)] = P[1]\ ? \rightarrow P[0] = P[1]\ ? \rightarrow A \neq N$
      - $q(2) = 0$
    - Voor $i = 4$ geldt $q(i) = 1$:
      - $P[q(3)] = P[3]\ ? \rightarrow P[0] = P[3]\ ? \rightarrow A = A$
      - $q(4) = q(3)+1 = 0+1 = 1$
    - Voor $i = 12$ geldt $q(i) = 3$:
      - $P[q(11)] = P[11]\ ? \rightarrow P[5] = P[11]\ ? \rightarrow A \neq O$
      - $P(q(5)) = P[11]\ ? \rightarrow P[2] = P[11]\ ? \rightarrow O = O$
      - $q(12) = q(5)+1 = 2+1 = 3$

	A	N	O	A	N	A	A	N	O	A	N	O	-
i	0	1	2	3	4	5	6	7	8	9	10	11	12
q(i)	-	0	0	0	1	2	1	1	2	3	4	5	3

- De prefixwaarden worden dus voor stijgende i berekend.
- Wat is de **efficiëntie**?
  - Er moeten p prefixwaarden berekend worden.
  - De recursierelatie wordt ook maar $p-1$ herhaald voor de voltallige bepaling van de prefixfunctie.
  - De methode is $\Theta(p)$.

Een eenvoudige lineaire methode

- Stel een string samen bestaande uit P gevolgd door T, gescheiden door een speciaal karakter dat in niet in beide strings voorkomt.
- Bepaal de prefixfunctie van deze nieuwe string, in $\Theta(n+p)$.
- Als de prefixwaarde van een positie i gelijk is aan p, werd P gevonden, beginnend bij index $i-p$ in T.

Het Knuth-Morris-Prattalgoritme

- Ook een lineaire methode, maar is efficiënter.
- Stel dat P op een bepaalde beginpositie vergeleken wordt met T, en dat er geen overeenkomst meer is tussen $P[i]$ en $T[j]$.

- Als $i = 0$, dan wordt P één positie naar rechts geschoven en begint het vergelijken met T weer bij $P[0]$.
- Als $i > 0$, dan is er een prefix van P met lengte i gevonden, dat we voor j op T kunnen leggen.
  - Verschuif P met een stap s kleiner dan i.
  - Er is nu een overlapping tussen het begin van P en het prefix van P dat we in T gevonden hebben.
  - De overlapping heeft lengte $i - s$.
  - De overlappende delen moeten wel overeenkomen.
  - De kleinste waarde van s waarbij dit mogelijk is, is $s = i - q(i)$.
  - Verschuif P met s en vergelijk verder vanaf $T[j]$ en $P[q(i)]$.

i	0	1	2	3	4	5	6	7	8	9
P	G	C	A	G	A	G	C	A	G	-
q(i)	-1	0	0	0	1	0	1	2	3	4
s	1	1	2	3	3	5	5	5	5	5

Tabel 1.1: Het patroon $P =$ `GCAGAGCAG`, de bijhorende prefixfunctie $q(i)$ en de s-waarden.

	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
	T	G	C	A	T	C	G	C	A	G	A	G	C	A	G	A	G	T	A	C	A	G	C	A	C	G
$i - q(i)$	sprong																									
3 - 0	3	G	C	A	G	A	G	C	A	G																
0 - (-1)	1				G	C	A	G	A	G	C	A	G													
0 - (-1)	1					G	C	A	G	A	G	C	A	G												
9 - 4	5						G	C	A	G	A	G	C	A	G											
6 - 1	5											G	C	A	G	A	G	C	A	G						
1 - 0	1																G	C	A	G	A	G	C	A	G	
1 - 0	1																	G	C	A	G	A	G	C	A	G

Tabel 1.2: Een eerste versie van het Knuth-Morris-Prattalgoritme, , waarbij $s = i - q(i)$.

- In tabel 1.4 kan opgemerkt worden dat bij de verschuiving, waarbij de fout op $T[16] = T$ veroorzaakt door het verkeerde karakter C, er opnieuw een C vergeleken wordt (Dit geldt niet voor $T[3]$ en het verkeerde karakter G omdat de verschuiving naar de eerste letter van het patroon is).
- Er is een **bijkomende voorwaarde**: de verschuiving s is enkel zinvol als $P[i - s] \neq P[i]$.
- Op basis van $q(i)$ wordt een nieuwe functie $q'(i)$ gedefinieerd, die een zinvolle verschuiving $s = i - q'(i)$ geeft, zodanig dat $P[i - s] \neq P[i]$.

$$q'(i) = \begin{cases} 0 & \text{als } q(i) = 0 \\ q(i) & \text{als } q[i] \neq 0 \text{ en } P[q(i) + 1] \neq P[i + 1] \\ q'(q(i)) & \text{als } q[i] \neq 0 \text{ en } P[q(i) + 1] = P[i + 1] \end{cases}$$

i	0	1	2	3	4	5	6	7	8	9
P	G	C	A	G	A	G	C	A	G	-
q(i)	-1	0	0	0	1	0	1	2	3	4
q'(i)	-1	0	0	0	1	0	0	0	0	4
s	1	1	2	3	3	5	6	7	8	5

Tabel 1.3: Het patroon $P =$ `GCAGAGCAG`, de nieuwe prefixfunctie $q'(i)$ en de nieuwe s-waarden.

