# Algoritmen en Datastructuren 2

Samenvatting

VUB Computerwetenschappen

Academiejaar 2025–2026

## Contents

# 1 Hoofdstuk 8: The Union-Find Problem

## 1.1 Introductie en Motivatie

**Het Union-Find Probleem**

Verschillende objecten kunnen dezelfde *waarde* vertegenwoordigen. Het union-find probleem gaat over het efficiënt bepalen of twee datawaarden **equivalent** zijn volgens een applicatie-specifieke, dynamisch evoluerende gelijkheidsdefinitie.

- De "gelijkheid" is **applicatie-specifiek** en evolueert dynamisch in de tijd.
- Twee sleuteloperaties: **union** (laat de relatie evolueren) en **find** (test equivalentie).

**Equivalentierelatie en equivalentieklassen**

Twee elementen $x$ en $y$ zijn *equivalent* ($x \equiv y$) als de relatie voldoet aan:

- **Reflexief:** $x \equiv x$
- **Symmetrisch:** $x \equiv y \Rightarrow y \equiv x$
- **Transitief:** $x \equiv y \wedge y \equiv z \Rightarrow x \equiv z$

De **equivalentieklasse** van $y$: $\overline{y} = \{x \in S \mid x \equiv y\}$, met:

- $y \in \overline{y} \neq \varnothing$
- $\overline{y} = \overline{z}$ of $\overline{y} \cap \overline{z} = \varnothing$ (klassen zijn *disjunct* of identiek)

Equivalentieklassen van equivalentierelaties worden ook **partities** genoemd.

## 1.2 Het Disjoint Sets ADT

**Abstract Datatype: disjoint-sets⟨ID⟩**

We willen objecten koppelen aan equivalentieklassen. We definiëren:

- `new(n)`: maak $n$ singleton-verzamelingen $\{0\}, \{1\}, \ldots, \{n-1\}$
- `find(s, nmbr)`: geeft de *identiteit* (ID) van de klasse waartoe `nmbr` behoort
- `union!(s, id1, id2)`: voegt de klassen met identiteiten `id1` en `id2` samen
- `same-set?(id1, id2)`: controleert of twee identiteiten dezelfde klasse voorstellen

Het type van de klasse-identiteit is generiek (`ID`); het enige vereiste is dat `same-set?` correct werkt.

**Voorbeeld:** Na `new(10)` hebben we $d = \{\{0\}, \{1\}, \ldots, \{9\}\}$.
Na `union!(d, find(d,1), find(d,8))` en vervolgens `union!(d, find(d,1), find(d,9))` wordt $d = \{\{0\}, \{2\}, \ldots, \{7\}, \{1, 8, 9\}\}$.

## 1.3 Toepassing: Doolhofgeneratie

**Maze Generation via Union-Find**

Een doolhof is een $N \times N$ raster. Elke cel slaat 2 bits op:

- Rechterbit: verticale muur (naar rechter buur)
- Linkerbit: horizontale muur (naar onder buur)

Opslagkost: $\lceil 2N^2/8 \rceil$ bytes.

**Algoritme:**

1. Initialisatie: alle muren "aan" (11 overal). Kies een uitgang in de laatste kolom.
2. Zolang `find(ingang)` $\neq$ `find(uitgang)`:
   - Kies willekeurig een muur (bit = 1) en bepaal de buurcel.
   - Als `find(cel)` $\neq$ `find(buur)`: breek de muur af en voer `union!` uit.

We nemen aan dat een reeks van $M$ gemixte `union!`/`find`-operaties wordt uitgevoerd, met $M \in \Omega(n)$.

## 1.4 Implementatie 1: Naive (Array van canonieke elementen)

**Naïeve implementatie**

Sla in een vector `E[i]` de klasse-identiteit (canoniek element) op van element $i$.

- `new(n)`: $E = [0, 1, \ldots, n-1]$ (elk element is zijn eigen klasse)
- `find(s, nmbr)`: `return E[nmbr]` $\Rightarrow$ $\boxed{O(1)}$
- `union!(s, e1, e2)`: voor elke $i$: als $E[i] = e_1$, stel $E[i] := e_2$ $\Rightarrow$ $\boxed{O(n)}$

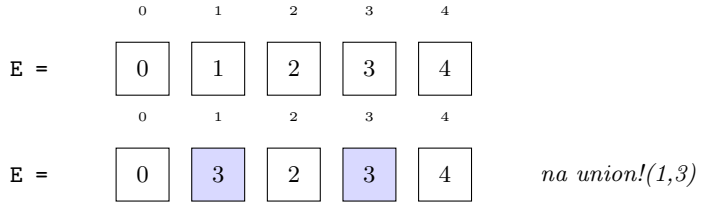



Figure 1: Naïeve representatie: array van canonieke elementen

## 1.5 Implementatie 2: Up-Trees

**Up-tree implementatie**

Elk element wijst naar zijn *ouder* in de boom (omgekeerd t.o.v. normale bomen). De **wortel** (leader) verwijst naar zichzelf.

- `new(n)`: voor elke $s$: `parent(s) = s` (singleton-bomen)
- `find(s, nmbr)`: volg ouderwijzers tot de wortel $\Rightarrow$ $\boxed{O(n)}$ worst-case
- `union!(s, e1, e2)`: `e1.parent = e2` $\Rightarrow$ $\boxed{O(1)}$

**Probleem: Degeneratie**

Zonder preventieve maatregelen kunnen up-trees degenereren tot een *gelinkte lijst*, waardoor `find` in het *slechtste geval* $O(n)$ wordt.

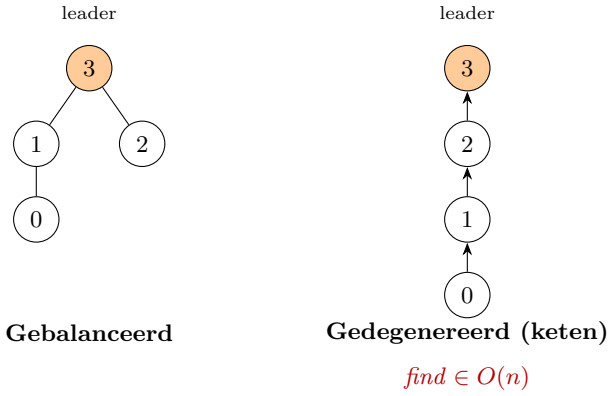



Figure 2: Up-tree: gebalanceerd vs. gedegenereerd

## 1.6 Geoptimaliseerde Implementatie

### 1.6.1 Optimalisatie 1: Ranked Trees (Rank-aware Union)

**Rank**

Elke wortel slaat een **rank** op, een bovenschatting van de hoogte van de boom:

- rank(leaf) $= 0$
- rank(internal node) $= 1 + \max(\text{rank(children)})$

**Union-regels met rank:**

1. Als $\text{rank}(e_1) > \text{rank}(e_2)$: hang $e_2$ onder $e_1$.
2. Als $\text{rank}(e_1) < \text{rank}(e_2)$: hang $e_1$ onder $e_2$.
3. Als $\text{rank}(e_1) = \text{rank}(e_2)$: kies willekeurig en verhoog de rank van de nieuwe leider met 1.

