FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Gevorderde algoritmen

Bert De Saffel

Master in de Industriële Wetenschappen: Informatica Academiejaar 2018–2019

Gecompileerd op 21 september 2019

UNIVERSITEIT
GENT

Inhoudsopgave

Deel I

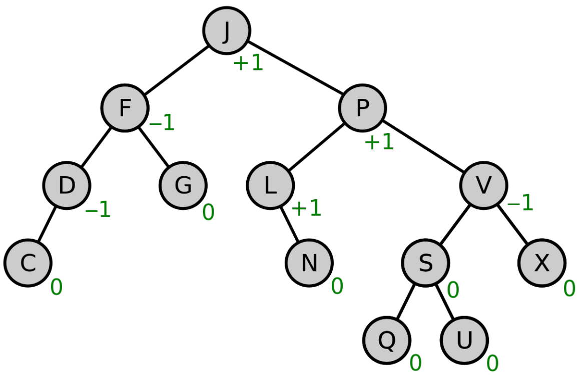
Gegevensstructuren II

Hoofdstuk 1

Efficiënte zoekbomen

1.1 Inleiding

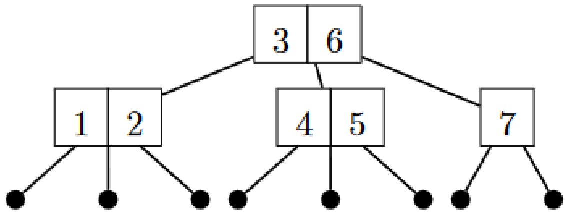
- Uitvoeringstijd van operaties (zoeken, toevoegen, verwijderen) op een binaire zoekboom met hoogte h is $O(h)$.
- De hoogte h is afhankelijk van de toevoegvolgorde:
  - In het slechtste geval bekomt men een gelinkte lijst, zodat $h = O(n)$.
  - Als elke toevoegvolgorde even waarschijnlijk is, dan is de verwachtingswaarde voor de hoogte $h = O(\lg n)$ met n het aantal gegevens.
    - ! Geen realistische veronderstelling.
- Drie manieren om de efficiëntie van zoekbomen te verbeteren:
  1. **Elke operatie steeds efficiënt maken.**
     (a) AVL-bomen (figuur 1.1).



Figuur 1.1: Een AVL-boom. De groene cijfers stellen de hoogteverschillen voor van de twee deelbomen voor elke knoop.

  - Hoogteverschil Δh van de twee deelbomen van elke knoop is kleiner of gelijk aan 1.
  - Δh wordt opgeslagen in de knoop zelf.

(b) 2-3-bomen (figuur 1.2).



Figuur 1.2: Een 2-3-boom.

- Elke knoop heeft 2 of 3 kinderen.
- Elk blad heeft dezelfde diepte.
- Bij toevoegen of verwijderen wordt het ideale evenwicht behouden door het aantal kinderen van de knopen te manipuleren.

(c) 2-3-4-bomen.

- Analoog aan een 2-3-boom, maar elke knoop heeft 2, 3 of 4 kinderen.

(d) Rood-zwarte bomen (sectie 1.2.1).

2. **Elke reeks operaties steeds efficiënt maken.**

(a) Splaybomen (sectie 1.3).

- De vorm van de boom wordt meermaals aangepast.
- Elke reeks opeenvolgende operaties is gegarandeerd efficiënt.
- Een individuele operatie kan wel traag uitvallen.
- *Geamortiseerd* is de performantie per operatie goed.

3. **De gemiddelde efficiëntie onafhankelijk maken van de operatievolgorde.**

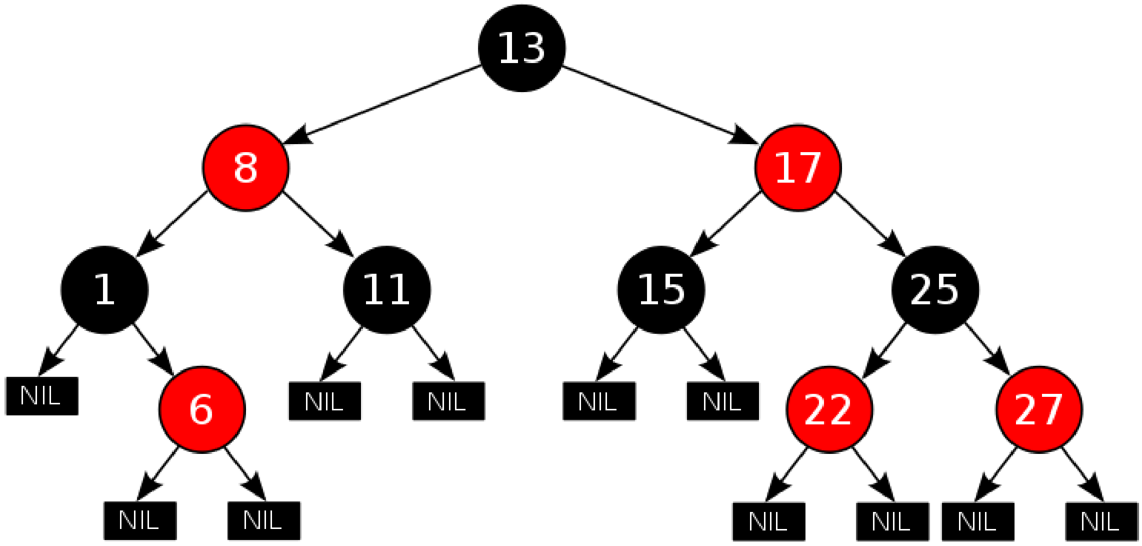
(a) Gerandomiseerde zoekbomen (sectie 1.4).

- Gebruik van een random generator.
- De boom is random, onafhankelijk van de toevoeg- en verwijdervolgorde.
- Verwachtingswaarde van de hoogte h wordt dan $O(\lg n)$.

1.2 Rood-zwarte bomen

1.2.1 Definitie en eigenschappen

- **Definitie:**
  - Een binaire zoekboom.
  - Elke knoop is rood of zwart gekleurd.
  - Elke virtuele knoop is zwart. Een virtuele knoop is een knoop die geen waarde heeft, maar wel een kleur. Deze vervangen de nullwijzers.
  - Een rode knoop heeft steeds twee zwarte kinderen
  - Elke mogelijke weg vanuit een knoop naar een virtuele knoop bevat evenveel zwarte knopen. Dit aantal wordt de *zwarte hoogte* genoemd
  - De wortel is zwart.
- **Eigenschappen:**
  - Een deelboom met wortel w en zwarte hoogte z heeft tenminste $2^z - 1$ inwendige knopen.



Figuur 1.3: Een rood-zwarte boom. De `NIL` knopen stellen virtuele knopen voor.

- De hoogte van een rood-zwarte boom met n knopen is steeds $O(\lg n)$.
  - Er zijn nooit twee opeenvolgende rode knopen op elke weg vanuit een knoop naar een virtuele knoop $\rightarrow z \geq h/2$.
  - Substitutie in de eerste eigenschap geeft:

$$
\begin{aligned}
n &\geq 2^z - 1 \geq 2^{h/2} - 1 \\
\rightarrow h &\leq 2\lg(n+1)
\end{aligned}
$$

1.2.2 Zoeken

- De kleur speelt geen rol, zodat de rood-zwarte boom een gewone binaire zoekboom wordt.
- De hoogte van een rood-zwarte boom is wel gegerandeerd $O(\lg n)$.
- Zoeken naar een willekeurige sleutel is dus $O(\lg n)$.

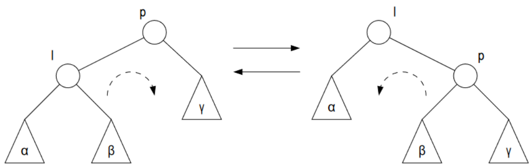
1.2.3 Toevoegen en verwijderen

- Element toevoegen of verwijderen, zonder rekening te houden met de kleur, is ook $O(\lg n)$.

! Geen garantie dat deze gewijzigde boom nog rood-zwart zal zijn.

- Twee manieren om toe te voegen:
  1. **Bottom-up:**
     - Voeg knoop toe zonder rekening te houden met de kleur.
     - Herstel de rood-zwarte boom, te beginnen bij de nieuwe knoop, en desnoods tot bij de wortel.
  2. **Top-down:**
     - Pas de boom aan langs de dalende zoekweg.
     - ✓ Efficiënter dan bottom-down aangezien er geen ouderwijzers of een stapel nodig is.

1.2.4 Rotaties



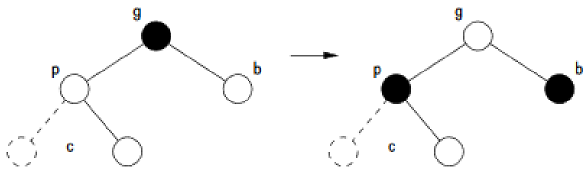
Figuur 1.4: Rotaties

- Wijzigen de vorm van de boom, maar behouden de inorder volgorde van de sleutels.
- Moet enkel pointers aanpassen, en is dus $O(1)$.
- **Rechtste rotatie** van een ouder p en zijn linkerkind l:
  - Het rechterkind van l wordt het linkerkind van p.
  - De ouder van p wordt de ouder van l.
  - p wordt het rechterkind van l.
- **Linkse rotatie** van een ouder p en zijn rechterkind r:
  - Het linkerkind van r wordt het rechterkind van p.
  - De ouder van r wordt de ouder van p.
  - p wordt het linkerkind van l.

1.2.5 Bottom-up rood-zwarte bomen

Toevoegen

- De knoop wordt eerst op de gewone manier toegevoegd.
- Welke kleur geven we die knoop?
  - **Zwart:** dit kan de zwarte hoogte van veel knopen ontregelen.
  - **Rood:** dit mag enkel als de ouder zwart is.
  - Kies voor rood omdat zwarte hoogte moeilijker te herstellen valt.
- Als de ouder zwart is, dan is toevoegen gelukt.
- Als de ouder rood is wordt deze storing verwijderd door rotaties en kleurwijzigingen door te voeren.
- Vaststellingen:
  - De ouder p van de nieuwe knoop c is rood.
  - De ouder g van p, is zwart.
- Er zijn zes mogelijke gevallen, die twee groepen van drie vormen, naar gelang dat p een linker- of rechterkind is van g.
- We onderstellen dat p een linkerkind is van g.
  1. **De broer b van p is rood.**

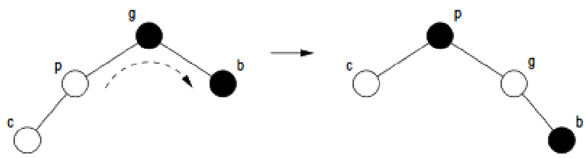


Figuur 1.5: Rode broer.

- ◦ Maak p en b zwart.
- ◦ Maak g rood.
- ◦ Als g een zwarte ouder heeft, is het probleem opgelost.
- ◦ Als g een rode ouder heeft, zijn er opnieuw twee opeenvolgende rode knopen.
- ◦ Het probleem wordt opgeschoven in de richting van de wortel.

2. **De broer p van p is zwart.**

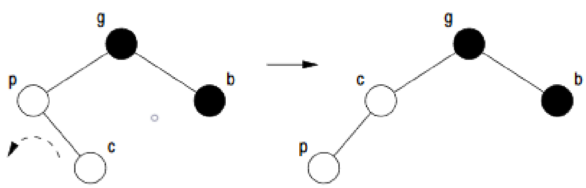
   (a) **Knoop c is een linkerkind van p.**



Figuur 1.6: Rode broer.

- ◦ Roteer p en g naar rechts.
- ◦ Maak p zwart.
- ◦ Maak g rood.

   (b) **Knoop c is een rechterkind van p.**



Figuur 1.7: Rode broer.

- ◦ Roteer p en c naar links.
- ◦ We krijgen nu het vorige geval.

- Hoogstens 2 rotaties om de boom te herstellen, voorafgegaan door eventueel $O(\lg n)$ opschuivingen.
- Roteren en opschuiven is $O(1)$, en afdalen is $O(\lg n)$ zodat toevoegen $O(\lg n)$ is.

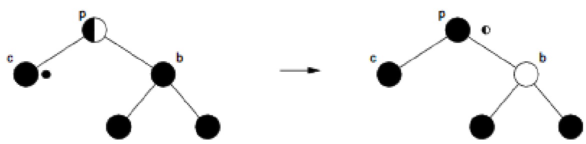
Verwijderen

- Verwijder eerst de knoop zoals bij een gewone zoekboom.
- Als de te verwijderen knoop rood is, is er geen gevolg voor de zwarte hoogte en is de operatie klaar.
- Als de te verwijderen knoop zwart is, zijn er drie mogelijkheden:

1. **De knoop heeft één rood kind.** Dit rood kind kan de zwarte kleur overnemen, zodat de zwarte hoogten intact blijven.
2. **De knoop heeft twee zwarte kinderen (virtueel of echt)**. De zwarte kleur wordt aan één van de kinderen gegeven, zodat die **dubbelzwart** wordt.

- In het eerste geval is de operatie klaar. In het tweede geval moet de boom, die nu een dubbelzwarte knoop bevat, herstelt worden.
- Als de dubbelzwarte knoop c de wortel is, kan deze extra zwarte kleur verdwijnen.
- Als c geen wortel is, en ouder p heeft, dan zijn er acht mogelijkheden die in groepen van twee uiteenvallen naargelang c een linker- of rechterkind van p is.
- We veronderstellen dat c een linkerkind is van p.

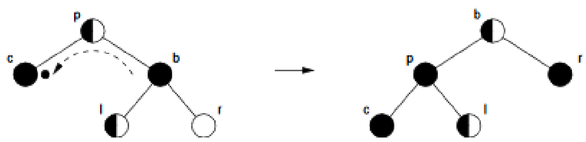
1. **De broer b van c is zwart.** De kleur van p is willekeurig. Hier zijn er drie gevallen mogelijk, afhankelijk van de kleur van de kinderen van b.
   (a) **Broer b heeft twee zwarte kinderen**.



Figuur 1.8

   - Knoop b kan rood worden.
   - De extra zwarte kleur van c kan aan p gegeven worden.
     - Als p rood was, dan is de operatie gelukt.
     - Als p reeds zwart was, dan verschuift het probleem zich naar boven.

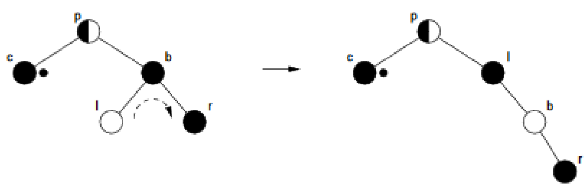
   (b) **Broer b heeft een rood rechterkind**. De kleur van het linkerkind l van b is willekeurig.



Figuur 1.9

   - Roteer p en b naar links.
   - Knoop p krijgt de extra zwarte kleur van c.
   - Het rechterkind r van b wordt zwart.
   - Knoop b krijgt de oorspronkelijke kleur van p.

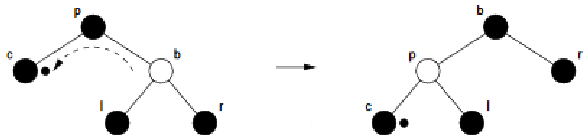
   (c) **Broer b heeft een zwarte rechterkind en een rood linkerkind**.



Figuur 1.10

- ◦ Roteer b en l naar rechts.
- ◦ Maak b rood en l zwart.
- ◦ Dit is nu het vorige geval.

2. **De broer b van c is rood.**



Figuur 1.11

- ◦ Roteer p en b naar links.
- ◦ Maak b zwart en p rood.
- ◦ Dit is nu het eerste geval.

- Hoogstens 3 rotaties nodig om de boom te herstellen, voorafgegaan door eventueel $O(\lg n)$ opschuivingen.
- Roteren en opschuiven is $O(1)$, en afdalen is $O(\lg n)$ zodat verwijderen $O(\lg n)$ is.

1.2.6 Top-down rood-zwarte bomen

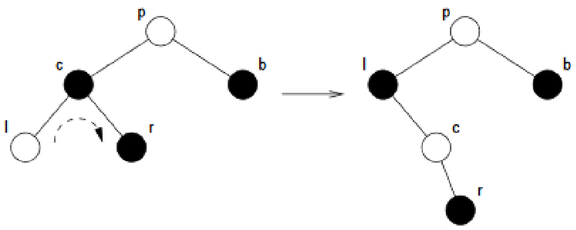
Toevoegen

- Ook hier worden nieuwe knopen rood gemaakt.
- Op de weg naar beneden mogen er geen rode broers zijn.
- Als we een **zwarte knoop met twee rode kinderen** tegenkomen, dan maken we die knoop rood en zijn kinderen zwart.
- Als zijn ouder rood is, kan dit met rotaties en kleurwijzigingen opgelost worden.
- Toevoegen daalt enkel in de boom en is $O(\lg n)$.

Verwijderen

- De zwarte hoogte van de fysisch te verwijderen knoop is één, omdat minstens één van zijn kinderen virtueel is.
- Om geen problemen te krijgen met de zwarte hoogte moet deze knoop rood zijn, maar dan moet zijn tweede kind ook virtueel zijn.
- De zoekknoop kan eender waar in de boom zitten, daarom wordt elke bolgende knoop op de zoekweg rood gemaakt.
  - ◦ Tijdens het afdalen komen we in een rode of rood gemaakte knoop p.
  - ◦ Die heeft dan zeker een zwart kind c, dat rood moet worden.
  - ◦ Er zijn acht mogelijkheden die in groepen van twee uiteenvallen naargelang c een linker- of rechterkind van p is.
- We veronderstellen dat c een linkerkind is van p.

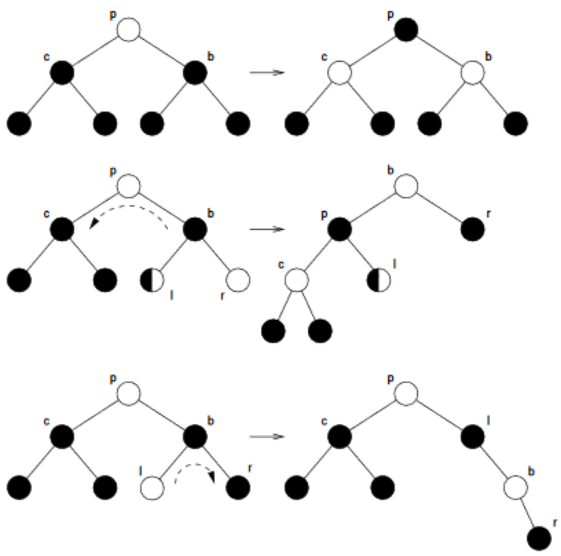
1. **Knoop c heeft minstens één rood kind.**



Figuur 1.12

- ◦ Als we naar een rode knoop moeten afdalen zitten we terug in de beginsituatie.
- ◦ Als c de fysisch te verwijderen knoop is of als we naar een zwarte knoop moeten afdalen:
  - ⋄ Roteer c samen met zijn rood kind zodat c nu als ouder zijn oorspronkelijk rood kind heeft.
  - ⋄ Wijzig de kleur van c naar zwart.
  - ⋄ Wijzig de kleur van zijn oorspronkelijk kind naar rood.

2. **Knoop c heeft twee zwarte kinderen.**



Figuur 1.13

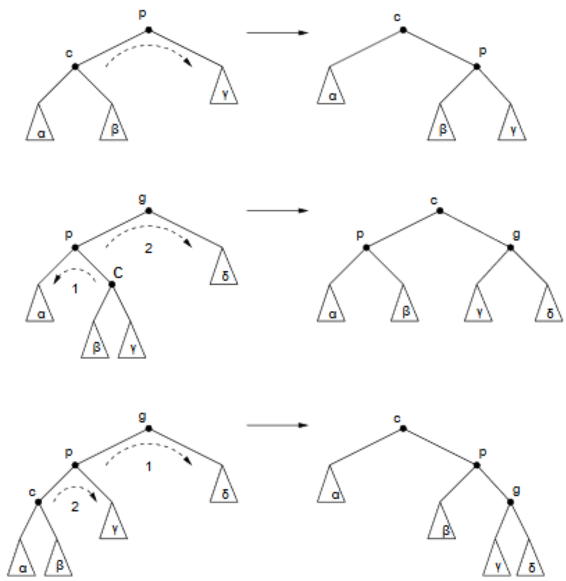
1.2.7 Vereenvoudigde rood-zwarte bomen

- De implementatie is omslachtig door de talrijke speciale gevallen.
- Eenvoudigere varianten bestaan:
  - ◦ Een **AA-boom** geeft aan dat enkel een rechterkind root moet zijn.
  - ◦ Een **Binary B-tree** beperkt het aantal gevallen maar behouden toch de asymptotische efficiëntie.
  - ◦ Een **left-leaning red-black-tree** stelt de eis dat een zwarte knoop enkel een rood rechterkind mag hebben als het reeds een rood linkerkind heeft.

1.3 Splaybomen

- Garanderen dat elke reeks opeenvolgende operaties efficiënt is.
- Als we m operaties verrichten op de splay tree, waarbij n keer toevoegen, dan is de performantie van deze reeks $O(m \lg n)$.
- Uitgemiddeld is dit $O(\lg n)$.
- Individuelde operaties mogen inefficiënt zijn, maar de boom moet zo aangepast worden zodat een reeks van die operaties efficiënt zijn.
- **Basisidee:** Elke knoop die gezocht wordt, toegevoegd of verwijdert wordt, zal de wortel worden van de boom, zodat opeenvolgende operaties op die knoop efficiënt zijn.
- Een willekeurige knoop tot wortel maken gebeurd via de ***splay-operatie***.
- De weg naar een diepe knoop bevat knopen die ook diep liggen. Terwijl we een knoop wortel maken, moeten de knopen op het zoekpad ook aangepast worden, zodat ook de toegangstijd van deze knopen verbetert, anders blijft de kans bestaan dat een reeks van operaties inefficiënt is.
- Er moet geen extra informatie bijgehouden worden voor knopen, wat geheugen uitspaart.
- De splay-operatie wordt gedefinieerd voor zowel bottom-up als top-down splaybomen.

1.3.1 Bottom-up splayboom



Figuur 1.14: Bottom-up splay.

- De knoop wordt eerst gezocht zoals bij een gewone zoekboom.
- De splay-operatie gebeurt van onder naar boven.
- Een knoop kan naar boven gebracht worden door hem telkens te roteren met zijn ouder.

- Om de toegangstijd van knopen op de zoekweg ook te verbeteren, zijn er drie verschillende rotaties:

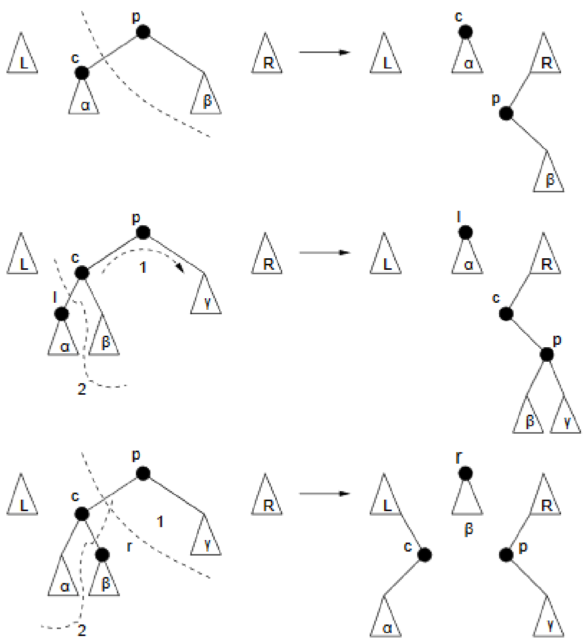
  1. **De ouder p van c is wortel.**
     - Roteer beide knopen zodat c de wortel wordt.
  2. **Knoop c heeft nog een grootouder.**
     - Er zijn vier gevallen, die uitvallen in groepen van twee, naar gelang dat p een linker- of rechterkind is van grootouder g.
     - We veronderstellen dat p linkerkind is van g.