	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
	T	G	C	A	T	C	G	C	A	G	A	G	C	A	G	A	G	T	A	C	A	G	C	A	C	G
$i - q'(i)$	sprong																									
3 - 0	3	G	C	A	G	A	G	C	A	G																
0 - (-1)	1				G	C	A	G	A	G	C	A	G													
0 - (-1)	1					G	C	A	G	A	G	C	A	G												
9 - 4	5						G	C	A	G	A	G	C	A	G											
6 - 0	6											G	C	A	G	A	G	C	A	G						
1 - 0	1																	G	C	A	G	A	G	C	A	G

Tabel 1.4: De tweede versie van het Knuth-Morris-Prattalgoritme, waarbij $s = i - q'(i)$.

1.2.3 Boyer-Moore

- Dit algoritme is een **variant** van het Knuth-Morris-Prattalgoritme.
- ! Het patroon wordt van achter naar voor overlopen bij het vergelijken met de tekst.
- Er worden **twee heuristieken** gebruikt die grotere verschuivingen mogelijk maakt. Het maximum van de twee heuristieken wordt dan gebruikt als verschuiving:
  1. **De heuristiek van het verkeerde karakter**.
  2. **De heuristiek van het juiste suffix**.

De heuristiek van het verkeerde karakter

- Het tekstkarakter waar een fout voorkomt wordt f genoemd (het verkeerde karakter in de tekst T).
- Als T ook dit karakter bevat, op een andere positie, kan P naar rechts verschoven worden.
- Om de verschuiving te bepalen wordt **de meest rechtse positie** i, links van $p - 1$ in P van elk karakter in het alfabet bijgehouden.
  - Dit wordt geïmplementeerd als een tabel, MRP genaamd, geïndexeerd op de karakters van het alfabet (tabel 1.5).

f	A	C	G	T
$MRP[f]$	7	6	5	-1

Tabel 1.5: De MRP-tabel voor $P =$ `GCAGAGCAG`. De waarden voor A en C zijn vanzelfsprekend. De waarde van G is niet 8, omdat dat sowieso het eerste karakter is dat vergeleken wordt, en telt niet mee. Een karakter dat niet in het patroon voorkomt krijgt de waarde -1.

  - Het volstaat nu om de waarde $k = MRP[f]$ op te zoeken, waarbij f het foute karakter in T is, op positie i in P, en P te verschuiven over $i - k$ posities.
    - ! In het geval dat $i - k < 0$, dan bedraagt de verschuiving 1 positie.
- Er zijn **drie varianten** van deze heuristiek:
  1. **Uitgebreide heuristiek van het verkeerde karakter**.
     - De MRP-tabel wordt uitgebreidt, zodat $MRP[f]$ de positie j teruggeeft, **links** van foutpositie i in het patroon.
     - Hiervoor is een tweedimensionale tabel nodig en is in het algemeen een vrij slechte uitbreiding.
  2. **Variant van Horspool**.
     - Dezelfde MRP-tabel als in de oorspronkelijke versie wordt gebruikt.

			j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
			T	G	C	A	T	C	G	C	A	G	A	G	C	A	G	A	G	T	A	C	A	G	C	A	C	G
f	$i-k$	sprong																										
C	4 − 6	1		G	C	A	G	A	G	C	A	G																
A	8 − 7	1			G	C	A	G	A	G	C	A	G															
G	6 − 5	1				G	C	A	G	A	G	C	A	G														
C	8 − 6	2					G	C	A	G	A	G	C	A	G													
/	/	1							G	C	A	G	A	G	C	A	G										
A	8 − 7	1								G	C	A	G	A	G	C	A	G									
G	6 − 5	2									G	C	A	G	A	G	C	A	G								
A	8 − 7	1											G	C	A	G	A	G	C	A	G						
C	8 − 6	2												G	C	A	G	A	G	C	A	G					
A	5 − 6	1														G	C	A	G	A	G	C	A	G			
C	8 − 6	2															G	C	A	G	A	G	C	A	G		
C	8 − 6	2																	G	C	A	G	A	G	C	A	G

Tabel 1.6: Het Boyer-Moore algoritme, enkel gebruik makend van de oorspronkelijke heuristiek van het verkeerde karakter.

