FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Gevorderde algoritmen

Bert De Saffel

Master in de Industriële Wetenschappen: Informatica Academiejaar 2018–2019

Gecompileerd op 31 juli 2019

UNIVERSITEIT
GENT

Inhoudsopgave

Deel I

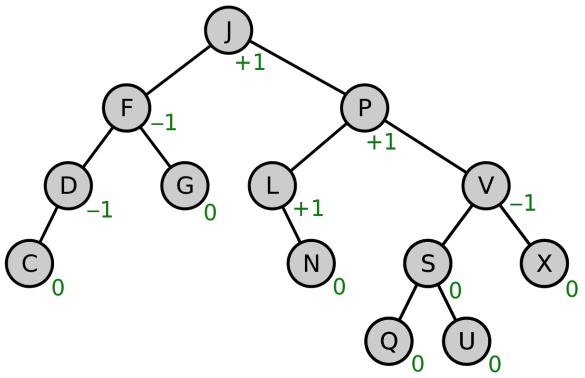
Gegevensstructuren II

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



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

     (b) 2-3-bomen.

- 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.

- Eenvoudiger dan 2-3-bomen om te implementeren.

(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.

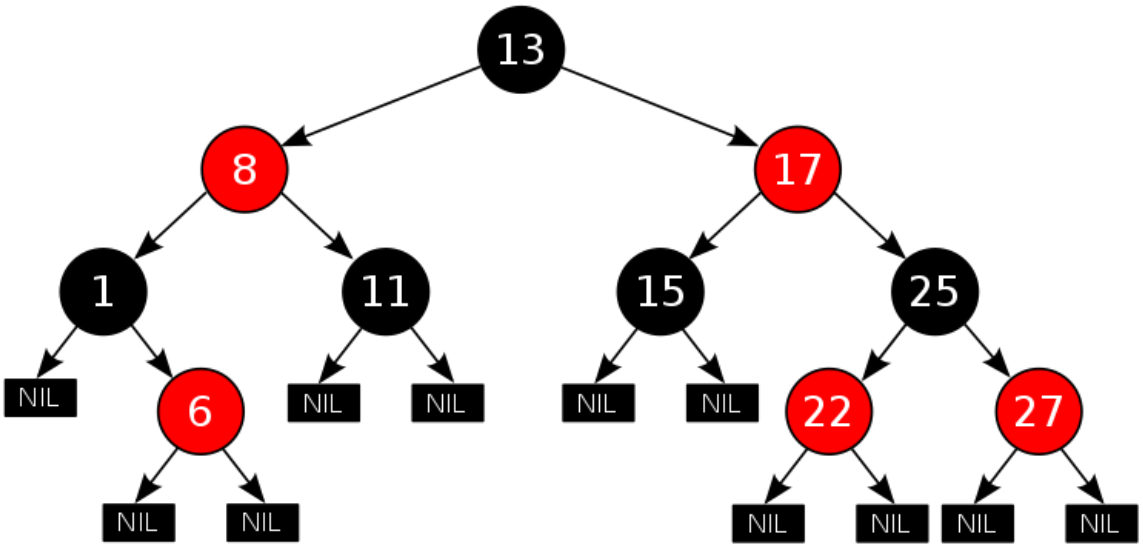
3. **De gemiddelde efficiëntie onafhankelijk maken van de operatievolgorde.**

(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



Figuur 1.2: Een rood-zwarte boom. De `NIL` knopen stellen virtuele knopen voor.

- **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.
  - 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:

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

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.

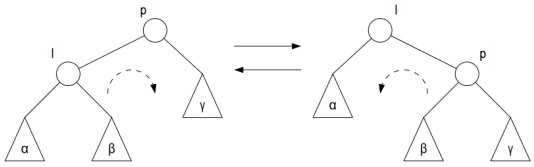
- Twee manieren om toe te voegen:

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

- 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.



Figuur 1.3: Rotaties

- 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

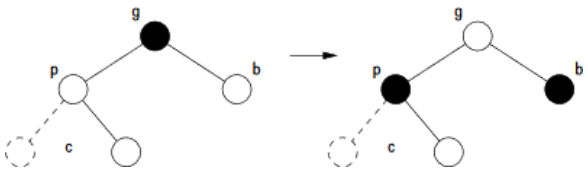
Verwijderen

1.2.6 Top-down 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.**
   - Maak p en b zwart.
   - Maak g rood.
   - Als g een zwarte ouder heeft, is het probleem opgelost.