     (a) **Knoop c is een rechterkind van p.**
       - Roteer p en c naar links.
       - Roteer g en c naar rechts.

     (b) **Knoop c is een linkerkind van p.**
       - Roteer g en p naar rechts.
       - Roteer p en c naar rechts.

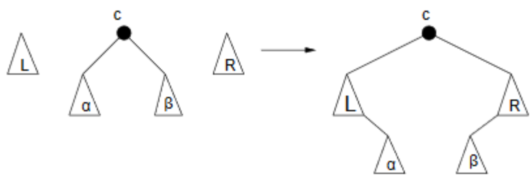
- De **woordenboekoperaties verlopen nu als volgt:**
  - **Zoeken.** De knoop wordt eerst gezocht zoals een gewone zoekboom. Daarna wordt deze tot wortel gemaakt via de splay-operatie.
  - **Toevoegen.** Toevoegen gebeurt ook zoals een gewone zoekboom. De nieuwe knoop wordt dan tot wortel gemaakt met de splay-operatie.
  - **Verwijderen.** Verwijderen gebeurt ook zoals een gewone zoekboom. Daarna wordt de ouder van die knoop tot wortel gemaakt met de splay-operatie.

1.3.2 Top-down splayboom



Figuur 1.15: Top-down splay.

- De splayoperatie wordt uitgevoerd tijdens de afdaling, zodat de gezochte knoop wortel zal zijn als we hem bereiken.
- De boom wordt in drie zoekbomen opgedeeld, L, M en R.
    - Alle sleutels in L zijn kleiner dan die in M.
    - Alle sleutels in R zijn groter dan die in M.
- Eerst is M de oorspronkelijke boom en zijn L en R ledig.
- De huidige knoop op de zoekweg is steeds de wortel van M.
- Stel dat we bij een knoop p uitkomen, en dan nog verder moeten naar een knoop c.
- Er zijn dan twee groepen van 3 gevallen, afhankelijk of c een linker- of rechterkind is van p.
- We veronderstellen dat c een linkerkind is van p.
    1. **Knoop c is de laatste knoop op de zoekweg.**
        - Knoop p wordt het nieuwe kleinste element in R samen met zijn rechtse deelboom.
        - Knoop c wordt de wortel van M.
    2. **Knoop c is niet de laatste knoop op de zoekweg.**
        - **We moeten verder afdalen naar het linkerkind l van c.**
            - Roteer p en c naar rechts.
            - Knoop c wordt het kleinste element in R samen met de rechtse deelboom van c.
            - De linkse deelboom van c wordt de nieuwe M met als wortel l.
        - **We moeten verder afdalen naar het rechterkind r van c.**
            - Knoop p wordt het kleinste element in R samen met de rechtse deelboom van p.
            - Knoop c wordt het nieuwe grootste element in L.
            - De rechtse deelboom van c wordt de nieuwe M met als wortel r.
- Als de gezochte knoop c wortel van M is, wordt de splayoperatie afgerond met een **join-operatie**.



Figuur 1.16: Samenvoegen na top-down splayen.

- De **woordenboekoperaties verlopen nu als volgt:**
    - **Zoeken.** De knoop met de gezochte sleutel wordt tot wortel gemaakt. Als de sleutel niet gevonden wordt dan is zijn opvolger of voorloper de wortel.
    - **Toevoegen.**
    - **Verwijderen.**

1.3.3 Performantie van splay trees

- Niet eenvoudig aangezien vorm van de boom vaak verandert.
- We willen aantonen dat een reeks van m operaties op een splay tree met maximaal n knopen een performantie van $O(m \lg n)$ heeft.
- Er wordt een **potentiaalfunctie** Φ gebruikt.
- Elke mogelijke vorm van een splayboom krijgt een reeël getal toegewezen aan de hand van deze potentiaalfunctie.
- Efficiënte operaties die minder tijd gebruiken dan de geamortiseerde tijd per operatie doen het potentiaal stijgen.
- Niet-efficiënte operaties die meer tijd gebruiken dan de geamortiseerde tijd per operatie doen het potentiaal dalen.
- De geamortiseerde tijd van een operatie wordt gedefinieerd als de som van haar werkelijke tijd en de toename van het potentiaal.
  - Stel t_i de werkelijke tijd van de $i-$de operatie.
  - Stel a_i de geamortiseerde tijd van die operatie.
  - Stel Φ_i het potentiaal na deze operatie.

$$\rightarrow a_i = t_i + \Phi_i - \Phi_{i-1}$$

- De geamortiseerde tijd van een reeks m operaties is de som van de individuelde geamortiseerde tijden:

$$\begin{aligned}\sum_{i=1}^{m} a_i &= \sum_{i=1}^{m}(t_i + \Phi_i - \Phi_{i-1}) \\ &= t_1 + \Phi_1 - \Phi_0 + t_2 + \Phi_2 - \Phi_1 + t_3 + \Phi_3 - \Phi_2 + \cdots + t_m + \Phi_m - \Phi_{m-1} \\ &= \Phi_m - \Phi_0 + \sum_{i=1}^{m} t_i\end{aligned}$$

- Als de potentiaalfunctie zo gekozen wordt zodat het eindpotentiaal Φ_m zeker niet kleiner is dan de beginpotentiaal Φ_0, dan vormt de totale geamortiseerde tijd een **bovengrens** van de werkelijke tijd want de boom zal zeker niet slechter zijn.
- De eenvoudigste potentiaalfunctie geeft voor elke knoop i een gewicht s_i die gelijk is aan het aantal knopen in de deelboom waarvan hij wortel is. De potentiaal van de boom is dan de som van de logaritmen van deze gewichten:

$$\Phi = \sum_{i=1}^{\Phi} \lg s_i$$

- We noemen $\lg s_i$ de rang r_i van knoop i.
- Performantie-analyse van bottom-up splayboom:
  - Performantie is evenredig met de diepte van de knoop, en dus met het aantal uitgevoerde rotaties.

- We willen aantonen dat de geamortiseerde tijd voor het zoeken naar een knoop c gevolgd door een splay-operatie op die knoop gelijk is aan

$$O(1 + 3(r_w - r_c))$$

waarbij r_w de rang van de wortel is en r_c de rang van de gezochte knoop.

  - Als c reeds de wortel is, dan is $r_w = r_c$ en blijft het potentiaal dezelfde.

$$O(1 + 3(r_w - r_c)) = O(1)$$

  - Anders moeten zoveel splay-operaties uitgevoerd worden als de diepte van de knoop (moet niet gekend zijn).
    - Een zig wijzigt de rang van c en p

$$a < 1 + r'_c - r_c$$

    - Een zig-zag wijzigt de rang van c, p en g

$$a < 2(r'_c - r_c)$$

    - Een zig-zig wijzigt de rang van c, p en g

$$a < 3(r'_c - r_c)$$

- De bovengrenzen voor de drie operaties bevatten dezelfde positieve term $r'_c - r_c$ maar met verschillende coëfficiënten.
- De totale geamortiseerde tijd is een som van dergerlijke bovengrenzen, maar kan niet vereenvoudigt worden als coëfficiënten niet gelijk zijn.
- Aangezien het bovengrenzen zijn, wordt de grootste coëfficiënt genomen.
- In de som vallen de meeste termen nu weg, behalve de rang van c voor en na de volledige splay-operatie.
- De geamortiseerde tijden van de woordenboekoperaties op een bottom-splay tree met n knopen zijn nu:
  - **Zoeken.** $O(1 + 3 \lg n)$ want $s_w = n$.
  - **Toevoegen.** $O(1 + 4 \lg n)$.
    Op de zoekweg worden de rang van knopen $p_1, p_2, ...p_k$ op de zoekweg gewijzigt. Stel s_{p_i} het gewicht van knoop p_i voor het toevoegen en s'_{p_i} het gewicht van knoop p_i na het toevoegen. De potentiaaltoename is dan

$$\lg\left(\frac{s'_{p_1}}{s_{p_1}}\right) + \lg\left(\frac{s'_{p_2}}{s_{p_2}}\right) + \cdots + \lg\left(\frac{s'_{p_k}}{s_{p_k}}\right) = \lg\left(\frac{s'_{p_1}}{s_{p_1}}\frac{s'_{p_2}}{s_{p_2}}\cdots\frac{s'_{p_k}}{s_{p_k}}\right)$$

    Deze is nooit groter dan

$$\lg\left(\frac{s'_{p_1}}{s_{p_k}}\right) \leq \lg(n+1)$$

  - **Verwijderen.** Het effect van verwijderen is nooit positief.
- De geamortiseerde tijd voor een reeks van m woordenboekoperaties is de som van de geamortiseerde tijden voor de individuele operaties.
- Stel n_i het aantal knopen bij de $i-$de operatie wordt die tijd $O(m + 4\sum_{i=1}^{m} \lg n_i) = O(m + 4m \lg n) = O(m \lg n)$.

1.4 Gerandomiseerde zoekbomen

- De performantie van de woordenboekoperaties op een gewone zoekboom is $O(\lg n)$ als elke toevoegvolgorde even waarschijnlijk is.
- Gerandomiseerde zoekbomen maken gebruik van een random generator om de operatievolgorde te neutraliseren.
- Deze bomen blijven steeds random.
- Een **treap** is een gerandomiseerde zoekboom.
  - Elke knoop krijgt naast een sleutel ook een prioriteit, die door de random generator wordt toegekend als de knoop toegevoegd wordt.
  - De prioriteiten van de knopen voldoen aan de heapvoorwaarde: de prioriteit van een kind is maximaal even hoog als die van zijn ouder.
- De woordenboekoperaties:
  - **Zoeken.** Zoeken moet geen rekening houden met de prioriteiten en verloopt zoals een normale binaire zoekboom.
  - **Toevoegen.** Eerst wordt er normaal toegevoegd. De knoop wordt nadien naar boven geroteerd om aan de heapvoorwaarde te voldoen.
  - **Verwijderen.** De te verwijderen knoop krijgt de laagste prioriteit, zodat die naar beneden geroteerd wordt. Dit blad kan dan verwijdert worden.

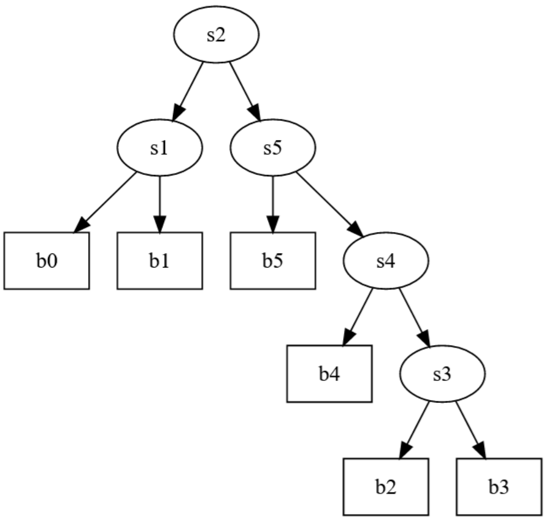
1.5 Skip lists

- Een meerwegszoekboom geïmplementeerd met gelinkte lijsten.
- Alle bladeren zitten op dezelfde diepte.
- Elke lijstknoop heeft plaats voor één sleutel en één kindwijzer.
- Een knoop met k kinderen bevat $k-1$ sleutels, zodat er één sleutelplaats over blijft.
- (zoekt gewoon eens een foto op)

Hoofdstuk 2

Toepassingen van dynamisch programmeren

2.1 Optimale binaire zoekbomen



Figuur 2.1: Een optimale binaire zoekboom met bijhorende kansentabel.

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

- Veronderstel dat de gegevens die in een binaire zoekboom moeten opgeslaan worden op voorhand gekend zijn.
- Veronderstel ook dat de waarschijnlijkheid gekend is waarmee de gegevens gezocht zullen worden.
- Tracht de boom zo te schikken zodat de verwachtingswaarde van de zoektijd minimaal wordt.
- De zoektijd wordt bepaald door de lengte van de zoekweg.

- De gerangschikte sleutels van de n gegevens zijn $s_1, \ldots, s_n$.
- De $n+1$ bladeren zijn $b_0, \ldots, b_n$.
  - Elk blad staat voor een afwezig gegeven die in een deelboom op die plaats had moeten zitten.
  - Het blad b_0 staat voor alle sleutels kleiner dan s_1.
  - Het blad b_n staat voor alle sleutels groter dan s_n.
  - Het blad b_i staat voor alle sleutels gorter dan s_i en kleiner dan s_{i+1}, met $1 \leq i < n$
- De waarschijnlijkheid om de i-de sleutel s_i te zoeken is p_i.
- De waarschijnlijkheid om alle afwezige sleutels, voorgesteld door een blad b_i, te zoeken is q_i.
- De som van alle waarschijnlijkheden moet 1 zijn:

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

- Als de zoektijd gelijk is aan het aantal knopen op de zoekweg (de diepte plus één), dan is de verwachtingswaarde van de zoektijd

$$\sum_{i=1}^{n} p_i(\text{diepte}(s_i) + 1) + \sum_{i=0}^{n} q_i(\text{diepte}(b_i) + 1)$$

- Deze verwachtingswaarde moet geminimaliseerd worden.
  - Boom met minimale hoogte is niet voldoende.
  - Alle mogelijke zoekbomen onderzoeken is ook geen optie, hun aantal is

$$\begin{aligned} \frac{1}{n+1}\binom{2n}{n} &\sim \frac{1}{n+1} \cdot \frac{2^{2n}}{\sqrt{\pi n}} \\ &\sim \frac{1}{n+1} \cdot \frac{4^n}{\sqrt{\pi n}} \\ &\sim \Omega\left(\frac{4^n}{n\sqrt{n}}\right) \end{aligned}$$

  ✓ Dynamisch programmeren biedt een uitkomst.
- Een optimalisatieprobleem komt in aanmerkingen voor een efficiënte oplossing via dynamisch programmeren als het:
  1. het een **optimale deelstructuur** heeft;
  2. de **deelproblemen onafhankelijk maar overlappend** zijn.
- Is dit hier van toepassing?
  - Is er een optimale deelstructuur?
    - Als een zoekboom optimaal is, moeten zijn deelbomen ook optimaal zijn.
    - Een optimale oplossing bestaat uit optimale oplossingen voor deelproblemen.
  - Zijn de deelproblemen onafhankelijk?
    - Ja want deelbomen hebben geen gemeenschappelijke knopen.
  - Zijn de deelproblemen overlappend?

- ◇ Elke deelboom bevat een reeks opeenvolgende sleutels $s_i, \ldots s_j$ met bijhorende bladeren $b_{i-1}, \ldots, b_j$.
- ◇ Deze deelboom heeft een wortel s_w waarbij $(i \leq w \leq j)$.
- ◇ De linkse deelboom bevat de sleutels $s_i, \ldots, s_{w-1}$ en bladeren $b_{i-1}, \ldots, b_{w-1}$.
- ◇ De rechtse deelboom bevat de sleutels $_{w+1}, \ldots, s_j$ en bladeren $b_w, \ldots, b_j$.
- ◇ Voor een optimale deelboom met wortel s_w moeten deze beide deelbomen ook optimaal zijn.
- ◇ Deze wordt gevonden door:
  1. achtereenvolgens elk van zijn sleutels $s_i, \ldots, s_j$ als wortel te kiezen;
  2. de zoektijd voor de boom te berekenen door gebruikte maken van de zoektijden van de optimale deelbomen;
  3. de wortel te kiezen die de kleinste zoektijd oplevert.

- We willen dus de kleinste verwachte zoektijd $z(i, j)$.
- Dit moet gebeuren voor alle i en j waarbij:
  - $1 \leq i \leq n + 1$
  - $0 \leq j \leq n$
  - $j \geq i - 1$
- De optimale boom heeft dus de kleinste verwachte zoektijd $z(1, n)$.
- Hoe $z_w(i, j)$ bepalen voor een deelboom met wortel s_w?
  - Gebruik de kans om in de wortel te komen.
  - Gebruik de optimale zoektijden van zijn deelbomen, $z(i, w - 1)$ en $z(w + 1, j)$.
  - Gebruik ook de diepte van elke knoop, maar elke knoop staat nu op een niveau lager.
    - ◇ De bijdrage tot de zoektijd neemt toe met de som van de zoekwaarschijnlijkheden.

$$g(i, j) = \sum_{k=i}^{j} p_k + \sum_{k=i-1}^{j} q_k$$

  - Hieruit volgt:

$$\begin{aligned} z_w(i, j) &= p_w + (z(i, w - 1) + g(i, w - 1)) + (z(w + 1, j) + g(w + 1, j)) \\ &= z(i, w - 1) + z(w + 1, j) + g(i, j) \end{aligned}$$

- Dit moet minimaal zijn → achtereenvolgens elke sleutel van de deelboom tot wortel maken.
  - De index w doorloopt alle waarden tussen i en j.

$$\begin{aligned} z(i, j) &= \min_{i \leq w \leq j} \{z_w(i, j)\} \\ &= \min_{i \leq w \leq j} \{z(i, w - 1) + z(w + 1, j)\} + g(i, j) \end{aligned}$$

- Hoe wordt de optimale boom bijgehouden?
  - Hou enkel de index w bij van de wortel van elke optimale deelboom.
  - Voor de deelboom met sleutels $s_i, \ldots, s_j$ is de index $w = r(i, j)$.
- Implementatie:
  - Een recursieve implementatie zou veel deeloplossingen opnieuw berekenen.

- Daarom **bottom-up** implementeren. Maakt gebruik van drie tabellen:
    1. Tweedimensionale tabel $z[1..n+1, 0..n]$ voor de waarden $z(i,j)$.
    2. Tweedimensionale tabel $g[1..n+1, 0..n]$ voor de waarden $g(i,j)$.
    3. Tweedimensionale tabel $r[1..n, 1..n]$ voor de indices $r(i,j)$.
- Algoritme:
    1. Initialiseer de waarden $z(i,i-1)$ en $g(i,i-1)$ op $q[i-1]$.
    2. Bepaal achtereenvolgens elementen op elke evenwijdige diagonaal, in de richting van de tabelhoek linksboven.
        - Voor $z(i,j)$ zijn de waarden $z(i,i-1), z(i,i), \ldots, z(i,j-1)$ van de linkse deelboom nodig en de waarden $z(i+1,j), \ldots, z(j,j), z(j+1,j)$ van de rechtse deelbom nodig.
        - Deze waarden staan op diagonalen onder deze van $z(i,j)$.

- Efficiëntie:
    - **Bovengrens:** drie verneste lussen $\rightarrow O(n^3)$.
    - **Ondergrens:**
        - Meeste werk bevindt zich in de binneste lus.
        - Een deelboom met sleutels $s_i, \ldots s_j$ heeft $j-i+1$ mogelijke wortels.
        - Elke test is $O(1)$.
        - Dit werk is evenredig met

$$\begin{aligned} & \sum_{i=1}^{n}\sum_{j=i}^{n}(j-i+1) \\ \Rightarrow \quad & \sum_{i=1}^{n} i^2 \\ \Rightarrow \quad & \frac{n(n+1)(2n+1)}{6} \\ \Rightarrow \quad & \Omega(n^3) \end{aligned}$$

    - **Algemeen:** $\Theta(n^3)$.
    - Kan met een bijkomende eigenschap (zien we niet in de cursus) gereduceerd worden tot $\Theta(n^2)$.

2.2 Langste gemeenschappelijke deelsequentie

- Een deelsequentie van een string wordt bekomen door nul of meer stringelementen weg te laten.
- Elke deelstring is een deelsequentie, maar niet omgekeerd.
- Een langste gemeenschappelijke deelsequentie (LGD) van twee strings kan nagaan hoe goed deze twee strings op elkaar lijken.
- Geven twee strings:
    - $X = \langle x_0, x_1, \ldots, x_{n-1}\rangle$
    - $Y = \langle y_0, y_1, \ldots, y_{m-1}\rangle$
- Hoe LGD bepalen?
    - Is er een optimale deelstructuur?

    - ◇ Een optimale oplossing maakt gebruik van optimale oplossingen voor deelproblemen.
    - ◇ De deelproblemen zijn paren prefixen van de twee strings.
    - ◇ Het prefix van X met lengte i is X_i.
    - ◇ Het prefix van Y met lengte j is Y_j.
    - ◇ De ledige prefix is X_0 en Y_0.
  - ○ Zijn de deelproblemen onafhankelijk?
    - ◇ Stel $Z = \langle z_0, z_1, \ldots, z_{k-1} \rangle$ de LGD van X en Y. Er zijn drie mogelijkheden:
      1. Als $n = 0$ of $m = 0$ dan is $k = 0$.
      2. Als $x_{n-1} = y_{m-1}$ dan is $z_{k-1} = x_{n-1} = y_{m-1}$ en is Z een LGD van X_{n-1} en Y_{m-1}.
      3. Als $x_{n-1} \neq y_{m-1}$ dan is Z een LGD van X_{n-1} en Y of een LGD van X en Y_{m-1}.
  - ○ Zijn de deelproblemen overlappend?
    - ◇ Om de LGD van X en Y te vinden is het nodig om de LGD van X en Y_{m-1} als van X_{n-1} en Y te vinden.

- De lengte $c[i,j]$ van de LGD van X_i en Y_j wordt door een recursieve vergelijking bepaald:

$$c[i,j] = \begin{cases} 0 & \text{als } i = 0 \text{ of } j = 0 \\ c[i-1, j-1] + 1 & \text{als } i > 0 \text{ en } j > 0 \text{ en } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{als } i > 0 \text{ en } j > 0 \text{ en } x_i \neq y_j \end{cases}$$

- De lengte van de LGD komt overeen met $c[n,m]$.
- De waarden $c[i,j]$ kunnen eenvoudig per rij van links naar rechts bepaald worden door de recursierelatie.
- Efficiëntie:
  - ○ We beginnen de tabel in te vullen vanaf $c[1,1]$ (als $i = 0$ of $j = 0$ zijn de waarden 0).
  - ○ De tabel c wordt rij per rij, kolom per kolom ingevuld.
  - ○ De vereiste plaats en totale performantie is beiden $\Theta(nm)$.

Hoofdstuk 3

Uitwendige gegevensstructuren

- Als de grootte van de gegevens de capaciteit van het intern geheugen overschrijdt, moeten deze gegevens opgeslagen worden in extern geheugen.
- We willen dat woordenboekoperaties nog steeds efficiënt uitgevoerd worden.
- Een harde schijf is veel trager dan een processor.
- Daarom moet het aantal schijfoperaties geminimaliseerd worden.

3.1 B-trees

- Uitwendige evenwichte zoekboom.
- Heeft een zeer kleine hoogte.
- Het aantal sleutels n is wel zeer groot.
- Er worden dus meerdere kinderen per knoop opgeslagen.
- Knopen kunnen best een volledige schijfpagina benutten.