- Bij en fout op tekstpositie m (positie waarbij $P[0]$ overeenkomt met T) en patroonpositie i, wordt P zodanig opgeschoven zodanig dat $T[m+p-1] = P[p-1]$.
- Zoek de positie van de meest rechtste positie van het karakter van $T[m+p-1]$: $k = MRP[T[m+p-1]]$.
- De verschuiving van P bedraagt bij een fout dan altijd $p-1-k$.
  - ✓ Verschuiving is onafhankelijk van patroonpositie i.
- Als deze variant gebruikt wordt, wordt de tweede heuristiek niet gebruikt, wat in het slechtste geval dus $O(pt)$ oplevert.

				j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
				T	G	C	A	T	C	G	C	A	G	A	G	C	A	G	A	G	T	A	C	A	G	C	A	C	G
m	$T[m+p-1]$	$p-1-k$	sprong																									
0	G	9 − 1 − 5	3	G	C	A	G	A	G	C	A	G																
3	C	9 − 1 − 6	2				G	C	A	G	A	G	C	A	G													
5	/	/	1						G	C	A	G	A	G	C	A	G											
6	A	9 − 1 − 7	1							G	C	A	G	A	G	C	A	G									
7	G	9 − 1 − 5	3								G	C	A	G	A	G	C	A	G								
10	C	9 − 1 − 6	2											G	C	A	G	A	G	C	A	G					
12	G	9 − 1 − 5	3													G	C	A	G	A	G	C	A	G			
15	C	9 − 1 − 6	2																G	C	A	G	A	G	C	A	G

Tabel 1.7: Het Boyer-Moore algoritme: Horspool variant.

3. **Variant van Sunday**.
   - ???

De heuristiek van het juiste suffix

- Hier wordt enkel de versie van de **originele Boyer-Moore** methode besproken, dus niet de varianten van Horspool of Sunday.
- In vele gevallen kan f aan de rechterkant van foutpositie i voorkomen, zodat $i - j < 0$, en er dus maar een verschuiving van 1 positie mogelijk is.
- Op positie i in P vinden we een verkeerd karakter f in T.
- Er is dus een **suffix** van P in T, met lengte $p - i - 1$.
- We willen weten of dit suffix s nog ergens in P voorkomt.
  - Als er meerdere plaatsen zijn waar s in P voorkomt, wordt de meeste rechtse genomen.
  - Suffixen kunnen overlappen.
- We willen dus de meeste rechtste positie j in P, waarbij $j \leq i$ waar een deelstring $s' = s$ begint.

- Analoog aan de prefixfunctie, is er nu een suffixfunctie $s(j)$:
  - Voor elke index j in P wordt de lengte van het grootste suffix van P bijgehouden, dat op index j begint.
  - De suffixwaarden is het omgekeerde van de prefixtabel voor het omgekeerde patroon P.
  - De grootste waarde voor j waarvoor $s(j) = p - i - 1$ is de waarde voor k.
  - Een verschuiving $v[i]$ voor foutpositie i is dan $i + 1 - k$. Als k niet gedefinieerd is dan is $v[i] = p - s[0]$.
- Voorbeeld:
  - Het patroon $P =$ `ABBABAB`.
  - Tabel 1.8 toont alle verschillende waarden:

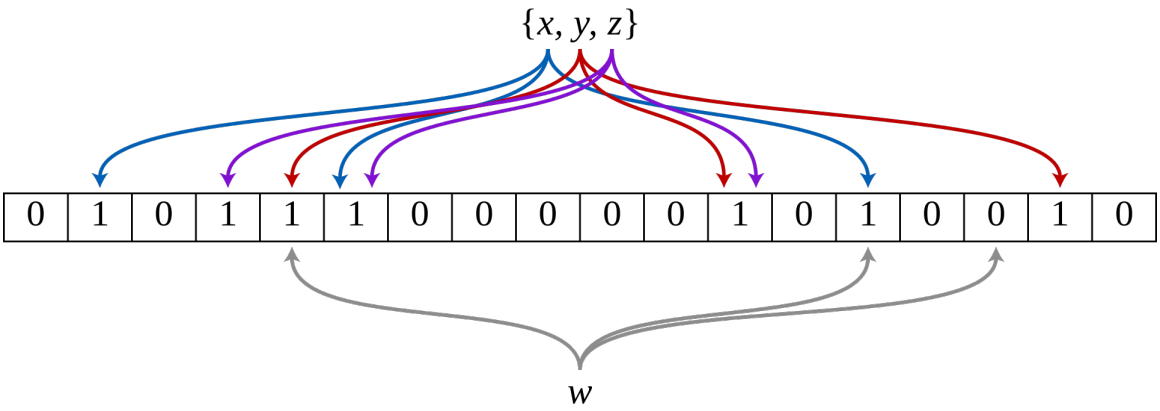
i	0	1	2	3	4	5	6
$p - i - 1$	6	5	4	3	2	1	0
$P[i]$	A	B	B	A	B	A	B
$s[i]$	2	1	3	2	1	0	0
k	/	/	/	2	3	4	6
$i + 1 - k$	/	/	/	2	2	2	1
$v[i]$	5	5	5	2	2	2	1