`union!` blijft $O(1)$.

**Examen: Invloed van rank-aware uptrees (Rank-aware uptrees)**

**Vraag:** Leg uit hoe rank-aware uptrees (union-by-rank) de complexiteit verbeteren.
**Antwoord:** Door bomen met een lagere rank (hoogte-overschatting) altijd onder bomen met een hogere rank te hangen, garanderen we dat de maximale rank (en dus de boomhoogte) logaritmisch groeit: rank $\leq \log_2 n$. Hierdoor daalt de worst-case complexiteit van `find` van $O(n)$ naar $O(\log n)$, terwijl `union!` $O(1)$ blijft.

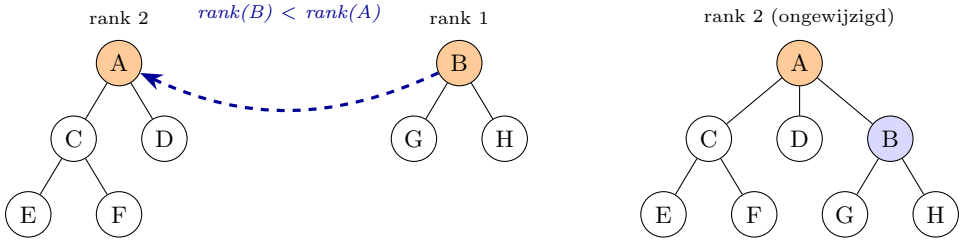



Figure 3: Rank-aware union: boom met lagere rank wordt onder hogere gehangen

**Bewijs: boom met rank $k$ heeft minstens $2^k$ knopen**

**Inductie:**

- *Basisgeval:* blad met rank 0 heeft $1 = 2^0$ knoop. ✓
- *Inductiestap:* een boom met rank $k+1$ ontstaat door twee bomen met rank $k$ te mergen (geval 3). Dat geeft $2^k + 2^k = 2^{k+1}$ knopen. Gevallen 1 en 2 voegen alleen maar meer knopen toe. ✓

**Gevolg:** $\text{rank(leader)} \leq \log_2 n$, dus `find` $\in O(\log_2 n)$.

| Operatie | Naïef | Up-trees | Ranked Up-trees |
|---|---|---|---|
| `union!` | $O(n)$ | $O(1)$ | $O(1)$ |
| `find` | $O(1)$ | $O(n)$ | $O(\log n)$ |

Table 1: Worst-case complexiteit tot nu toe

### 1.6.2 Optimalisatie 2: Path Compression

**Path Compression**

Tijdens een `find`-operatie is het pad naar de wortel *irrelevant* voor toekomstige operaties. Daarom laten we `find` **alle bezochte knopen rechtstreeks naar de wortel wijzen** tijdens het terugkeren.

Listing 1: find met path compression

```
find(s, nmbr):
  if nmbr.parent == nmbr:
    return nmbr
  else:
    nmbr.parent = find(s, nmbr.parent)   // herlink naar wortel
    return nmbr.parent
```

**Examen: Invloed van padcompressie**

**Vraag:** Wat is de invloed van padcompressie op de complexiteit van `find` en `union`?
**Antwoord:** Padcompressie vlakt de bomen af door alle bezochte knopen rechtstreeks aan de wortel te hangen. Hierdoor worden alle toekomstige `find`-operaties voor die knopen versneld naar $O(1)$. In combinatie met rank-aware union daalt de *geamortiseerde* kost voor een reeks van `find`/`union` operaties naar vrijwel constant: $O(\alpha(n))$, hoewel een *enkele* `find` nog worst-case $O(\log n)$ kan duren.

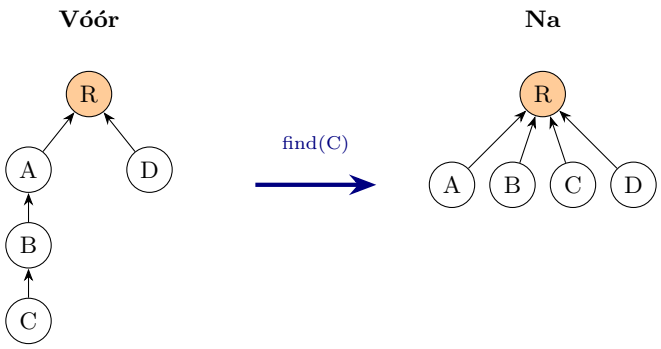



Figure 4: Path compression: alle bezochte knopen wijzen direct naar de wortel

**Effecten:**
- Ranks worden *overschattingen* van de werkelijke hoogte, maar blijven bruikbaar voor union-beslissingen.
- Worst-case van één enkele `find` blijft $O(\log n)$.
- Over een reeks operaties: "doe nu werk, profiteer later" $\Rightarrow$ **geamortiseerde analyse** nodig.

## 1.7 Geamortiseerde Analyse

**Geamortiseerde complexiteit**

De "gemiddelde" worst-case kost over een *reeks* operaties (niet over willekeurige invoer), onder de aanname dat we weten welke reeksen van operaties mogelijk zijn.

### 1.7.1 Methode 1: Aggregatiemethode

**Aggregatiemethode**

Bepaal de bovengrens $f_M(n)$ op de totale kost van $M$ individuele operaties, en bereken de gemiddelde kost als $f_M(n)/M$.

**Voorbeeld: ArrayList Doubling.** Laat $c_i$ de kost van de $i$-de `push!` operatie zijn:

$$c_i = \begin{cases} i & \text{als } i-1 \text{ een macht van 2 is} \\ 1 & \text{anders} \end{cases}$$

Totale kost over $M$ push-operaties:

$$\sum_{i=1}^{M} c_i = M + \sum_{j=0}^{\lfloor \log_2(M-1) \rfloor} 2^j \leq 3M \in O(M)$$

Geamortiseerde kost per `push!`: $O(M)/M = O(1)$.

