FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Gevorderde algoritmen

Bert De Saffel

Master in de Industriële Wetenschappen: Informatica Academiejaar 2018–2019

Gecompileerd op 8 december 2019

UNIVERSITEIT
GENT

Inhoudsopgave

Deel I

Strings

Hoofdstuk 1

Gegevensstructuren voor strings

1.1 Inleiding

- Efficiënte gegevensstructuren kunnen een zoeksleutel lokaliseren door elementen één voor één te testen.
- Dit heet **radix search**.
- Meerdere soorten boomstructuren die radix search toepassen.
  - **Digitale zoekbomen**: deze bomen hebben als nadeel dat de structuur van de boom afhankelijk is van de toevoegvolgorde.
  - **Tries**: de structuur van een trie is niet afhankelijk van de toevoegvolgorde.
  - **Ternaire zoekbomen**: een alternatieve voorstelling van een meerwegstrie, die minder geheugen gebruikt en toch efficiënt blijft.

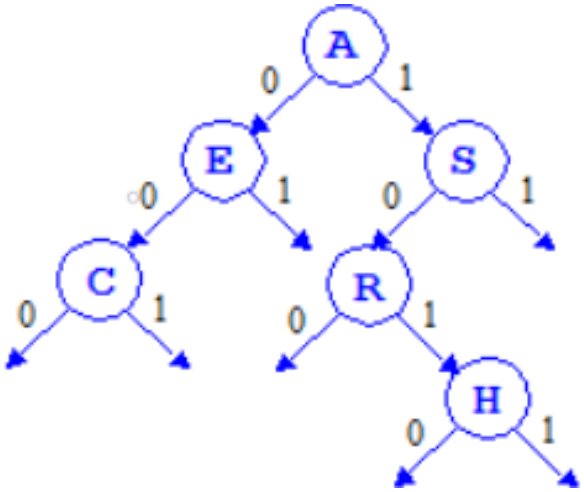
! Veronderstel dat geen enkele sleutel een prefix is van een ander. Dit wordt de **prefixvoorwaarde** genoemd.

De sleutels `test` en `testen` zullen dus nooit samen voorkomen in de boom aangezien `test` een prefix is van `testen`. Dit is noodzakelijk: stel dat een langere sleutel reeds in de boom zit. Als de kortere sleutel gezocht wordt, of toegevoegd moet worden, zullen er uiteindelijk geen sleutelelementen overblijven om ze te onderscheiden.

Dit kan opgelost worden door een speciaal karakter toe te voegen die in geen enkele sleutel zal voorkomen. Zo kunnen de sleutels `test$` en `testen$` wel samen voorkomen.

1.2 Digitale zoekbomen

- Sleutels worden opgeslagen in de knopen.
- Zoeken en toevoegen verloopt analoog als een normale binaire zoekboom.
- Slechts één verschil:
  - De juiste deelboom wordt niet bepaald door de zoeksleutel te vergelijken met de sleutel in de knoop.
  - Wel door enkel het volgende element (van links naar rechts) te vergelijken.



Figuur 1.1: Een digitale zoekboom voor zes sleutels: $A = 00001, S = 10011, E = 00101, R = 10010, C = 00011, H = 10100$, die ook in deze volgorde toegevoegd worden.

  - Bij de wortel wordt het eerste sleutelelement gebruikt, een niveau dieper het tweede sleutelelement, enz.

- In de cursus zijn de sleutelelementen beperkt tot bits → **binaire digitale zoekbomen**.
- Bij een knoop op diepte i wordt bit $(i + 1)$ van de zoeksleutel gebruikt om af te dalen in de juiste deelboom.

! De zoekboom overlopen in inorder levert de zoeksleutels niet noodzakelijk in volgorde op.

  - Sleutels in de linkerdeelboom van een knoop op diepte i zijn zeker kleiner dan deze in de rechterdeelboom.
  - Maar, de sleutel van de knoop op diepte i kan toch in beide deelbomen terechtkomen als hij later werd toegevoegd.

- De hoogte van een digitale zoekboom wordt bepaald door het aantal bits van de langste sleutel.
- Performantie is vergelijkbaar met rood-zwarte bomen:
  - Voor een groot aantal sleutels met relatief kleine bitlengte is het zeker beter dan een binaire zoekboom en vergelijkbaar met die van een rood-zwarte boom.
  - Het aantal vergelijkingen is nooit meer dan het aantal bits van de zoeksleutel.

  ✓ Implementatie van een digitale zoekboom is eenvoudiger dan die van een rood-zwarte boom.

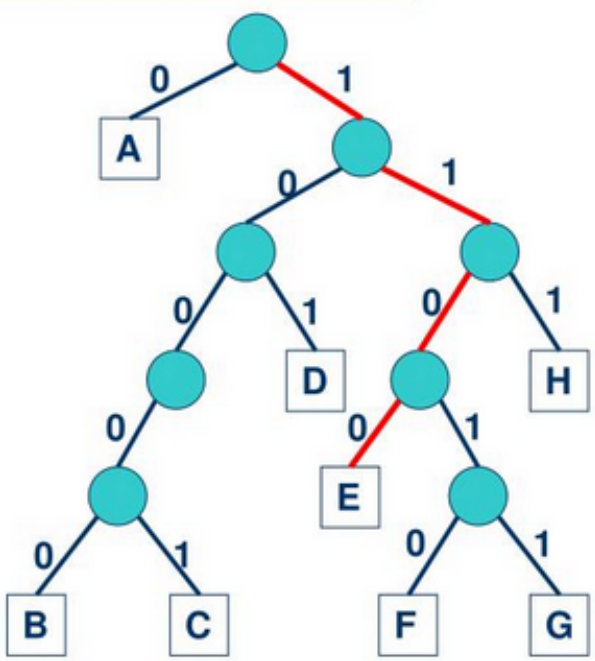
  ! De beperkende voorwaarde is echter dat er efficiënte toegang nodig is tot de bits van de sleutels.

1.3 Tries

- Een digitale zoekstructuur die wel de volgorde van de opgeslagen sleutels behoudt.
- Ze moeten echter voldoen aan de **prefixvoorwaarde**: een sleutel mag geen prefix zijn van een andere sleutel.
  - Dit kan opgelost worden door elke sleutel te laten volgen door een afsluitteken. Dit werkt echter niet bij binaire tries.

1.3.1 Binaire tries

string	node
01011	A
10000	B
10001	C
10101	D
11001	E
11010	F
11011	G
11101	H



Figuur 1.2: Een voorbeeld van een binaire trie met opgeslagen sleutels A, B, C, D, E, F, G en H. Elk van deze sleutels heeft een (willekeurig gekozen) bitrepresentatie die de individuele elementen van de sleutels voorstelt. De zoekweg van de sleutel E wordt aangegeven door rode verbindingen.

- Zoekweg wordt bepaald door de opeenvolgende bits van de zoeksleutel.
- Sleutels worden enkel opgeslaan in de bladeren, met als gevolg dat de structuur is onafhankelijk van de toevoegvolgorde van de sleutels.
  - De boom *inorder* overlopen geeft de sleutels gerangschikt terug.
  - De zoeksleutel moet niet meer vergeleken worden met elke knoop op de zoekweg.
- Twee mogelijkheden bij **zoeken** en **toevoegen**:
  1. Indien een lege deelboom bereikt wordt, bevat de boom de zoeksleutel niet. De zoeksleutel kan dan in een nieuw blad op die plaats toegevoegd worden.
  2. Anders komen we in een blad. De sleutel in dit blad **kan** eventueel gelijk zijn aangezien ze zeker dezelfde beginbits hebben.
     - Als we bijvoorbeeld `10011` zoeken maar de boom bevat enkel de sleutel `10010`, zullen we in het blad met de sleutel `10010` uitkomen aangezien de eerste 4 elementen hetzelfde zijn. De sleutels zijn echter niet gelijk.
     - Indien de sleutels niet hetzelfde zijn, kunnen twee mogelijkheden voorkomen bij **toevoegen**:
       (a) **Het volgende bit verschilt.** Het blad wordt vervangen door een knoop met twee kinderen die de twee sleutels bevat.
       (b) **Een reeks van opeenvolgende bits is gelijk.** Het blad wordt vervangen door een reeks van inwendige knopen, zoveel als er gemeenschappelijke bits zijn. Bij het eerste verschillende bit krijgen we terug het eerste geval.

! Wanneer opgeslagen sleutels veel gelijke bits hebben, zijn er veel knopen met één kind.

- Het aantal knopen is dan ook hoger dan het aantal sleutels.
- Een trie met n gelijkmatige verdeelde sleutels heeft gemiddeld $n/\ln 2 \approx 1.44n$ inwendige knopen.