Tabel 1.8

- Er zijn **drie speciale gevallen** die zich kunnen voordoen:
  1. **Het patroon P werd gevonden**.
     - Er is geen foutief patroonpositie ($i = -1$) en het juiste suffix is nu P zelf.
     - Toch mogen er geen p posities opgeschoven worden, want een nieuwe P in T kan de vorige gedeeltelijk overlappen.
     - De overlapping is het langst mogelijke suffix van P, korter dan p.
     - De verschuiving is dus $v[-1] = p - s[0]$ (virtueel tabelelement, kan geïmplementeerd worden als constante).
  2. **Er is geen juist suffix**.
     - Als $i = p - 1$, dan is er geen juist suffix.
     - Er is geen waarde voor de verschuiving, dus de waarde van de eerste heuristiek moet gebruikt worden.
  3. **Het juiste suffix komt niet meer in P voor**.
     - Er is geen index j gevonden waarvoor $s(j) = p - i - 1$.
     - De verschuiving is opnieuw $v[i] = p - s[0]$ voor $0 < i < p$.

1.2.4 Onzekere algoritmen

- Algoritmen die een zekere waarschijnlijkheid hebben om een geheel foutief resultaat te geven.
- Zulke algoritmen worden ook **Monte Carloalgoritmen** genoemd.
- Er zijn redenen waarom zulke algoritmen toch nuttig kunnen zijn;
  1. Zulke algoritmen zijn vaak sneller.
     - Een voorbeeld is een **Bloomfilter** (figuur 1.1).



Figuur 1.1: Een bloomfilter, die de verzameling $\{x, y, z\}$ beschrijft. De logische OF met al deze elementen is al reeds uitgevoerd. De controle of w ook in deze verzameling zit zegt dat deze er niet in zit, want een bit van de hashwaarde van w in de bloomfilter is 0.

- We willen een verzameling van objecten in gehashte vorm bijhouden.
- Een Bloomfilter houdt de logische bitsgewijze OF bij van de hashwaarden van alle elementen.
- Om te weten of een object in de verzameling zit wordt deze eerst gehasht. Daarna wordt de logische EN operatie gebruikt op de bloomfilter met deze waarde.
- Als het resultaat verschilt van de hashwaarde dan zit het object er zeker niet in.
- Anders weten we het niet, en moet de verzameling doorzocht worden.

2. Men tracht de kans dat er een fout voorkomt zo klein mogelijk te maken.

1.2.5 Het Karp-Rabinalgoritme

- Herleidt het vergelijken van strings tot het vergelijken van getallen.
- Aan elke mogelijke string die even lang is als P wordt een getal toegekend.
- Er zijn d^p verschillende strings met lengte p, zodat de getallen groot kunnen worden.
  - Daarom worden de getallen beperkt tot deze die in één processorwoord (met lengte w bits) voorgesteld kunnen worden, via een modulobewerking.
- Meerdere strings zullen met hetzelfde getal moeten overeenkomen ($\equiv$ hashing).
- Gelijke strings betekent nog altijd gelijke getallen, maar een gelijk getal betekent niet meer dezelfde string.
  - Bij een gelijk getal moet het patroon nog steeds vergeleken worden met de tekst op die positie.
- Hoe worden de getallen gedefinieerd?
  - Ze moeten in $O(1)$ berekend kunnen worden voor elk van de $O(t)$ deelstrings in de tekst.
  - Een hashwaarde voor een string met lengte p in $O(1)$ berekenen is niet realistisch.
  - Daarom wordt de hashwaarde voor de deelstring op positie $j + 1$ berekend op basis van de deelstring op basis j.
  - De eerste hashwaarde berekenen ($j = 0$) mag dan langer duren.