### 1.7.2 Methode 2: Boekhoudmethode (Accounting Method)

**Boekhoudmethode**

Wijs elke $i$-de operatie een fictieve kost $\hat{c}_i$ toe (in "euro's", waarbij 1 euro = 1 rekenstap). Een deel wordt direct "uitgegeven", de rest gaat naar de bank. Bij dure operaties halen we geld van de bank.
**Voorwaarde:** De bankbalans mag *nooit negatief* worden:

$$\sum_{i=1}^{M} \hat{c}_i \geq \sum_{i=1}^{M} c_i$$

**Voorbeeld: ArrayList Doubling.** Kies $\hat{c}_i = 3$: 1 euro om het element op te slaan, 2 euro om in de bank te stoppen (voor toekomstige kopieerkosten). De bank wordt nooit negatief, en:

$$\sum_{i=1}^{M} \hat{c}_i \leq 3M \Rightarrow \text{geamortiseerde kost per operatie} = O(1)$$

### 1.7.3 Methode 3: Potentiaalmethode

**Potentiaalmethode**

Definieer een **potentiaalfunctie** $\Phi : \{D_i\} \to \mathbb{R}$ op de datastructuur, met $\Phi(D_0) = 0$ en $\Phi(D_i) \geq 0$ voor alle $i$.
De **geamortiseerde kost** van de $i$-de operatie is:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = c_i + \Delta\Phi_i$$

Over $M$ operaties:

$$\sum_{i=1}^{M} \hat{c}_i = \sum_{i=1}^{M} c_i + \Phi(D_M) - \Phi(D_0) \geq \sum_{i=1}^{M} c_i$$

omdat $\Phi(D_M) \geq 0$ en $\Phi(D_0) = 0$.

**Voorbeeld: ArrayList.** Met $\Phi(D_i) = 2i - 2^{\lceil \log(i) \rceil}$:

$$\hat{c}_i = 3 \quad \text{(in beide gevallen)} \Rightarrow \sum_{i=1}^{M} \hat{c}_i = 3M \Rightarrow O(1) \text{ per operatie.}$$

| Methode | Creatieve stap |
|---|---|
| Aggregatie | Tel hoeveel operaties duur zijn in een reeks van $M$ |
| Boekhouding | Kies $\hat{c}_i$ zodat de bank nooit negatief wordt |
| Potentiaal | Kies $\Phi(D_i) \geq 0$; hoe scherper $\Phi$, hoe beter het resultaat |

Table 2: Samenvatting van analyse-methoden

## 1.8 Geamortiseerde Analyse van Union-Find

### 1.8.1 Wiskundig gereedschap

**Tetration (Power Tower)**

$$2 \uparrow\uparrow k = \begin{cases} 1 & \text{als } k = 0 \\ 2^{2 \uparrow\uparrow (k-1)} & \text{als } k > 0 \end{cases} = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{k \text{ niveaus}}$$

Groeit *extreem snel*: $2 \uparrow\uparrow 0 = 1$, $2 \uparrow\uparrow 1 = 2$, $2 \uparrow\uparrow 2 = 4$, $2 \uparrow\uparrow 3 = 16$, $2 \uparrow\uparrow 4 = 65536$, $2 \uparrow\uparrow 5 = 2^{65536}$.

**Superlogaritme**

$$\log_2^*(k) = \begin{cases} 0 & \text{als } k = 0 \text{ of } k = 1 \\ 1 + \log_2^*(\lceil \log_2 k \rceil) & \text{anders} \end{cases}$$

Groeituiterst *traag*: $\log_2^*(4) = 2$, $\log_2^*(16) = 3$, $\log_2^*(65536) = 4$, $\log_2^*(2^{65536}) = 5$.
Aangezien $2^{65536}$ groter is dan het aantal atomen in het heelal, geldt $\log_2^* < 5$ voor alle praktische doeleinden.

### 1.8.2 Lemma's over ranks en blokken

**Sleutellemma's**

1. **Lemma 8.3:** Op een pad van knoop naar wortel nemen ranks *strikt monotoon toe*.
2. **Lemma 8.4:** Zodra een knoop geen leider meer is, verandert zijn rank niet meer.
3. **Lemma 8.5:** Een boom met rank $r$ bevat minstens $2^r$ knopen.
4. **Lemma 8.6:** Er zijn maximaal $n/2^r$ knopen met rank $r$.
5. **Lemma 8.7:** $\text{rank}(x) \leq \lfloor \log_2 n \rfloor$ voor elke knoop $x$.

**Blokindeling van knopen**

$$\text{block}(x) = \log_2^*(\text{rank}(x))$$

- Er zijn maximaal $\log_2^*(n)$ blokken.
- Het aantal knopen in blok $b$ is begrensd door: $n_b \leq \dfrac{n}{2 \uparrow\uparrow b}$.

### 1.8.3 Union-Find met de Boekhoudmethode

**Kosten met de boekhoudmethode**

**UNION** (als knoop $x$ geen leider meer wordt):

$$\hat{c}_i = 1 + 2 \uparrow\uparrow \text{block}(x)$$

De 1 is voor de union zelf; de rest gaat naar de bankrekening van $x$.
**FIND:**

$$\hat{c}_i = \log_2^*(n) + 2$$

Financiering tijdens `find` (pad $x_0, x_1, \ldots, x_k$):

- $x_k$ en $x_{k-1}$: betaald uit $\hat{c}_i$ (de "+2").
- $x_j$ met $x_j$ en ouder $x_{j+1}$ in *verschillende* blokken: betaald uit $\hat{c}_i$ ($\leq \log_2^*(n)$ blokken).
- $x_j$ met $x_j$ en ouder $x_{j+1}$ in *hetzelfde* blok: betaald uit $x_j$'s bankrekening.

**Resultaat:** De totale kost voor $K$ unions en $M - K$ finds is:

$$K + n \log_2^*(n) + (M - K) \log_2^*(n) = O\big((n + M) \log_2^*(n)\big)$$

Met $M = \Omega(n)$: de **geamortiseerde kost is** $O(\log_2^*(n))$ per operatie.

### 1.8.4 Verbetering met de Potentiaalmethode: Inverse Ackermann

**Ackermanfunctie**

$$A_k(j) = \begin{cases} j + 1 & \text{als } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{als } k \geq 1 \end{cases}$$

waarbij $f^{(j)}(x) = f(f^{(j-1)}(x))$ ($j$-voudige toepassing). Voorbeelden: $A_1(j) = 2j + 1$, $A_2(j) = 2^{j+1}(j + 1) - 1$.

**Inverse Ackermanfunctie**

$$\alpha(n) = \min\{k \mid A_k(1) \geq n\}$$

Groeit *onvoorstelbaar traag*: $\alpha(n) \leq 4$ voor alle $n \leq 16^{512}$.

**Eindresultaat Union-Find**

Met de potentiaalmethode: een reeks van $M$ union- en find-bewerkingen (met path compression en ranked unions) heeft een totale kost in $O(M \cdot \alpha(n))$.
**Geamortiseerde kost per operatie:** $O(\alpha(n))$, *praktisch* $O(1)$.

**Geamortiseerde Analyse & $\log^* n$**

De strikte wiskundige bewijsvoering (*accounting method* / potentiaalmethode) wijst credits toe aan knopen en toont aan dat de boom afvlakt na verloop van tijd. Historisch bewezen Hopcroft en Ullman dat rank-aware union met path compression $O(\log^* n)$ is (de iteratieve logaritme: het aantal keer dat je de logaritme moet nemen om $\leq 1$ te bekomen). Tarjan verbeterde deze grens later tot de inverse Ackermann functie $O(\alpha(n))$, wat een nog tragere stijging is.