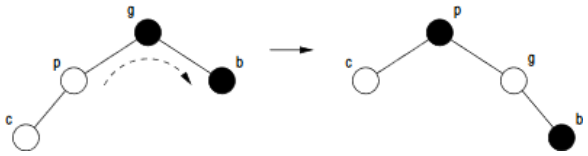


Figuur 1.4: Rode broer.

- 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.**

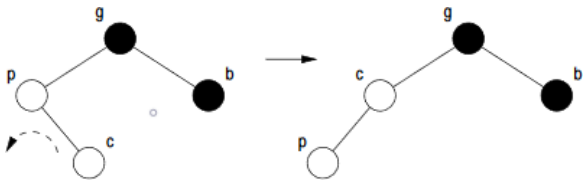
   (a) **Knoop c is een linkerkind van p.**



Figuur 1.5: Rode broer.

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

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



Figuur 1.6: 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.

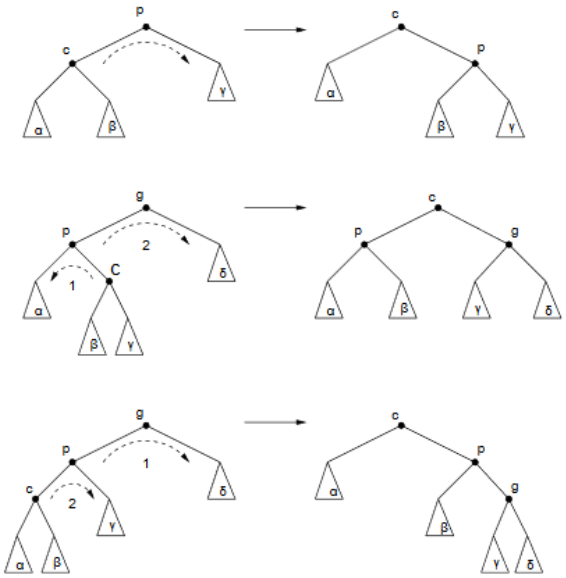
Verwijderen

- 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, heeft hij ofwel geen kinderen ofwel minstens één rood kind.

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 1.7: 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.
  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**.

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

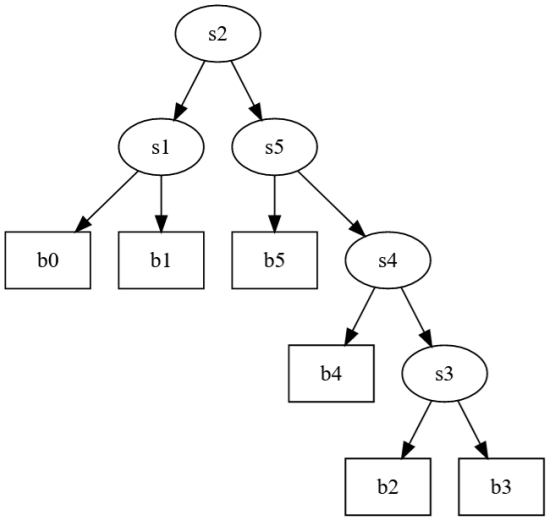
1.3.2 Top-down splayboom

1.4 Gerandomiseerde zoekbomen

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 geminimaliseert 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.

- LGD kan bepaald worden door recursieve vergelijking:

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

- 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

Samenvoegbare heaps

- Een samenvoegbare heap is een heap die geoptiamliseerd zijn o mde 'join' operatie uit te voeren.
- De join operatie voegt twee heaps samen, zodat de **heapvoorwaarde** nog steeds geldig is.
- Een lijst van voorbeelden die niet gekend moeten zijn (In de cursus is er geen uitleg over hoe dat je de samenvoegoperatie zou implementeren voor deze heaps):
  - Leftist tree.
  - Skew heaps.
  - Fibonacci heaps.
  - Relaxed heaps