- De voorstelling van P:
  - We beschouwen een string als een getal in een $d-$tallig talstelsel omdat elk stringelement d waarden kan aannemen zodat elk stringelement wordt voorgesteld door een cijfer tussen 0 en $d-1$.

$$H(P) = \sum_{i=0}^{p-1} P[i]d^{p-i-1} = P[0]d^{p-1} + P[1]d^{p-2} + \cdots + P[p-2]d + P[p-1]$$

  - Om de beperkte waarde te bekomen, wordt de rest bij deling door een getal r genomen. Dit wordt de **fingerprint** genoemd.

$$H_r(P) = H(P) \bmod r$$

  - Dit is geen efficiënte operatie omdat de individuele getallen van de som in $H(p)$ groot kunnen worden, maar gelukkig

$$(a+b) \bmod r = (a \bmod r + b \bmod r) \bmod r$$

    Dit geldt ook voor verschil en het product.
  - Omdat elk tussenresultaat nu binnen een processorwoord past, is $H_r(P)$ berekenen slechts $\Theta(p)$.
- De voorstelling van T:
  - De waarde T_0 bij beginpositie $j = 0$ wordt op dezelfde manier berekend als P.

$$H(T_0) = \sum_{i=0}^{p-1} T[i]d^{p-i-1} = T[0]d^{p-1} + T[1]d^{p-2} + \cdots + T[p-2]d + T[p-1]$$

  - Er is nu een eenvoudig verband tussen het getal voor de deelstring T_{j+1} bij beginpositie $j+1$ en dat voor T_j bij beginpositie j:

$$H(T_{j+1}) = (H(T_j) - T[j]d^{p-1})d + T[j+p]$$

    (De waarde $T[j]d^{p-1}$ aftrekken en die van $T[j+p]$ optellen en er ook voor zorgen dat de macht die bij $T[j+1], T[j+2], ...T[j+p-1]$ hoort met 1 verhoogt wordt door te vermenigvuldigen met d)
    - Stel een string T = `ABCDE`, $d = 5$ en $p = 3$ (wat P is maakt niet uit voor dit voorbeeld). De waarden van de stringelementen zijn $A = 1, B = 2, C = 3, D = 4, E = 5$.
    - De opeenvolgende waarden T_j zijn dan:
      -

$$\begin{aligned} H(T_0) &= \sum_{i=0}^{2} T[i]5^{2-i} \\ &= A \cdot 5^2 + B \cdot 5^1 + C \\ &= 1 \cdot 5^2 + 2 \cdot 5^1 + 3 \\ &= 25 + 10 + 3 = 38 \end{aligned}$$

      -

$$\begin{aligned} H(T_1) &= (H(T_0) - T[0]5^2) \cdot 5 + T[3] \\ &= (A \cdot 5^2 + B \cdot 5 + C - A \cdot 5^2) \cdot 5 + D \\ &= B \cdot 5^2 + C \cdot 5 + D \\ &= 2 \cdot 5^2 + 3 \cdot 5 + 4 \\ &= 50 + 15 + 4 = 69 \end{aligned}$$

*

$$\begin{aligned} H(T_2) &= (H(T_1) - T[1]5^2) \cdot 5 + T[4] \\ &= (B \cdot 5^2 + C \cdot 5 + D - B \cdot 5^2) \cdot 5 + E \\ &= C \cdot 5^2 + D \cdot 5 + E \\ &= 3 \cdot 5^2 + 4 \cdot 5 + 5 \\ &= 75 + 20 + 5 = 100 \end{aligned}$$

- Analoog aan $H_r(P)$ worden de waarden $H(T)$ ook modulo r genomen, zodat

$$H_r(T_{j+1}) = H(T_{j+1}) \bmod r$$

- Het berekenen van $H_r(P)$, $H(T_0)$ en $d^{p-1} \bmod r$ vereist $\Theta(p)$ operaties.
- Het berekenen van alle andere fingerprints $H_r(T_j)(0 < j \leq t - p)$ vergt $\Theta(t)$ operaties.
- Dit is $\Theta(t + p)$.
- Maar, de strings moeten nog vergeleken worden als de fingerprints hetzelfde zijn.
- In het slechtste geval zijn de fingerprints op elke positie gelijk, zodat de totale performantie **O(tp)** is.
- Er zijn nu nog twee mogelijkheden om r te bepalen:
  1. **Vaste** r
     - Kies r als een zo groot mogelijk priemgetal zodat $rd \leq 2^w$.
     - Priemgetallen zorgt ervoor dat gelijkaardige deelstrings dezelfde fingerprinters zouden opleveren.
     - Een groot priemgetal zorgt voor een groot aantal mogelijke fingerprints.
     - Er is nu wel een nieuw verband tussen $H_r(T_{j+1})$ en $H_r(T_j)$:

$$H_r(T_{j+1}) = \Big(\big((H_r(T_j) + r(d-1) - T[j](d^{p-1} \bmod r)) \bmod r \big) d + T[j+1] \Big) \bmod r$$

       (De term $r(d-1)$ wordt toegevoegd om een negatief tussenresultaat te vermijden.)
  2. **Random** r
     - Soms is een vaste r nadelig: er kan bijvoorbeeld een slechte waarde gekozen worden.
     - De veiligste implementatie gebruikt een willekeurige priem r uit een bepaald bereik.
     - Een groter bereik reduceert de kans op fouten.
     - Het aantal priemgetallen kleiner of gelijk aan k is $\frac{k}{\ln k}$.
     - Door k groot te kiezen zal slechts een klein deel van die priemgetallen een fout veroorzaken.
     - De kans dat r één van die priemen is wordt klein.
     - Voor $k = t^2$ is de kans op één enkele foute $O(1/t)$.
     - Om fouten helemaal te vermijden zijn er twee mogelijkheden:
       - Overgaan naar een andere methode als de fout gesignaleerd wordt.
       - Herbeginnen met een nieuwe random priem r.