1.3.2 Meerwegstries

- Heeft als doel de hoogte van een trie met lange sleutels te beperken.
- Meerdere sleutelbits in één enkele knoop vergelijken.
- Een sleutelelement kan m verschillende waarden aannemen, zodat elke knoop (potentiaal) m kinderen heeft $\rightarrow$ m-wegsboom.
- **Zoeken** en **toevoegen** verloopt analoog als bij een binaire trie:
  - In elke knoop moet nu enkel een $m-$wegsbeslissing genomen worden, op basis van het volgende sleutelelement.
  - Dit kan in $O(1)$ door per knoop een tabel naar wijzers van de kinderen bij te houden, geïndexeerd door het sleutelelement.
- Ook hier is de structuur onafhankelijk van de toevoegvolgorde van de sleutels, en de boom in inorder overlopen zorgt ook voor een gerangschikte lijst.
- De performantie is ook analoog met die van binaire tries.
  - Zoeken of toevoegen van een willekeurige sleutel vereist gemiddeld $O(\log_m n)$ testen op het aantal sleutelelementen.
  - De boomhoogte wordt ook beperkt door de lengte van de langste opgeslagen sleutel.
  - Er zijn gemiddeld $n/\ln m$ inwendige knopen.
  - Het aantal wijzers per knoop is wel $mn \ln m$.

! Het grootste nadeel is dat meerwegstries veel geheugen gebruiken. Mogelijke verbeteringen zijn:

- In plaats van een tabel met m wijzers te voorzien, waarvan de meeste toch nullwijzers zijn, kan een gelinkte lijst bijgehouden worden. Elk element van de gelinkte lijst bevat een sleutelelement en een wijzer naar een kind. De lijst is ook gerangschikt volgens de sleutelelementen, zodat niet altijd de hele lijst moet onderzocht worden om het juiste element te vinden.

  Op de hogere niveaus is een tabel met m wijzers toch beter, omdat daar meer kinderen kunnen zijn.
- Een trie kan ook enkel voor de eerste niveaus gebruikt worden, en daarna een andere gegevensstructuur gebruiken. Vaak stopt men als een deelboom niet meer dan s sleutels bevat. Deze sleutels worden dan opgeslaan in een korte lijst, die dan sequentieel doorzocht kan worden. Het aantal inwendige knopen daalt met een factor s, tot ongeveer $n/(s \ln m)$.

1.4 Variabelelengtecodering

- Normaal worden gegevens opgeslaan in gegevensvelden met een vaste grootte.
  - Een karakter in ASCII-codering wordt bevat altijd 7 bits.

- Een integer datastructuur voorziet altijd 32 bits.

- Soms is het nuttig om variabele lengte te voorzien:
  1. **Verhoogde flexibiliteit**: Wanneer blijkt dat er meer bits nodig zijn, is het eenvoudig om meer bits te voorzien.
  2. **Compressie**: Veelgebruikte letters kunnen een kortere bitlengte krijgen om de grootte van de totale gegevens te reduceren.

- In beide gevallen hebben we een **alfabet**, waarbij we niet elke letter door evenveel bits laten voorstellen.

! Een belangrijk nadeel is dat eerst de hele codering ongedaan moet gemaakt worden vooraleer er in gezocht kan worden. Variabelelengtecodering is dan ook enkel nuttig als dit niet uitmaakt.

- Bij het **decoderen** is er een **prefixcode**.
  - Dit is een codering waarbij een **codewoord**, nooit het prefix van een ander codewoord kan zijn.
  - Een codering is een mapping die elke letter van het alfabet afbeeldt op een codewoord. Bijvoorbeeld, de letters A, C, G en T van een DNA-string kunnen volgende codewoorden krijgen:

$$A \to 0$$
$$C \to 10$$
$$G \to 110$$
$$T \to 111$$

  - Op die manier weten we dat het einde van een codewoord is bereikt zonder het begin van het volgende codewoord te moeten analyseren.
    - Stel dat volgende codering binnenkomt:

```
01101011111110
```

    - Het decoderen komt dan neer op het inlezen van opeenvolgende bits totdat een blad in de trie bereikt is:

0	1 1 0	1 0	1 1 1	1 1 1	1 0
A	G	C	T	T	C

  - Een typische prefixcode voor natuurlijke getallen schrijft het getal op in een 128-delig stelsel en elk cijfer wordt apart opgeslaan in een aparte byte. Bij het laatste cijfer wordt er 128 opgeteld, zodat de laatste byte een 1-bit heeft op de meest significante plaats.
  - In geschreven taal wordt er gewacht tot een spatie of leesteken tegengekomen wordt om het onderscheidt tussen verschillende woorden te maken.

- Een trie is geschikt om een invoerstroom te decoderen die gecodeerd is met een prefixcode.
  - Alle codewoorden worden eerst opgeslaan in de trie.
  - Aan het begin van een codewoord starten we bij de wortel.
  - Per ingelezen bit of byte (afhankelijk van het probleem, bij strings zeker een byte) gaan we een niveau omlaag in de trie.
  - Bij een blad is het codewoord compleet.

1.4.1 Universele codes

- Deze codes zijn onafhankelijk van de gekozen brontekst.
- De codes worden hier geïllustreerd als de codering voor de verschillende positieve gehele getallen.

	Elias' gammacode	Elias' deltacode	Fibonaccicode
1	1	1	11
2	010	0100	011
3	011	0101	0011
4	00100	01100	1011
5	00101	01101	00011
6	00110	01110	10011
7	00111	01111	01011
8	0001000	00100000	000011
9	0001001	00100001	100011
...			
23	000010111	001010111	01000011
...			
45	00000101101	0011001101	001010011
...			

De Elias' gammacode

- Gegeven een getal n:
  - Stel het getal voor met zo weinig mogelijk bittekens (k) en laat dit voorafgaan door $k-1$ nulbits.
  - Een getal n wordt voorgesteld door $2\lfloor \log_2 n \rfloor + 1$ bittekens.
- Voorbeeld $n = 14$
  - Het getal voorstellen met k bittekens: $1110 \rightarrow k = 4$.
  - Deze voorstelling vooraf laten gaan door $k - 1 = 3$ nulbits: 0001110.

De Elias' deltacode

- Gegeven een getal n:
  - Gebruik de laatste $k - 1$ bittekens van het getal en laat dit voorafgaan door de Elias' gammacode voor k.
  - Een getal n wordt voorgesteld door $\lfloor \log_2 n \rfloor + 2\lfloor \log_2(\log_2 n + 1) \rfloor + 1$ bittekens.
- Voorbeeld $n = 14$
  - Het getal voorstellen met k bittekens: $1110 \rightarrow k = 4$.
  - De gammacode van $k = 4$ is 00100.
  - Stel de gammacode samen met de laatste $k - 1$ bittekens van n: 00100110.