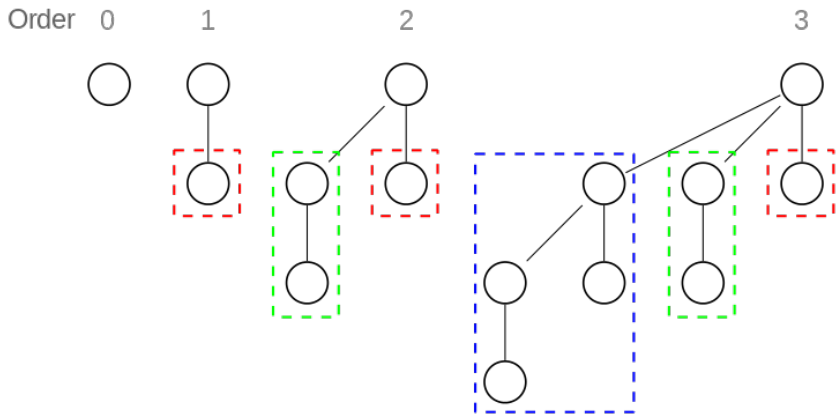
De belangrijke samenvoegbare heaps zijn: **Binomial queues** en **Pairing heaps**.

3.1 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 3.1)
- 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:

- **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.

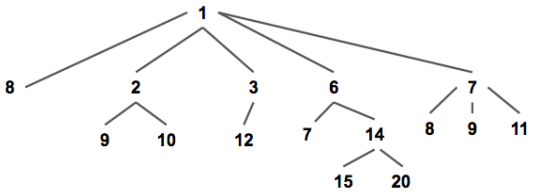


Figuur 3.1: Verschillende ordes van binomiaalbomen.

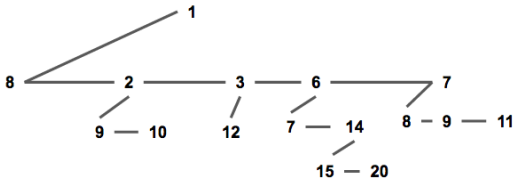
- **Toevoegen**: Maak een triviale binomiaalqueue met één knoop en voeg deze samen met de andere binomiaalqueue.
- **Minimum verwijderen**: Zoek binomiaalboom B_k met het kleinste wortelelement. Verwijder deze uit de binomiaalqueue. Verwijder wortel van B_k. Voeg beide binomiaalqueues terug samen.

3.2 Pairing heaps

- Een algemene boom waarvan de sleutels voldoen aan de heapvoorwaarde.
- Elke knoop heeft een wijzer naar zijn linkerkind en rechterbroer (figuur 3.2 en 3.3). 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 3.2: Een pairing heap in boomvorm.



Figuur 3.3: Een pairing heap.

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 4

Toepassingen van diepte-eerst zoeken

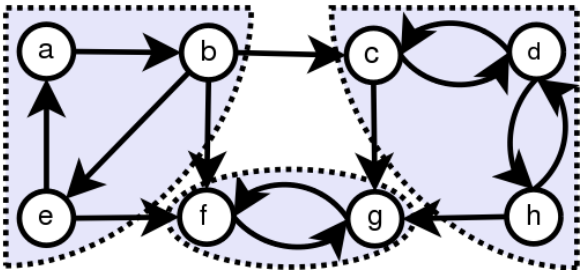
4.1 Enkelvoudige samenhang van grafen

4.1.1 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.

4.1.2 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 4.1).



Figuur 4.1: Een sterk samenhangende graaf.

- Een **zwak samenhangende gerichte graaf** is een graaf die niet sterk, maar toch samenhangend is indien de richtingen buiten beschouwing gelaten worden. Een graaf die niet sterk samenhangend is, bestaat uit zo groot mogelijke sterk samenhangende componenten.
- Sommige algoritmen gaan ervan uit de een graaf sterk samenhangend is. Men moet dus eerst deze componenten bepalen, meestal via een **componentengraaf** die:
  - 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, anders wil dit zeggen dat die knoop zelf nog opgesplitst zou kunnen worden in twee sterk samenhangende componenten.
- De sterk samenhangende componenten 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 de resterende knoop met het hoogste postordernummer. Het resultaat is een diepte-eerst bos, waarvan de bomen sterk samenhangende componenten zijn.
- 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.

4.2 Dubbele samenhang van ongerichte grafen

Twee definities:

- **Brug.** 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.
- **Scharnierpunt.** 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 postorder 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.