**Let op:** één enkele `find` blijft worst-case $O(\log_2 n)$; één enkele `union!` is $O(1)$.

## 1.9 Samenvattende Tabel

| | **Naïef** (worst-case) | **Up-trees** (worst-case) | **Ranked** (worst-case) | **+ Path Compr.** (geamortiseerd) |
|---|---|---|---|---|
| `union!` | $O(n)$ | $O(1)$ | $O(1)$ | $O(\alpha(n))$ |
| `find` | $O(1)$ | $O(n)$ | $O(\log n)$ | $O(\alpha(n))$ |

Table 3: Volledige overzichtstabel performance Union-Find

# 2 Hoofdstuk 9: Grafen

## 2.1 Concepten en Definities

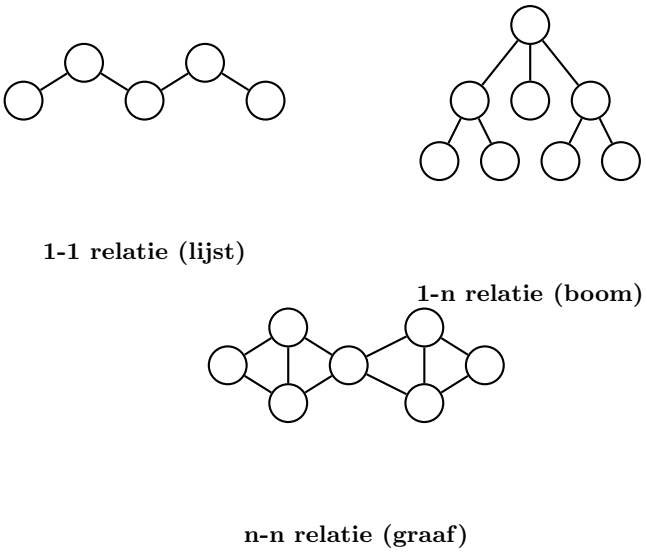



Figure 5: Overzicht: lijsten, bomen en grafen als relatie-types

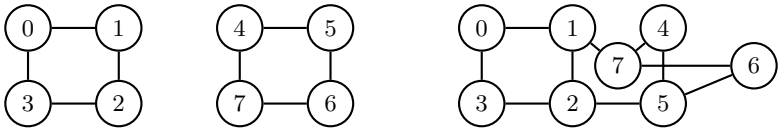
**Ongerichte graaf**

Een **ongerichte graaf** is een koppel $G = (V, E)$ met:
- $V$: eindige verzameling **knopen** (nodes)
- $E$: eindige verzameling **bogen** (edges), waarbij elke $e = \{u, v\} \subseteq V$ met $u \neq v$

De **orde** van de graaf is $|V|$. De **graad** van een knoop $u$ is het aantal bogen $e$ waarvoor $u \in e$.

**Let op:** Self-loops ($\{u, u\}$) zijn *niet* toegestaan in onze definitie.

Maximum aantal bogen: $|E| \leq \dfrac{|V|(|V|-1)}{2} = C(|V|, 2)$.




*(a) niet samenhangend* *(b) samenhangend*

Figure 6: Ongerichte grafen (naar Figuur 9.1 uit het cursusboek)

**Dense vs. Sparse**

- **Dense (dicht):** $|E| \sim |V|(|V|-1)/2$
- **Sparse (ijl):** $|E| \ll |V|(|V|-1)/2$

De keuze van representatie en algoritme hangt hier sterk van af.

**Gerichte graaf (digraph)**

Een **gerichte graaf** is een koppel $G = (V, E)$ met $E \subseteq V \times V$, d.w.z. elke boog $e = (u, v)$ is een *geordend* koppel. Self-loops zijn wél toegestaan.

- **In-graad** van $u$: $|\{(v, u) \in E\}|$
- **Uit-graad** van $u$: $|\{(u, v) \in E\}|$
- **Graad** van $u$: in-graad + uit-graad

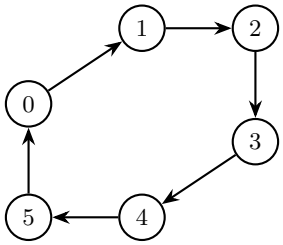



Figure 7: Gerichte graaf: cyclus $0 \to 1 \to 2 \to 3 \to 4 \to 5 \to 0$

**Gewogen graaf**

Een graaf $G = (V, E)$ is **gewogen** als er een functie $w_G : E \to \mathbb{R}$ bestaat die aan elke boog een numerieke waarde (gewicht) toekent.

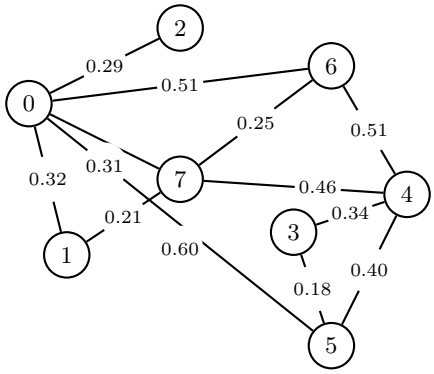



Figure 8: Gewogen ongerichte graaf (naar Figuur 9.5 uit het cursusboek)

### 2.1.1 Paden, Cycli en Connectiviteit

**Pad en Cyclus**

- Een **pad** is een rij knopen $v_0 v_1 \ldots v_k$ waarbij $v_i$ en $v_{i+1}$ adjacent zijn.
- Een **eenvoudig pad** is een pad waarin alle knopen *verschillend* zijn.
- Een **cyclus** is een pad waarbij $v_0 = v_k$ en alle tussenliggende knopen verschillend zijn.

**Connectiviteit**

- **Deelgraaf:** $G' = (V', E')$ met $V' \subseteq V$ en $E' \subseteq E$.
- Een ongerichte graaf is **samenhangend** (connected) als er een pad bestaat tussen elk paar knopen.
- Een niet-samenhangende graaf bestaat uit **samenhangende componenten**: maximaal samenhangende deelgrafen.
- Een gerichte graaf is **sterk samenhangend** (strongly connected) als er een *gericht* pad bestaat van elke knoop naar elke andere knoop.
- **Sterk samenhangende componenten (SCC's):** maximaal sterk samenhangende deelgrafen.

**Bos en Boom (als graaf)**

- Een **bos** (forest) is een ongerichte graaf *zonder cycli*.
- Een **boom** (tree) is een *samenhangend* bos.
- Een **gewortelde boom** (rooted tree) heeft één aangewezen knoop als wortel.