3.1.1 Definitie

- Een B-tree heeft een orde m waarbij $m > 2$.
  - Elke inwendige knoop heeft hoogstens m kinderen.
  - Elke inwendige knoop, behalve de wortel, heeft minstens $\lceil m/2 \rceil$ kinderen.
  - Elke inwendige knoop met $k + 1$ kinderen bevat k sleutels.
  - Elk blad bevat hoogstens $m - 1$ en minstens $\lceil m/2 \rceil - 1$ sleutels.
  - De wortel bevat tenminste 2 kinderen, tenzij hij een blad is.
  - Als de wortel een blad is bevat hij minstens 1 sleutel.
  - Alle bladeren bevinden zich op hetzelfde niveau.
- Elke knoop bevat het volgende:
  - Een geheel getal k dan het huidig aantal sleutels in de knoop aanduidt.
  - Een tabel voor maximaal m pointers naar de kinderen van de knoop.

  - Een tabel voor maximaal $m-1$ sleutels, die stijgend gerangschikt zijn.
    - Er is ook een tabel die bijbehorende informatie per sleutel bijhoudt.
    - De k geordende sleutels van de inwendige knoop verdelen het sleutelbereik in $k+1$ deelgebieden.
    - De sleutels uit de deelboom van het i-de kind c_i liggen tussen de sleutels s_{i-1} en s_i.
  - Een logische waarde b die aanduidt of de knoop een blad is of niet.
- $2-3$ bomen ($m=3$) of $2-3-4$ bomen ($m=4$) zijn eenvoudige voorbeelden van B-trees. Normaal is m wel groter.

3.1.2 Eigenschappen

- Het minimaal aantal knopen voor een boom met hoogte h is

$$1+2+2g+\cdots+2g^{h-1}=1+2\sum_{i=0}^{h-1}g^i=1+2\left(\frac{1-g^h}{1-g}\right)$$

  - De wortel van een minimale boom heeft slechts 1 sleutel en twee kinderen.
  - Elk kind heeft minimum $g=\lceil m/2\rceil$ kinderen.
- De hoogte is bijgevolg $O(\lg n)$.
  - Elke knoop heeft minstens $g-1$ sleutels, behalve de wortel, die er minstens één heeft.

$$\begin{aligned} n &\geq 1+2(g-1)\left(\frac{g^h-1}{g-1}\right) \\ \rightarrow n &\geq 2g^h-1 \\ \rightarrow h &\leq log_g\left(\frac{n+1}{2}\right) \end{aligned}$$

- Een B-tree met n uniform verdeelde sleutels gebruikt ongeveer $\frac{n}{m\ln 2}$ schijfpaginas.

3.1.3 Woordenboekoperaties

Zoeken

- In elke knoop moet een meerwegsbeslissing genomen worden.
- De knoop moet eerst in het geheugen ingelezen worden.
- De sleutel wordt opgezocht in de gerangschikte tabel met sleutels.
  - Normaal zou binair zoeken efficiënter zijn, maar deze winst is vrij onbelangrijk.
  - Lineair zoeken kan bij kleine tabellen efficiënter uitvallen door het aantal cachefouten te minimaliseren.
- Er kunnen zich nu drie situaties voordoen:
  1. Als de sleutel in de tabel zit stopt het zoeken, met als resultaat een verwijzing naar de knoop op de schijf.
  2. Als de sleutel niet gevonden is en is de knoop een blad, dan zit de sleutel niet in de boom.

3. Als de sleutel niet gevonden is en de knoop is inwendig, wordt een nieuwe knoop in het geheugen ingelezen waarvan de wortel een kind is van de huidige knoop. Het zoekproces start opnieuw met deze knoop.

- Performantie:
  - Het aantal schijfoperaties is $O(h) = O(log_g n)$.
  - De processortijd per knoop is $O(m)$.
  - De totale performantie is $O(m \log_g n)$.

Toevoegen

- Toevoegen gebeurt **bottom-up**. Een top-down implementatie is ook mogelijk maar wordt minder gebruikt.
- De structuur van de boom kan gewijzigd worden.
- Toevoegen gebeurt altijd aan een blad.
- Vanuit de wortel wordt eerst het blad gezocht waarin de sleutel zou moeten zitten.
- Drie gevallen:
  - **De B-tree is ledig.**
    - De wortelknoop moet in het geheugen aangemaakt worden en gedeeltelijk ingevuld worden.
    - De knoop wordt dan naar de schijf gekopieërd.
  - **De B-tree is niet ledig.**
    - Het blad waarin de sleutel moet zitten wordt opgezocht. Er zijn dan twee gevallen.
    - **Het blad bevat minder dan m sleutels.**
      - De sleutel wordt in de juiste volgorde toegevoegd aan de tabel met sleutels.
    - **Het blad bevat m sleutels.**
      - Het blad wordt opgeslitst bij de middelste seutel.
      - Er wordt een nieuwe knoop op hetzelfde niveau aangemaakt, waarin de gegevens rechts van de middelste sleutel terechtkomen.
      - De middelste sleutel gaat naar de ouder.
      - Normaal gezien heeft de ouder plaats voor deze knoop, anders wordt er opnieuw geplitst.
- Performantie:
  - In het slechtste geval worden er $h + 1$ knopen gesplitst.
  - Een knoop splitsen vereist drie schijfoperaties en een processortijd van $O(m)$.
  - In het slechtste geval moet de boom tweemaal doorlopen worden.
    - Eerst om de sleutel te vinden.
    - Daarna eventueel tot de wortel splitsen.
    - ✓ Maar het aantal schijfoperaties per niveau is constant.
  - Het totaal aantal schijfoperaties is $\Theta(h)$.
  - De totale performantie is dan $O(mh) = O(m \log_g n)$.

Verwijderen

- Ook hier wordt enkel de **bottom-up** versie besproken.
- De gezochte sleutel kan zowel in een blad als in een inwendige knoop zitten.
  - **De sleutel zit in een blad.**
    - Er zijn geen kinderen meer dus kan de sleutel verwijderd worden.
    - Het kan zijn dat het blad nu te weinig sleutels heeft (minder dan $\lceil m/2 \rceil - 1$).
    - Er wordt een sleutel geleend van de ouder.
    - In het slechtste geval gaat dit ontlenen door tot aan de wortel.
    - Een sleutel ontlenen van een wortel die slechts één sleutel bevat maakt hem ledig, zodat de wortel verwijdert wordt.
  - **De sleutel zit in een inwendige knoop.**
    - De sleutel wordt vervangen door zijn voorloper of opvolger.
    - Als een knoop nu te weinig sleutels overhoudt, gebeurt er een **rotatie**.
      - Een sleutel van zijn broer gaat naar zijn ouder.
      - Een sleutel van de ouder gaat naar de knoop, die ook een kindwijzer van de broer overneemt.
      - Dit kan enkel als er een broer is die sleutels kan missen.
      - Als geen enkele broer een sleutel kan missen, wordt de knoop samengevoegd met een broer.
- Performantie:
  - Analoog aan toevoegen en is dan $O(m \log_g n)$.

3.1.4 Varianten van B-trees

- Nadelen van een gewone B-tree:
  - De bladeren moeten plaats reserveren voor kindwijzers die toch niet gebruikt worden.
  - Inwendige knopen kunnen gegevens bevatten en dat maakt verwijderen veel ingewikkelder.
  - Zoeken naar een opvolger van een sleutel kan $O(\log_g n)$ schijfoperaties vereisen.

B^+-tree

- Alle gegevens en bijhorende informatie zitten in de bladeren.
- Inwendige knopen worden gebruikt als index om de gegevens snel te lokaliseren.
- Bladeren en inwendige knopen hebben dus een verschillende structuur.
- Er is ook een **sequence set**, een gelinkte lijst van alle bladeren in stijgende sleutelvolgorde.
- De inwendige knopen moeten enkel sleutels bevatten en geen bijhorende informatie zodat de maximale graad groter is dan de bladeren.
- De bladeren moeten geen plaats reserveren voor kindwijzers, zodat ze meer gegevens kunnen bevatten.

Prefix B^+-tree

- Een variant van een B^+-tree voor strings.
- Strings kunnen echter veel plaats innemen.
- Om twee deelbomen van elkaar te onderscheiden wordt de kleinst mogelijke prefix bijgehouden.

B^*-tree

- In plaats van enkel gegevens over te brengen naar een buur tijdens het splitsen, worden de gegevens verdeeld over **drie** knopen.
- Beter gevulde knopen betekent een minder hoge boom.

3.2 Uitwendige hashing

- Waneer de volgorde van de sleutels niet belangreijk is.
- De woordenboekoperaties vereisen gemiddeld slechts $O(1)$.
- Er wordt een binaire trie (hoofdstuk 10) gebruikt.
  - Wanneer een sleutel gezocht wordt, worden de sleutels niet vergeleken maar wel de opeenvolgende bits van de sleutel.
  - Elke deelboom van een trie bevat alle sleutels met een gemeenschappelijk prefix.
  - Alle sleutels van een deelboom kan in één pagina ondergebracht worden.
  - Als de pagina vol geraakt, wordt de knoop (en dus de pagina) gesplitst, en beide pagina's krijgen een nieuwe trieknoop als ouder.
  - De vorm van een trie is onafhankelijk van de toevoegvolgorde.
  - Daarom wordt niet de bits van de sleutels gebruikt, maar de hashwaarde.
- Dit hoofdstuk bespreekt twee methoden: **extendible hashing** en **linear hashing** die beiden het zoeken in de trie elimineren.

3.2.1 Extendible hashing

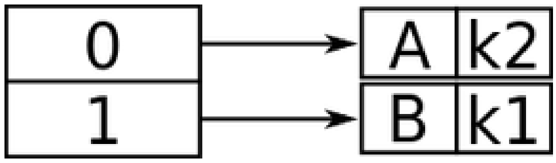
- Het zoeken in de trie wordt geëlimineerd door de langst mogelijke prefix uit de trie als index te gebruiken in een hashtabel.
- Kortere prefixen komen overeen met meerdere tabelelementen die allemaal een verwijzing naar dezelfde pagina moet bevatten.
- Implementatie:
  - Er is een hashtabel die wijzers naar schijfpagina's bevat, waarbij elke schijfpagina maximaal m sleutels met bijbehorende gegevens bevat.
  - De hashwaarden zijn gehele getallen, waarvan het bereik bepaald wordt door de breedte w van een processorwoord.
  - De laatste d bits van die getallen dienen als indices in de hashtabel, zodat de tabel 2^d elementen bevat.
  - De **globale diepte** is d en is de lengte van het langste prefix in de trie.

- Alle sleutels waarvan de hashwaarde met dezelfde d bits eindigt komen bij hetzelfde tabelelement terecht.
- Een pagina kan sleutels met hashwaarden bevatten waarvan de laatste d bits verschillend zijn.
- Het aantal bits k is de **lokale diepte** en is het aantal waarmee al haar hashwaarden eindigen.

- De **woordenboekoperaties:**
  - **Zoeken.**
    - Bereken de hashwaarde van de sleutel.
    - Zoek de overeenkomstige pagina via de hashtabel.
    - Zoek sequentieel in deze pagina.
  - **Toevoegen.**
    - Als de pagina niet vol is moet gemiddeld helft van de elementen opgeschoven worden, maar dat is verwaarloosbaar.
    - Als de pagina vol is moet deze gesplitst worden.
    - Alle hashwaarden in die pagina beginnen met dezelfde k bits.
    - Er wordt daarom gesplitst op het volgende bit $k + 1$. Alle elementen in de pagina waarbij die bit één is wordt overgebracht naar de nieuwe pagina.
    - De waarde van k wordt één groter zowel in de nieuwe pagina als in de oude pagina.
    - De hashtabel moet ook aangepast worden.
      - **Als k kleiner was dan d:** de helft van de wijzers van de oude pagina moeten naar de nieuwe pagina wijzen.
      - **Als k gelijk was aan d:** er was maar één wijzer naar de oude pagina. De waarde van d moet ook met één toenemen en de grootte van de hashtabel moet **verdubbelt** worden.
  - **Verwijderen.**
    - Zien we niet.
- Als er n uniform verdeelde sleutels opgeslagen zijn, dan is de verwachtingswaarde van het aantal pagina's $n/(m \ln 2) \equiv 1.44n/m$.

Voorbeeld

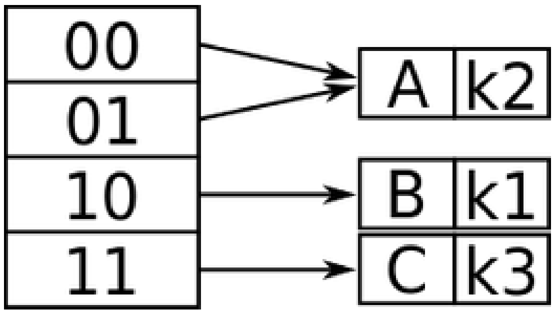
- Veronderstel een hashfunctie $h(k)$.
- Een pagina heeft slechts plaats om één hashwaarde op te slaan.
- De eerste i bits van elke hashwaarde wordt gebruikt als index in de hashtabel.
- De waarde van i is het kleinste getal zodanig dat elk item in de hashtabel uniek is.
- De volgende hashwaarden worden gebruikt:
  - $h(k_1) = 100100$
  - $h(k_2) = 010110$
  - $h(k_3) = 110110$
  - $h(k_4) = 011110$
- **k_1 en k_2 worden toegevoegd.**



Figuur 3.1

- ◦ Deze hashwaarden kunnen onderscheiden worden door het eerste bit, dus $d = 1$.
- ◦ De hashtabel bevat nu twee elementen (figuur 3.1), met twee wijzers: één naar pagina A en één naar pagina B.
- ◦ De lokale diepte k van beide elementen is gelijk aan d, $k = d = 1$.

- **k_3 wordt toegevoegd.**



Figuur 3.2

- ◦ Er zijn niet genoeg bits om k_3 te onderscheiden van k_1.
- ◦ Er wordt vergeleken op bit $k + 1$.
- ◦ Er wordt een nieuwe pagina C aangemaakt waarin de sleutel k_3 opgeslagen wordt.
- ◦ Aangezien k gelijk was aan d, verdubbelt de grootte van de hashtabel.

3.2.2 Linear hashing

- Er wordt geen hashtabel gebruikt door ervoor te zorgen dat pagina's opeenvolgende adressen hebben.
- De d eindbits van de hashwaarde worden niet gebruikt als index, maar rechtstreeks als adres van een pagina.
- Het gaat hier over **logische adressen**, die eenvoudig manipuleerbaar zijn en niet de **fysische adressen** die het besturingssysteem beheert.
- Er zijn 2^d adressen en evenveel pagina's.
- Als een pagina vol is wordt deze gesplitst, maar niet noodzakelijk de hele pagina.
- Pagina's worden in sequentiële volgorde gesplitst, of ze nu vol zijn of niet.
- Elke pagina die niet vol is (alle pagina's behalve de volle die het splitsen veroorzaakt heeft) krijgt een overflow pagina.

- Als de pagina aan de beurt is om te splitsen, worden zijn gegevens verdeeld over zijn overflow pagina en de pagina zelf.
- De **woordenboekoperaties:**
  - **Zoeken.**
    - Bereken de hashwaarde van de sleutel.
    - We moeten echter weten hoeveel eindbits er nodig zijn om de pagina te adresseren.
    - Er wordt een variabele p bijgehouden, die het adres van de volgende te splitsen pagina bijhoudt.
    - Het adres gevormd door de d eindbits wordt vergeleken met p.
    - Als $d < p$ dan is de gezochte pagina reeds gesplitst en moeten $d+1$ eindbits gebruikt worden. Anders volstaan d bits.
    - De sleutel kan in de pagina binair of lineair gezocht worden.
  - **Toevoegen.**
    - Eerst wordt de juiste pagina gelokaliseerd.
    - Als de pagina vol zit moet ze gesplitst worden.
    - Splitsen gebeurt sequentieël zodat $p = 0$ in het begin.
    - p wordt met één verhoogd tot alle 2^d pagina's gesplitst zijn.
    - De waarde van d wordt dan verhoogd met één, en p wordt terug 0.
    - Als pagina p gesplitst wordt, is het adres van de nieuwe pagina $p + 2^d$.
  - **Verwijderen.**
    -

Hoofdstuk 4

Meerdimensionale gegevensstructuren

- Gegevens met meer dan één sleutel zijn meerdimensionaal.
- Gegevensstructuren moeten toelaten om op al die sleutels, of in een bereik van meerdere sleutels te zoeken.
- De meeste gegevensstructuren zijn efficiënt voor een klein aantal dimensies.
- De gegevens worden zo gemodelleert zodat ze een geometrische structuur vormen.
- Elke sleutel is een punt in een meerdimensionale Euclidische ruimte.
- Een meerdimensionaal punt zoeken is een speciaal geval van zoeken van alle punten in een meerdimensionale hyperrechthoek.
- Notatie:
  - Het aantal punten is n.
  - Het aantal dimensies is k.

4.1 Projectie

- Per dimensie wordt er een gegevensstructuur (bv gelinkte lijst) bijgehouden die de gesorteerde punten volgens die dimensie bijhoudt.
- Elk punt wordt dus geprojecteerd op elke dimensie.
- Zoeken in een hyperrechthoek gebeurt door een dimensie te kiezen en alle punten te zoeken die voor die dimensie binnen de hyperrechthoek liggen.
- Deze methode werkt als de zoekrechthoek een zijde heeft die de meeste punten uitsluit.
- De **gemiddelde performantie** is $O(n^{1-\frac{1}{k}})$.

4.2 Rasterstructuur

- De zoekruimte wordt verdeelt met behulp van een raster.
- Voor elk rastergebied (een hyperrechthoek) wordt een gelinkte lijst bijgehouden met de punten die erin liggen.
- De punten vinden die in een hyperrechthoek liggen komt neer op het vinden van de rastergebieden die overlappen, en welke van de punten in hun gelinkte lijsten binnen die rechthoek vallen.
- Het aantal rastergebieden is best een constante fractie van n, zodat het gemiddeld aantal punten in elk rastergebied een kleine constante wordt.

4.3 Quadtrees

- Een quadtree verdeelt de zoekruimte in 2^k hyperrechthoeken, waarvan de zijden evenwijdig zijn met het assenstelsel.
- Deze verdeling wordt opgeslaan in een 2^k-wegsboom: elke knoop staat voor een gebied.
- Een quadtree is niet geschikt voor hogere dimensies: er zouden te veel knopen zijn.
- Deze cursus behandelt enkel twee dimensies en er worden enkel **twee-dimensionale punten** opgeslaan.

4.3.1 Point quadtree

- Elke inwendige knoop bevat een punt, waarvan de coördinaten de zoekruimte opdelen in vier rechthoeken.
  - Elk (deel)zoekruimte is de wortel van een deelboom die alle punten in de overeenkomstige rechthoek bevat.
- Woordenboekoperaties:
  - **Zoeken en toevoegen.**
    - Het zoekpunt wordt telkens vergeleken met de punten van de opeenvolgende knopen.
    - Als het zoekpunt niet aanwezig is, eindigt de zoekoperatie in een ledig deelgebied, maar kan het punt wel toegevoegd worden als inwendige knoop.
    - De structuur van een point quadtree is afhankelijk van de toevoegvolgorde, maar is in het gemiddelde geval $O(\lg n)$. In het slechtste geval is het $O(n)$.
  - **Toevoegen als de gegevens op voorhand gekend zijn.**
    - Er kan voor gezorgd worden dat geen enkel deelgebied meer dan de helft van de punten van die van zijn ouder bevat.
    - De punten worden lexicografisch geranschikt en de wortel is de mediaan.
    - Alle punten voor de mediaan vallen dan in twee van zijn deelbomen, deze erachter in de andere twee.
    - Bij elk kind gebeurt hetzelfde.
    - Deze constructie is $O(n \lg n)$.
  - **Verwijderen.**
    - Een punt verwijderen zorgt ervoor dat een deelboom geen ouder meer heeft.
    - Om dit op te lossen worden alle punten in die deelboom opnieuw toegevoegd aan de boom.

4.3.2 PR quadtree

- Point-region quadtree.
- De zoekruimte **moet een rechthoek zijn**.
  - De zoekruimte kan gegeven worden.
  - De zoekruimte kan ook bepaald worden als de kleinste rechthoek die alle punten omvat.
- Elke knoop verdeelt de zoekruimte in vier **gelijke rechthoeken**.
- De opdeling loopt door tot dat elk deelgebied nog één punt bevat.
- Inwendige knopen bevatten geen punten.
- Woordenboekoperaties:
  - **Zoeken.**
    - De opeenvolgende punten vanuit de wortel worden gebruikt om de rechthoek te vinden waarin het punt zou moeten liggen.
  - **Toevoegen.**
    - Als de gevonden rechthoek geen punt bevat kan het punt toegevoegd worden.
    - Als de gevonden rechthoek wel een punt bevat, moet deze rechthoek opnieuw opgesplitst worden tot elk van de punten in een eigen gebied ligt.
  - **Verwijderen.**
    - Een punt verwijderen kan ervoor zorgen dat een deelgebied ledig wordt.
    - Als er nog slechts 1 punt zit in één van de vier deelgebieden, kunnen deze deelgebieden samengevoegd worden.
- De vorm van een PR quadtree is wel onafhankelijk van de toevoegvolgorde.
- Er is geen verband tussen de hoogte h en het aantal opgeslagen punten n omdat een PR quadtree nog steeds onevenwichtig kan uitvallen.
- Er is wel een verband tussen de hoogte h en de kleinste afstand a tussen twee zoekpunten.
  - Stel z de grootste zijde van de zoekruimte.
  - De grootste zijde van een gebied op diepte d is dan $\frac{z}{2^d}$.
  - De maximale afstand tussen twee punten in dat gebied is de lengte van de diagonaal in dat gebied

$$
\begin{aligned}
\sqrt{\left(\frac{z}{2^d}\right)^2 + \underset{k-2}{\cdots} + \left(\frac{z}{2^d}\right)^2} &= \sqrt{k\left(\frac{z}{2^d}\right)^2} \\
&= \frac{z\sqrt{k}}{2^d}
\end{aligned}
$$

  - Op elke diepte d is

$$
\begin{aligned}
a &\leq \frac{z\sqrt{k}}{2^d} \\
2^d &\leq \frac{z\sqrt{k}}{a} \\
d &\leq \lg\left(\frac{z\sqrt{k}}{a}\right) \\
d &\leq \lg\left(\frac{z}{a}\right) + \lg(\sqrt{k}) \\
d &\leq \lg\left(\frac{z}{a}\right) + \frac{\lg k}{2}
\end{aligned}
$$

  - De hoogte h is de maximale diepte van een inwendige knoop plus één:

$$h \leq d + 1 \leq \lg \left(\frac{z}{a} \right) + \frac{\lg k}{2} + 1$$

- Performantie:
  - Op elk niveau bedekken de gebieden van de inwendige knopen de verzameling punten, en al deze gebieden bevatten punten.
  - Per niveau is het aantal inwendige knopen $O(n)$.
  - Het totaal aantal inwendige knopen i in een boom met hoogte h is $O(hn)$.
  - Elke inwendige knoop heeft 4 kinderen, zodat het aantal bladeren $3i + 1$ is.
  - Het aantal knopen is ook $O(hn)$.
  - De constructietijd van de boom is $O(hn)$.