De Fibonaccicode

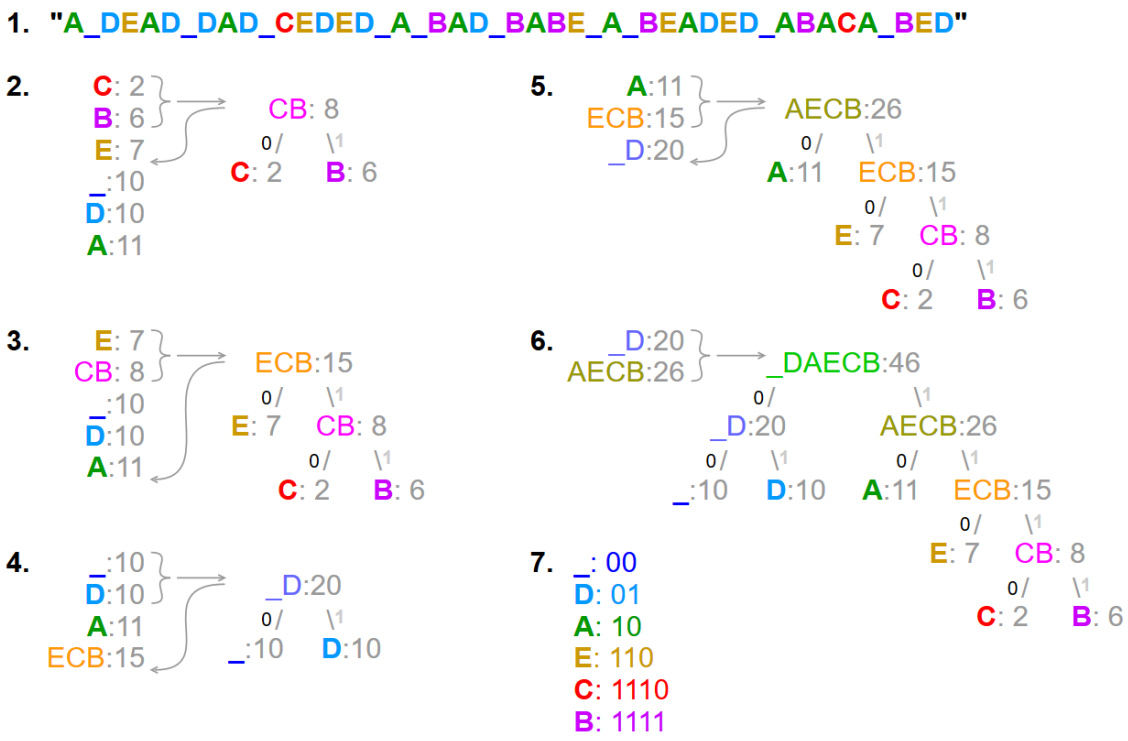
- De Fibonaccireeks
$$1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...$$
- Dit heeft als eigenschap dat een getal i geschreven kan worden als de som van verschillende Fibonaccigetallen zodanig dat er nooit twee getallen in de reeks worden gebruikt die onmiddellijke buren zijn van elkaar.
- Gegeven een getal n:
  - Overloop de Fibonaccireeks van klein naar groot en gebruik een éénbit voor elk getal dat in de berekende som voorkomt, en een nulbit voor alle andere getallen. Voeg daarna op het einde nog een éénbit toe.
  - Een getal n wordt voorgesteld door $k + 1$ bittekens.
- Voorbeeld $n = 72$
  - De som van fibonaccigetallen is $72 = 1 + 3 + 13 + 55$.
  - De reeks van Fibonacci overlopen en een éénbit gebruiken voor elk getal dat in de berekende som voorkomt levert volgende bitstring op: 101001001.
  - Dit moet nog gevolgd worden door een 1, zodat dit een prefixcode wordt: 1010010011.

1.5 Huffmancodering

- Sommige letters in een tekst kunnen meer voorkomen dan een andere.
- Minder bittekens gebruiken voor die letters speelt ten voordele van de grootte van de hele tekst.

1.5.1 Opstellen van de decoderingsboom

- Er wordt een prefixcode toegepast waarbij elke letter een apart codewoord krijgt die voor de hele tekst geldt.
- We zullen bitcodes gebruiken, en dus ook een binaire trie.
- Om de optimale code op te stellen moet nagegaan worden hoe vaak elk codewoord gebruikt zal worden.
- Er is een alfabet $\Sigma = \{s_i | i = 0, \cdots, d - 1\}$.
- We bekomen de frequenties f_i door elke letter s_i te tellen in de tekst.
- We zoeken een trie met n bladeren die de optimale code oplevert.
  - Neem een willekeurige binaire trie met d bladeren, elk met een letter uit Σ.
  - Ken aan elke knoop een gewicht toe:
    - Een blad krijgt als gewicht de frequentie f_i van de overeenkomstige letter.
    - Een inwendige knoop krijgt als gewicht de som van de gewichten van zijn kinderen.
  - Stel dat het bestand gecodeerd wordt met de bijhorende code en dat deze trie gebruikt wordt om te decoderen.
  - Het totaal aantal bits in het gecodeerde bestand is de som van de gewichten van alle knopen samen, met uitzondering van de wortel.



Figuur 1.3: Een visualisatie van huffmancodering. De te coderen tekst wordt weergegeven bij stap 1. In stap 2 wordt eerst elke letter gesorteerd in een lijst bijgehouden (eigenlijk een bos van bomen) volgens zijn niet-stijgende frequenties f_i. Stap 2 tot 6 neemt dan altijd de twee minst frequente bomen en combineert ze om een nieuwe boom te bekomen. Die boom wordt terug in het bos gestoken. Stap 7 toont de werkelijke codering. Stap 8 toont de gecodeerde versie van de tekst in stap 1.

  - De wortel heeft gewicht n (de som van alle frequenties), dus we zoeken een trie waarvoor n minimaal wordt.

- Stel een knoop k met gewicht w_k op diepte d_k. en een knoop l met gewicht w_l op diepte w_l, zodanig dat k niet onder l hangt en l niet onder k.
- Er kan een nieuwe trie gemaakt worden k, inclusief de bijbehorende deelboom, van plaats te verwisselen met l.
  - Er waren d_k knopen boven k in de trie, die verliezen gewicht w_k maar krijgen gewicht w_l.
  - Er waren d_l knopen boven l in de trie, die verliezen gewicht w_l maar krijgen gewicht w_k.
- De totale gewichtsverandering van de totale trie is

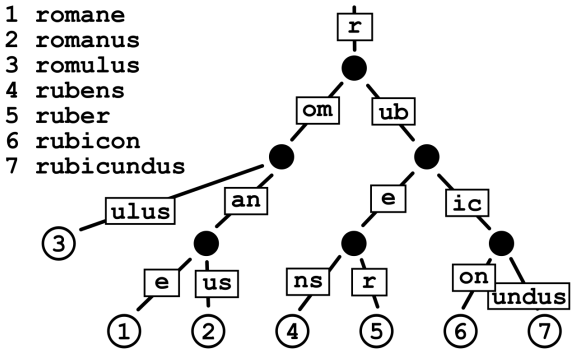
$$(d_k - d_l)(w_l - w_k)$$

- Als l een groter gewicht en kleinere diepte dan k heeft, is er een betere trie bekomen.
- De optimale trie heeft volgende eigenschappen:
  - Geen enkele knoop heeft een groter gewicht dan een knoop op een kleinere diepte.

- Geen enkele knoop heeft een groter gewicht dan een knoop links (of rechts) van hem op dezelfde diepte, want dan kunnen de twee knopen omgewisseld worden.

- Constructie van de coderingsboom:
  - Op elk moment is er een bos van deelbomen die aan elkaar gehangen moeten worden.
    - Dit bos wordt geïmplementeerd met een prioriteitswachtrij met als prioriteit het gewicht van de deelbomen.
  - In het begin bestaat het bos uit enkel bladeren.
  - De twee bomen met het lichtste gewicht worden uit het bos gehaald en worden verenigd onder een nieuwe knoop en wordt terug in het bos gestoken.
  - De diepte h van de boom is onbekend, maar wel weten we dat:
    - alle knopen op niveau h zijn zeker bladeren,
    - dat h een even getal is.
  - We kunnen bladeren twee aan twee samen nemen, telkens de lichtste (kleinst gewicht) die overblijven.
  - De resulterende bomen hebben altijd een groter gewicht, dus komen later in het gerangschikte bos.
  - Dit blijft herhaald worden tot dat er maar één boom overblijft (stap 2 tot 6 in figuur 1.3).

1.6 Patriciatries



Figuur 1.4: Een patriciatrie. Elk blad bevat een verwijzing naar een woord in een lijst en knopen met maar één kind worden samengevoegd.

! Veel trieknopen hebben maar één kind zodat er veel ongebruikt geheugen is.

! Er zijn ook twee soorten knoopen: inwendige knoop zonder sleutel maar met wijzers naar kinderen, en bladeren met sleutel maar zonder wijzers naar kinderen.

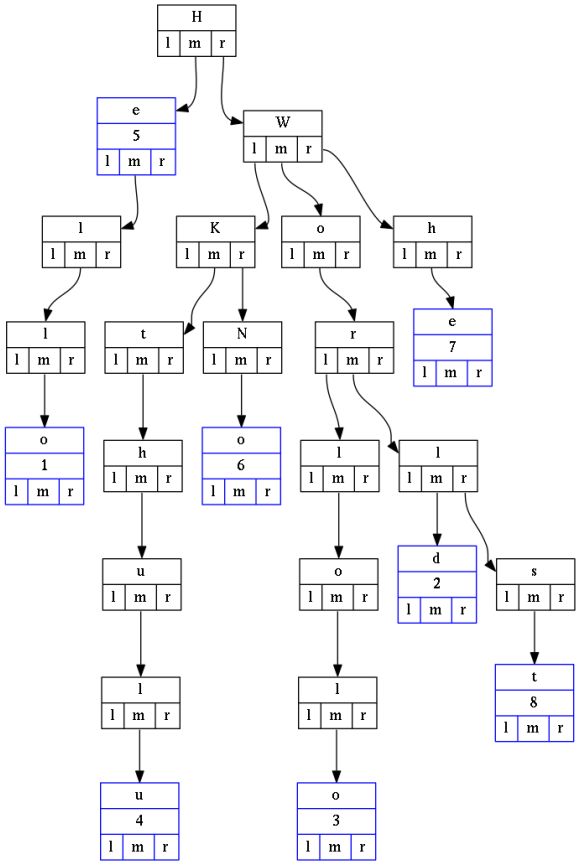
- Een **Patriciatrie** (Practical Algorithm to Retrive Information Coded In Alphanumeric) verwijdert deze problemen door enkel **knopen met meer dan één kind te behouden**.
- Bij een gewone meerwegstrie kunnen knopen voorkomen met maar één kind.
- Zo een knoop kan weggelaten worden en zijn kind kan in de plaats gezet worden.
- Twee gevolgen:

1. Als we in een kind komen, moeten we weten hoeveel voorouders er ontbreken. Dit lossen we op door een **testindex** in de knoop bijhouden, de index van het te testen karakter.
2. De karakters die niet getest worden kunnen tot conflict leiden bij een zoekstring waarbij die karakters niet overeenkomen.