## 2.2 Graaf-ADT's

### 2.2.1 ADT: Ongewogen Grafen

**ADT unweighted-graph**

| | |
|---|---|
| `new(d, n)` | Maak een nieuwe graaf met $n$ knopen; `d` = `#t` voor gericht, `#f` voor ongericht |
| `order(g)` | Geeft het aantal knopen $\|V\|$ |
| `nr-of-edges(g)` | Geeft het aantal bogen $\|E\|$ |
| `directed?(g)` | Is de graaf gericht? |
| `add-edge!(g, u, v)` | Voeg boog $(u, v)$ toe |
| `delete-edge!(g, u, v)` | Verwijder boog $(u, v)$ |
| `adjacent?(g, u, v)` | Zijn $u$ en $v$ verbonden door een boog? |
| `for-each-node(g, f)` | Pas $f$ toe op elke knoop |
| `for-each-edge(g, u, f)` | Pas $f$ toe op elke buur van $u$ |

### 2.2.2 ADT: Gewogen Grafen

**ADT weighted-graph**

Gelijkaardig aan `unweighted-graph`, met als verschillen:

- `add-edge!(g, u, v, w)`: extra parameter `w` voor het gewicht
- `weight(g, u, v)`: geeft het gewicht van de boog; $+\infty$ als geen boog bestaat
- `adjacent?(g, u, v)`: `weight(g,u,v)` $\neq +\infty$
- `for-each-edge(g, u, f)`: $f$ krijgt twee argumenten: gewicht en doelknoop

### 2.2.3 Elementaire Grafalgoritmen

**Basisfuncties met het ADT**

1. **Graadberekening:** itereer over alle knopen met `for-each-node`, en voor elke knoop over alle buren met `for-each-edge`. Tel het aantal bogen.
2. **In-/Uit-graad (gericht):** gelijkaardig, maar bijhoud twee vectoren (in-degs en out-degs).
3. **Connectiviteitstest:** recursief DFS met een *node-indexed vector* `visited` om oneindige recursie te voorkomen bij cycli.

**Concept: Node-indexed vector.** Een vector van lengte $|V|$ die per knoop berekende informatie opslaat. Wordt zowel als uitvoer (resultaat) als als stuurelement (visited-markering) gebruikt.

## 2.3 Grafrepresentaties

Er bestaan drie fundamentele manieren om grafen in het geheugen op te slaan.

### 2.3.1 Bogenlijst (Edge List)

**Edge List**

Sla alle bogen op in een gelinkte lijst.

- `add-edge!`: $O(1)$
- `for-each-edge`: $O(|E|)$ (moet *alle* bogen doorlopen)
- `adjacent?`: $O(|E|)$

$\Rightarrow$ Niet praktisch, wordt niet verder bestudeerd.

### 2.3.2 Adjacentielijst (Adjacency List)

**Adjacency List**

De graaf is een vector met per knoop $n$ een **gesorteerde gelinkte lijst** van alle knopen adjacent aan $n$ (de *adjacentielijst*).

- Ongerichte bogen worden in *beide* richtingen opgeslagen.
- Geheugen: $O(|V| + |E|)$

**Complexiteit:**

| | |
|---|---|
| `add-edge!` | $O(|V|)$ (invoegen in gesorteerde lijst) |
| `delete-edge!` | $O(|V|)$ |
| `adjacent?` | $O(|V|)$ (maar gemiddeld sneller door sortering) |
| `for-each-edge` | $O(|V|)$ (traverseer adjacentielijst) |

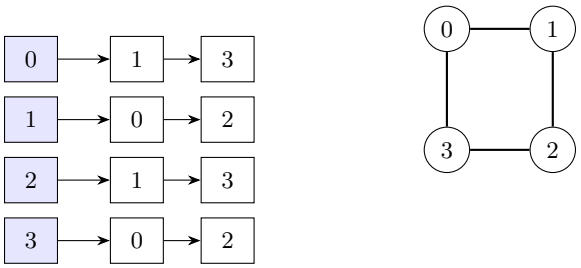



Figure 9: Adjacentielijst-representatie van een ongerichte graaf

**Gewogen variant:** Adjacentielijsten slaan *boog-objecten* (gewicht + doelknoop) op i.p.v. gewone nummers. Alle operaties blijven $O(|V|)$.

### 2.3.3 Adjacentiematrix (Adjacency Matrix)

**Adjacency Matrix**

De graaf is een 2D vector van booleans (ongewogen) of gewichten (gewogen). Entry $(u, v)$ = `#t` (of gewicht $w$) als er een boog $(u, v)$ bestaat.
**Optimalisatie voor ongerichte grafen:** sla een *onderdriehoeksmatrix* op (halve geheugenruimte), want $\{u, v\} = \{v, u\}$. Bij access: gebruik $\max(u, v)$ als rij-index en $\min(u, v)$ als kolomindex.

- Geheugen: $O(|V|^2)$, ongeacht $|E|$

**Complexiteit:**

| | |
|---|---|
| `add-edge!` | $O(1)$ |
| `delete-edge!` | $O(1)$ |
| `adjacent?` | $O(1)$ |
| `for-each-edge` | $\Theta(|V|)$ (hele rij doorlopen, ook als weinig bogen) |

**Gerichte graaf: volle matrix**

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | F | T | F | F |
| 1 | F | F | T | F |
| 2 | F | F | F | T |
| 3 | T | F | F | F |

**Ongerichte: onderdriehoek**

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | | | | |
| 1 | T | | | |
| 2 | F | T | | |
| 3 | T | F | T | |

Figure 10: Adjacentiematrix: volle matrix (gericht) vs. onderdriehoeksmatrix (ongericht)

**Gewogen variant:** Gebruik het gewicht i.p.v. een boolean; gebruik $+\infty$ voor afwezige bogen. `adjacent?(u,v)` $\Leftrightarrow$ `weight(u,v)` $\neq +\infty$.

## 2.4 Samenvattende Tabel: Grafrepresentaties

| | **Edge List** | **Adjacency List** | **Adjacency Matrix** |
|---|---|---|---|
| **Geheugen** | $O(\|E\|)$ | $O(\|V\| + \|E\|)$ | $O(\|V\|^2)$ |
| `add-edge!` | $O(1)$ | $O(\|V\|)$ | $O(1)$ |
| `delete-edge!` | $O(\|E\|)$ | $O(\|V\|)$ | $O(1)$ |
| `adjacent?` | $O(\|E\|)$ | $O(\|V\|)$ | $O(1)$ |
| `for-each-edge` | $\Theta(\|E\|)$ | $O(\|V\|)$ | $\Theta(\|V\|)$ |

Table 4: Vergelijking van grafrepresentaties

**Keuze van representatie**

- **Sparse grafen** $\Rightarrow$ adjacentielijst is beter (geheugen $O(|V| + |E|) \ll O(|V|^2)$)
- **Dense grafen** $\Rightarrow$ adjacentiematrix is attractiever (operaties in $O(1)$, geheugen vergelijkbaar)
- Sommige algoritmen zijn sneller op bepaalde representaties; bijv. $O(|V|^2)$ vs. $O(|E| \log |E|)$

## 2.5 Variatie: Gewogen Grafen

**Weighted Graphs**

In een **gewogen graaf** zijn bogen (en/of knopen) voorzien van een numeriek gewicht of kost (bijv. afstand op een wegenkaart).
Extra operaties t.o.v. het standaard ADT:

- `weight(g, node)`: lees het gewicht
- `weight!(g, node, w)`: schrijf het gewicht
- `add-edge!(g, u, v, weight)`: voeg een gewogen boog toe
- `for-each-node` en `for-each-edge`: de callback ontvangt ook het gewicht

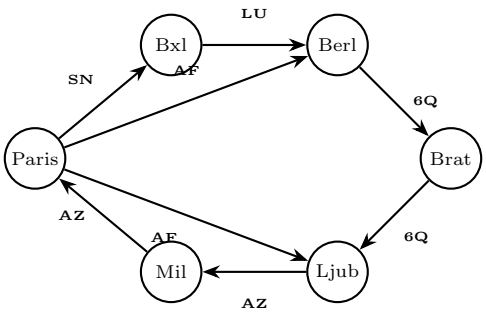


Figure 11: Gelabelde graaf: vliegroutes met luchtvaartmaatschappijen

## 2.6 Grafproblemen en -algoritmen (Overzicht)

**Drie categorieën grafalgoritmen**

1. **Maten berekenen:** bijv. het *chromatisch getal* (kleinste $k$ zodat de graaf $k$-kleurbaar is).
2. **Booleaanse queries:** bijv. "bevat de graaf cycli?"
3. **Deelgrafen berekenen:** bijv. opspannende bossen/bomen (spanning forests).

Bijna alle niet-triviale grafalgoritmen vereisen een **traversal** (doorlopen) van de graaf, hetzij *depth-first* (DFT), hetzij *breadth-first* (BFT). Deze worden uitgebreid behandeld in Hoofdstuk 10.

# 3 Hoofdstuk 10: Graaf-Traversals

Bijna elk niet-triviaal grafalgoritme is een instantie van een van twee fundamentele traversal-strategieën: **Depth-First Traversal (DFT)** of **Breadth-First Traversal (BFT)**. Beide worden geïmplementeerd als Scheme higher-order procedures die zes (DFT) resp. vier (BFT) callback-procedures ontvangen.

## 3.1 Intuïtie

**DFT vs. BFT, Kernidee**

- **DFT:** "ga zo diep mogelijk". Wanneer een knoop $v$ ontdekt wordt, worden eerst *al* zijn onbezochte buren recursief verkend vóór we terugkeren naar de buren van $v$'s voorganger. *Stack*-gestuurd (impliciet via de Scheme-runtime-stack).
- **BFT:** "ga zo breed mogelijk". Alle buren van $v$ worden eerst in een *queue* geplaatst; pas daarna worden ze verwerkt. Knopen worden nooit opnieuw bezocht.

**Niet-uniekheid**

Er bestaat niet "de" DFT of "de" BFT van een graaf. De uitkomst hangt af van:

1. de keuze van de *startknoop* (root),
2. de volgorde waarin buren beschouwd worden (`for-each-edge`),
3. de selectie van roots voor niet-samenhangende componenten.

## 3.2 DFT, Het Algoritme

**DFT Signatuur (7+1 argumenten)**

```
(dft graph
  root-discovered    ; root -> bool
  node-discovered    ; node -> bool
  node-processed     ; node -> bool
  edge-discovered    ; from to -> bool
  edge-processed     ; from to -> bool
  edge-bumped        ; from to -> bool
  . roots)           ; optioneel: lijst van roots
```

- Elk callback retourneert een boolean: zolang `#t` ⇒ traversal gaat door; `#f` ⇒ **abort** via `call/cc`.
- `edge-bumped`: opgeroepen bij boog naar een *reeds bezochte* knoop.
- `edge-discovered`/`edge-processed`: opgeroepen vóór/na het recursief verkennen van een *onbezochte* buurknoop.
- `node-discovered`/`node-processed`: opgeroepen wanneer een knoop voor het eerst ontdekt/volledig afgewerkt is.
- Optionele `roots`: beperkt de traversal tot knopen bereikbaar vanuit deze lijst.

**Null-operaties** ("no-ops") voor ongebruikte callbacks:

```
(define (root-nop root) #t)
(define (node-nop node) #t)
(define (edge-nop from to) #t)
```

## 3.3 DFT voor Ongerichte Grafen, Drie Eigenschappen

### 3.3.1 Eigenschap 1: Spanning Forest

**Opspannend Bos (Spanning Forest)**

Gegeven $G = (V, E)$. Een **opspannend bos** is een deelgraaf $G' = (V, E')$ die een bos is (geen cycli) en hetzelfde aantal samenhangende componenten heeft als $G$. Een opspannend bos van een samenhangende graaf is een **opspannende boom**.

DFT genereert automatisch een opspannend bos: elke boog naar een onbezochte knoop wordt een *tree edge*; bogen naar reeds bezochte knopen worden genegeerd.

```
(define (dft-forest g)
  (define children (make-vector (order g) '()))
  (define roots '())
  (dft g
    (lambda (root) (set! roots (cons root roots)))
    node-nop node-nop
    (lambda (from to)                    ; edge-discovered
      (vector-set! children from
        (cons to (vector-ref children from))))
    edge-nop edge-nop)
  (cons roots children))
```

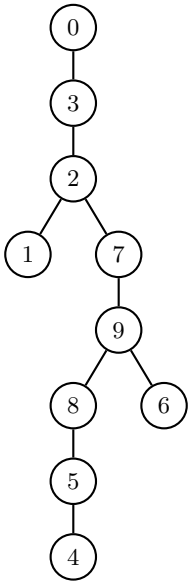



Figure 12: DFT-boom: smal en diep

### 3.3.2 Eigenschap 2: Edge Labeling ($E = T \cup B$)

**Boogclassificatie, Ongerichte Graaf + DFT**

DFT classificeert de bogen van een ongerichte graaf in twee groepen:

- **Tree edges ($T$):** bogen die deel uitmaken van het opspannend bos.
- **Back edges ($B$):** bogen naar een reeds bezochte knoop (die een voorouder is in de boom).

*Back edges corresponderen met cycli* in de oorspronkelijke graaf.

### 3.3.3 Eigenschap 3: Knoopnummering

Drie nummeringen onthullen de volgorde van bezoek:

1. **Pre-order $e(u)$:** oplopend nummer bij `node-discovered`.
2. **Post-order $o(u)$:** oplopend nummer bij `node-processed`.

3. **Visit time** $(d(u), f(u))$**:** discovery- en finishing-tijd (beide met dezelfde globale `time`-teller).

**Parenthesis Theorem (ongerichte grafen)**

Voor alle knopen $u, v$ geldt exact één van:

$$[d(u), f(u)] \cap [d(v), f(v)] = \emptyset, \qquad [d(u), f(u)] \subset [d(v), f(v)], \qquad [d(u), f(u)] \supset [d(v), f(v)].$$

Indien $u$ en $v$ adjacent zijn: $e(u) < e(v) \Leftrightarrow u$ is voorouder van $v$; $o(u) < o(v) \Leftrightarrow u$ is afstammeling van $v$.