4.4 K-d trees

- Vermijdt de grote vertakkingsgraad van een point quadtree door een binaire boom te gebruiken.
- Elke inwendige knoop bevat een punt, dat de deelzoekruimte slechts opsplitst in één dimensie.
- Opeenvolgende knopen gebruiken opeenvolgende dimensies om te splitsen.
- De opdeling kan doorgang tot slechts één punt in elk gebied is, of men kan vroeger stoppen en gelinkte lijsten bijhouden per gebied.
- Door de (eventueel random) afwisselende dimensies zijn er geen rotaties mogelijk om een dergelijke boom evenwichtig te maken. Daarom wordt verwijderen ook nooit echt gedaan, maar eerder met **lazy deletion**.
- Men kan wel af en toe een deelboom reconstrueren, en dan ook de te verwijderen knopen effectief verwijderen.

Hoofdstuk 5

Samenvoegbare heaps

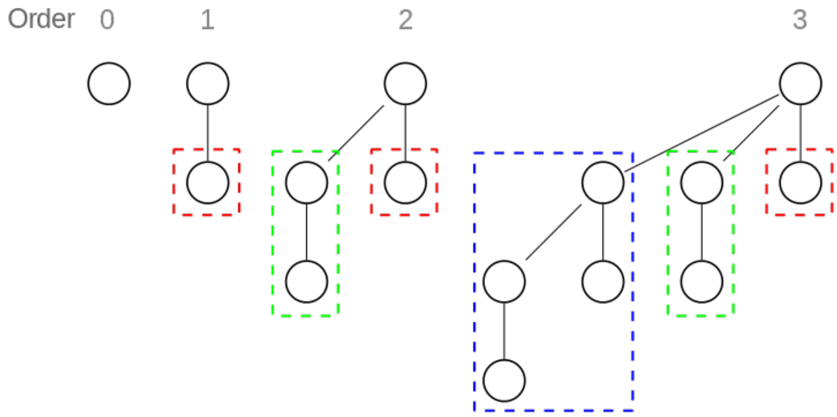
- Een samenvoegbare heap is een heap die geoptimaliseerd zijn om de 'join' operatie uit te voeren.
- De join operatie voegt twee heaps samen, zodat de **heapvoorwaarde** nog steeds geldig is.
- Een lijst van bekende heaps.
  - Leftist heaps.
    - Deze heaps proberen zo onevenwichtig mogelijk te zijn.
    - De linkerkant is diep en de rechterkant ondiep.
    - De operaties zijn efficiënt omdat al het werk in de rechterkant gebeurt.
  - Skew heaps.
    - Gelijkaardig aan een leftist heap, maar er is een vormbeperking.
    - In het slechtste geval kunnen individuele operaties $O(n)$ zijn.
  - Fibonacci heaps.
  - Relaxed heaps

De belangrijke samenvoegbare heaps zijn: **Binomial queues** en **Pairing heaps**.

5.1 Binomiale queues

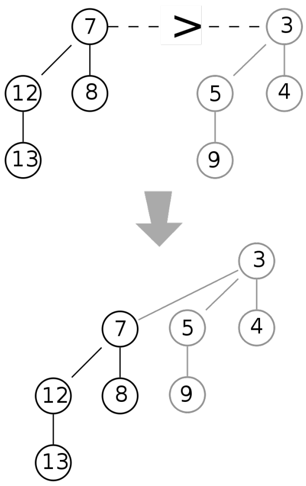
- Bestaat uit bos van binomiaalbomen.
- Binomiaalboom B_n bestaat uit twee binomiaalbomen B_{n-1}. B_0 bestaat uit één knoop.
- De tweede binomiaalboom is de meest linkse deelboom van de wortel van de eerste.
- Een binomiaalboom B_n bestaat uit een wortel met als kinderen $B_{n-1}, ..., B_1, B_0$ (zie figuur 5.1)
- Op diepte d zijn er $\binom{n}{d}$ knopen.
- Voorbeeld: Een prioriteitswachtrij met 13 elementen wordt voorgesteld als $\langle B_3, B_2, B_0 \rangle$.

De operaties op een binomiaalqueue:



Figuur 5.1: Verschillende ordes van binomiaalbomen.

- **Minimum vinden**: Overloop de wortel van elke binomiaalboom. Het minimum kan ook gewoon bijgehouden worden.
- **Samenvoegen**: Tel de bomen met dezelfde hoogte bij elkaar op, $B_h + B_h = B_{h+1}$. Maak de wortel met de grootste sleutel het kind van deze met de kleinste.

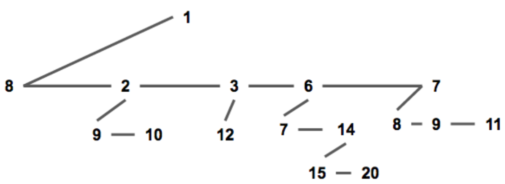


Figuur 5.2: Hier worden twee binomiaalbomen van orde 3 samengevoegd. De boom met de waarde 7 voor de wortel wordt het linkerkind van de boom met waarde 3 voor de wortel. Het wordt een boom van orde 4.

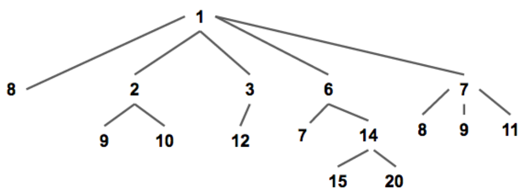
- **Toevoegen**: Maak een triviale binomiaalqueue met één knoop en voeg deze samen met de andere binomiaalqueue.
- **Minimum verwijderen**: Zoek binomiaalboom B_k met het kleinste wortelelement. Verwijder deze uit de binomiaalqueue. De deelbomen van deze binomiaalboom vormen een nieuw binomiaalbos die samengevoegd kan worden met de originele heap.

5.2 Pairing heaps

- Een algemene boom waarvan de sleutels voldoen aan de heapvoorwaarde.
- Elke knoop heeft een wijzer naar zijn linkerkind en rechterbroer (figuur 5.3 en 5.4). Als verminderen van prioriteit moet ondersteund worden, heeft elke knoop als linkerkind een wijzer naar zijn ouder en als rechterkind een wijzer naar zijn linkerkind.



Figuur 5.3: Een pairing heap.



Figuur 5.4: Dezelfde pairing heap, maar in boomvorm.

De operaties op een pairing heap.

- **Samenvoegen**: Verlijk het wortelelement van beide heaps. De wortel met het grootste element wordt het linkerkind van deze met het kleinste element.
- **Toevoegen**: Maak een nieuwe pairingheap met één element, en voeg deze samen met de oorspronkelijke heap.
- **Prioriteit wijzigen**: De te wijzigen knoop wordt losgekoppeld, krijgt de prioriteitwijziging, en wordt dan weer samengevoegd met de oorspronkelijke heap.
- **Minimum verwijderen**: De wortel verwijderen levert een collectie van c heaps op. Voeg deze heaps van links naar rechts samen in $O(n)$ of voeg eerst in paren toe, en dan van rechts naar links toevoegen in geamortiseerd $O(\lg n)$.
- **Willekeurige knoop verwijderen**: De te verwijderen knoop wordt losgekoppeld, zodat er twee deelheaps onstaan. Deze twee deelheaps worden samengevoegd.

Deel II

Grafen II

Hoofdstuk 6

Toepassingen van diepte-eerst zoeken

- Notatie:
  - Het aantal knopen is n.
  - Het aantal verbindingen is m.

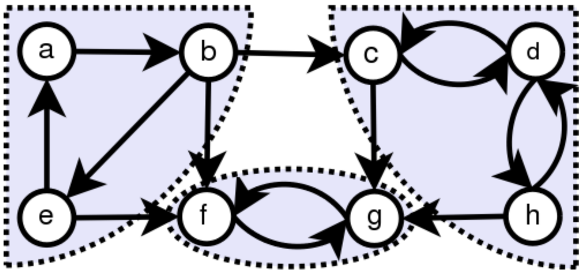
6.1 Enkelvoudige samenhang van grafen

6.1.1 Samenhangende componenten van een ongerichte graaf

- Een **samenhangende ongerichte graaf** is een graaf waarbij er een weg bestaat tussen elk paar knopen.
- Een **niet samenhangende ongerichte graaf** bestaat dan uit zo groot mogelijke samenhangende componenten.
- Diepte-eerst zoeken vindt alle knopen die met wortel van de diepte-eerst boom verbonden zijn.
  - Een ongerichte graaf is samenhangend wanneer die boom alle knopen bevat.
  - Als er meerdere bomen zijn, vormen deze de samenhangende componenten.
  - Diepte-eerst zoeken is $\Theta(n + m)$.

6.1.2 Sterk samenhangende componenten van een gerichte graaf

- Een **sterk samenhangende gerichte graaf** is een graaf waarbij er een weg tussen elk paar knopen in beide richtingen (niet perse dezelfde verbindingen) bestaat (cfr. figuur 6.1). Een graaf die niet sterk samenhangend is, bestaat uit zo groot mogelijke sterk samenhangende componenten.
- Een **zwak samenhangende gerichte graaf** is een graaf die niet sterk, maar toch samenhangend is indien de richtingen buiten beschouwing gelaten worden.
- Sommige algoritmen gaan ervan uit de een graaf sterk samenhangend is. Men moet dus eerst deze componenten bepalen, meestal via een **componentengraaf** die:



Figuur 6.1: De componenten van een sterk samenhangende graaf. Merk op dat in 'in beide richtingen' enkel betrekken heeft tot de richtingen die er zijn. De knopen van het component $A - B - E$ bevat maar één richting maar is wel sterk samenhangend, omdat er een weg bestaat tussen elk paar knopen in beide richtingen, maar hier is er maar één richting en is ook geldig. De knopen van het component $C - D - H$ is wel een geval van beide richtingen.

- een knoop heeft voor elk sterk samenhangend component,
- en een verbinding van knoop a naar knoop b indien er in de originele graaf een verbinding van één van de knopen van a naar één van de knopen van b is.

- De componentgraaf bevat geen lussen. Mocht dit wel zo zijn, zouden de knopen die de lus veroorzaken zich in hetzelfde sterk samenhangende component bevinden.
- De sterk samenhangende componenten **in een gerichte graaf** kunnen bekomen worden met behulp van diepte-eerst zoeken (Kosaraju's Algorithm):
  1. Stel de omgekeerde graaf op, door de richting van elke verbinding om te keren.
  2. Pas diepte-eerst zoeken toe op deze omgekeerde graaf, waarbij de knopen in postorder genummered worden.
  3. Pas diepte-eerst zoeken toe op de originele graaf, met als startknoop steeds de resterende knoop met het hoogste postordernummer. Het resultaat is een diepte-eerst bos, waarvan de bomen de knopen bevatten die elk sterk samenhangende componenten zijn.
- We willen aantonen dat de wortel van elke boom in beide richtingen verbonden is met elk van zijn knopen. Op die manier is elke andere knoop in beide richtingen verbonden door de wortel en klopt het algoritme.
  - Via de boomtakken is er een weg van de wortel w naar elk van de knopen u in de boom.
  - Er is dan ook een weg van u naar w in de omgekeerde graaf.
  - De wortel w is altijd een voorouder van u in een diepte-eerst boom van de omgekeerde graaf.
  - Hieruit volgt dat er een weg van w naar u bestaat in de omgekeerde graaf.
  - Er is dan ook een weg van u naar w in de originele graaf.
- Diepte-eerst zoeken is $\Theta(n + m)$ voor ijle en $\Theta(n^2)$ voor dichte grafen. Het omkeren van de graaf is ook $\Theta(n + m)$ voor ijle en $\Theta(n^2)$ voor dichte grafen.

6.2 Dubbele samenhang van ongerichte grafen

Twee definities:

- Een **brug** is een verbinding dat, indien deze wordt weggenomen, de graaf in twee deelgrafen opsplitst. Een graaf zonder bruggen noemt men **dubbel lijnsamenhangend**; als er tussen elk paar knopen minstens twee onafhankelijke wegen bestaan, dan is een graaf zeker dubbel lijnsamenhangend.
- Een **scharnierpunt** is een knoop dat, indien deze wordt weggenomen, de graaf in ten minste twee deelgrafen opsplitst. Een graaf zonder scharnierpunten noemt men **dubbel knoopsamenhangend** (of dubbel samenhangend). Een graaf met scharnierpunten kan onderverdeeld worden in dubbel knoopsamenhangende componenten. Als er tussen elk paar knopen twee onafhankelijke wegen bestaan, dan is de graaf dubbel knoopsamenhangend.

Scharnierpunten, dubbel knoopsamenhangende componenten, bruggen en dubbel lijnsamenhangende componenten kunnen opnieuw via diepte-eerst zoeken gevonden worden:

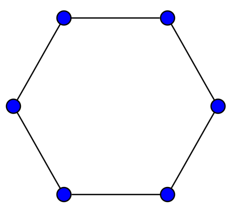
1. Stel de diepte-eerst boom op, waarbij de knopen in preorder genummerd worden.
2. Bepaal voor elke knoop u de laagst genummerde knoop die vanuit u kan bereikt worden via een weg bestaande uit nul of meer dalende boomtakken gevolgd door één terugverbinding.
3. Indien alle kinderen van een knoop op die manier een knoop kunnen bereiken die hoger in de boom ligt dan hemzelf, dan is de knoop zeker niet samenhangend. De wortel is een scharnierpunt indien hij meer dan één kind in heeft. ToDo: hoe bruggen vinden?

Diepte-eerst zoeken is $\Theta(n + m)$ voor ijle en $\Theta(n^2)$ voor dichte grafen.

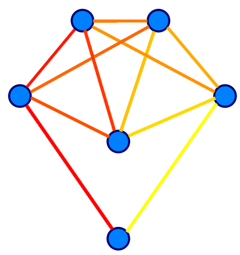
6.3 Eulercircuit

Een eulercircuit is een **gesloten pad** (begin- en eindknoop is dezelfde) in een graaf die **alle verbindingen** éénmaal bevat.

6.3.1 Ongerichte grafen



(a) Een Eulegraaf met 6 knopen en 6 verbindingen.



(b) Een Eulergraaf waarbij de volgorde van de verbindingen die het Eulercircuit opmaken gekleurd worden van rood naar geel.

- Een Eulergraaf is een graaf met een eulercircuit.
  - Heeft als vereiste dat er geen knopen zijn met oneven graad (eigenschap 2).
- Volgende eigenschappen zijn equivalent.
  1. Een samenhangende graaf G is een Eulergraaf.

   - Dit volgt uit de derde eigenschap.
   - Stel dat L één van de lussen van G is.
   - Als L een Eulercircuit is dan is G een Eulergraaf.
   - Zoniet bestaat er een andere lus L' die een gemeenschappelijke knoop k heeft met L.
   - Aangezien elke verbinding tot één lus behoort, kunnen deze twee lussen bij knoop k samengevoegd worden.
   - Uiteindelijk bekomen we een Eulercircuit.
2. De graad van elke knoop van G is even.
   - Dit volgt uit de eerste eigenschap.
   - Als een knoop k voorkomt op een Eulercircuit, draagt dat twee bij tot zijn graad.
     - Er is een verbinding nodig om de knoop te bereiken, en ook een verbinding om de knoop te verlaten.
     - Elke verbinding komt precies éénmaal voor op een Eulercircuit.
3. De verbindingen van G kunnen onderverdeeld worden in lussen, waarbij elke verbinding slechts behoort tot één enkel lus.
   - Dit volgt uit de tweede eigenschap.
   - Stel dat er n knopen zijn.
   - Er zijn minstens n verbindingen (want moet terug in startknoop eindigen).
     - Eenvoudigste Eulergraaf is een cyclusgraaf (cfr. Figuur 6.2a).
   - G bevat dan minstens één lus.
   - Als de lus verwijdert wordt, blijft er een niet noodzakelijke samenhangende graaf H over waarvan alle knoopgraden nog steeds even zijn.
   - Elk van de samenhangende componenten van H kan opnieuw in lussen onderverdeeld worden.

- Het **algoritme van Hierholzer** geeft een Eulercircuit voor een Eulergraaf.
  - De eerste lus L begint bij een willekeurige knoop. Er worden willekeurig verbindingen gekozen tot dat de knoop opnieuw bereikt wordt.
  - De volgende lus L' begint bij één van de knopen van L waarvan nog niet alle verbindingen doorlopen zijn. Opnieuw worden willekeurig verbindingen gekozen tot de knoop opnieuw bereikt wordt.
  - Er worden lussen gegenereerd zolang niet alle verbindingen van een knoop opgebruikt zijn.

6.3.2 Gerichte grafen

- Een Eulercircuit in een gerichte graaf is slechts mogelijk als de graaf een sterk samenhangende Eulergraaf is.
- De constructie verloopt analoog aan de ongerichte Eulergraaf.

Hoofdstuk 7

Kortste afstanden II

- Traditioneel kortste afstanden tussen twee knopen: algoritme van Dijkstra.
- Probleem:
    - Dijkstra gebruikt het feit dat indien een pad naar A → C bestaat met kost $K_{A,C}$, er geen korter pad A → B → C kan zijn met kost $K_{A,B} + K_{A,C}$, daarom is het algoritme performant, maar er mogen dus geen negatieve verbindingen zijn. Indien $K_{A,B}$ negatief zou zijn dan klopt Dijkstra niet want dan

$$K_{A,B} + KA, C > K_{A,C}$$

- Volgende algoritmen hebben enkel betrekking tot **gerichte grafen**.

7.1 Kortste afstanden vanuit één knoop

7.1.1 Algoritme van Bellman-Ford

Dit algoritme werkt voor verbindingen met negatieve gewichten, iets wat het algoritme van Dijkstra niet kon. Het algoritme is dan natuurlijk ook trager.

- Werkt voor negatieve verbindingen.
- Geen globale kennis nodig van heel het netwerk, zoals bij Dijkstra, maar slechts enkel de buren van een bepaalde knoop. Daarom gebruiken routers Bellman-Ford (distance vector protocol).

! Zal niet stoppen indien er een negatieve lus in de graaf zit, aangezien het pad dan zal blijven dalen tot $-\infty$.

Het algoritme berust op een belangrijke eigenschap: Indien een graaf geen negatieve lussen heeft, zullen de kortste wegen evenmin lussen hebben en hoogstens $n-1$ verbindingen bevatten. Hieruit kan een recursief verband opgesteld worden tussen de kortste wegen met maximaal k verbindingen en de kortste wegen met maximaal $k-1$ verbindingen.

$$d_i(k) = \min(d_i(k-1), \min_{j \in V}(d_j(k-1) + g_{ji}))$$

met

- $d_i(k)$ het gewicht van de kortste weg met maximaal k verbindingen vanuit de startknoop naar knoop i,
- g_{ji} het gewicht van de verbinding (j, i),
- $j \in V$ is elke knoop j.

Er bestaan twee goede implementaties:

1. - Niet nodig om in elke iteratie alle verbindingen te onderzoeken. Als een iteratie de voorlopige kortste afstand tot een knoop niet aanpast, is het zinloos om bij de volgende iteratie de verbindingen vanuit die knoop te onderzoeken.
   - Plaats enkel de knopen waarvan de afstand door de huidige iteratie gewijzigd werd in een wachtrij.
   - Enkel de buren van deze knopen worden in de volgende iteratie getest.
   - Elke buur moet, indien hij getest wordt en nog niet in de wachtrij zit, ook in de wachtrij gezet worden.
2. - Gebruik een deque in plaats van een wachtrij.
   - Als de afstand van een knoop wordt aangepast, en als die knoop reeds vroeger in de deque zat, dant voegt men vooraan toe, anders achteraan.
   - Kan in bepaalde gevallen zeer inefficiënt uitvallen.

7.2 Kortste afstanden tussen alle knopenparen

- Voor dichte grafen → Floyd-Warshall (Algoritmen I).
- Voor ijle grafen → Johnson.

7.2.1 Het algoritme van Johnson

- Maakt gebruik van Bellman-Ford en Dijkstra.
- Omdat we Dijkstra gebruiken, moet elk gewicht positief worden.
  1. Breidt de graaf uit met een nieuwe knoop s, die verbindingen van gewicht nul krijgt met elke andere knoop.
  2. Voer Bellman-Ford uit op de nieuwe graaf om vanuit s de kortste afstand d_i te bepalen tot elke originele knoop i.
  3. Het nieuwe gewicht $\hat{g}_{ij}$ van een oorspronkelijke verbinding g_{ij} wordt gegeven door:
  $$\hat{g}_{ij} = g_{ij} + d_i - d_j$$
- Het algoritme van Dijkstra kan nu worden toegepast op elke originele knoop, die alle kortste wegen zullen vinden. Om de kortste afstanden te bepalen moeten de originele gewichten opgeteld worden op deze wegen.
- Dit algoritme is $O(n(n + m) \lg n)$ want:
  - Graaf uitbreiden is $\Theta(n)$.
  - Bellman-Ford is $O(nm)$.
  - De gewichten aanpassen is $\Theta(m)$.
  - n maal Dijkstra is $O(n(n + m) \lg n)$. Dit is de belangrijkste term, al de andere termen mogen verwaarloosd worden.

7.3 Transitieve sluiting

Sluiting = algemene methode om één of meerdere verzamelingen op te bouwen. ('als een verzameling deze gegevens bevat, dan moet ze ook de volgende gegevens bevatten').

- **Fixed point**: Een sluiting wordt fixed point genoemd omdat op een bepaald moment verdere toepassing niets meer verandert, $f(x) = x$.
- **Least fixed point**: De kleinste x zoeken zodat $f(x) = x$ voldaan wordt.

Transitieve sluiting = 'Als (a, b) en (b, c) aanwezig zijn dan moet ook (a, c) aanwezig zijn.'

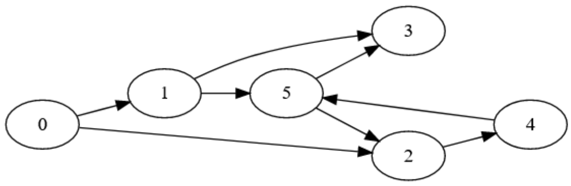
- Transitieve sluiting van een gerichte graaf is opnieuw een gerichte graaf, maar:
  - er wordt een nieuwe verbinding van i naar j toegevoegd indien er een weg bestaat van i naar j in de oorspronkelijke graaf.
- 3 algoritmen:
  1. **Diepte-of breedte-eerst zoeken**:
     - Spoor alle knopen op die vanuit een startknoop bereikbaar zijn en herhaal dit met elke knoop.
     - Voor ijle grafen $\rightarrow \Theta(n(n + m))$.
     - Voor dichte grafen $\rightarrow \Theta(n^3)$.
  2. **Met de componentengraaf**:
     - Interessant wanneer men verwacht dat de transitieve sluiting een dichte graaf zal zijn, want dan zijn veel knopen onderling bereikbaar, zodat er een beperkt aantal sterk samenhangende componenten zijn. Die kunnen in $\Theta(n + m)$ bepaald worden.
     - Maak dan de componentengraaf (kan in $O(n + m)$).
     - Als nu blijkt dat component j beschikbaar is vanuit component i, dan zijn alle knopen van j beruikbaar vanuit knopen van i.
  3. **Het algoritme van Warshall**:
     - Maak een reeks opeenvolgende $n \times n$ matrices $T^{(0)}, T^{(1)}, ..., T^{(n)}$ die logische waarden bevatten.
     - Element $t_{ij}^{(k)}$ duidt aan of er een weg tussen i en j met mogelijke intermediaire knopen $1, 2, ..., k$ bestaat.
     - Bepalen opeenvolgende matrices:

$$t_{ij}^{(0)} = \begin{cases} \text{onwaar} & \text{als } i \neq j \text{ en } g_{ij} = \infty \\ \text{waar} & \text{als } i = j \text{ of } g_{ij} < \infty \end{cases}$$

       en

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \text{ OF } \left(t_{ik}^{(k-1)} \text{ EN } t_{kj}^{(k-1)}\right) \qquad \text{voor } 1 \leq k \leq n$$

     - $T^{(n)}$ is de gezochte burenmatrix.
     - Alle berekeningen kunnen in dezelfde tabel T gebeuren. Er moet geen plaats voorzien zijn voor andere tabellen.
     - **Voorbeeld**



Figuur 7.1: Een gerichte graaf met 6 knopen.