4.3 Eulergraaf

Een eulercircuit is een gesloten omloop in een graaf die alle verbindingen éénmaal bevat. Een eulergraaf is een graaf met een eulercircuit, die volgende eigenschappen heeft:

- De graaf is knoopsamenhangend.
- De graad van elke knoop is even.
- De verbindingen kunnen onderverdeeld worden in lussen, waarbij elke verbinding slechts behoort tot één enkel lus.

Hoofdstuk 5

Kortste afstanden II

5.1 Kortste afstanden vanuit één knoop

5.1.1 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. (Dijkstra gebruikt het feit dat indien een pad naar A $\rightarrow$ C bestaat, er geen korter pad A $\rightarrow$ B $\rightarrow$ C kan zijn, daarom is het algoritme performant, maar er mogen dus geen negatieve verbindingen zijn. Indien B negatief zou zijn dan klopt Dijkstra niet.).

- 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.

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

met

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

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.

5.2 Kortste afstanden tussen alle knopenparen

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

5.2.1 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.

5.3 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 veel 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 bevat.
     - 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.

Hoofdstuk 6

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**.

6.1 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**.
  - Iteratief algoritme.
  - 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 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 (komt bijna nooit voor maar just in case).

- De verzameling van stromen voor alle mogelijke knopenparen in beide richtingen wordt een **stroomverdeling** genoemd.
- De verzameling mogelijke stroomtoename tussen elk paar knopen wordt het **restnetwerk** genoemd.
  - Het restnetwerk bevat dezelfde knopen, maar niet noodzakelijk dezelfde verbindingen en capaciteiten.
  - In het restnetwerk wordt de **vergrotende weg** gezocht.
  - Als er geen vergrotende weg is dan zal de networkstroom maximaal zijn.
  - Voor elk mogelijke stroomverdeling bestaat er een overeenkomstig restnetwerk.

Deel III

Strings

Hoofdstuk 7

Gegevensstructuren voor strings

7.1 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.

! 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`.

7.2 Digitale zoekbomen

- Sleutels worden opgeslagen in de knopen.
- Zoeken en toevoegen verloopt analoog.
- 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.
- Hier 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 zoeksleutels zijn niet noodzakelijk in volgorde van toevoegen.

  - Sleutels in de linkerdeelboom van een knoop op diepte i zijn kleiner dan deze in de rechterdeelboom, maar ToDo: wat?

- De hoogte van een digitale zoekboom wordt bepaald door het aantal bits van de langste sleutel.

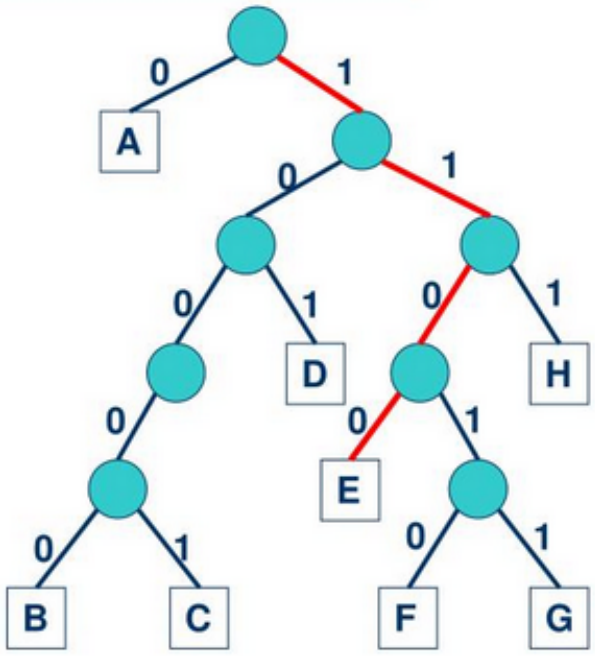
- Performantie is vergelijkbaar met rood-zwarte bomen:
  - Voor een groot aantal sleutels met relatief kleine bitlengte is het zeker beter dan een binaire zoekboom en vergelijkbaar met die van een rood-zwarte boom.
  - Het aantal vergelijkingen is trouwens nooit meer dan het aantal bits van de zoeksleutel.
  - ✓ 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.