- Een knoop is **expliciet** als hij nog voorkomt in de boom.
- Een knoop is **impliciet** als hij enkel wordt aangeduid door een indexsprong aangegeven in de nakomeling.
- We gaan ervan uit dat de trie **niet ledig** is.
- **Zoeken.**
  - Test altijd op het karakter aangegeven door de testindex.
  - Als dit leidt naar een nulpointer zit de string niet in de boom.
  - Als we in een blad komen, weten we niet zeker of dat dit de gezochte string is: karakters die niet getest zijn kunnen verschillen.
  - Dus in een blad wordt de zoekstring compleet vergeleken met de string die in het blad zit.
- **Toevoegen.**
  - Het kan zijn dat we een blad moeten toevoegen aan een implicite knoop.
  - We houden een **verschilindex** bij die de eerste plaats aanduidt waar de nieuwe string verschilt van de meest gelijkende string in de trie (deze met de langst gemeenschappelijke prefix).
  - De zoekoperatie eindigt altijd in een explicite knoop. Er zijn dan drie mogelijkheden als de nieuwe string nog niet in de trie zit:
    1. **De expliciete knoop is geen blad**
       (a) **testindex = verschilindex**
       De knoop heeft geen kind voor het karakter in de string aangeduid door de verschilindex. Er kan een blad toegevoegd worden voor de nieuwe string.
       (b) **testindex > verschilindex**
       Er moet een expliciete knoop toegevoegd worden met als testindex de verschilindex. De knoop krijgt twee kinderen: de oude explicite knoop en het nieuwe blad.
    2. **De expliciete knoop is een blad**
       Beschouw een blad als een expliciete knoop met een oneindig grote testindex, dan heb je het vorige geval.

1.6.1 Binaire patriciatrie

- Elke expliciete inwendige knoop heeft twee kinderen.
- _ToDo: idk man wtf is this shit



Figuur 1.5: Een ternaire zoekboom voor de volgende woorden: `Hello, World, Kthulu, Wololo, No, We, He, Worst`. In deze versie hebben de woorden geen afsluitelement. De blauwe knopen stellen het laatste karakter van elk woord voor, dus daar is een sleutel gevonden en daar zit de bijhorende data (getallen in dit geval).

1.7 Ternaire zoekbomen

- Een alternatieve voorstelling van een meerwegstrie.
- ! De snelste implementatie van een meerwegstrie gebruikt een tabel van m kindwijzers in elke knoop, wat onnodig veel geheugen vereist.
- Men gebruikt dan een ternaire zoekboom waarvan elke knoop een **sleutelelement** bevat.
- **Zoeken** vergelijkt telkens het sleutelelement met het element in de huidige knoop. Er zijn dan drie mogelijkheden:
  - Is het zoeksleutelelement kleiner, dan zoeken we verder in de linkse deelboom, met **hetzelfde zoeksleutelelement**.
  - Is het zoeksleutelelement groter, dan zoeken we verder in de rechtse deelboom, met **hetzelfde zoeksleutelelement**.
  - Is het zoeksleutelelement gelijk, dan zoeken we verder in de middelste deelboom, met het **volgende zoeksleutelelement**.
- Om te voorkomen dat een sleutel geen prefix is van elke andere sleutel, wordt er terug een afsluitkarakter gekozen.

- Een zoeksleutel wordt gevonden wanneer we met zijn afsluitelement bij een knoop met datzelfde element uitkomen.
- Een ternaire zoekboom behoudt de volgorde van de opgeslagen sleutels.
- De **voordelen** van een ternaire zoekboom:
  - Het past zich goed aan bij onregelmatig verdeelde zoeksleutels.
    - De Unicode standaard bevat meer dan 1000 karakters, waarvan enkelen heel vaak gebruikt worden. In dit geval zouden meerwegstries ook te veel geheugen nodig hebben voor de tabellen met wijzers.
  - Zoeken naar afwezige sleutels is efficiënt. Er wordt maar vergelijken met slechts enkele sleutelelementen. Een normale binaire boom vereist $\Omega(\lg n)$ sleutelvergelijkingen.
  - Complexe zoekoperaties zijn mogelijk zoals sleutels opsporen die in niet meer dan één element verschillen van de zoeksleutel of zoeken naar sleutels waarvan bepaalde elementen niet gespecifieerd zijn.
- Mogelijke **verbeteringen**:
  - Het aantal knopen kan beperkt worden door een combinatie te maken van een trie en een patriciatrie: enkel sleutels opslaan in bladeren en knopen met maar één kind samenvoegen.
  - De wortel kan vervangen worden door een meerwegstrieknoop, wat resulteert in een tabel van ternaire zoekbomen.

    Als het aantal mogelijke sleutelelementen m niet te groot is, volstaat een tabel van m^2 ternaire zoekbomen, zodat er een zoekboom overeenkomt met elk eerste paar sleutelelementen.

Hoofdstuk 2

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`
  - $t = 25$

2.1 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**.

2.1.1 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

$$\langle \cdots \rangle$$

  waarin $\cdots$ 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":

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

2.1.2 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**
    $$rr* \rightarrow r+$$
  - **Optionele uitdrukking**
    $$r|\epsilon \rightarrow r?$$
  - **Unies van symbolen**
    $$a|b|c \rightarrow [abc]$$
    $$a|b|\ldots|z \rightarrow [a-z]$$
- 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.
-

2.2 Variabele tekst

2.2.1 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	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	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	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	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	A	G	C	A	G	T	A	C	A	G	C	A	C	G
						G	C	A	G	A	G	C	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	A	G	A	G	T	A	C	A	G	C	A	C	G
								G	C	A	G	A	G	C	A	G	A	G	G	T	A	C	A	G	C	A	C	G
									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	T	A	C	A	G	C	A	C	G
											G	C	A	G	A	G	C	A	G	G	T	A	C	A	G	C	A	C	G
												G	C	A	G	A	G	C	A	G	G	T	A	C	A	G	C	A	C	G
													G	C	A	G	A	G	C	A	G	G	T	A	C	A	G	C	A	C	G
														G	C	A	G	A	G	C	A	G	G	T	A	C	A	G	C	A	C	G
															G	C	A	G	A	G	C	A	G	G	T	A	C	A	G	C	A	C	G
																G	C	A	G	A	G	C	A	G	G	T	A	C	A	G	C	A	C	G
																	G	C	A	G	A	G	C	A	G	G	T	A	C	A	G	C	A	C	G
																		G	C	A	G	A	G	C	A	G	G	T	A	C	A	G	C	A	C	G
																			G	C	A	G	A	G	C	A	G	G	T	A	C	A	G	C	A	C	G
																				G	C	A	G	A	G	C	A	G	G	T	A	C	A	G	C	A	C	G
																					G	C	A	G	A	G	C	A	G	G	T	A	C	A	G	C	A	C	G

2.2.2 Knuth-Morris-Pratt

De prefixfunctie

- Gegeven een string P en index i met $i \leq p$.
- De deelstring van P eindigend voor i wordt de prefix van 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.

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

  - De waarden van de prefixfunctie voor $P =$ `GCAGAGCAG` zijn als volgt:

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

- De prefixwaarden worden 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.
  - Dit komt neer op het berekenen van de prefixfunctie van T, maar de karakters nog steeds vergelijken met P.
- Bepaal de prefixfunctie van deze nieuwe string, in $\Theta(n+p)$.
- Als $q(j) = p$ voor $0 < j \leq t$, dan werd P gevonden beginnend bij index $j - p$ in T.

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
q(j)	-	1	2	3	0	0	1	2	3	4	5	6	7	8	9	0	1	0	0	0	0	1	2	3	0

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 2.1: 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 2.2: Een eerste versie van het Knuth-Morris-Prattalgoritme, waarbij $s = i - q(i)$.