- De initieële tabel $T^{(0)}$ is gewoon een kopie van de burenlijst van de graaf.

$$T^{(0)} = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

- De tabel $T^{(1)}$ geeft een uitbreiding van $T^{(0)}$, waarbij knoop 1 een intermediaire knoop mag zijn in een weg naar knopen knopen. Het is logisch dat enkel knopen die 1 als buur hebben een nieuwe weg kunnen vinden.

$$T^{(1)} = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

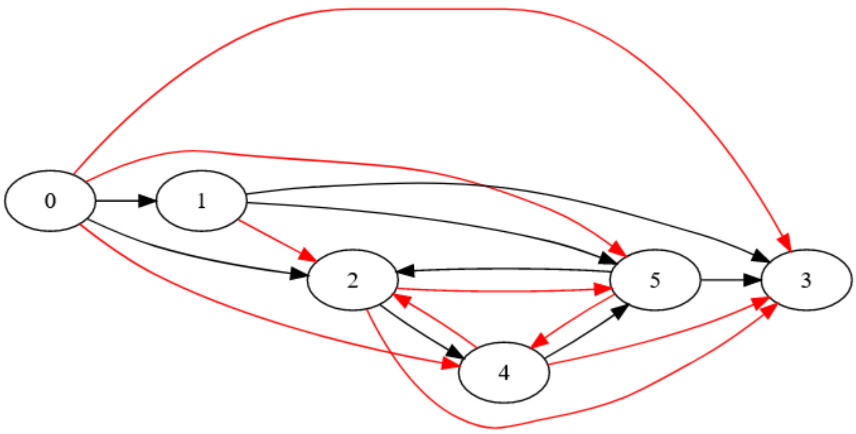
- De tabel $T^{(2)}$ geeft een uitbreiding van $T^{(1)}$, waarbij knoop 2 een intermediaire knoop mag zijn in een weg naar twee knopen. Het is logisch dat enkel knopen die 2 als buur hebben (in de nieuwe matrix $T^{(1)}$) een nieuwe weg kunnen vinden.

$$T^{(2)} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

- Via diezelfde redenering wordt uiteindelijk $T^{(5)}$ bekomen, die de burenmatrix voorstelt van de transitieve sluiting.

$$T^{(5)} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

- Figuur 7.2 toont de transitieve sluiting.



Figuur 7.2: De transitieve sluiting van de graaf op figuur 7.1. De transitieve sluiting bevat dezelfde verbindingen als de graaf (zwarte verbindingen) en ook de nieuwe verbindingen die de transitieve eigenschap vastleggen (rode verbindingen).

Hoofdstuk 8

Stroomnetwerken

- Eigenschappen van een **stroomnetwerk**:
  - Is een gerichte graaf.
  - Heeft twee speciale knopen:
    1. Een **producent**.
    2. Een **verbruiker**.
  - Elke knoop van de graaf is bereikbaar vanuit de producent.
  - De verbruiker is vanuit elke knoop bereikbaar.
  - De graaf mag lussen bevatten.
  - Elke verbinding heeft een capaciteit.
  - Alles wat in een knoop toestroomt, moet ook weer wegstromen. De stroom is dus **conservatief**.

8.1 Maximalestroomprobleem

- Zoveel mogelijk materiaal van producent naar verbruiker laten stromen, zonder de capaciteiten van de verbindingen te overschrijden.
- Wordt opgelost via de methode van **Ford-Fulkerson**.
  - Een iteratieve methode. Het wordt een **methode** genoemd en geen algoritme omdat de implementatie van de vergrotende paden ontbreekt.
  - Bij elke iteratie neemt de nettostroom vanuit de producent toe, tot het maximum bereikt wordt.
- Elke verbinding (i, j) heeft:
  - een capaciteit $c(i, j)$;
    - Als er geen verbinding is tussen twee knopen, dan wordt er toch een verbinding gemaakt met capaciteit 0. Dit dient om wiskundige notaties te vereenvoudigen.
  - de stroom $s(i, j)$ die er door loopt, waarbij $0 \leq s(i, j) \leq c(i, j)$.
- De totale nettostroom f van alle knopen K uit producent p in de graaf is dan

$$f = \sum_{j \in K} (s(p, j) - s(j, p))$$

  - $s(p, j)$ is de uitgaande stroom vanuit de producent p naar knoop j.
  - $s(j, p)$ is de totale inkomende stroom van elke knoop j naar producent p.

- De verzameling van stromen voor alle mogelijke knopenparen in beide richtingen wordt een **stroomverdeling** genoemd.
- De verzameling mogelijke stroomtoenamen tussen elk paar knopen wordt het **restnetwerk** genoemd.
  - Het restnetwerk bevat dezelfde knopen, maar behoudt enkel de verbindingen die meer stroom kunnen doorlaten.
  - Een verbinding van knoop i naar knoop j wordt opgenomen als:
    - $s(i, j) < c(i, j)$, en/of
    - er loopt stroom over de verbinding (j, i) die kleiner kan gemaakt worden.
  - Een verbinding in het restnetwerk krijgt de capaciteit $c_r(i, j) = c(i, j) - s(i, j) + s(j, i)$.
  - De verindingen van het restnetwerk vormen niet noodzakelijk een deelverzameling van de originele verbindingen:
    - Stel dat er geen verbinding $(i, j)(c(i, j) = 0)$ is, maar wel een verbinding (j, i) waarover een positieve stroom loopt.
    - Het restnetwerk krijgt toch een verbinding (i, j) omdat de stroom over (j, i) eventueel nog kleiner kan gemaakt worden.
- In het restnetwerk wordt de **vergrotende weg** van producent naar verbruiker gezocht.
  - Dit is een enkelvoudige weg zonder lus van producent naar verbruiker.
  - Elke verbinding op die weg heeft een positieve restcapaciteit, en kan nog meer stroom doorlaten.
  - Er is dan extra stroom mogelijk gelijk aan de kleinste restcapaciteit op die weg.
  - De stroom in de overeenkomstige verbindingen in het eigenlijke stroomnetwerk wordt hiermee aangepast.
- Is de methode van Ford-Fulkerson correct?
  - Een **snede** (P, V) van een samenhangende graaf is een verzameling verbindingen die de knopenverzameling in twee niet-ledige stukken P en V verdeelt.
    -
    - Een verbinding (i, j) zit in (P, V) als $i \in P$ en $j \in V$ of $i \in V$ en $j \in P$.
    - Bij stroomnetwerken zijn nuttige sneden waarbij de producent p tot P behoort en de verbuiker v tot V.
    - De capaciteit $c(P, V)$ van de snede wordt gedefinieerd als de som van alle capaciteiten $c(i, j)$, met i in P en j in V.
    - De nettostroom $f(P, V)$ van de snede is de som van alle voorwaartse stromen $s(i, j)$, min de som van alle achterwaartse stromen $s(j, i)$, met i in P en j in V.
  - De conservatieve eigenschap van een stroomnetwerk heeft als gevolg dat de netwerkstroom f gelijk is aan de nettostroom $f(P, V)$ van elke mogelijke snede.
    - De stroom van het netwerk vanuit de producent p werd reeds gedefinieerd

$$f = \sum_{j \in K} (s(p, j) - s(j, p))$$

    - In alle andere knopen i van P is de stroom conservatief:

$$\sum_{j \in K} (s(i, j) - s(j, i)) = 0$$

- Gecombineerd, voor alle knopen in P, is dit dan

$$f = \sum_{i \in P} \sum_{j \in K} (s(i,j) - s(j,i))$$

- Voor alle knopen j uit P komt elke stroom $s(i,j)$ tweemaal voor in deze dubbele som, met tegengesteld teken.
- Er blijven enkel nog knopen j uit $V = K \backslash P$ over, en dat is de nettostroom van de snede (P, V)

$$f = \sum_{i \in P} \sum_{j \in V} (s(i,j) - s(j,i)) = f(P,V)$$

- De **max-flow min-cut** stelling zegt dat f maximaal wordt als het overeenkomstige restnetwerk geen vergrotende weg meer heeft.
  - De volgende eigenschappen zijn equivalent:
    1. De netwerkstroom f is maximaal.
    2. Er is geen vergrotende weg meer te vinden in het restnetwerk.
    3. De netwerkstroom f is gelijk aan de capaciteit van *een snede* in de oorspronkelijke graaf.
  - ??

- **Hoe** moet nu de **vergrotende weg bepaald** worden?
  - **Performantie afhankelijk van de capaciteiten**
    De performantie van volgende implementaties zijn afhankelijk van de graaf (n en m) en door de grootte van de capaciteiten.
    1. - Stel dat alle capaciteiten geheel zijn, en C is de grootste capaciteit.
       - De maximale netwerkstroom is dan $O(nC)$.
       - Bij elke iteratie van Ford-Fulkerson zal de stroomtoename langs een vergrotende weg ook geheel zijn.
       - Het aantal iteraties is $O(nC)$.
       - Het restnetwerk bepalen is $O(m)$ en daarin een vergrotende weg vinden met diepte-eerst of breedte-eerst zoeken is ook $O(m)$.
       - De totale performantie is $\mathbf{O(mnC)}$.
    2. - Neem steeds de vergrotende weg die de grootste stroomtoename mogelijk maakt.
       - Dit kan door een kleine wijziging aan het algoritme van Dijkstra (kortste afstanden vervangen door grootste capaciteiten).
       - Het aantal iteraties is $O(m \lg C)$ (zonder bewijs).
       - Elke iteratiestap is $O(m \lg n)$ (van Dijkstra).
       - De totale performantie is $\mathbf{O(m^2 \lg n \lg C)}$
    3. - Stel een cutoff $c = 2^{\lfloor \lg C \rfloor}$ in.
       - Een vergrotende weg vinden die een stroomtoename van minstens c eenheden toelaat, of vaststellen dat die er niet is, kan in $O(m)$.
       - Als er geen vergrotende weg gevonden is, dan is de minimale snedecapaciteit van het restnetwerklager dan mc.
       - c wordt in elke fase gehalveerd, tot dat uiteindelijk $c = 1$. Hiervoor zijn er $O(m \lg C)$ iteraties nodig.
       - De totale performantie is $\mathbf{O(m^2 \lg C)}$
  - **Performantie onafhankelijk van de capaciteiten**
    - Als de vergrotende weg het minimum aantal verbindingen heeft, dan stijgt de lengte van de vergrotende weg na hoogstens m iteraties.

    - De maximale lengte is $n - 1$, zodat er $O(nm)$ iteraties nodig zijn.
    - In elke iteratie wordt nu breedte-eerst zoeken gebruikt en is $O(m)$.
    - De totale performantie is $\mathbf{O(nm^2)}$

- Alle algoritmen die een maximale stroom zoeken via vergrotende wegen hebben als nadeel dat die stroomtoename langs de hele weg van p naar v moet gebeuren, wat in het slechtste geval $O(n)$ vereist.
- Een meer recentere techniek is de **preflow-push** methode, die de stroomtoename van een weg opsplitst in de stroomtoename langs zijn verbindingen.
  - De preflow duidt op het feit dat er meer stroom kan binnenkomen in een knoop dat er buiten gaat.
  - Knopen met een positief overschot heten 'actief'.
  - Zolang er actieve knopen zijn, voldoet de oplossing niet.
  - Er wordt willekeurig een actieve knoop geselecteerd, en trachten om zijn overschot weg te werken via zijn buren.
  - Als er geen actieve knopen zijn, voldoet de stroom aan de conservatieve eigenschap, en is bovendien maximaal (zonder bewijs).
  - Enkele performanties van deze methode, in vergelijking met Ford-Fulkerson:
    - De eenvoudigste implementatie haalt een performantie van $O(n^2m)$.
    - Het FIFO preflow-push algoritme selecteert de actieve knopen met een wachtrij, en is $O(n^3)$.
    - Het highest-label preflow-push algoritme neemt de actieve knoop die het verst van v ligt, en is $O(n^2\sqrt{m})$.
    - Het excess-scaling algoritme duwt stroom van een actieve knoop met voldoend groot overschot naar een knoop met een voldoend klein overschot, en is $O(nm + n^2 \lg C)$.

8.2 Verwante problemen

Het maximalestroomprobleem kan uitgbreid worden om verwante problemen op te lossen:

1. **Meerdere producenten en verbruikers**
   - Men wil de gezamenlijke nettostroom van alle producten maximaliseren.
   - Dit kan eenvoudig door een nieuw stroomnetwerk aan te maken met twee nieuwe knopen: een totaalproducent en totaalverbruiker.
   - Vanuit de totaalproducent zijn er verbindingen naar alle producenten met oneindige capaciteit.
   - Naar de totaalverbuiker komen er verbindingen toe van alle verbuikers, ook met oneindige capaciteit.
   - De totaalproducent produceert het geheel van alle producenten, en de totaalverbruiker verbruikt alles wat bij de verbuikers samenkomt
2. **Capaciteiten toekennen aan knopen**
   - Men wil capaciteiten toekennen aan knopen.
   - Dit kan ook omgevormd worden tot een normaal stroomnetwerk door elke knoop te dupliceren, en een verbinding te maken tussen elke knoop en zijn duplicant.
   - De capaciteit van de verbinding is dan de knoopcapaciteit.

- Elke inkomende verbinding van de originele knoop blijft bij de knoop.
- Elke uitgaande verbinding komt terecht bij de duplicant.

3. **Een ongericht stroomnetwerk**
   - Een normaal stroomnetwerk verwacht een gerichte graaf.
   - Elke ongerichte verbinding kan vervangen worden door een paar gerichte verbindinge, één in elke richting, en beide verbindinge krijgen de originele capaciteit.
4. **Ondergrenzen toekennen aan verbindingen**
   - Eerst gaat men na of dat een netwerkstroom mogelijk is.
   - Indien ja, wordt die getransformeerd tot een maximale stroom.
   - Dit heeft als praktisch nut dat er voorkomen wordt dat de 'flow' stilstaat.
5. **Meerdere soorten materiaal door de verbindingen**
   - Voor elk soort materiaal is er één producent en ook één verbruiker.
   - In elke knoop is de stroom van elk materiaal apart conservatief.
   - De gezamenlijke stroom van alle materialen door een verbinding mag haar capaciteit niet overschrijden.
6. **Een kost per stroomeenheid**
   - Het **minimalekostprobleem** zoekt niet alleen de maximale stroom, maar bovendien die met de minimale kost.
   - Het maximalestroomprobleem is een specifiek geval van het minimalekostprobleem.

8.2.1 Meervoudige samenhang in grafen

- Definities:
  - Een graaf is **k-voudig knoopsamenhangend** als er tussen elk paar knopen van een graaf k onafhankelijke wegen bestaan **zonder gemeenschappelijke knopen**.
  - Een graaf is **k-voudig lijnsamenhangend** als er tussen elk paar knopen van een graaf k onafhankelijk wegen bestaan **zonder gemeenschappelijke verbindingen**.
- Voor $k < 4$ kunnen knoopsamenhang en lijnsamenhang efficiënt via diepte-eerst zoeken onderzocht worden.
- Voor grotere k moeten stroomnetwerken gebruikt worden.
- Als een maximale netwerkstroom gevonden is, dan is ook de minimale snede gevonden (max-flow min-cut stelling).
- De fundamentele eigenschap van samenhang in een graaf wordt gegeven door de **stelling van Menger**.
  - Vier versies: zowel voor gerichte als ongerichte grafen en zowel voor meervoudige knoopsamenhang als meervoudige lijnsamenhang.
  - Voorbeeld voor een meervoudig lijnsamenhangende gerichte graaf:

    Het minimum aantal verbindingen dat moet verwijderd worden om een knoop v van een gerichte graaf onbereikbaar te maken vanuit een andere knoop p is gelijk aan het maximaal aantal lijnonafhankelijke wegen van p naar v. Hierbij is v geen buur van p.

    - Deze stelling volgt uit de eigenschappen van een stroomnetwerk met eenheidscapaciteiten.
  -

Hoofdstuk 9

Koppelen

- Een **koppeling** in een **ongerichte graaf** is een deelverzameling van de verbindingen waarin elke knoop hoogstens éénmaal voorkomt

9.1 Koppelen in tweeledige grafen

- Een **tweeledige graaf** heeft volgende eigenschappen:
  - Een ongerichte graaf.
  - De knopen kunnen in twee deelverzameling L en R verdeeld worden.
  - Alle verbindingen bevatten als eindknopen steeds één uit L en één uit R.
- Kan bijvoorbeeld gebruikt worden om uit te voeren taken toe te wijzen aan uitvoerders. De verbindingen duiden aan welke taken een uitvoerder aankan.

9.1.1 Ongewogen koppeling

- Een **maximale ongewogen koppeling** is een koppeling met het grootst aantal verbindingen waarbij de verbindingen geen gewichten hebben.
- Er is een nauw verband met een maximale ongewogen koppelinge en de maximale stroom in stroomnetwerken.
- De graaf wordt eerst omgevormd naar een stroomnetwerk:
  - Maak van de ongerichte graaf eerst een gerichte graaf, door de originele verbindingen te vervangen door verbindingen van L naar R.
  - Voeg een producent p in, die naar alle knopen van L verbonden wordt.
  - Voeg een verbruiker v in, waarnaar alle knopen uit R naar verbonden worden.
  - Stel alle capaciteiten in op 1.
  - De maximale stroom zoeken in dit stroomnetwerk komt overeen met het grootste aantal verbindingen vinden, en dus de maximale ongewogen koppeling.
- Een koppeling met k verbindingen komt overeen met een gehele stroomverdeling met als netwerkstroom k.
- Het getal k is niet groter dan het aantal knopen in de kleinste van de twee verzamelingen L en R, en is $O(n)$.

9.2 Stabiele koppeling

- Gegeven één of twee verzamelingen van elementen.
- Elk element van die verzamelingen heeft een gerangschikte voorkeurslijst van andere elementen.
- De elementen moeten gekoppeld worden, rekening houdend met hun voorkeuren en zodanig dat de koppeling stabiel is.
- Een **koppeling is onstabiel** wanneer ze twee niet met elkaar gekoppelde elementen bevat, die liever met elkaar zouden gekoppeld zijn dan in de huidige toestand te blijven.
- Drie problemen:
  1. **Stable marriage**
     - Twee verzamelingen met dezelfde grootte.
     - De elementen worden *mannen* en *vrouwen* genoemd.
     - Elke man heeft een voorkeurslijst die alle vrouwen bevat.
     - Elke vrouw heeft een voorkeurslijst die alle mannen bevat.
     - Elke man **moet** gekoppeld worden aan een vrouw, zodanig dat de koppeling stabiel is.
  2. **Hospitals/Residents**
  3. **Stable roommates**

9.2.1 Stable marriage

Het Gale-Shapley-algoritme

- Er is tenminste één stabiele koppeling bij een stable marriage probleem.
- Er is een **actieve groep**, die aanzoeken stuurt naar de **passieve groep**.
- Het **Gale-Shapley-algoritme** garandeert dat de actieve groep de beste elementen zal krijgen die het kan hebben in een stabiele koppeling.
  - In de man-georiënteerde versie zijn de mannen de actieve groep, die aanzoeken sturen naar vrouwen, die dan de passieve groep zijn.
  - In de vrouw-georiënteerde versie zijn de vrouwen de actieve groep, die aanzoeken sturen naar mannen, die dan de passieve groep zijn.
- Op elk moment in het algoritme is een persoon ofwel verloofd, ofwel vrij.
- De personen in de actieve groep kunnen afwisselend verloofd of vrij is, maar de personen in de passieve groep blijven verloofd eens ze een aanzoek gekregen hebben (maar kan wel van partner veranderen).
- Een aanzoek gebeurt door een persoon in de actieve groep die nog vrij is.
  - Een aanzoek doen aan een persoon in de passieve groep die ook vrij is, moet verloven.
  - Een aanzoek doen aan een persoon in de passieve groep die al verloofd is, vergelijkt eerst met de huidige partner, en verwerpt de laagst geklasseerde. Als de persoon die verwerpt wordt de partner was, wordt die terug vrij.
- Een persoon uit de actieve groep zal aanzoeken versturen in volgorde van de voorkeurslijst.

Eigenschappen van de oplossing

- We veronderstellen dat mannen nu de actieve groep zijn, en vrouwen de passieve groep.
- Het algoritme stopt altijd en de oplossing is steeds stabiel.
  - Geen enkele man wordt afgewezen door alle vrouwen want hij kan niet afgewezen worden door de laatste vrouw op zijn lijst.
  - In elke iteratie is er een aanzoek, en geen enkele man doet dat twee maal aan dezelfde vrouw. Er zijn maximaal n^2 aanzoeken.
  - De oplossing is stabiel:
    - Stel een man m_1 en een vrouw v_1.
    - m_1 verkiest v_1 boven zijn huidige vrouw v_2.
    - v_1 moet m_1 in het verleden dus hebben afgewezen (omdat m_1 zeker eerst aan v_1 een aanzoek zou sturen in plaats van v_2) omdat v_1 een andere man m_2 verkoos.
    - Er is geen ongekoppeld paar dat de stabiliteit van die koppeling in gevaar kan brengen, want v_1 zal enkel nog mannen aanvaarden die nog hoger gerangschikt staan dan m_2 (en dus ook m_1).
- **Elke mogelijke aanzoekvolgorde geeft dezelfde oplossing**.

Implementatie

- Er zijn voorkeurslijsten voor de passieve groep, die de volgorde aanduiden van elk element van de actieve groep. Deze lijsten moeten opgesteld worden en dat is $\Theta(n^2)$.
- Er is ook een lijst van deelnemers in de actieve groep om snel te achterhalen wie nog aanzoeken kan doen.
- Het algoritme stopt als het laatst overgebleven element uit de passieve groep een aanzoek heeft gekregen.
- Elk van de $n-1$ elementen in de passieve groep kunnen n aanzoeken krijgen, zodat in het slechtste geval het algoritme $\Theta(n^2)$ is.

Uitbreidingen

- **Verzamelingen van ongelijke grootte**
  - Het aantal mannen n_m is verschillend van het aantal vrouwen n_v.
  - Een koppeling wordt als onstabiel beschouwd als er een man m en vrouw v bestaan zodat:
    1. m en v geen partners zijn.
    2. m ofwel ongekoppeld blijft, ofwel v verkiest boven zijn partner.
    3. v ofwel ongekoppeld blijft, ofwel m verkiest boven haar partner.
  - Er wordt verondersteld dat iemand liever gekoppeld wordt dan alleen te moeten blijven.
- **Onaanvaardbare partners**
  - De voorkeurslijsten moeten niet meer alle andere personen bevatten van de andere groep.
  - Stabiele koppeling kan nu gedeeltelijk zijn, zodat niet noodzakelijk iedereen een partner krijgt.
  - Een koppeling wordt als onstabiel beschouwd als er een man m en vrouw v bestaan zodat:
    1. m en v geen partners zijn, maar wel aanvaardbaar zijn voor elkaar.