7.3 Tries

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

7.3.1 Binaire tries

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



Figuur 7.1: Een voorbeeld van een binaire trie met opgeslagen sleutels A, B, C, D, E, F, G en H. Elk van deze sleutels heeft een (willekeurige) bitrepresentatie die de individuele elementen van de sleutels voorstelt. De zoekweg van de sleutel E wordt aangegeven door rode verbindingen.

- Zoekweg wordt bepaald door de opeenvolgende bits van de zoeksleutel.
- Sleutels worden enkel opgeslaan in de bladeren.
  - 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 `testen` zoeken maar de boom bevat enkel de sleutel `test`, zullen we in het blad met de sleutel `test` uitkomen aangezien de eerste 4 elementen hetzelfde zijn. De sleutels zijn echter niet gelijk.
     - Indien de sleutels niet hetzelfde zijn, zijn er terug twee mogelijkheden:
       (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 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.

- De structuur is onafhankelijk van de toevoegvolgorde van de sleutels.
- De sleutels in de zoekweg worden enkel getest op de bit die op dat niveau van toepassing is.

7.3.2 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.

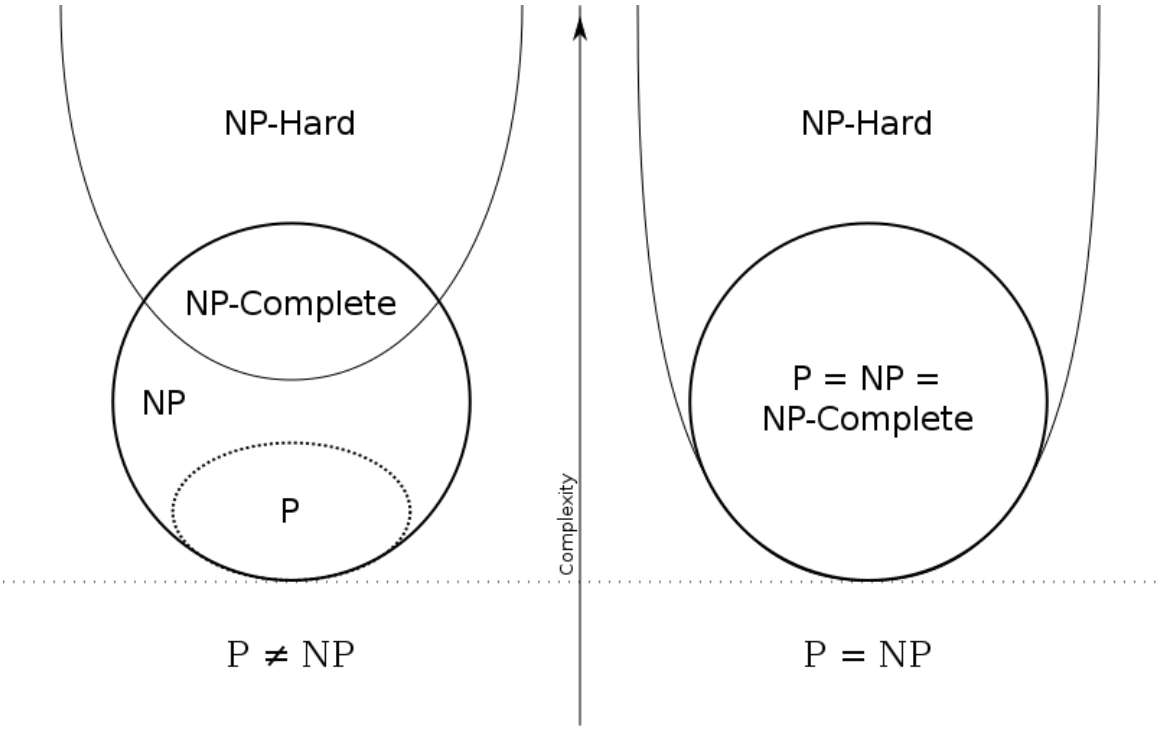
Deel IV

Hardnekkige problemen

Hoofdstuk 8

NP

8.1 Complexiteit: P en NP



Figuur 8.1: De linkse deelfiguur toont de verschillende complexiteitsklassen indien $P \neq NP$. De rechtse deelfiguur toont hetzelfde indien $P = NP$.