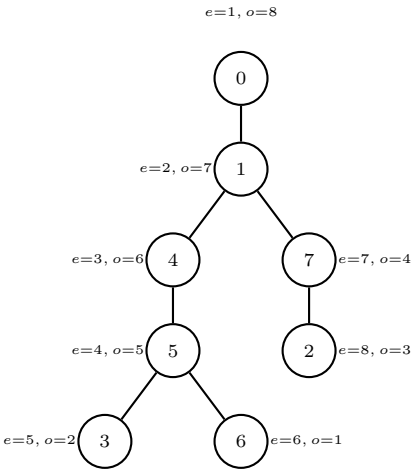



Figure 13: Pre-order en post-order nummering bij DFT

## 3.4 DFT voor Gerichte Grafen

### 3.4.1 Eigenschap 1: Spanning Forest

Analoog, maar *meer bomen* doordat pijlen in de "verkeerde" richting genegeerd worden.

### 3.4.2 Eigenschap 2: Edge Labeling ($E = T \cup B \cup C \cup D$)

**Vier soorten bogen bij gerichte DFT**

| | |
|---|---|
| **Tree (T)** | Boog naar een nog onontdekte knoop. |
| **Back (B)** | Boog naar een voorouder in de DFT-boom $\Rightarrow$ *cyclus*. Zelflussen zijn ook back edges. |
| **Down (D)** | Boog naar een *afstammeling* (bereikt via een ander pad). |
| **Cross (C)** | Boog naar een knoop in een *andere subboom* (reeds volledig verwerkt). |

Classificatie via pre/post-order nummers voor boog $(u, v)$ naar een reeds bezochte knoop:

- $o(v) > o(u) \Rightarrow$ **Back**
- $o(v) < o(u)$ en $e(v) < e(u) \Rightarrow$ **Cross**
- $o(v) < o(u)$ en $e(v) > e(u) \Rightarrow$ **Down**

**Examen: Categoriseer bogen bij DFT (DFS Edges)**

**Vraag:** Bespreek de 4 soorten edges bij DFS en het verschil tussen gerichte en ongerichte grafen.
**Antwoord:** In een *gerichte* graaf deelt DFT bogen in 4 categorieën in: Tree (naar onbezochte knoop), Back (naar voorouder, vormt cyclus), Down (naar afstammeling via korter pad), en Cross (naar andere subboom). In een *ongerichte* graaf bestaan er enkel Tree en Back edges, omdat bogen in beide richtingen bewandeld mogen worden (elke cross of down edge in de ene richting is onmiddellijk een back of tree edge in de andere). Bij *BFT* (ongericht) zijn er enkel Tree en Cross edges, doordat we op breedte zoeken.

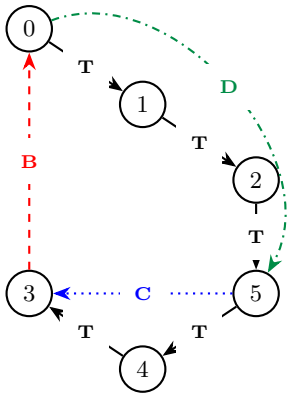



Figure 14: Gerichte boogclassificatie door DFT

## 3.5 BFT, Het Algoritme

**BFT Signatuur (5+1 argumenten)**

```
(bft graph
  root-discovered    ; root -> bool
  node-discovered    ; node -> bool
  edge-discovered    ; from to -> bool
  edge-bumped        ; from to -> bool
  . roots)           ; optioneel: lijst van roots
```

- Geen `node-processed` of `edge-processed`: BFT keert nooit terug naar een knoop.
- `edge-discovered` retourneert `#f` ⇒ `to` wordt *niet* geënqueued (krachtig maar gevaarlijk: geen echte BFT meer!).
- Queue-gebaseerd ⇒ iteratief van nature.

## 3.6 BFT voor Ongerichte Grafen, Drie Eigenschappen

### 3.6.1 Eigenschap 1: Spanning Forest

BFT genereert ook een opspannend bos, maar de bomen zijn **breed en ondiep** (tegenovergestelde van DFT).

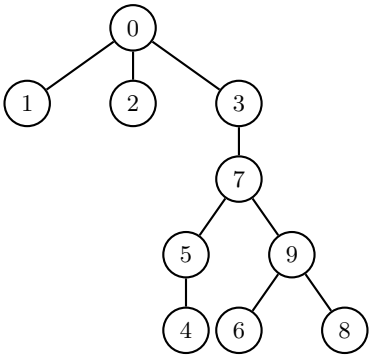



Figure 15: BFT-boom: breed en ondiep

### 3.6.2 Eigenschap 2: Edge Labeling ($E = T \cup C$)

**Boogclassificatie, Ongerichte Graaf + BFT**

Slechts twee categorieën:

- **Tree edges ($T$):** bogen naar een onontdekte knoop.
- **Cross edges ($C$):** bogen naar een reeds bezochte knoop. Geen back/down edges mogelijk: BFT voegt buren "gretig" toe $\Rightarrow$ kruisverbindingen zijn altijd horizontaal.

### 3.6.3 Eigenschap 3: Afstand als Knoopnummering

**BFT en Afstandsmaat**

Bij BFT vanuit root $r$: als $d(u)$ de door BFT toegekende afstand is (elke tree edge verhoogt de teller met 1), dan is $d(u)$ het **kortste pad** (in termen van aantal bogen) van $r$ naar $u$.

Dit maakt BFT bijzonder geschikt voor *shortest-path*-problemen in ongewogen grafen.

## 3.7 BFT voor Gerichte Grafen

Edge labeling bij gerichte BFT: $E = T \cup B \cup C$. *Geen down edges.*

- **Back edge:** boog naar een voorouder in dezelfde boom.
- **Cross edge:** boog naar een knoop in een andere subboom of een reeds verwerkte knoop.

## 3.8 Performantie

| | **Adjacency List** | **Adjacency Matrix** |
|---|---|---|
| **DFT** | $O(\|V\| + \|E\|)$ | $\Theta(\|V\|^2)$ |
| **BFT** | $O(\|V\| + \|E\|)$ | $\Theta(\|V\|^2)$ |

Table 5: Performantie van DFT en BFT

**Keuze DFT vs. BFT**

- DFT: geschikt voor cyclusdetectie, topologisch sorteren, SCC's, bruggen, articulatiepunten.
- BFT: geschikt voor kortste-pad (ongewogen), afstandsberekening, bipartiteness.
- Beide: connected components, opspannende bossen.

# 4 Hoofdstuk 11: Ongerichte Grafalgoritmen

Drie grote thema's: (1) basisalgoritmen, (2) connectiviteitsvarianten, (3) minimale opspannende bossen.

## 4.1 Basisalgoritmen

### 4.1.1 Cyclusdetectie

**Cycli in ongerichte grafen (DFT)**

Een ongerichte graaf bevat een cyclus $\Leftrightarrow$ DFT ontdekt een *back edge*.