- In tabel 2.2 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	0	0	0	4
s	1	1	2	3	3	5	6	7	8	5

Tabel 2.3: Het patroon $P =$ `GCAGAGCAG`, de nieuwe prefixfunctie $q'(i)$ en de nieuwe s-waarden.

2.2.3 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.

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

- 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 2.5).

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

Tabel 2.5: 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.

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

- 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.
   - 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 2.7: 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 in P is dan $i + 1 - k$. Als k niet gedefinieerd is dan is $v[i] = p - s[0]$.

- 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 $v[i] = p - s[0]$.

i	-1	0	1	2	3	4	5	6	7	8
P	-	G	C	A	G	A	G	C	A	G
$s(i)$	/	4	3	2	1	2	1	0	0	0
$p - i - 1$	/	8	7	6	5	4	3	2	1	0
k	/	-	-	-	-	-	1	4	5	8
$i + 1 - k$	/	-	-	-	-	-	5	3	3	1
$v[i]$	5	5	5	5	5	5	5	3	3	1

Tabel 2.8: De waarden voor k worden als volgt berekent: zoek de grootste i zodanig dat $s(i) = p - i - 1$. De gekleurde cijfers in de tabel toont voor elke k de relatie met index i en $s(i) = p - i - 1$.

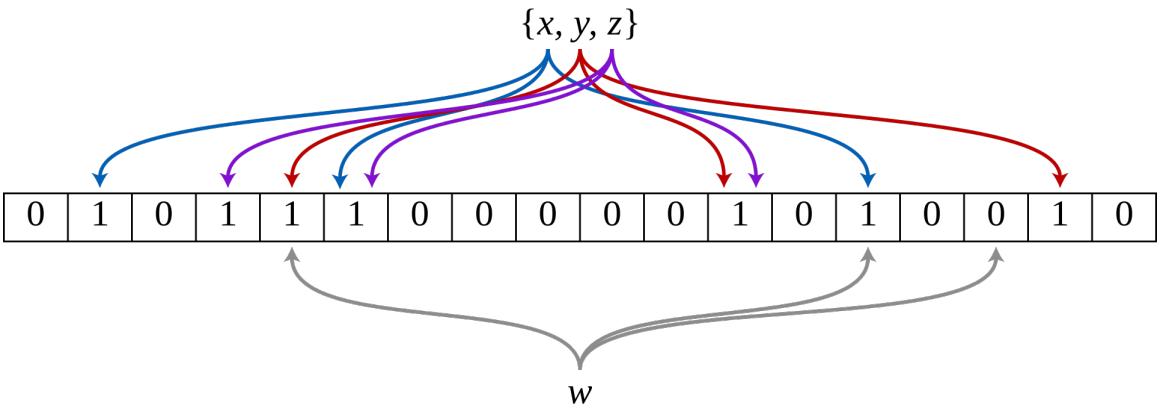
	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
	T	G	C	A	T	C	G	C	A	G	A	G	C	A	G	A	G	T	A	C	A	G	C	A	C	G
i	$v[i]$																									
4	5	G	C	A	G	A	G	C	A	G																
-1	5						G	C	A	G	A	G	C	A	G											
8	1											G	C	A	G	A	G	C	A	G						
8	1												G	C	A	G	A	G	C	A	G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

Tabel 2.9: Het Boyer-Moore algoritme, enkel gebruik makend van de heuristiek van het juiste suffix.

2.2.4 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 2.1).
   - We willen een verzameling van objecten in gehashte vorm bijhouden.



Figuur 2.1: 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.

    - 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.

2.2.5 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 (in de voorbeelden is $r = 29$) 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.

  - Op deze manier wordt $H_r(P)$ als volgt gedefinieerd:

$$H_r(P) = \sum_{i=0}^{p-1} P[i]d^{p-i-1} \bmod r$$

  ! In de voorbeelden wordt deze eigenschap niet gebruikt om het overzichtelijk te houden.

- Omdat elk tussenresultaat nu binnen een processorwoord past, is $H_r(P)$ berekenen slechts $\Theta(p)$.
- Voor het alfabet $\Sigma = \{A, C, G, T\}$ gelden volgende waarden voor de stringelementen:

A	1
C	2
G	3
T	4

$$\begin{aligned} H(P) &= \sum_{i=0}^{8} P[i] \cdot 4^{8-i} \\ &= G \cdot 4^8 + C \cdot 4^7 + A \cdot 4^6 + G \cdot 4^5 + A \cdot 4^4 + G \cdot 4^3 + C \cdot 4^2 + A \cdot 4^1 + G \cdot 4^0 \\ &= 3 \cdot 4^8 + 2 \cdot 4^7 + 1 \cdot 4^6 + 3 \cdot 4^5 + 1 \cdot 4^4 + 3 \cdot 4^3 + 2 \cdot 4^2 + 1 \cdot 4^1 + 3 \cdot 4^0 \\ &= 237031 \\ H_r(P) &= 237031 \bmod 29 \\ &= 14 \end{aligned}$$

- 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:

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

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

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

(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)

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
$H(T_j)$	238311	166813	142967	309726	190329	237031	161693	122487	227808	124801	237062	161817	122983	229790	132729	268774	26523	/	/	/	/	/	/	/	/
$H_r(T_j)$	18	5	26	6	2	14	18	20	16	14	16	26	23	23	25	2	17	/	/	/	/	/	/	/	/
$H(T_0)$	$3 \cdot 4^8 + 2 \cdot 4^7 + 1 \cdot 4^6 + 4 \cdot 4^5 + 2 \cdot 4^4 + 3 \cdot 4^3 + 2 \cdot 4^2 + 1 \cdot 4^1 + 3 \cdot 4^0 = 238311$																								
$H(T_1)$	$(H(T_0) - T[0] \cdot 4^8) \cdot 4 + T[9] = (238311 - 3 \cdot 4^8) \cdot 4 + 1 = 166813$																								
$H(T_2)$	$(H(T_1) - T[1] \cdot 4^8) \cdot 4 + T[10] = (166813 - 2 \cdot 4^8) \cdot 4 + 3 = 142967$																								
$H(T_3)$	$(H(T_2) - T[2] \cdot 4^8) \cdot 4 + T[11] = (142967 - 1 \cdot 4^8) \cdot 4 + 2 = 309726$																								
$H(T_4)$	$(H(T_3) - T[3] \cdot 4^8) \cdot 4 + T[12] = (309726 - 4 \cdot 4^8) \cdot 4 + 1 = 190329$																								
$H(T_5)$	$(H(T_4) - T[4] \cdot 4^8) \cdot 4 + T[13] = (190329 - 2 \cdot 4^8) \cdot 4 + 3 = 237031$																								
$H(T_j)$											...														
P	G	C	A	G	A	G	C	A	G																
$T[5 \cdots 13]$	G	C	A	G	A	G	C	A	G																
$T[9 \cdots 17]$	A	G	C	A	G	A	G	T	A																

Tabel 2.10: Het Karp-Rabinalgoritme.

- 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.

2.2.6 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 $P_G(\{s_0\}, F)$

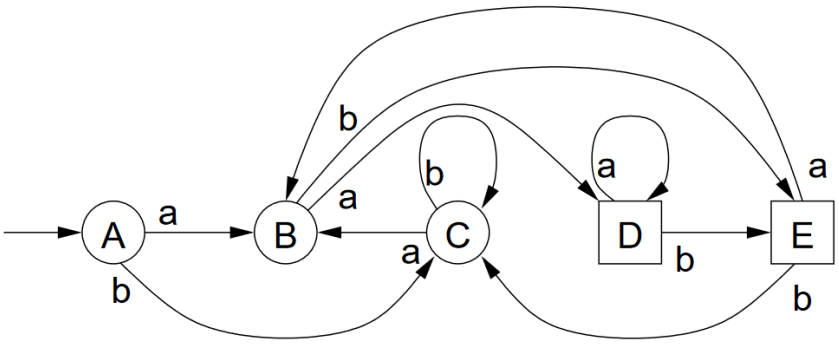
Niet-deterministische automaten

- 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 2.2 toont de DA geconstrueerd uit de NA van figuur 2.5.

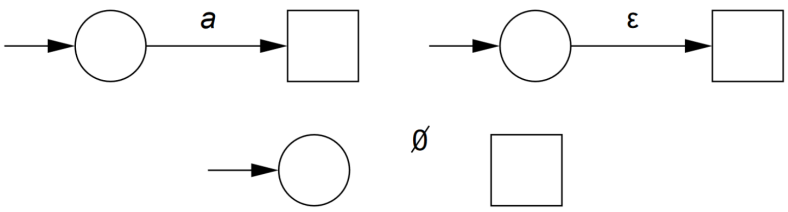


		a	b
$\rightarrow$	$A = \{1, 2, 3, 4, 8\}$	B	C
	$B = \{2, 3, 4, 5, 7, 8, 9, 10, 11\}$	D	E
	$C = \{2, 3, 4, 6, 7, 8\}$	B	C
$\leftarrow$	$D = \{2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 14\}$	D	E
$\leftarrow$	$E = \{2, 3, 4, 6, 7, 8, 13, 14\}$	B	C

Figuur 2.2: Deterministische automaat geconstrueerd uit de NA van figuur 2.5.