- Alle besproken algoritmen hadden 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.

8.1.1 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 $\rightarrow$ **P** vs **NP** probleem (Figuur 8.1).
  - 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.

8.2 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.

8.2.1 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:

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

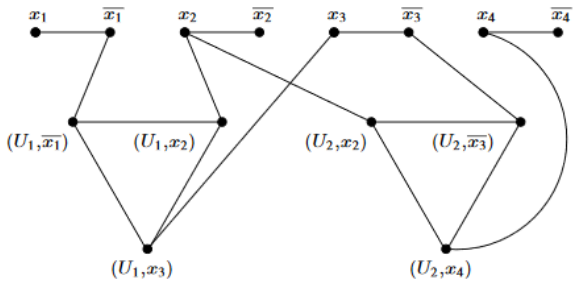
8.2.2 Vertex Cover

- 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

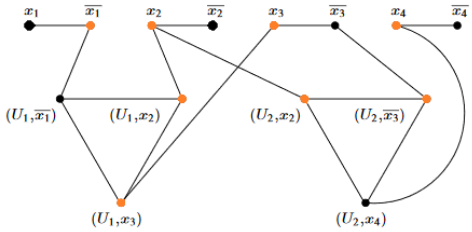
$$U_1 = \overline{x}_1 \vee x_2 \vee x_3$$
$$U_2 = x_2 \vee \overline{x}_3 \vee x_4$$

Bijhorende graaf:



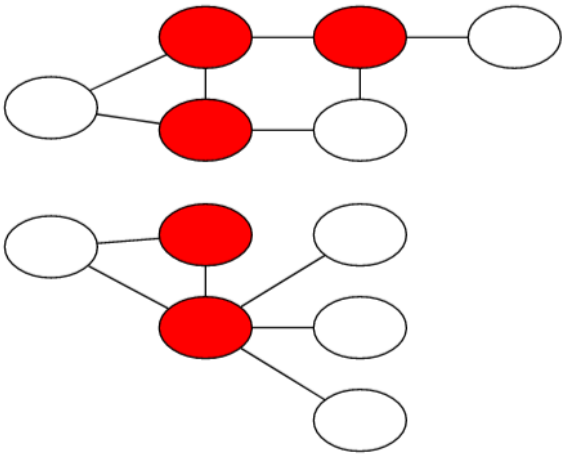
Figuur 8.2: Een graaf voor het 3SAT-probleem.

Voorbeeld van een minimale vertex cover:



Figuur 8.3: Een minimale vertex cover van de graaf op figuur 8.2

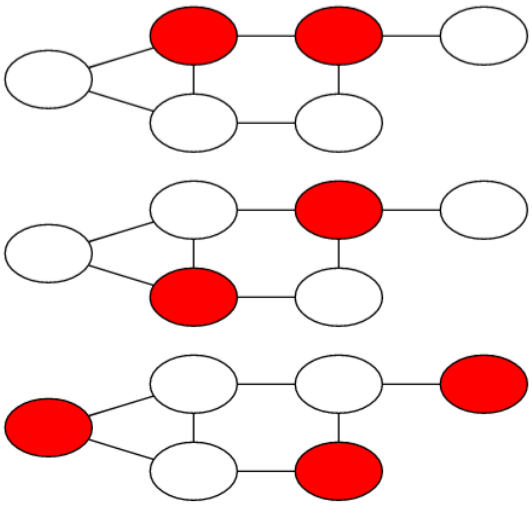
- Andere voorbeelden van minimale vertex covers:



Figuur 8.4: De minimale vertex cover van twee verschillende grafen.

8.2.3 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 8.5: Voorbeelden van dominating sets.

8.2.4 Graph Coloring

8.2.5 Clique

8.2.6 Independent set

8.2.7 Hamilton path

8.2.8 Minimum cover

8.2.9 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.

8.2.10 Partition

8.2.11 TSP

8.2.12 Longest path

8.2.13 Bin packing

8.2.14 Knapsack

Hoofdstuk 9

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.

9.1 Combinatorische optimisatie

- Abstracte representatie van de problemen nodig.
  - Het zijn **optimisatie**-problemen.
  - Voor een verzameling $\mathcal{S}$ moet 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.