1.2.6 Zoeken met automaten

- Een **automaat** is een informatieverwerkend eenheid.

Deterministische automaten

- Een deterministische automaat (DA) bestaat uit:
  - Een eindige verzameling invoersymbolen Σ.
  - Een eindige verzameling staten S.
  - Een begintoestand $s_0 \in S$.
  - Een eindige verzameling eindstaten $F \subset S$.
  - Een overgangsfunctie $p(t, a)$ die de nieuwe toestand geeft wanneer de DA in toestand t invoersymbool a ontvangt.
- Een DA kan voorgesteld worden een gelabelde multigraaf G.
  - De knopen zijn de toestanden.
  - De verbindingen zijn de overgangen.
- Als de DA zich in een eindstaat bevindt na het invoeren van een string, dan wordt deze string herkent door de DA.
- Een taal die door een DA herkent wordt is regulier.
  - Dit is de verzameling van labels $P_G(\{s_0\}, F)$

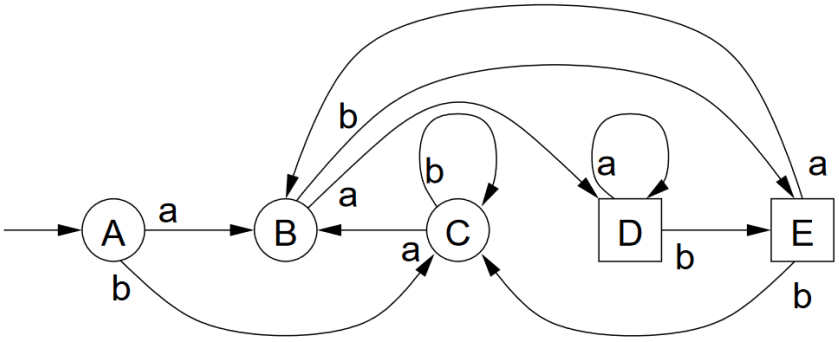
Niet-deterministische automaten

- Een niet-deterministische automaat (NA) bestaat uit:
  - Een eindige verzameling invoersymbolen Σ.
  - Een eindig aantal statenbits. De verzameling van statenbits die de waarde 1 hebben is de staat van de NA.
  - Een beginbit b_0 en een verzameling eindbits.
  - De overgangsfunctie $s(i, a)$ is de verzameling statenbits die een signaal krijgen van statenbit i als de inkomende letter a is. Een statenbit dat een signaal binnenkrijgt krijgt de waarde 1.
  - Nul of meerdere ϵ-overgangen. Een ϵ-overgang van statenbit i naar statenbit j zorgt ervoor dat wanneer i een signaal binnenkrijgt, dit signaal direct doorstuurt naar j.
- Een NA herkent een string als op het einde van die string er één of meer eindbits aan staan.

De deelverzamelingconstructie

- Een NA is een alternatieve voorstelling van een DA.
- Elke NA kan omgezet worden in een DA.
  - Een DA is eenvoudiger om te implementeren: de nieuwe toestand wordt opgezocht in een tweedimensionale tabel.
- Elke staat van de NA komt overeen met een verzameling statenbits die aanstaan.
- Als er k statenbits zijn, dan zijn er 2^k mogelijke deelverzamelingen.
  - ✓ Slechts een klein aantal van deze deelverzamelingen worden effectief bereikt.
- Er is een impliciet gegeven multigraaf met 2^k knopen.

- Om een DA te construeren zijn er twee hulpoperaties nodig:
  1. **ϵ-sluiting(T)**: De deelverzameling van statenbits bereikbaar via ϵ-overgangen vanuit een verzameling statenbits T.
  2. $p(T, a)$: De deelverzameling van statenbits rechtstreeks bereikbaar vanuit een toestand t uit T voor het invoersymbool a.
- Om een DA te construeren moet er een verzameling van toestanden S met begin- en eindtoestanden en een overgangstabel M opgesteld worden.
  - De begintoestand is ϵ-sluiting(b_0).
  - Elke andere staat wordt bekomen door ϵ-sluiting($p(T, a)$) voor elke andere staat T en invoersymbool a.
  - Een eindtoestand van de DA bevat minstens één eindtoestand van de NA. Vervolgens wordt voor elke toestand T de overgang voor elk invoersymbool a bepaald.
- Figuur 1.2 toont de DA geconstrueerd uit de NA van figuur 1.5.