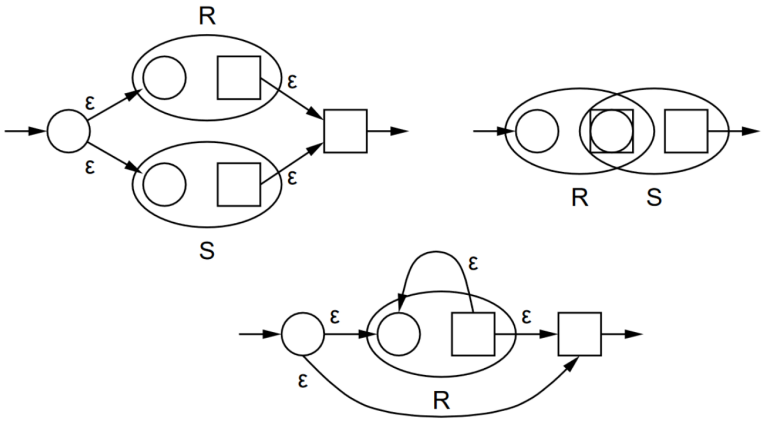
Automaten voor regexps

- Een reguliere taal kan herkend worden door een DA of NA.
- Een NA kan opgebouwd worden vanuit een regexp door de **constructie van Thompson**.



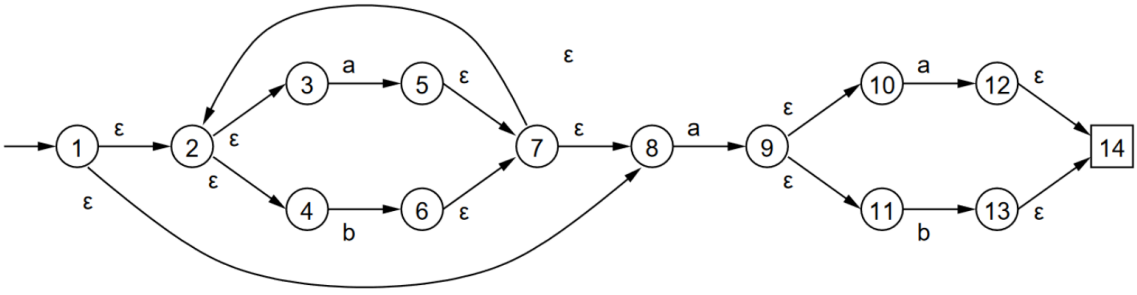
Figuur 2.3: Constructie van Thompson: basiselementen.

- Er worden NA's gedefinieerd voor basiselementen van een regexp: elk element uit Σ en ϵ (figuur 2.3).
- Deze NA's kunnen samengesteld worden voor elk van de drie basisoperatoren: unie, concatenatie en Kleenesluiting (figuur 2.4).



Figuur 2.4: 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 2.5 toont de NA bekomen met deze constructie voor de regexp $(a|b) * a(a|b)$.



Figuur 2.5: 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

2.2.7 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 2.11). Een bit i van woord $S[s]$ is waar als karakter s op plaats i in P voorkomt.

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

Tabel 2.11: 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.
  - 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:

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

R_j		R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}	R_{15}	R_{16}	R_{17}	R_{18}	R_{19}	R_{20}	R_{21}	R_{22}	R_{23}	R_{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
$R[0]$	G	1	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1	0	0	0	1
$R[1]$	C	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0
$R[2]$	A	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
$R[3]$	G	0	0	0	0	0	0	0	0	1	0	0	0	0	1	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	1	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	1	0	0	0	0	0	0	0	0	0
$R[6]$	C	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
$R[7]$	A	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
$R[8]$	G	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

Tabel 2.12: De opeenvolgende tabellen R_j. Er moet slechts één tabel in het geheugen bijgehouden worden, en dat is die op het huidige karakterpositie j.

- 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.

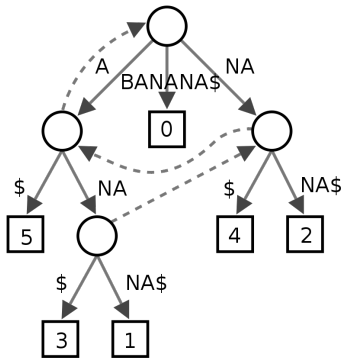
$$R_{j+1}^1 = \text{Schuif}(R_j^0) \text{ OF } (\text{Schuif}(R_j) \text{ EN } S[T[j+1]])$$

Hoofdstuk 3

Indexeren van vaste tekst

- Sommige zoekoperaties gebeuren op een vaste tekst T waarin frequent gezocht wordt naar een veranderlijk patroon P.
- Voorbereidend werk op de tekst om efficiënter te doorzoeken.
- Alle zoekmethoden in hoofdstuk 2 verrichten voorbereidend werk op het patroon.
  - In het slechtste geval is dit $O(t + p)$.
  - Dit kan gereduceerd worden tot $O(p)$ door eerst $O(t)$ voorbereidend werk te doen op T via **suffixen**.
  - Als een patroon in de tekst voorkomt, moet het een prefix zijn van één van de suffixen.
  - Een suffix dat begint op lokatie i wordt aangeduidt met suff_i.

3.1 Suffixbomen



Figuur 3.1: Een suffixboom voor de string `BANANA$`. Elk van de suffixen `BANANA$`, `ANANA$`, `NANA$`, `ANA$`, `NA$` en `A$` kan gevonden worden in deze boom. Het suffix `NANA$` wordt gevonden door twee keer de rechterdeelboom te nemen vanuit de wortel. De index 2 wijst erop dat de suffix begint bij $T[2]$. De gestreepte verbindingen zijn staartpointers.

- Een **gewijzigde patriciatrie**:
  1. In de bladeren wordt enkel de index i van suff_i opgeslagen, en niet het suffix zelf.

2. In elke knoop wordt er een begin- en eindindex opgenomen, in plaats van een testindex.
3. Elke inwendige knoop kan een staartpointer opnemen die de opbouw van de suffixboom eenvoudiger maakt.
   - De staart van een string s is **staart(s)** en wordt bekomen door het eerste karakter van de string te verwijderen.
   - De staart van een suffix is zelf ook een suffix.

- In de trie is er een expliciete inwendige knoop horend bij de niet-lege string α, als de trie twee strings $\alpha\beta$ en $\alpha\gamma$ bevat zodat de eerste letter van β verschilt van de eerste letter van γ.
- **Twee eenvoudige toepassingen** van een suffixboom:
  1. **Zoeken van een patroon P in een tekst T.**
     - Constureer de suffixboom voor T.
     - Zoek de knoop die overeenkomt met P.
       - Als deze niet bestaat zit P niet in T.
       - Als deze wel bestaat, zitten de gezochte beginposities in T bij alle bladeren die opvolger zijn van de gevonden knoop.
  2. **De langste gemeenschappelijke deelstring.**
     - Er is een verzameling van k verschillende strings $S = \{s_1, s_2, \cdots, s_k\}$ met totale lengte t.
     - De langste gemeenschappelijke deelstring van van al die strings wordt gezocht.
     - Er wordt een **veralgemeende suffixboom** opgesteld:
       - Elk blad bevat de beginpositie van het suffix, en ook tot welke strings ze behoort.
       - Elk blad moet ook de begin- en eindindex opslagen, per eventuele string.
     - Elke inwendige knoop komt overeen met een prefix van een suffix, en deze deelstring komt voor in elk string die vermeld wordt bij een blad dat opvolger is van die knoop.
     - De boom wordt overlopen om de lengte van al de prefixen en het aantal verschillende strings te bepalen waarin ze voorkomen, en dus ook het langste prefix dat in alle strings voorkomt.

3.2 Suffixtabellen

- We veronderstellen dat het vroeger gebruikte patroon P (`GCAGAGCAG`) nu de tekst is, om voorbeelden in te korten.
- Een tabel met de gerangschikte suffixen van een string.
- De elementen van de tabel zijn indices, die de startpositie van het suffix aanduiden van de string.

i	0	1	2	3	4	5	6	7	8
$T[i]$	G	C	A	G	A	G	C	A	G
$SA[i]$	7	2	4	6	1	8	3	5	0
$T[SA[i]]$	A	A	A	C	C	G	G	G	G

Tabel 3.1: De suffixtabel SA voor `GCAGAGCAG`.