2. m ofwel ongekoppeld blijft, ofwel v verkiest boven zijn partner.
3. v ofwel ongekoppeld blijft, ofwel m verkiest boven haar partner.

- **Gelijke voorkeuren**
  - De voorkeurslijsten mogen meerdere personen bevatten met dezelfde rangschikking.
  - Er zijn dan drie gevallen om stabiliteit te definiëren:
    1. **Superstabiliteit.** Een koppeling is onstabiel als er een man en een vrouw bestaan die geen partners zijn, en elkaar minstens evenzeer verkiezen als hun partners.
    2. **Sterke stabiliteit.** Een koppeling is onstabiel als er een man een vrouw bestaan die geen partners zijn, waarvan de ene de andere strik verkiest boven de partner, en de andere de eerste minstens even graag heeft als de partner.
    3. **Zwakke stabiliteit.** Een koppeling is onstabiel als er een man en een vrouw bestaan die geen partners zijn, en die elkaar strikt verkiezen boven hun partners.

Deel III

Strings

Hoofdstuk 10

Gegevensstructuren voor strings

10.1 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.

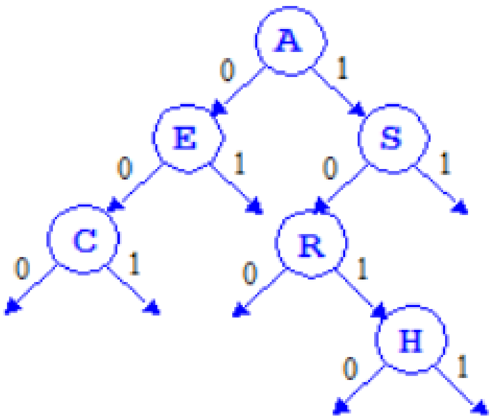
! Veronderstel dat geen enkele sleutel een prefix is van een ander.

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 willen toevoegen, 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.

10.2 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.
  - Bij de wortel wordt het eerste sleutelelement gebruikt, een niveau dieper het tweede sleutelelement, enz.



Figuur 10.1: 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.

- 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.

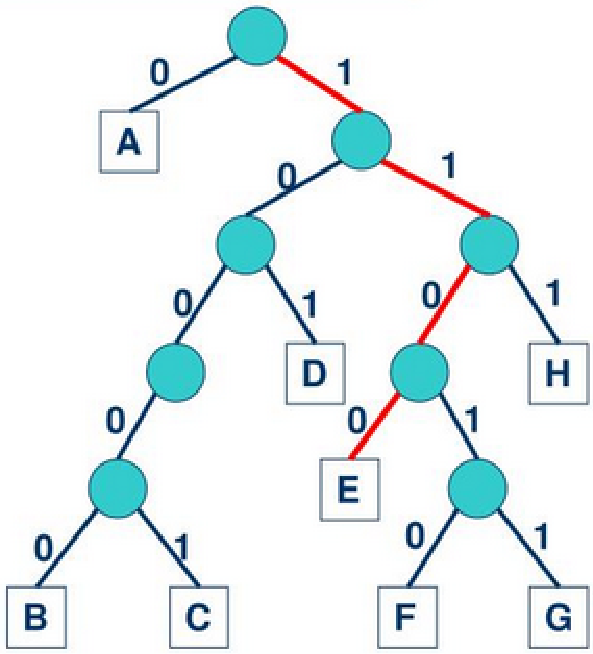
10.3 Tries

- Een digitale zoekstructuur die wel de volgorde van de opgeslagen sleutels behoudt.

10.3.1 Binaire tries

- 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.

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



Figuur 10.2: 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.

- ○ 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:
       (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.

10.3.2 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 $n/(s \ln m)$.

10.4 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, B, C en D kunnen volgende codewoorden krijgen:

$$A \rightarrow 10$$
$$B \rightarrow 1111$$
$$C \rightarrow 1110$$
$$D \rightarrow 01$$

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

10.4.1 Universele codes

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

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.
  - Anders gezegd: zet n om in zijn binaire representatie. Deze representatie heeft k bits. Laat deze representatie voorafgaan door $k-1$ nulbits.

	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' 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.

De Fibonaccicode

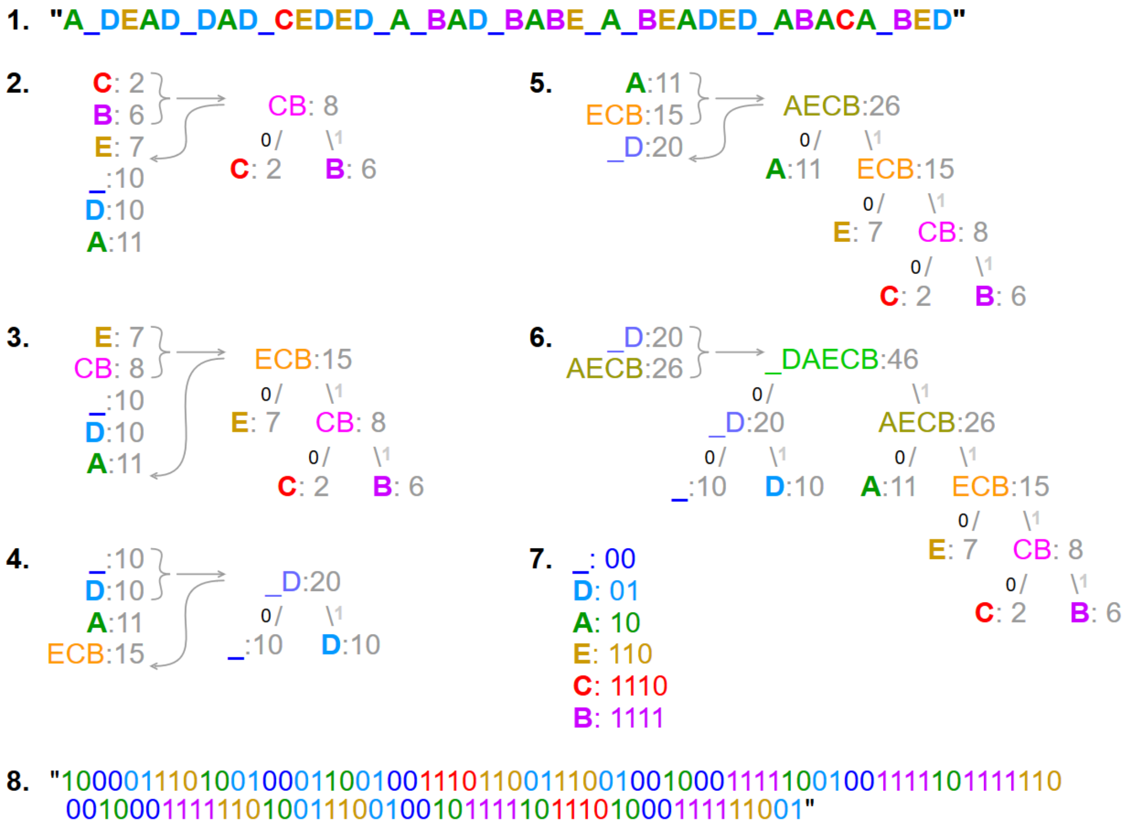
- De Fibonaccireeks
$$1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \ldots$$
- Dit heeft als eigenschap dat een getal i geschreven kan worden als de som van verschillende Fibonaccigetallen zodanig dat er nooit twee getallen in de reeks worden gebruikt die onmiddellijke buren zijn van elkaar.
- 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.

10.5 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.

10.5.1 Opstellen van de decoderingsboom

- Er wordt een prefixcode toegepast waarbij elke letter een apart codewoord krijgt die voor de hele tekst geldt.



Figuur 10.3: 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.

- We zullen bitcodes gebruiken, en dan 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 $\Sigma = \{s_i | i = 0, ..., d-1\}$
- 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.

  - 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

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

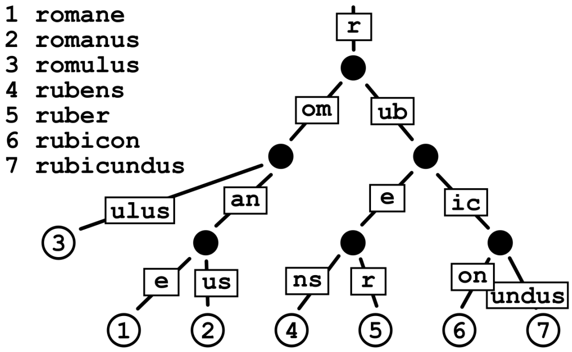
- Als l een groter gewicht en kleinere diepte dan k heeft, is er een betere trie bekomen.
- De optimale trie heeft volgende eigenschappen:
  - Geen enkele knoop heeft een groter gewicht dan een knoop op een kleinere diepte.
  - Geen enkele knoop heeft een groter gewicht dan een knoop links (of rechts) van hem op dezelfde diepte, want dan kunnen de twee knopen omgewisseld worden.
- Constructie van de coderingsboom: (dunno actaully)
  - Op elk moment is er een bos van deelbomen die aan elkaar gehangen moeten worden.
  - In het begin bestaat het bos uit enkel bladeren.
  - Er worden twee bomen 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 10.3).

10.5.2 Patriciatries

! 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:

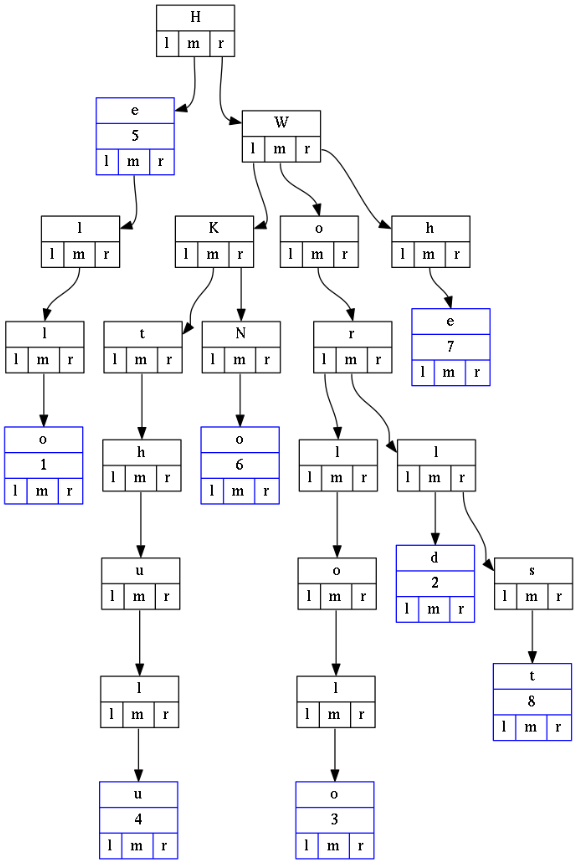


Figuur 10.4: Een patriciatrie. Elk blad bevat een verwijzing naar een woord in een lijst en knopen met maar één kind worden samengevoegd.

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.

10.6 Ternaire zoekbomen



Figuur 10.5: 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).

- 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 11

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.

11.1 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).

11.1.1 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:

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

    - De kleine letters zijn elementen van Σ en ϵ stelt de lege string voor.
    - 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":

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

11.1.2 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$(\epsilon) = \{\epsilon\}$.
  3. Voor elke $a \in \Sigma$ is 'a' een regexp, met als taal Taal$('a') = \{'a'\}$.
- Regexps kunnen gecombineerd worden via drie operaties:
- Vaak worden verkorte notaties gebruikt:

Operatie	Regexp	Operatie op taal/talen
Concatenatie	(RS)	Taal(R) $\cdot$ Taal(S)
Of	(R—S)	Taal(R) $\cup$ Taal(S)
Kleenesluiting	(R)*	Taal(R)*

- **Minstens eenmal herhalen**

$$r+ \leftrightarrow rr*$$

- **Optionele uitdrukking**

$$r? \leftrightarrow r|\epsilon$$

- **Unies van symbolen**

$$[abc] \leftrightarrow a|b|c$$

$$[a-z] \leftrightarrow a|b|\ldots|z$$

- Regexps kunnen verbonden worden met graafproblemen.
- **Stelling:** 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

$$P_G(a,b) = \begin{cases} \{\}, & \text{als } a \neq b \\ \{\epsilon\}, & \text{als } a = b \end{cases}$$

  - 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.
-

11.2 Variabele tekst

11.2.1 Een eenvoudige methode

- Vanaf positie j in T wordt P vergeleken door de overeenkomstige karakters van beide te vergelijken.
- $P[i]$ vergelijken met $T[j+1]$ voor $0 < i \leq p$.
- Stoppen zodra er een verschil is of het einde van P bereikt is.
- Dan verder doen voor $j + 1$.
- $P[0]$ zal vaak verschillen van $T[j]$, zodat de test op veel beginposities j reeds na één karaktervergelijking stopt.
- De gemiddelde uitvoeringstijd is $O(t)$.
- In het slechtste geval is dit $O(tp)$.

11.2.2 Zoeken met de prefixfunctie

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 $q()$ 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:
  - $q(i) < i$
  - $q(1) = 0$
  - $q(0) = -$ (niet gedefinieerd)
- De waarde van $q(i+1)$ kan bepaald worden als de waarden van de vorige posities gekend zijn.

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

  - Stel de string `ANOANAANOANO`
  - Dan zijn de waarden van de prefixfunctie als volgt:

	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 kunne 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 $s = i - q(i)$.

11.2.3 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**.
     - 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.
  2. Men tracht de kans dat er een fout voorkomt zo klein mogelijk te maken.

11.2.4 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 uniek getal toegekend.
- In plaats van P en de even lange deeltekst op een bepaalde positie te vergelijken, worden de overeenkomstige getallen vergeleken.
- Gelijke strings betekent gelijke getallen en omgekeerd is dit ook waar.
- 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.
- 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.
- Af en toe vergissen is dus mogelijk.
- 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:

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

    (term met de hoogste macht aftrekken en die met de kleinste opstellen)
    - 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:

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

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

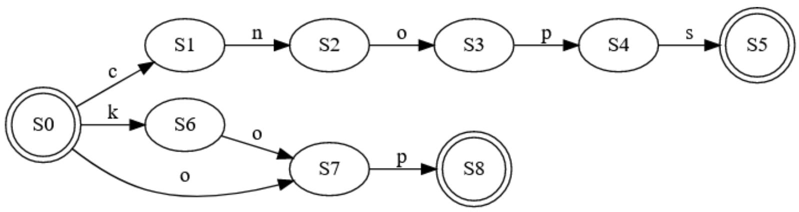
*
$$\begin{aligned}
H(T_2) &= (H(T_1) - T[1]5^2) \cdot 5 + T[4] \\
&= (B \cdot 5^2 + C \cdot 5 + D - B \cdot 5^2) \cdot 5 + E \\
&= C \cdot 5^2 + D \cdot 5 + E \\
&= 3 \cdot 5^2 + 4 \cdot 5 + 5 \\
&= 75 + 20 + 5 = 100
\end{aligned}$$

- De fingerprint is dan
$$H_r(T_{j+1}) = ((H(T_j) - T[j]d^{p-1})d + T[j+p]) \bmod$$
- 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.

11.2.5 Zoeken met automaten

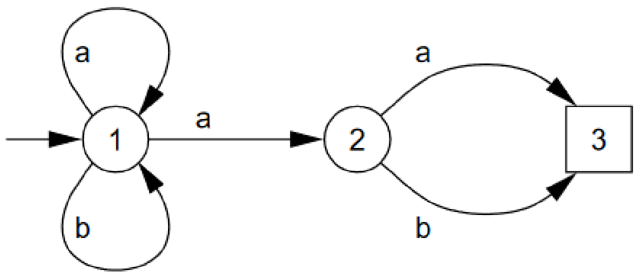
- Automaten beschrijven algemene informatieverwerkende eenheden met een eindig geheugen.
- Het geheugen wordt voorgesteld door **staten**.
  - Er zijn evenveel staten als er mogelijkheden zijn.
  - Een geheugenmodule van 32 kilobyte heeft 256^{32000} mogelijke staten.
- Een automaat modelleert ook de tijd als een
- **Deterministische automaten.**



Figuur 11.1: Een deterministische automaat die de woorden `CNOPS`, `KOP` en `OP` herkent. S_0 is de startstaat, S_5 en S_8 zijn eindstaten.

  - Een deterministische automaat (DA) bestaat uit:
    - ◇ Een (eindige) verzameling invoersymbolen Σ.
    - ◇ Een (eindige) verzameling toestanden S.
    - ◇ Een begintoestand $s_0 \in S$.
    - ◇ Een verzameling eindtoestanden $F \subset S$.
    - ◇ Een overgangsfunctie $p(t, a)$ die een nieuwe toestand geeft wanneer de automaat in staat t symbool a ontvangt.
  - Een DA wordt voorgesteld door een gerichte geëtiketteerde multigraaf G, de **overgangsgraaf**.
    - ◇ De knopen zijn de verschillende staten.
    - ◇ De verbindingen zijn de overgangen met als etiket het overeenkomstig invoersymbool.
  - Een DA start altijd in zijn begintoestand, en maakt de gepaste toestandsovergangen bij elk ingevoerd symbool.
  - Als een DA zich in een eindtoestand bevindt, dan wordt de string **herkend** door de DA. De verzameling strings die herkend wordt door een DA is de taal van die automaat.

- **Niet-deterministische automaten.**
  - Heeft geen staten, maar wel **statenbits**.
  - De 'staat' van een NA wordt aangeduid door de verzameling statenbits die aan staan.
  - De beginstaat wordt aangeduid met een speciale statenbit, de beginbit, die aanstaat in het begin terwijl alle andere uit staan.
  - De eindstaten worden aangeduid door de eindbits.
  - De overgang van een staat naar de volgende werkt bit per bit.
  - Een statenbit die aan staat reageert op een invoersymbool door een signaal naar nul of meer statenbits te sturen.
  - Een statenbit die één of meer signalen binnekrijgt zet zichzelf aan, anders uit.
  - Als i een statenbit is en a een letter uit het alfabet, dan is $s(i, a)$ de verzameling statenbits die rechtstreeks een signaal van i krijgen als de inkomenden letter a is.
  - Er zijn ook $\epsilon-$overgangen. Een $\epsilon-$overgang van statenbit i naar statenbit j zorgt ervoor dat i direct een signaal uitstuurt naar j, zonder vertraging.

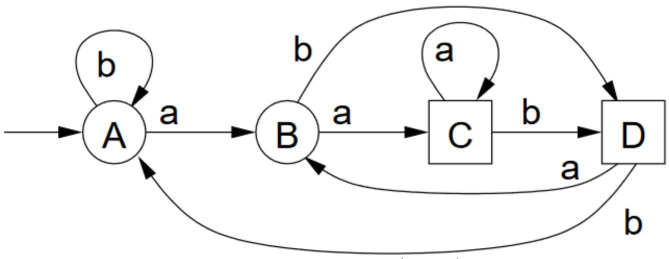


		a	b
$\rightarrow$	1	$\{1,2\}$	$\{1\}$
	2	$\{3\}$	$\{3\}$
$\leftarrow$	3		

Figuur 11.2: Een niet-deterministische automaat en bijhorende statentabel voor de reguliere expressie $(a|b) * a(a|b)$.

De deelverzamelingconstructie

- Een NA is een alternatieve voorstelling van een DA, maar laat geen efficiënte implementatie toe:
  - Bij elke binnenkomende letter moeten alle statenbits die aanstaan overlopen worden, en de daarbijhorende bits die een signaal krijgen aanduiden.
  - Bij een DA moet voor elke binnenkomende letter enkel de nieuwe staat opgezocht worden in de tabel.
- Een NA is wel eenvoudiger om op te stellen. Een reguliere uitdrukking kan eenvoudig omgezet worden tot een NA.
- **Een NA omzetten naar een DA** wordt de **deelverzamelingconstructie** genoemd.
  - Als een NA k statenbits heeft, zijn er 2^k mogelijke deelverzamelingen.
  - Die allemaal nagaan is niet efficiënt aangezien de meeste deelverzamelingen al niet bereikbaar zijn vanuit de begintoestand. Op figuur 11.2 is te zien dat enkel de deelverzamelingen $\{1\}$, $\{1, 2\}$ en $\{3\}$ (3 van de 8 deelverzamelingen) op elk moment beschikbaar kunnen zijn.



		a	b
$\rightarrow$	$A = \{1\}$	B	A
	$B = \{1, 2\}$	C	D
$\leftarrow$	$C = \{1, 2, 3\}$	C	D
$\leftarrow$	$D = \{1, 3\}$	B	A

Figuur 11.3: De deterministische automaat geconstrueerd uit die van figuur 11.2.