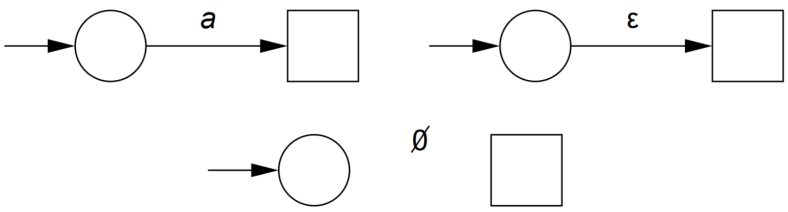


		a	b
$\rightarrow$	$A = \{1, 2, 3, 4, 8\}$	B	C
	$B = \{2, 3, 4, 5, 7, 8, 9, 10, 11\}$	D	E
	$C = \{2, 3, 4, 6, 7, 8\}$	B	C
$\leftarrow$	$D = \{2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 14\}$	D	E
$\leftarrow$	$E = \{2, 3, 4, 6, 7, 8, 13, 14\}$	B	C

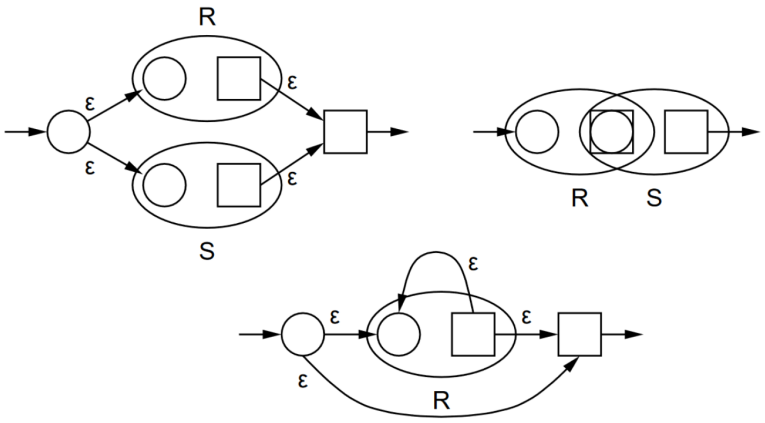
Figuur 1.2: Deterministische automaat geconstrueerd uit de NA van figuur 1.5.

Automaten voor regexps

- Een reguliere taal kan herkend worden door een DA of NA.
- Een NA kan opgebouwd worden vanuit een regexp door de **constructie van Thompson**.
  - Er worden NA's gedefinieerd voor basiselementen van een regexp: elk element uit Σ en ϵ (figuur 1.3).
  - Deze NA's kunnen samengesteld worden voor elk van de drie basisoperatoren: unie, concatenatie en Kleenesluiting (figuur 1.4).

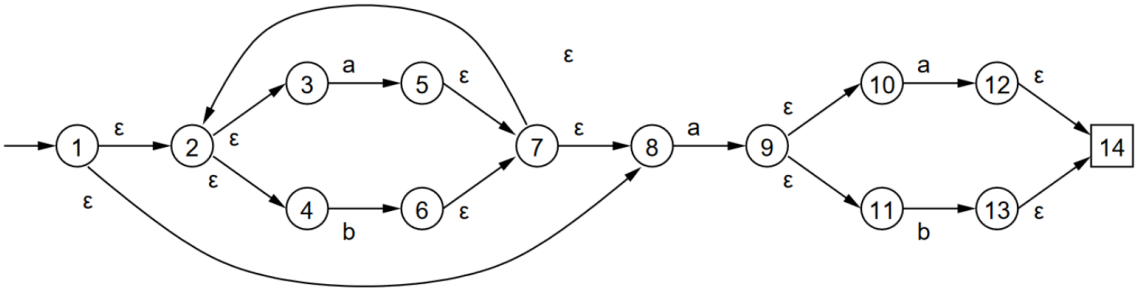


Figuur 1.3: Constructie van Thompson: basiselementen.



Figuur 1.4: Constructie van Thompson: unie, concatenatie en Kleenesluiting.

- Een NA die op deze manier geconstrueerd is, zal alle strings genereerd door de regexp herkennen en geen andere.
- Figuur 1.5 toont de NA bekomen met deze constructie voor de regexp $(a|b) * a(a|b)$.



Figuur 1.5: Constructie van Thompson: NA voor $(a|b) * a(a|b)$.

- Een NA uit de constructie van Thompson heeft drie eigenschappen:
  - Er is slechts één beginbit en één eindbit.
  - Het aantal bits is niet groter dan tweemaal het aantal elementen in de regexp.
  - Vanuit elke bit vertrekken er hoogstens twee overgangen: ofwel één overgang voor een symbool uit Σ, ofwel hoogstens twee ϵ-overgangen.