- Deze tabel wordt bekomen door de suffixboom in inorder te overlopen.
  - Andere, meer efficiënte, methoden bestaan, maar worden niet in de cursus besproken.

- Om een suffixtabel te gebruiken is er een hulpstructuur nodig, de **LGP**-tabel (Langste Gemeenschappelijke Prefix), die dan wordt omgezet in de **LCP**-tabel die gerangschikt is in de volgorde gegeven door de suffixtabel.
- Tabel 3.2 toont de constructie van de LCP-tabel.
  - Overloop i, $0 \leq i \leq t$.
  - Zoek j zodanig dat $SA[j] = i$.
  - De opvolger van $suff_i$ is dan $suff_{SA[j+1]}$.
  - Vergelijk $suff_i$ met $suff_{SA[j+1]}$ vanaf beginpositie $P[i+l]$.
    - Initieel is $l = 0$.
    - Verhoog l totdat $T[i+l] \neq T[SA[j+1]+l]$.
    - Als $l > 0$, dan moet voor $i+1$ slechts vergeleken worden vanaf $T[i+l-1]$.

			i	0	1	2	3	4	5	6	7	8
			$T[i]$	G	C	A	G	A	G	C	A	G
			$SA[i]$	7	2	4	6	1	8	3	5	0
			$T[SA[i]]$	A	A	A	C	C	G	G	G	G
i	**opvolger**	**LGP[i]**	1	0	1	2	3	4	5	6	7	8
0	-	0	/									
1	8	0	$suff_1$	C	A	G	A	G	C	A	G	
			$suff_8$	G								
2	4	2	$suff_2$	A	G	A	G	C	A	G		
			$suff_4$	A	G	C	A	G				
3	5	1	$suff_3$	G	A	G	C	A	G			
			$suff_5$	G	C	A	G					
4	6	0	$suff_4$	A	G	C	A	G				
			$suff_6$	C	A	G						
5	0	4	$suff_5$	G	C	A	G					
			$suff_0$	G	C	A	G	A	G	C	A	G
6	1	3	$suff_6$	C	A	G						
			$suff_1$	C	A	G	A	G	C	A	G	
7	2	2	$suff_7$	A	G							
			$suff_2$	A	G	A	G	C	A	G		
8	3	1	$suff_8$	G								
			$suff_3$	G	A	G	C	A	G			
			$LGP[i]$	0	0	2	1	0	4	3	2	1
			$LCP[i]$	4	0	0	1	2	0	3	1	2

Tabel 3.2: De constructie van de LCP-tabel voor `GCAGAGCAG`. Karakters die rood of groen zijn worden effectief vergeleken. Oranje karakters moeten niet meer vergeleken worden.

- Een patroon P opzoeken in de suffixtabel gebeurt met **binair zoeken**.

3.3 Tekstzoekmachines

3.3.1 Inleiding

- Tekstzoekmachines zijn in eerste instantie gelijkaardig aan databanksystemen.
  - Documenten worden bewaard in een repository.

  - Er worden indexen bijgehouden om snel documenten te doorlopen.
  - Er kunnen queries uitgevoerd worden relevante documenten te zoeken.
- Maar ze verschillen ook van databanksystemen.
  - Een query voor een tekstzoekmachine bestaat enkel uit woorden of zinnen.
  - In een databanksysteem zal de query resultaten geven die voldoen aan een logische uitspraak, maar bij een tekstzoekmachine is dit vager.
  - Een tekstzoekmachine geeft niet alle resultaten terug, maar enkel de meest relevante. Het begrip relevantie is ook niet exact, aangezien dit afhangt van de gebruiker.
- Het gebruik van **indices** om tekst te indexeren is onmisbaar.

3.3.2 Zoeken van tekst en informatie verzamelen

Queries

- In een traditionele databank hebben gegevens een unieke sleutel, wat niet het geval is bij tekstdocumenten op het internet.
- Soms hebben tekstdocumenten *metadata* zoals de auteur, het onderwerp en het aantal pagina's, maar deze zijn slechts occasioneel nuttig.
- De meest voorkomende manier om in tekst te zoeken is het zoeken naar **inhoud** aan de hand van een **query**.
- Aangezien dat een tekstzoekmachine probeert relevante documenten weer te geven, moet gemeten kunnen worden hoe goed deze documenten zijn.
- Een tekstzoekmachine heeft een bepaalde **effectiveness** voor een getal r waarbij de meeste van de eerste r resultaten relevant zijn.
  - De *effectiveness* wordt vaak bepaald door de **precision** en **recall**.
  - De *precision* is de verhouding van documenten dat relevant zijn.
  - De *recall* is de verhouding van relevante documenten die gekozen zijn.
  - Voorbeeld:
    - Een tekstdatabank bevat 20 documenten.
    - Een gebruiker zoekt in deze databank met een query en er worden 8 resultaten teruggegeven.
    - De gebruiker vindt dat 5 van deze resultaten relevant zijn voor hem, en dat er nog 2 andere documenten in de tekstdatabank zitten die niet door de tekstzoekmachine gegeven worden.
    - De textitprecision is 5/8.
    - De *recall* is 5/7.
  - Veel van de technieken zorgen ervoor dat *effectiveness* vrij hoog blijft.

Voorbeelddatabanken

- De **Keeper databank**.

  1 The old night keeper keeps the keep in the town.
  2 In the big old house in the bog old gown.
  3 The house in the town had the big old keep.
  4 Where the old night keeper never did sleep.
  5 The night keeper keeps the keep in the night.
  6 And keeps in the dark and sleeps in the night.

  - Bevat 6 documenten elk met 1 lijn.
  - Verschillende eenvoudige technieken om in deze databank te zoeken.
    - De query `big old house` waarbij de query als één enkele string beschouwd wordt zal enkel document 2 geven.
    - De query `big old house` waarbij elk woord in een verzameling van woorden komt (**bag-of-word**, {`big`, `old`, `house`}) zal documenten 2 en 3 teruggeven. De volgorde van de woorden in deze verzameling spelen geen rol en elk woord wordt afzonderlijk bekeken of ze voorkomt in het document of niet.
  - Meerdere technieken om de **woordenschat** van een tekstdatabank te reduceren:
    - **Zonder aanpassingen**
      `And and big dark did gown had house In in keep keeper keeps light never night old sleep sleeps The the town Where`
    - **Hoofdletter-invariantie**
      `and big dark did gown had house in keep keeper keeps light never night old sleep sleeps the town where`
    - **Verwijderen meerdere varianten van hetzelfde woord**
      `and big dark did gown had house in keep light never night old sleep the town where`
    - **Verwijderen van vaak voorkomende woorden**
      `big dark did gown house keep light night old sleep town`

- Twee hypothetische databanken om efficiëntie te bespreken:

	NewsWire	Web
Grootte in gigabytes	1	100
Aantal Documenten	400 000	12 000 000
Aantal woorden	180 000 000	11 000 000 000
Aantal unieke woorden	400 000	16 000 000
Aantal unieke woorden per document, opgesomd	70 000 000	3 500 000 000

- Elke tekstzoekmachine moet aan een aantal voorwaarden voldoen:
  - De queries moeten goed geanalyseerd worden.
  - De queries moeten snel geanalyseerd worden.
  - Minimaal gebruik van resources zoals geheugen en bandbreedte.
  - Schaalbaar naar grote volumes van data.
  - Resistent tegen het wijzigen van documenten.