- Er is dus een implicite multigraaf met 2^k knopen die doorlopen kan worden met breedte-eerst of diepte-eerst zoeken.
- Knopen die niet bereikbaar zijn zijn overbodig voor de DA.
- Buren in deze impliciete multigraaf kunnen niet opgezocht worden in een burenlijst. Er zijn hulpoperaties nodig:
  - De **$\epsilon-$sluiting(T)** geeft de deelverzameling van statenbits bereikbaar via $\epsilon-$overgangen vanuit een verzameling statenbits T (gewoon via diepte eerst zoeken zoals pseudocode 11.1 in cursus).
  - De overgangsfunctie $p(t, a)$ kan uitgebreidt worden voor een verzameling van statenbits tot $p(T, a)$: de deelverzameling van alle statenbits rechtstreeks bereikbaar vanuit een toestand t uit T voor het invoersymbool a.
- Voor een DA hebben we verzameling van toestanden D en overgangstabel M nodig.
- De begintoestand van de DA is $\epsilon-$sluiting(b_0).
- dunno man

11.2.6 De Shift-AND-methode

- Bitgeoriënteerde methode, die efficiënt werkt voor **kleine patronen**.
- Voor elke positie j in de tekst T bijhouden welke prefixen van het patroon P overeenkomen met de tekst, eindigend op positie j.
- Maakt gebruik van een tabel R met p logische waarden. Het $i-$de element komt overeen met prefix van lengte i.
  - R_j stelt de waarde van tabel R na verwerking van $T[j]$.
  - $R_j[i-1]$ is waar als de eerste i karakters van P overeenkomen met de i testkarakters eindigend in j.
  - De tabel R_{j+1} kan afgeleidt worden uit R_j, aangezien sommige prefixen verlengd kunnen

worden:

$$R_{j+1}[0] = \begin{cases} 1, & \text{als } P[0] = T[j+1] \\ 0, & \text{als } P[0] \neq T[j+1] \end{cases}$$

$$R_{j+1}[i] = \begin{cases} 1, & \text{als } R_{j-1} = 1 \text{ en } P[i] = T[j+1] \\ 0, & \text{anders} \end{cases} \qquad \text{voor } 1 \leq i \leq p$$

- Bij de berekening van R_{j+1} moeten we weten of $T[j+1]$ gelijk is aan $P[i]$, voor elke mogelijke waarde van i.
- Er wordt een tweedimensionaletabel S opgesteld met d (lengte van alfabet) bitpatronen. Een bit i van woord $S[s]$ is waar als het karakter s op plaats i in P voorkomt.
- Om alle bits R_{j+1} gelijktijdig te berekenen wordt de schuifoperatie naar rechts gebruikt (bit i wordt bit $i+1$, en er wordt vooraan een éénbit ingeschoven), gevolgd door een bit-per-bit EN-operatie met $S[T[j+1]]$

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

- **Voorbeeld:**
  - Stel $\Sigma = \{A, C, G, T\}$ en $d = 4$.
  - Stel $P =$ `GCAGAGAG`.
  - Stel $T =$ `GCATCGCAGAGAGTATACAGTACG`.
  - De tabel S kan uit P berekent worden:

	$S[A]$	$S[C]$	$S[G]$	$S[T]$
G	0	0	1	0
C	0	1	0	0
A	1	0	0	0
G	0	0	1	0
A	1	0	0	0
G	0	0	1	0
A	1	0	0	0
G	0	0	1	0

  - De tabellen R_j worden dan:

		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

◇ Start vanuit $R_0 = \begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix}$ (want $R_0[0] = P[0]$).

$$
\begin{aligned}
R_1 &= \text{Schuif}(R_0) \text{ EN } S[T[1]] \\
&= \begin{bmatrix}1 & 1 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix} \text{ EN } \begin{bmatrix}0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix} \\
&= \begin{bmatrix}0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix} \\
R_2 &= \text{Schuif}(R_1) \text{ EN } S[T[2]] \\
&= \begin{bmatrix}1 & 0 & 1 & 0 & 0 & 0 & 0 & 0\end{bmatrix} \text{ EN } \begin{bmatrix}0 & 0 & 1 & 0 & 1 & 0 & 1 & 0\end{bmatrix} \\
&= \begin{bmatrix}0 & 0 & 1 & 0 & 0 & 0 & 0 & 0\end{bmatrix} \\
R_3 &= \text{Schuif}(R_2) \text{ EN } S[T[3]] \\
&= \begin{bmatrix}1 & 0 & 0 & 1 & 0 & 0 & 0 & 0\end{bmatrix} \text{ EN } \begin{bmatrix}0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix} \\
&= \begin{bmatrix}0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix} \\
R_4 &= \text{Schuif}(R_3) \text{ EN } S[T[4]] \\
&= \begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix} \text{ EN } \begin{bmatrix}0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix} \\
&= \begin{bmatrix}0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix} \\
R_5 &= \text{Schuif}(R_4) \text{ EN } S[T[5]] \\
&= \begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix} \text{ EN } \begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix} \\
&= \begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix} \\
&\ldots \\
R_8 &= \text{Schuif}(R_7) \text{ EN } S[T[8]] \\
&= \begin{bmatrix}1 & 0 & 0 & 1 & 0 & 0 & 0 & 0\end{bmatrix} \text{ EN } \begin{bmatrix}1 & 0 & 0 & 1 & 0 & 1 & 0 & 1\end{bmatrix} \\
&= \begin{bmatrix}1 & 0 & 0 & 1 & 0 & 0 & 0 & 0\end{bmatrix} \\
&\ldots \\
R_{12} &= \text{Schuif}(R_{11}) \text{ EN } S[T[12]] \\
&= \begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0 & 0 & 1\end{bmatrix} \text{ EN } \begin{bmatrix}1 & 0 & 0 & 1 & 0 & 1 & 0 & 1\end{bmatrix} \\
&= \begin{bmatrix}1 & 0 & 0 & 1 & 0 & 0 & 0 & \mathbf{1}\end{bmatrix}
\end{aligned}
$$

- Bij R_{12} is $R_{12}[7] = 1$, zodat P gevonden is en begint in T op positie $T[12-7] = T[5]$.

- De totale performantie is $\mathbf{\Theta(t + p)}$

11.3 De Shift-AND methode: benaderende overeenkomst

- De Shift-AND methode kan aangepast worden om fouten in het gevonden patroon toe te laten.
- Veronderstel dat er één karakter op een willekeurige plaats in P mag vervangen worden.
  - We zoeken dus alle deelstrings in T niet langer dan $m+1$ die P als deelsequentie bevatten.
  - Er is een nieuwe tabel R_j^1 die alle prefixen aanduidt in de tekst eindigend bij positie j, met hoogstens één vervanging.
  - $R_j^1[i]$ is waar als de eerste i karakters van P overeenkomen met de i van de $i+1$ karakters die in de tekst eindigen bij positie j.

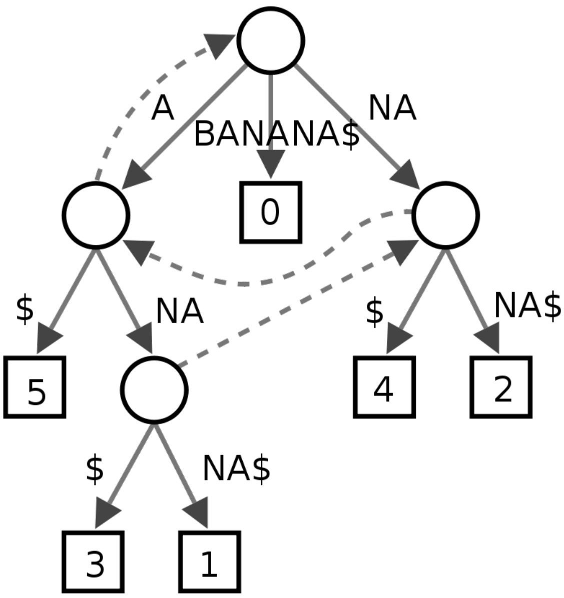
Hoofdstuk 12

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 11 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 aangeduid met suff_i.

12.1 Suffixbomen

- Gebaseerd op de **Patriciatrie**.
- Het aantal inwendige knopen is $O(t)$ en de vereiste geheugenruimte is $O(|\Sigma|t)$.
- Kan geconstrueerd worden in $O(t)$.
- Er zijn een aantal **wijzigingen** ten opzichte van een originele Patriciatrie:
  1. Een patriciatrie slaat strings op bij de bladeren. Hier volstaat de index i van suff_i.
  2. De testindex wordt vervangen door een begin- en eindindex, die een substring aangeeft van T in elke knoop.
  3. In elke inwendige knoop kan een staartpointer opgenomen worden.
     - De **staart(s)** van een string s is de string bekomen door het eerste karakter te verwijderen.
     - Er is een staartpointer van een inwendige knoop x naar een andere inwendige knoop y als de padstring van y hetzelfde is als staart(s).
     - Op figuur 12.1 is er bijvoorbeeld een staartpointer van de rechtse inwendige knoop met als padstring `NA` naar de linkse inwendige knoop met als padstring `A` omdat staart(`NA`) = `A`.
- De voorwaarde dat een string geen prefix mag zijn van een ander werd vroeger opgelost door een extra afsluitend karakter te introduceren, maar dat is hier moeilijker.



Figuur 12.1: Een suffixboom voor het woord `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.

- ◦ Elk karakter van T wordt één per één toegevoegd in de suffixboom.
- ◦ Na k iteraties zitten er suffixen van $T[0] \cdots T[k-1]$ in de boom zonder afsluitteken.
- ◦ _ToDo: ...
- ◦ Dus om ervoor te zorgen dat deze voorwaarde geldig is, moet T eindigen op een karakter dat nergens anders voorkomt in de tekst. Op figuur 12.1 is dit het karakter `$`.

12.2 Suffixtabellen

- Eenvoudiger alternatief voor een suffixboom, maar vereist minder geheugen.
- Een tabel met de gerangschikte suffixen (hun startindices) van T.

! Een suffixtabel bevat geen informatie over het gebruikte alfabet.

- Een suffixtabel construeren kan door eerst de suffixboom op te stellen in $O(t)$ en daarna deze in inorder te overlopen, ook in $O(t)$.
  - ◦ De suffixtabel, geconstrueerd uit de suffixboom uit figuur 12.1.

$$A = \begin{bmatrix} 6 & 5 & 3 & 1 & 0 & 4 & 2 \end{bmatrix}$$

Het eerste element ($A[0] = 6$) is een verwijzing naar het eindkarakter, maar zit niet in de boom.

- Er is echter nog een belangrijke hulpstructuur nodig, de LGP-tabel.
  - ◦ Langste Gemeenschappelijke Prefix - tabel.
  - ◦ Voor suff_i is $LGP[i]$ de lengte van het langste gemeenschappelijke prefix van suff_i.

- De alfabetische opvolger van suff_i wordt gegeven door **opvolger**$(\textbf{suff}_{SA_{|j|}}) = \textbf{suff}_{SA_{|j+1|}}$.

- De LGP-tabel wordt opgesteld via de suffixtabel:
  - Start met suff_0.
  - Zoek j zodat $A[j] = 0$.
  - Bepaal het langste gemeenschappelijke suffix:
    - Start met $l = 0$.
    - Verhoog l tot $T[i + l]$ niet meer overeenkomt.

12.3 Tekstzoekmachines

12.3.1 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.

12.3.2 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:
- Elke tekstzoekmachine moet aan een aantal voorwaarden voldoen:
  - De queries moeten goed geanalyseerd worden.
  - De queries moeten snel geanalyseerd worden.

	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

- 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**.

12.3.3 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 12.1.
- Er kan nu een **query evaluatie** algoritme opgesteld worden (gevisualiseerd op figuur 12.2).
  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.
  $$\langle d, f_{d,t}, p_1, \cdots, p_{f_{d,t}} \rangle$$

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

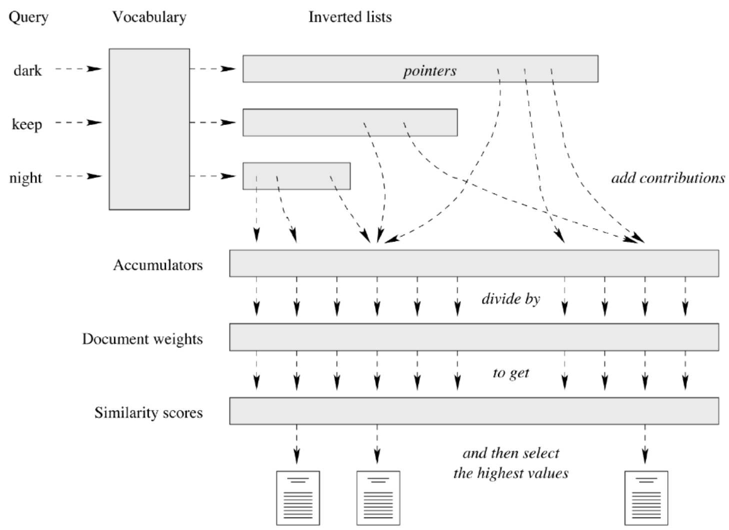
Tabel 12.1: Een op document niveau geïnverteerd bestand voor de *Keeper* databank. Elk woord t bestaat uit f_t en een lijst van paren, waarbij elk paar bestaat uit een sleutel d van een document en de frequentie $f_{d,t}$ van het woord t in d. Ook zijn de waarden van W_d te zien, berekend volgens $W_d = \sqrt{\sum_t w_{d,t}^2} = \sqrt{\sum_t f_{d,t}^2}$.

12.3.4 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

12.3.5 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**



Figuur 12.2: Het gebruik van een geïnverteerd bestand en een verzameling van accumulators om gelijkaardigheidswaarden te berekenen.

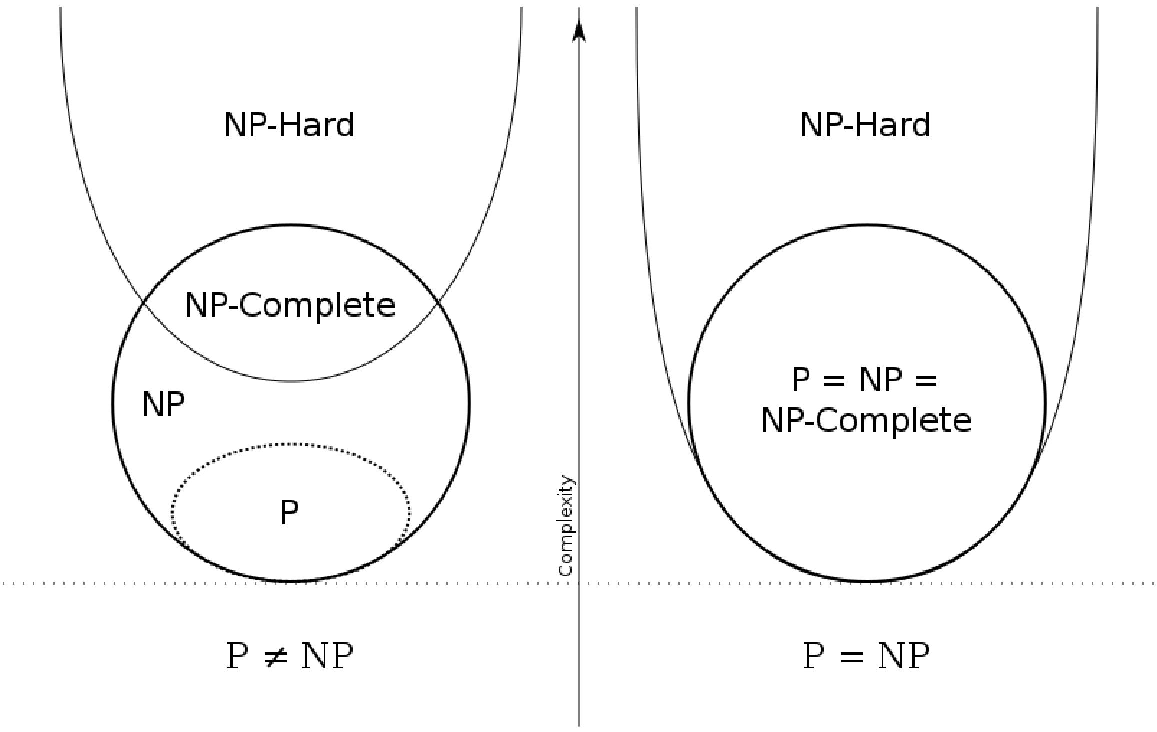
Deel IV

Hardnekkige problemen

Hoofdstuk 13

NP

13.1 Complexiteit: P en NP



Figuur 13.1: De linkse deelfiguur toont de verschillende complexiteitsklassen indien $P \neq NP$. De rechtse deelfiguur toont hetzelfde indien $P = NP$.

- Alle besproken algoritmen hebben een efficiënte oplossing.
- Hun uitvoeringstijd wordt begrensd door een **veelterm** zoals $O(n^2)$ of $O(n^2m)$.
- Sommige problemen hebben hebben geen efficiënte oplossing.
- Problemen worden onderverdeeld in **complexiteitsklassen**.
  - Beperking tot **beslissingsproblemen**, waarbij de uitvoer *ja* of *nee* is.

- Niet echt een beperking omdat elk probleem als een beslissingsproblemen kan geformuleerd worden.

13.1.1 Complexiteitsklassen

- De klasse **P** (**P**olynomiaal) bevat alle problemen waarvan de uitvoeringstijd begrensd wordt door een veelterm.
  - Op een realistisch computermodel.
    - Heeft een polynomiale bovengrens voor het werk dat in één tijdseenheid kan verricht worden.
  - Met een redelijke voorstelling van de invoergegevens (geen overbodige informatie, compact, ...).
  - Al de problemen in **P** worden als efficiënt oplosbaar beschouwd.
  - ! Waarom een veelterm? $O(n^{100})$ kan nauwelijks efficiënt genoemd worden.
    1. Meestal is de graad van de veelterm beperkt tot twee of drie.
    2. Veeltermen vormen de kleinste klasse functies die kunnen gecombineerd worden, en opnieuw een veelterm opleveren.
       - Men noemt dit een **gesloten klasse**.
       - Efficiënte algoritmen voor eenvoudigere problemen kunnen dus gecombineerd worden tot een efficiënt algoritme voor een complex probleem.
    3. De efficiëntiemaat blijft onafhankelijk van het computermodel.
- De klasse **NP** (**N**iet-deterministisch **P**olynomiaal) bevat alle problemen die door een niet-deterministische computer in polynomiale tijd kunnen opgelost worden en waarvan de oplossing kan gecontroleerd worden in polynomiale tijd.
  - Een niet-deterministische computer bevat hypothetisch een oneindig aantal processoren, waarvan er op tijdstap t er k kunnen aangesproken van worden. De processoren werken niet samen, maar kunnen wel hun deel van het probleem oplossen.
  - Elk probleem uit **P** behoort tot **NP**.
  - Niet geweten of er probleem in **NP** zit die niet tot **P** behoort → **P** vs **NP** probleem (Figuur 13.1).
  - Wel geweten dat er problemen zijn die niet in **NP** zitten, en dus ook niet in **P**.
- De klasse **NP-hard** bevat alle problemen die minstens even zwaar zijn als elk **NP**-probleem.
  - Een probleem X dat gereduceerd kan worden naar een probleem Y betekent dat Y minstens even zwaar is als X.
- De klasse **NP-compleet** bevat alle problemen die **NP-hard** zijn, maar toch nog in **NP** zitten.
  - Als er één **NP-compleet** probleem bestaat die efficiënt oplosbaar zou zijn (en dus in **P** behoort), dan zouden alle problemen uit **NP** ook efficiënt oplosbaar zijn, zodat **P** = **NP**.
  - NP-complete problemen kunnen op verschillende manieren aangepakt worden:
    - Backtracking en snoeien.
    - Speciale gevallen oplossen met efficiënte algoritmen.
    - Het kan zijn dat de gemiddelde uitvoeringstijd toch goed is.
    - Gebruik een benaderend algoritme.
    - Maak gebruik van heuristieken.

13.2 NP-complete problemen

- Overzicht van belangrijke NP-complete (optimalisatie)problemen.
- Om na te gaan of een probleem NP-compleet is, moet het herleid kunnen worden naar een basisvorm.

13.2.1 Het basisprobleem: SAT (en 3SAT)

- Gegeven:
  - Een verzameling logische variabelen $\mathcal{X} = \{x_1, \ldots, x_{|\mathcal{X}|}\}$.
  - Een verzameling logische uitspraken $\mathcal{F} = \{f_1, \ldots, f_{|\mathcal{F}|}\}$.
  - Elke uitspraak bestaat uit automaire uitspraken (atomen) samengevoegd met OF-operaties:

$$f_1 = x_2 \vee \overline{x_5} \vee x_7 \vee x_8$$

- Gevraagd:
  - Hoe moeten de waarden toegekend worden aan de variabelen uit $\mathcal{X}$ zodat elke uitspraak in $\mathcal{F}$ waar is?
- Elk NP-compleet probleem is reduceerbaar tot SAT.
- Een uitspraak met meer dan drie atomen kan herleidt worden naar een reeks uitspraken met elk drie atomen:

$$\begin{aligned} f_1 =& x_2 \vee \overline{x_5} \vee x_n \\ f_1' =& \overline{x_n} \vee x_7 \vee x_8 \end{aligned}$$

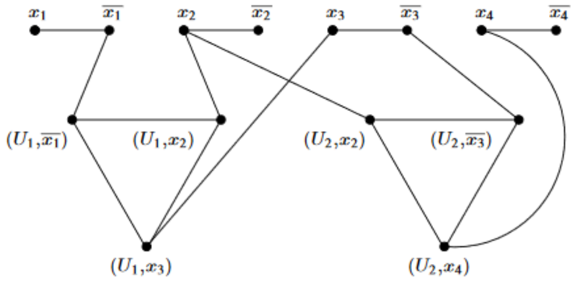
13.2.2 Vertex Cover

- Gegegeven:
  - Een ongerichte graaf.
- Gevraagd:
  - Hoe kan de kleinste groep knopen bepaald worden die minsten één eindknoop van elke verbinding bevat.
- Voorbeeld:
  - 3SAT kan gereduceerd worden tot vertex cover.
    - Voor elke logische variabele x_i worden er twee knopen gemaakt: voor x_i en $\overline{x}_i$. Deze knopen zijn verbonden.
    - Voor elke uitspraak worden er drie knopen gemaakt, één voor elk atoom. Deze drie knopen worden verbonden. Elk atoom wordt ook verbonden met de knopen voor de individuele logische variabelen.
  - Voor elke logische variabele moet zeker één van de twee knopen opgenomen worden.
  - Voor elke uitspraak moeten zeker twee van de drie knopen opgenomen worden.
  - Minstens $|\mathcal{X}| + 2|\mathcal{F}|$ knopen.

- Stel volgende uitspraken

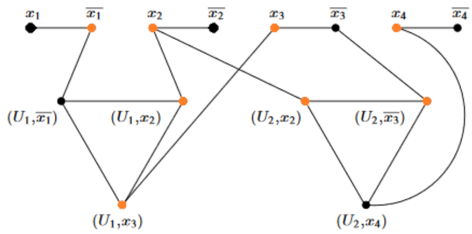
$$U_1 = \overline{x}_1 \vee x_2 \vee x_3$$
$$U_2 = x_2 \vee \overline{x}_3 \vee x_4$$

Bijhorende graaf:



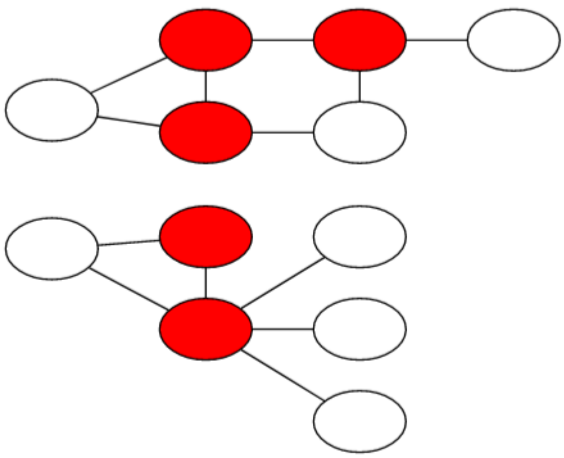
Figuur 13.2: Een graaf voor het 3SAT-probleem.

Voorbeeld van een minimale vertex cover:



Figuur 13.3: Een minimale vertex cover van de graaf op figuur 13.2

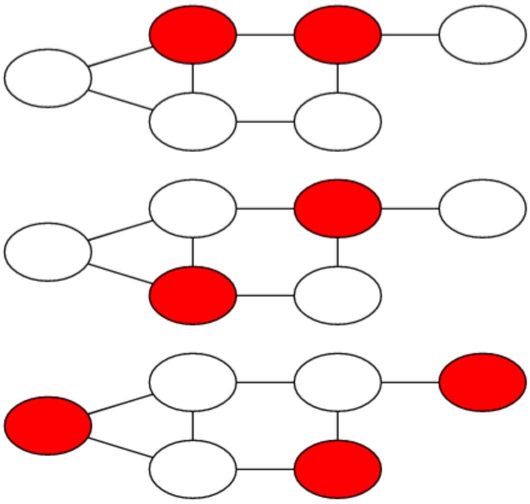
- Andere voorbeelden van minimale vertex covers:



Figuur 13.4: De minimale vertex cover van twee verschillende grafen.