- **Stelling 2** Een taal kan herkend worden door een eindige deterministische automaat als en slechts als ze regulier is.
- **Bewijs:**
  - Een taal die herkend wordt door een automaat is zeker regulier.
    - Een taal bestaande uit labels van de paden vertrekkend uit een beginstaat en eindigd in een eindstaat is steeds regulier.
  - Als een taal regulier is, kan ze beschreven worden door een regexp en voor deze regexp kan eerst een NA opgebouwd worden, en vervolgens tot een DA omgevormd worden via de deelverzamelingconstructie.
- **Gevolg:**
  - Niet alle contextvrije talen zijn regulier.
    - Stel de strings beginnend met een aantal 'a's gevolgd door evenveel 'b's
    - Als er een 'b' tegengekomen wordt, moet er ergens bijgehouden worden hoeveel 'a's er al zijn geweest.
    - Dit aantal is niet begrensd en kan niet vooraf in een eindig geheugen geplaatst worden.

Minimalisatie van een automaat

1.2.7 De Shift-AND-methode

- Bitgeoriënteerde methode, die zeer efficiënt werkt voor **kleine patronen**.
- Hou voor elke positie j in T bij welke prefixen van P overeenkomen met de tekst, eindigend op j.
- Er is een tabel S met d woorden (tabel 1.9). Een bit i van woord $S[s]$ is waar als karakter s op plaats i in P voorkomt.

	G	C	A	G	A	G	A	G
S['G']	**1**	0	0	**1**	0	**1**	0	**1**
S['C']	0	**1**	0	0	0	0	0	0
S['T']	0	0	0	0	0	0	0	0
S['A']	0	0	**1**	0	**1**	0	**1**	0

Tabel 1.9: De tabel S bevat een bitpatroon voor elk karakter s in het alfabet. Karakters die niet in het patroon voorkomen krijgen een bitpatroon bestaande uit p nulbits.

- Een tabel R_j van p logische waarden geeft voor het i-de element het prefix van lengte i, die hoort bij tekstpositie j.

		R_0 G	R_1 C	R_2 A	R_3 T	R_4 C	R_5 G	R_6 C	R_7 A	R_8 G	R_9 A	R_{10} G	R_{11} A	R_{12} G	R_{13} T	R_{14} A	R_{15} T	R_{16} A	R_{17} C	R_{18} A	R_{19} G	R_{20} T	R_{21} A	R_{22} C	R_{23} G
$R[0]$	G	**1**	0	0	0	0	**1**	0	0	**1**	0	**1**	0	**1**	0	0	0	0	0	0	**1**	0	0	0	**1**
$R[1]$	C	0	**1**	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$R[2]$	A	0	0	**1**	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$R[3]$	G	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$R[4]$	A	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$R[5]$	G	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0
$R[6]$	A	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0
$R[7]$	G	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0

  - De startabel R_0 wordt opgebouwd als $R_0[0] = 1$ en $R_0[1 \cdots p-1] = 0$.

- Opeenvolgende tabellen R_{j+1} kunnen via efficiënte bitoperaties bekomen worden:

$$R_{j+1} = \text{Schuif}(R_j) \text{ EN } S[T[j+1]]$$

- De schuif-operatie verschuift een bitpatroon naar rechts, en voegt links een éénbit toe.

- Het patroon wordt gevonden in de tekst op positie $j - p$ als $R_j[p-1] = 1$.
- Er zijn ook **benaderingen mogelijk**. Deze benaderingen maken gebruik van dezelfde tabel $R_j = R_j^0$, en een nieuwe tabel R_j^1, waarvan de definitie afhankelijk is van het soort benadering.
  - **Karakters inlassen**
    - De tabel R_j^1 duidt alle prefixen aan eindigend op positie j met hoogstens één inlassing.

$$R_{j+1}^1 = R_j^0 \text{ OF } (\text{Schuif}(R_j) \text{ EN } S[T[j+1]])$$

    .
  - **Karakters verwijderen**
    - De tabel R_j^1 duidt alle prefixen aan eindigend op positie j met hoogstens één verwijdering.

$$R_{j+1}^1 = \text{Schuif}(R_{j+1}^0) \text{ OF } (\text{Schuif}(R_j) \text{ EN } S[T[j+1]])$$

  - **Karakters vervangen**
    - De tabel R_j^1 duidt alle prefixen aan eindigend op positie j waarbij $i-1$ van de eerste i karakters van P overeenkomen met $i-1$ karakters van de i karakters die in de tekst eindigen bij positie j.

$$R_{j+1}^1 = \text{Schuif}(R_j^0) \text{ OF } (\text{Schuif}(R_j) \text{ EN } S[T[j+1]])$$