Gelijkaardigheidsfuncties

- Elke tekstzoekmachine maakt gebruik van een rankingsysteem om documenten te ordenen.
- Om documenten te ordenen wordt er gebruik gemaakt van een gelijkaardigheidsfunctie.
- Hoe hoger de waarde van deze functie, hoe hoger de kans dat de gebruiker dit document als relevant zal beschouwen.
- De r meest relevante documenten worden dan gegeven aan de gebruiker.
- In **bag-of-words** queries wordt de gelijkaardigheidsfunctie samengesteld door een aantal statistische variabelen:
  - $f_{d,t}$ is de frequentie van het woord t in document d.
  - $f_{q,t}$ is de frequentie van het woord t in de query q.
  - f_t is het aantal documenten dat één of meer keer het woord t bevat.
  - F_t is het aantal keer dat t voorkomt in de hele tekstdatabank.
  - N is het aantal documenten in de tekstdatabank.
  - n het aantal geïndexeerde woorden in de tekstdatabank.
- Deze waarden kunnen gecombineerd worden om drie vaststellingen te maken:
  1. Een woord dat in veel documenten voorkomt krijgt een kleiner gewicht.
  2. Een woord dat veel in één document voorkomt krijgt een groter gewicht.
  3. Een document dat veel woorden bevat krijgt een kleiner gewicht.
- Er is een **query vector** $\vec{w}_q$ en een **document vector** $\vec{w}_d$, waarbij elk component in deze vector gedefinieerd wordt als

$$w_{q,t} = \ln\left(\frac{N}{f_t}\right) \qquad w_{d,t} = f_{d,t}$$

- De maat van gelijkheid $S_{q,d}$, de maat in hoeverre het document d relevant is voor query q, kan bekomen worden door de cosinus van de hoek tussen deze twee vectoren te nemen.

$$S_{q,d} = \frac{\vec{w}_d \cdot \vec{w}_q}{||\vec{w}_d|| \cdot ||\vec{w}_q||} = \frac{\sum_t w_{d,t} \cdot w_{q,t}}{\sqrt{\sum_t w_{d,t}^2} \cdot \sqrt{\sum_t w_{q,t}^2}}$$

- De grootheid $w_{q,t}$ encodeert de **inverse document frequentie** van een woord t.
- De grootheid $w_{d,t}$ encodeert de **woord frequentie** van een woord t.
- Het nadeel aan deze methode is dat elk document in beschouwing genomen moet worden, maar dat slechts r documenten gevonden moeten worden.
- Voor de meeste documenten is de gelijkaardigheidswaarden insignificant.
- Deze **brute-force** methode kan uitgebreidt worden tot betere methoden, via **indices**.

3.3.3 Indexeren en query-evaluatie

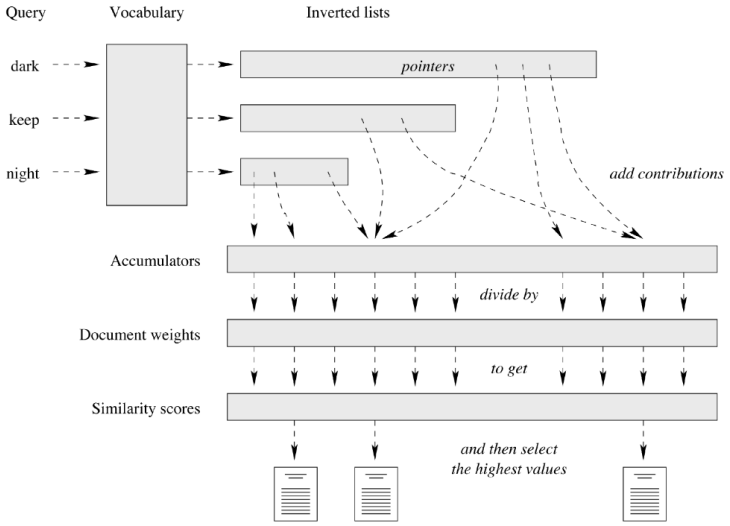
- Een **index** in deze context is een datastructuur dat een woord afbeeldt op documenten dat dit woord bevat.
- Het verwerken van een query kan dan enkel uitgevoerd worden op documenten die minstens één van de query woorden bevat.
- Er zijn vele soorten indices, maar de meest gebruikte is een **inverted file index**: een collectie van lijsten, één per woord, dat documenten bevat dat dit woord bevat.
- Een **normale inverted file index** bestaat uit twee componenten.
  1. Voor elk woord t houdt de **zoekstructuur** het volgende bij:
     - een getal f_t van het aantal documenten dat t bevat, en
     - een pointer naar de start van de correspondeerde geïnverteerde lijst.
  2. Een **verzameling van geïnverteerde lijsten**, waarbij elk lijst het volgende bijhoudt voor een woord t:
     - de sleutels van documenten d die t bevatten, en
     - de verzameling van frequencies $f_{d,t}$ van woorden t in document d.
     - $\rightarrow \langle d, f_{d,t} \rangle$ paren.
- Samen met W_d en deze twee componenten zijn geordende queries mogelijk.
- Een *inverted file* voor de *keeper database* is te zien op tabel 3.3.

woord t	f_t	Geïnverteerde lijst voor t
`and`	1	$\langle 6,2 \rangle$
`big`	2	$\langle 2,2 \rangle$ $\langle 3,1 \rangle$
`dark`	1	$\langle 6,1 \rangle$
`did`	1	$\langle 4,1 \rangle$
`gown`	1	$\langle 2,1 \rangle$
`had`	1	$\langle 3,1 \rangle$
`house`	2	$\langle 2,1 \rangle$ $\langle 3,1 \rangle$
`in`	5	$\langle 1,1 \rangle$ $\langle 2,2 \rangle$ $\langle 3,1 \rangle$ $\langle 5,1 \rangle$ $\langle 6,2 \rangle$
`keep`	3	$\langle 1,1 \rangle$ $\langle 3,1 \rangle$ $\langle 5,1 \rangle$
`keeper`	3	$\langle 1,1 \rangle$ $\langle 4,1 \rangle$ $\langle 5,1 \rangle$
`keeps`	3	$\langle 1,1 \rangle$ $\langle 5,1 \rangle$ $\langle 6,1 \rangle$
`light`	1	$\langle 6,1 \rangle$
`never`	1	$\langle 4,1 \rangle$
`night`	3	$\langle 1,1 \rangle$ $\langle 4,1 \rangle$ $\langle 5,1 \rangle$
`old`	4	$\langle 1,1 \rangle$ $\langle 2,2 \rangle$ $\langle 3,1 \rangle$ $\langle 4,1 \rangle$
`sleep`	1	$\langle 4,1 \rangle$
`sleeps`	1	$\langle 6,1 \rangle$
`the`	6	$\langle 1,3 \rangle$ $\langle 2,2 \rangle$ $\langle 3,3 \rangle$ $\langle 4,1 \rangle$ $\langle 5,3 \rangle$ $\langle 6,2 \rangle$
`town`	2	$\langle 1,1 \rangle$ $\langle 3,1 \rangle$
`where`	1	$\langle 4,1 \rangle$

d	1	2	3	4	5	6
W_d	4	4.2	4	2.8	4.1	4

Tabel 3.3: Een op document niveau geïnverteerd bestand voor de *Keeper* databank. Elk woord t bestaat uit f_t en een lijst van paren, waarbij elk paar bestaat uit een sleutel d van een document en de frequentie $f_{d,t}$ van het woord t in d. Ook zijn de waarden van W_d te zien, berekend volgens $W_d = \sqrt{\sum_t w_{d,t}^2} = \sqrt{\sum_t f_{d,t}^2}$.

- Er kan nu een **query evaluatie** algoritme opgesteld worden (gevisualiseerd op figuur 3.2).



Figuur 3.2: Het gebruik van een geïnverteerd bestand en een verzameling van accumulators om gelijkaardigheidswaarden te berekenen.

1. Er wordt een accumulator A_d bijgehouden voor elk document d. Initieel is elke $A_d = 0$.
2. Voor elk woord t in de query worden volgende operaties uitgevoerd:
   (a) Bereken $w_{q,t} = \ln\left(\frac{N}{f_t}\right)$ en vraag de geïnverteerde lijst op van t.
   (b) Voor elk paar $\langle d, f_{d,t}\rangle$ in de geïnverteerde lijst worden volgende operaties uitgevoerd:
      i. Bereken $w_{d,t}$.
      ii. Stel $A_d = A_d + w_{q,t}w_{d,t}$.
3. Voor elke $A_d > 0$, stel $S_d = A_d/W_d$.
4. Identificeer de r grootste S_d waarden en geef de correspondeerde documenten terug.

- Het is ook nog mogelijk om **de posities van de woorden in het document te indexeren**.
  - Het paar $\langle d, f_{d,t}\rangle$ kan uitgebreidt worden om de posities p bij te houden waar dat t voorkomt in d.

$$\langle d, f_{d,t}, p_1, \cdots, p_{f_{d,t}}\rangle$$

3.3.4 Queries met zinnen

- Een query kan een expliciete zin bevatten, aangeduid met aanhalingstekens, zoals `"philip glass"` of `"the great flydini"`.
- Soms is het ook impliciet zoals `Albert Einstein` of `San Francisco hotel`.
- _ToDo: idk

3.3.5 Constructie van een index

- Het volume van de data is veel te groot om alles in het geheugen te doen.
- Er zijn drie methoden:

1. **In-memory Inversion**
   - Alle documenten wordt tweemaal overlopen.
     (a) Een eerste keer telt de frequentie f_t van alle verschillende woorden van alle documenten.
     (b) Een tweede maal plaatst de pointers in de juiste positie.
2. **Sort-Based Inversion**
3. **Merge-Based Inversion**