13.2.3 Dominating set

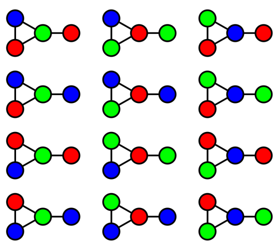
- Gegeven:
  - Een ongerichte graaf.
- Gevraagd:
  - Een kleinste groep knopen zodat elke andere knoop met minstens een van de knopen uit de groep verbonden is.
- Voorbeelden:



Figuur 13.5: Voorbeelden van dominating sets.

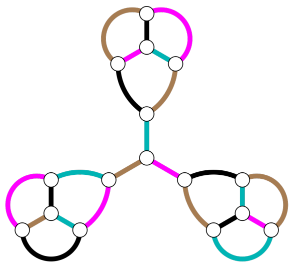
13.2.4 Graph Coloring

- **Vertex coloring**:
  - Gegeven:
    - Een ongerichte graaf.
  - Gevraagd:
    - Kleur de knopen met een minimum aantal kleuren, zodat de eindknopen van elke verbinding een verschillende kleur hebben.
  - Voorbeeld:



Figuur 13.6: De knopen van een graaf kunnen op meerdere manieren gekleurd worden.

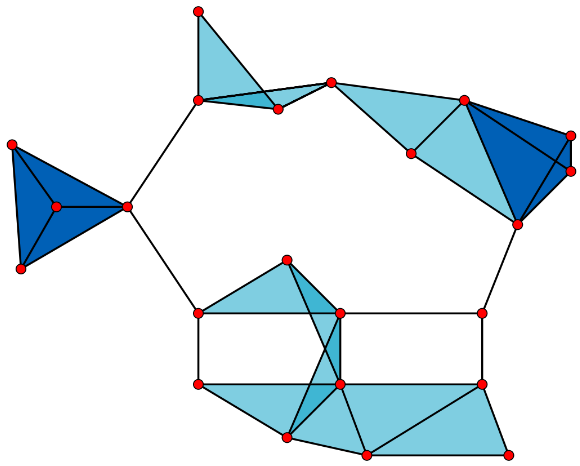
- **Edge coloring**:
    - Gegeven:
        - Een ongerichte graaf.
    - Gevraagd:
        - Kleur de verbindingen met een minimum aantal kleuren, zodat verbindingen met dezelfde eindknoop een verschillende kleur hebben.
    - Voorbeeld:



Figuur 13.7: Elke knoop heeft graad 3 dus zijn er hoogstens 4 kleuren nodig om de verbindingen te kleuren.

13.2.5 Clique

- Gegeven:
    - Een ongerichte graaf.
- Gevraagd:
    - De grootste groep knopen die allemaal met elkaar verbonden zijn.
- Voorbeeld:

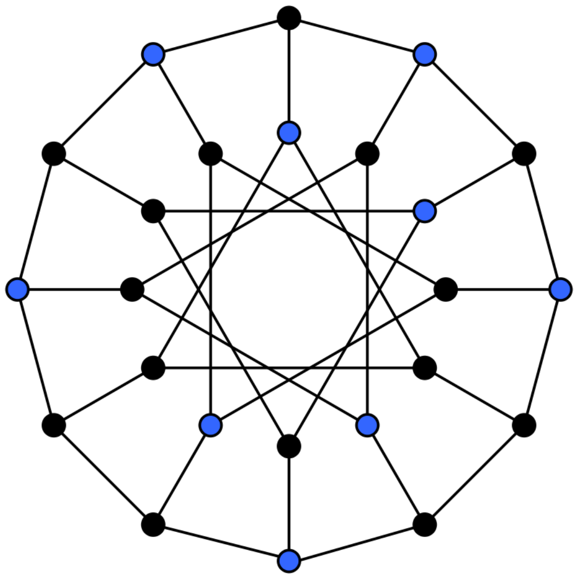


Figuur 13.8: Deze graaf bevat 23 1-knoop cliques (de knopen), 49 2-knoop cliques (door de verbindingen), 19 3-knoop cliques (lichtblauwe driehoeken) en 2 4-knoop cliques (donkerblauwe oppervlakken). In deze graaf zijn de twee 4-knoop de grootste groep van knopen.

- Het SAT probleem kan herleidt worden tot clique.
  - Voor elk voorkomen van een atoom in een uitspraak wordt twee knopen (F_i, x_j) en $(F_i, \overline{x_j})$ aangemaakt.
  - Er is een verbinding tussen knopen die horen bij verschillende uitspraken als de atomen elkaar niet uitsluiten:
    - (F_i, x_k) met (F_j, x_l) voor alle k en l.
    - $(F_i, \overline{x_k})$ met $(F_i, \overline{x_l})$ voor alle k en l.
    - (F_i, x_k) met $(F_i, \overline{x_l})$ als $k \neq l$.
- Een clique kan hoogstens $|\mathcal{F}|$ elementen bevatten.

13.2.6 Independent set

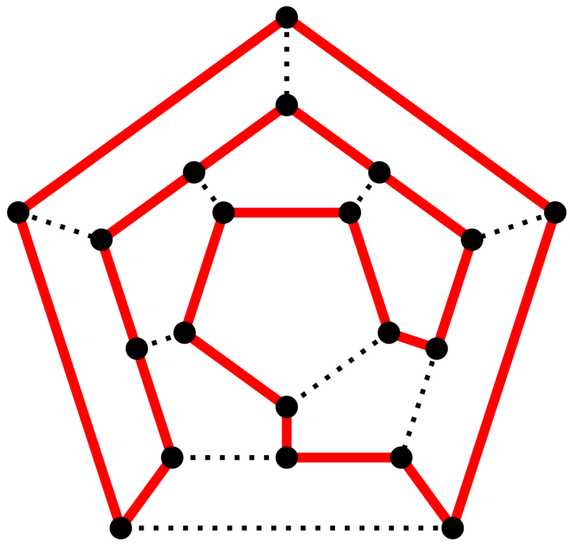
- Gegeven:
  - Een ongerichte graaf.
- Gevraagd:
  - De grootste groep knopen zonder gemeenschappelijke verbindingen.
- Voorbeeld:



Figuur 13.9: De blauwe knopen vormen de independent set.

13.2.7 Hamilton path

- Gegeven:
  - Een al dan niet gerichte graaf.
- Gevraagd:
  - Bestaat er een circuit dat elke knoop eenmaal bevat?
- Voorbeeld:



Figuur 13.10: Een tweedimensionale weergave van een Hamiltonpad door een dodecaëder. De zwarte stippelijnen zijn ook verbindingen.

13.2.8 Minimum cover

- Gegeven:
  - Een verzameling $\mathcal{S}$.
  - Een collectie deelverzamelingen $\mathcal{C}$ van $\mathcal{S}$.
- Gevraagd:
  - De kleinste deelverzameling $\mathcal{C}'$ van $\mathcal{C}$ zodat elk element van $\mathcal{S}$ tot minstens één van de deelverzamelingen uit $\mathcal{C}'$ behoort.
- Voorbeeld:
  - Dit kan efficiënt opgelost worden als de deelverzamelingen in $\mathcal{C}$ niet meer dan twee elementen bevatten.
  - Als elke $C \in \mathcal{C}$ exact twee elementen heeft is het equivalent met edge cover.
  - Als er $C \in \mathcal{C}$ zijn met slechts één element kunnen deze geschrapt worden en dan edge cover oplossen.
  - SAT kan ook herleidt worden tot minimum cover.
    - Neem $\mathcal{S} = \mathcal{F} \cup \mathcal{X}$
    - Voor elke logische variabele x_i zijn er twee elementen van C

$$C_{x_i} = \{x_i\} \cup \{F \in \mathcal{F} : x_i \in F\}$$
$$C_{\overline{x_i}} = \{x_i\} \cup \{F \in \mathcal{F} : \overline{x_i} \in F\}$$

13.2.9 Subset sum

- Gegeven:
  - Een verzameling elementen met elk een positieve gehele grootte.
  - Een positief geheel getal k.

- Gevraagd:
  - Een deelverzameling van die elementen, zodat de som van hun grootten gelijk is aan k.

13.2.10 Partition

- Gegeven:
  - Een zak van positieve gehele getallen.
- Gevraagd:
  - Kan die opgesplitst worden in twee deelverzamelingen, zodat de som van de grootten van de ene gelijk is aan de som van de grootten van de andere.

13.2.11 Travelling salesman problem

- Gegeven:
  - Een aantal steden, en de afstand tussen elk paar steden.
- Gevraagd:
  - Het kortste circuit dat elke stad eenmaal bevat.

13.2.12 Longest path

- Gegeven:
  - Een gerichte of ongerichte gewogen graaf.
  - Twee knopen s en t.
- Gevraagd:
  - De langste weg zonder lussen van s naar t.

13.2.13 Bin packing

- Gegeven:
  - Een verzameling van n objecten met afmetingen $s_1, \cdots, s_n$.
  - Een verzameling van m bakken met capaciteiten $c_1, \cdots, c_m$.
- Gevraagd:
  - Sla alle objecten op in zo weinig mogelijk bakken

13.2.14 Knapsack

- Gegeven:
  - Een verzameling van n objecten met elk een afmeting.
  - Een knapzak met een zekere capaciteit.
- Gevraagd:
  - Steek objecten in de zak zonder zijn capaciteit te overschrijden, zodat hun totale waarde zo groot mogelijk is.

Hoofdstuk 14

Metaheuristieken

- **Heuristieken** zijn vuistregels bij het zoeken naar een oplossing van een probleem.
- Garanderen niet dat er een oplossing gevonden wordt, maar versnelt wel de zoektocht ernaar.

14.1 Combinatorische optimisatie

- Abstracte representatie van de problemen nodig.
  - Het zijn **optimisatie**-problemen.
  - Voor een verzameling $\mathcal{S}$ moet hieruit de beste gekozen worden.
  - De verzameling $\mathcal{S}$ is een eindige verzameling van strings over een eindig alfabet.
  - Het beste individu wordt bepaald door een evaluatiefunctie f.
  - De beste waarde komt overeen met de kleinste waarde voor f.
- Voorbeeld bij het Travelling Salesman probleem:
  - Het alfabet is hier de verzameling verbindingen.
  - De verzameling $\mathcal{S}$ bestaat uit een reeks verbindingen zo dat elke verbinding vertrekt uit het eindpunt van de vorige en zo dat alle steden bezocht worden.
  - Het beste individu is die met de kleinste lengte.
- Vaak zijn alle strings in $\mathcal{S}$ even lang.
  - Elementen van $\mathcal{S}$ kunnen dan beschreven worden door variabelen.
  - Elke letter $s \in \mathcal{S}$ geeft de waarde aan van een variabele.
  - Voorbeeld bij het lessenroosterprobleem:
    - Er zijn een aantal lessen, een aantal docenten, een aantal klaslokalen, een aantal mogelijke tijdsslots en een aantal groepen studenten.
    - Er moet een lessenrooster opgesteld worden zodanig dat er minimale negatieve elementen zijn (springuren, onevenwichtige verdelingen van de studenten, ...).
    - Elke string van $\mathcal{S}$ is even lang en beschrijft e en lessenrooster zonder conflicten door aan elke les een lokaal en een tijdsslot toe te kennen.
    - $\mathcal{S}$ heeft twee letters: de eerste duidt de tijdsslot aan en de tweede het lokaal.
- In principe zijn zulke problemen oplosbaar met backtracking.

- Maar voor problemen die te groot zijn voor backtracking, zijn metaheuristieken toch nodig.
- Er wordt daarom eerst een steekproef genomen uit de totale verzameling $\mathcal{S}$ en voor elk individu wordt de $f-$waarde berekend.
- Het individu met de beste $f-$waarde wordt altijd bijgehouden.
- Het zoeken stop na een bepaald criterium, zoals bijvoorbeeld een tijdslimiet of als het gevonden individu reeds goed genoeg is.
- De verschillende metaheuristieken verschillen in de manier waarop individuen worden uitgekozen om te vergelijken.
- Elke metaheuristiek leent zich ertoe om lokale optimalisaties door te voeren.

14.2 Vooronderstellingen

- Bij de keuze van methodieken moet er eerst nagegaan worden of het probleem aan een aantal voorwaarden voldoet. Het ontbreken van een bepaalde voorwaarde kan bijvoorbeeld een metaheuristiek uitsluiten. Er zijn **vier** gebieden waarin onderscheidt tussen de verschillende optimisatietechnieken gemaakt kan worden.

  1. **Het moet zinvol zijn om op zoek te gaan naar betere individuen in de buurt van een gegeven individu.** Het kan zijn dat we een individu van de grond af opbouwen (zeker bij het eerste individu), maar er moeten ook pogingen ondernomen worden om een reeds bestaand individu te verbeteren.
  2. **Soms is het niet zinvol om bij het opbouwen van een nieuw individu van $\mathcal{S}$ uit te gaan van reeds bekende individuen.** Een nieuw individu van de grond opbouwen kan op twee manieren:
     (a) Het individu wordt opgebouwd uit componenten. Deze componenten komen overeen met letters van de strings in $\mathcal{S}$. Dit gebeurt at random, eventueel met een bepaalde heuristiek die zeer slechte individuen uitsluit.
     (b) Het individu wordt rechtstreeks aangemaakt.

     In het travelling salesman probleem kan een individu beschouwd worden als een pad in een graaf. Via de eerste methode is het pad leeg en worden verbindingen toegevoegd. In het tweede geval is het individu een permutatie van de verzameling verbindingen.
  3. **Een heuristiek moet aangeven dat een bepaalde keuze beter is dan de andere.** Bij het opbouwen van de componenten van een individu zijn sommige keuzes niet meer zinvol. Die mogen dan zeker niet meer gekozen worden. Er zijn twee mogelijkheden om dit te realiseren.
     (a) **Shortlisting.** Eerst worden de beste kandidaten geselecteerd met behulp van een bepaalde cutoff. Daarna wordt hieruit gekozen met gelijke waarschijnlijkheid.
     (b) **Gewogen keuze.** Elke mogelijkheid krijgt een een gewicht w_i. De kans $p(i)$ dat i gekozen wordt is dan
     $$p(i) = \frac{w_i}{\sum_j w_j}$$
  4. **Er zijn drie fundamentele methoden om een nieuw individu $\mathcal{S}$ te kiezen uitgaande van andere individuen.**
     (a) Kleine wijzigingen aanbrengen in een reeds bekeken individu s van $\mathcal{S}$.
     (b) Kruisen van twee bekeken individuen.
     (c) Het vermengen van een grote hoeveelheid al bekeken individuen.

14.3 Lokaal versus globaal zoeken

- Bij het opbouwen van nieuwe individuen van $\mathcal{S}$ moet het zinvol zijn om een individu van $\mathcal{S}$ te verbeteren door kleine aanpassingen.
- Een kleine aanpassing hangt af van het probleem.
  - Voor elke $s \in \mathcal{S}$ is er een **omgeving** $\mathcal{N}(s)$: de omgeving van s die bestaat uit s zelf en alle strings uit $\mathcal{S}$ die door één kleine wijziging uit s bekomen kunnen worden.
- Een goed individu kan zeker in een slecht individu omgevormd worden, maar vaker blijft het een goed individu.
- Lokale verbeteringen kunnen iteratief toegepast worden:
  - Beginnend van een individu $s_0 \in \mathcal{S}$
  - Genereer een rij van individuen $s_1, \cdots, s_n \in \mathcal{S}$.
  - Hierbij geldt dat voor alle $i = 0, \cdots, n-1, s-i+1 \in \mathcal{N}(s_i)$ en $f(s_{i+1}) < f(s_i)$.
  - Dit eindigt met een lokaal minimum, een individu s_n zodat $\forall s \in \mathcal{N}(s_n) : f(s) \geq f(s_n)$.
- Het vinden van zo een rij heet **lokale optimisatie**.
- Zo een rij vinden kan op meerdere manieren:
  1. **Steepest hill climbing.** (of steepest slope descending)
     Soms is het gemakkelijk om een lokaal minimum te vinden omdat $\mathcal{N}(s)$ weinig individuen bevat.
  2. **Randomisatie.**
     Veranderingen worden random gekozen.
- Er kunnen veel lokale minima zijn in $\mathcal{S}$, elk met een verschillende $f-$waarde.
- Er moet een manier zijn om te ontsnappen aan een lokaal minimum $\rightarrow$ **exploratie**.
  1. Exploratie door helemaal opnieuw te beginnen.
  2. Exploratie door grotere wijzigingen aan te brengen aan gekende individuen.

14.4 Methodes zonder recombinatie

- Random zoeken kan efficiënter gemaakt worden.
  1. Lokaal zoeken verbetert vaak random aangemaakt individuen.
  2. Heuristieken gebruiken om het aanmaken van de individuen te sturen.

14.4.1 Simulated Annealing

- Tijdens lokale optimisatie maakt het gebruik van een kans op een individu s', ook al is s' slechter dan s.
- De kans hangt af van:
  - De evaluatiewaarden $f(s)$ en $f(s')$.
  - Een 'temperatuur' T.

- De kans wordt groter indien T kleiner is of indien $f(s') - f(s)$ kleiner is (Boltzmanndistributie).

$$\rho(T, f(s'), f(s)) = \exp\left(\frac{f(s') - f(s)}{T}\right)$$

- T wordt vrij hoog gekozen, zodat exploratie bevorderd wordt en wordt iteratief gedaald tot bijna nul.
- Dit algoritme vindt zeer waarschijnlijk een globaal optimum als er een $\Gamma > 0$ bestaat waarvoor

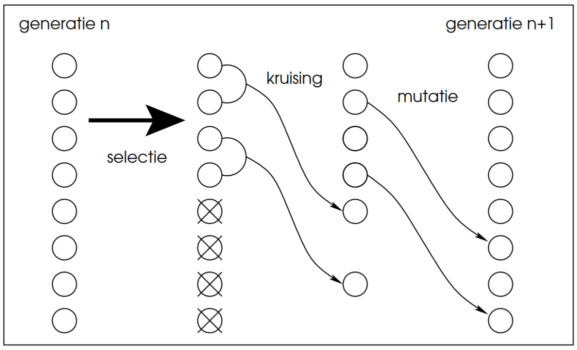
$$\sum_{k=1}^{\infty} \exp\left(\frac{-\Gamma}{T_k}\right) = \infty$$

14.4.2 Tabu Search

- Houdt een taboelijst bij: een lijst van al gecontroleerde individuen die niet opnieuw mogen bezocht worden.
- Alle gecontroleerde individuen bijhouden is niet efficiënt, daarom maar een beperkt aantal.
- De lijst bevorderd exploratie: bij een grote lijst zal een nieuwe individu veel verder in de omgeving $\mathcal{N}(s)$ liggen.
- De taboelijst kan ook eigenschappen bijhouden en geen individuen.

14.5 Genetische algoritmen

- Meeste genetische algoritmen maken gebruik van **kruising**.
- Kruising is het combineren van twee individuen.
- Enkel zinvol als een kruising een beter individu *kan* opleveren.
- Kruising combineerd meestal de componenten van beide individuen die goed kunnen zijn.
- Handig als één van de individuen een slechte $f-$waarde heeft.
- Twee manieren om te kruisen:
  1. **Gerichte kruising.**
     Enkel haalbaar als er een algemeen idee is van de structuur van individuen.
  2. **Blinde kruising.**
     Gerandomiseerde kruising van componenten.
- Er is een **populatie nodig**.
- Het doel is dan om de algemen populatie te verbeteren.
- Men spreekt van **generaties**. De overstap van een generatie naar een andere gebeurt als volgt (figuur 14.1):
  1. De populatie wordt uitgedund zodat enkel de beste individuen overblijven.
  2. De populatie wordt terug aangevuld op basis van de overblijvende individuen door kruising en mutatie.
- Twee verfijningen:



Figuur 14.1: Overgang naar een volgende generatie.

- Twee componenten kunnen elkaar positief beïnvloeden. Het is dan interessant om deze componenten bij elkaar te houden. Dit kan ingebouwd worden bij gerichte kruising. Er kan ook een maat voor gelijkaardigheid ingevoerd worden.
- Kruising moet niet enkel zorgen voor betere individuen, maar ook voor diversificatie en zo voor exploratie. Zo voert men **niching** in, dat de fitheid van individuen verminderd naarmate er meer gelijkaardige individuen in de populatie zitten.

14.6 Vermenging

- Niet twee individuen kruisen, maar de globale eigenschappen van de hele populatie wordt in beschouwing genomen.
  1. **Componentniveau.** Hierbij gaat men ervan uit dat het succes van een individu afhangt van de individuele componenten die ze bevat.
  2. **Combinatieniveau.** Hierbij gaat men ervan uit dat het succes van een individu afhangt van de combinatie van componenten die ze bevat.

14.6.1 Recombinatie op componentniveau

- Er wordt een $f-$waarde toegekend niet enkel aan de individuen, maar ook aan de componenten zelf.
- Het construeren van een nieuwe populatie gebeurt pseudorandom waarbij gewicht w_i van component i dient als gewicht voor de gewogen keuze.
- De gewogen random keuze kiest uit een verzameling $\mathcal{V}$ van componenten een component i met waarschijnlijkheid

$$p(i) = \frac{w_i}{\sum_{j \in V} w_j}$$

- Twee belangrijke metaheuristieken die gebruik maken van dit soort recombinatie. Ze verschillen in de manier waarop gewichten aan componenten toegekend worden.
  - **Gene Pool Recombiniation (GPR).**
    Elk component krijgt een gewicht evenredig met het aantal individuen in de overblijvende populatie die het component bevatten. Het gewicht van een component is dan enkel afhankelijk van de vorige generatie.

- ◦ **Ant Colony Optimisation (ACO)**.
  Elke s krijgt eerst een kwaliteitswaarde $F(s)$, die groter is naarmate $f(s)$ kleiner is. Een nieuw gewicht voor een component i is dan

$$w_i = (1-\rho)w_i + \sum_{s \text{ bevat } i} F(s)$$

  met ρ de vergeetfactor. Het gewicht van meerdere generaties zal invloed hebben op het huidige gewicht.

14.6.2 Recombinatie op combinatieniveau

- Er zijn twee voorwaarden om dergerlijke methoden te gebruiken:
  1. Opeenvolgende letters in het individu moeten een sterk onderling verband vertonen.
  2. Een deelstring van een goed individu gebruiken in een ander individu verhoogt de kans dat het andere individu ook goed is.
- De componenten van de individuen worden de knopen van een ongerichte gewogen graaf.
- Als twee componenten samen kunnen voorkomen is er een verbinding tussen de overeenkomstige knopen in de graaf.
- Er zijn nu een aantal **agents** die proberen een individu uit $\mathcal{S}$ te construeren. Dit wordt voorgesteld door een pad in de graaf.
  1. Op een bepaald punt bepaalt de agent of het pad een individu definieert. Zo ja dan stopt het.
  2. Anders worden de componenten bepaalt die nog in aanmerking komen.
  3. Als er geen mogelijkheden zijn wordt er gestopt, anders wordt er een gewogen keuze gemaakt tussen de verbindinge.
- Nadat elke agent geprobeerd heeft om een individu te construeren, worden de gewichten aangepast.
- Elke agent die een individu s heeft gevonden, past het gewicht aan van elke verbinding dat het gebruikt heeft met $F(s)$

$$\tau_{ij} = (1-\rho)\tau_{ij} + \sum_{s \in A_{ij}} F(s)$$

  Hierbij is A_{ij} de verzameling individuen die verbinding v_{ij} gebruikt hebben.
- Meestal zijn alle τ_{ij} eerst gelijk, maar andere beginwaarden zijn ook mogelijk.