**Subtiliteit:** in een ongerichte graaf wordt de boog $\{u, v\}$ tweemaal bezocht. Wanneer we van $v$ terugkijken naar $u$ (de ouder van $v$), is dat géén back edge. We moeten daarom de DFT-boom bijhouden om de ouder te herkennen.

```
(define (cyclic? g)
  (define tree (make-vector (order g) '()))
  (define cyclic #f)
  (dft g root-nop node-nop node-nop
    (lambda (from to) (vector-set! tree to from))  ; edge-disc.
    edge-nop
    (lambda (from to)                               ; edge-bumped
      (if (not (eq? (vector-ref tree from) to))
        (set! cyclic #t))))
  cyclic)
```

### 4.1.2 Paden Vinden

**Bestaat er een pad? (DFT)**

Start een DFT vanuit `from` (optionele roots = `(list from)`). Stop zodra `to` ontdekt wordt via `node-discovered`. Geeft géén garantie over padlengte.

**Kortste pad (BFT)**

Start een BFT vanuit `from`. BFT garandeert dat het **kortste pad** (in termen van het aantal bogen) gevonden wordt.
Elk ontdekt knooppunt krijgt een pad via een `paths`-vector: `(vector-set! paths to (cons to (vector-ref paths from)))`.

### 4.1.3 Afstanden (BFT)

Triviaal: elke `edge-discovered` zet `distances[to] := distances[from] + 1`.

## 4.2 Connectiviteit

### 4.2.1 Samenhangende Componenten

**Connected Components (DFT of BFT)**

Elke `root-discovered` ontdekt een nieuwe component. `node-discovered` kent het component-nummer toe.

**Performantie:** $O(|V| + |E|)$ (adj. lijst) / $\Theta(|V|^2)$ (adj. matrix). Één van de weinige problemen die even eenvoudig met DFT en BFT op te lossen zijn.

### 4.2.2 Edge-Connectivity (Bruggen)

**Brug (Bridge)**

Een boog $e \in E$ is een **brug** als het verwijderen van $e$ het aantal samenhangende componenten verhoogt. Een samenhangende graaf zonder bruggen is **edge-connected**.

**Brugendetectie-Algoritme (Hopcroft–Tarjan, 1973)**

Kernidee: boog $\{m, n\}$ (tree edge, $n$ kind van $m$) is een brug als en slechts als *géén* back edge vanuit $n$ of een afstammeling van $n$ naar $m$ of een voorouder van $m$ leidt.

**Datastructuur:** `highest-back-edge[v]` = kleinste pre-order nummer bereikbaar vanuit $v$ (via afstammelingen + back edges).

- `node-discovered`: zet `preorder[v] := highest-back-edge[v] := time++`.
- `edge-processed`: propageer: `highest-back-edge[from] := min(hbe[from], hbe[to])`. *Test:* als `hbe[to] = preorder[to]` $\Rightarrow$ {`from, to`} is een brug.
- `edge-bumped`: als echte back edge (niet de ouder): `hbe[from] := min(hbe[from], preorder[to])`.

**Examen: One-edge connectivity vs Biconnectivity**

**Vraag:** Wat is one-edge connectivity (bruggen) en wat is het belang/toepassing ervan ten opzichte van biconnectiviteit?
**Antwoord: One-edge connectivity** betekent dat een graaf samenhangend is en géén bruggen (kwetsbare bogen) heeft. **Biconnectiviteit** betekent dat een graaf geen articulatiepunten (kwetsbare knopen) heeft. *Toepassingen:* One-edge connectivity is cruciaal bij netwerk-robuustheid voor *verbindingen* (bv. kabelbreuk mag het netwerk niet splitsen), terwijl biconnectiviteit belangrijk is voor robuustheid van *knooppunten* (bv. als een server of router crasht, moet de rest kunnen communiceren).

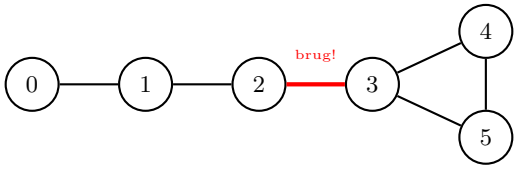



Figure 16: Boog $\{2, 3\}$ is een brug

### 4.2.3 Biconnectivity (Articulatiepunten)

**Articulatiepunt**

Een knoop waarvan de verwijdering (samen met alle bijbehorende bogen) de graaf in meerdere componenten splitst. Een graaf zonder articulatiepunten is **biconnected**.

**Articulatiepunten (Hopcroft–Tarjan)**

Vrijwel identiek aan het bruggenalgoritme, met twee gevallen:

1. **Interne knoop $v$:** $v$ is een articulatiepunt als $\exists$ kind $w$ waarvoor `highest-back-edge[w]` $\geq$ `preorder[v]`.
2. **Root van de DFT-boom:** de root is een articulatiepunt als en slechts als de root $\geq 2$ kinderen heeft in de DFT-boom.

### 4.2.4 Bipartiteness

**Bipartiet**

Een graaf $G = (V, E)$ is **bipartiet** als $V = V_1 \cup V_2$ met $V_1 \cap V_2 = \emptyset$ zodanig dat alle bogen tussen $V_1$ en $V_2$ lopen. Equivalent: $\chi(G) = 2$ (kleurbaar met 2 kleuren).

**Bipartiteness-Test (DFT)**

- Gebruik een `colors`-vector met booleans.
- `root-discovered`: root krijgt kleur `#t`.
- `edge-discovered`: kind krijgt kleur $\neg$ ouder.
- `edge-bumped`: als `colors[from] = colors[to]` $\Rightarrow$ niet bipartiet.

**Let op:** gebruik `and` om informatie van meerdere bumped edges te combineren.

## 4.3 Minimale Opspannende Bossen (MST)

**Minimum Spanning Forest**

Een opspannend bos van een gewogen graaf waarvoor de som van de gewichten van alle bogen minimaal is. MST's zijn uniek als alle gewichten verschillend zijn.

### 4.3.1 Kruskal's Algoritme (1956)

**Kruskal**

1. Verzamel alle bogen en **sorteer** op gewicht (quicksort).
2. Overloop bogen van licht naar zwaar; voeg een boog toe aan het bos als het géén cyclus veroorzaakt.
3. Cyclusdetectie via **disjoint sets** (union-find): als `find(from)` $\neq$ `find(to)` $\Rightarrow$ veilig.

**Performantie:** $O(|E| \cdot \log |V|)$ (sorting domineert). Met radix sort: $O(|E| \cdot \alpha(|V|))$.

**Examen: Correctheidsbewijs Kruskal**

**Vraag:** Wat is MST? Leg Kruskal uit en bewijs correctheid.
**Antwoord:** Een MST is een opspannende boom met minimaal totaal booggewicht. **Bewijs (Cut Property):** Kruskal kiest iteratief de globaal lichtste boog $e = (u, v)$ die geen cyclus vormt in het huidige bos $F$. Omdat $e$ geen cyclus vormt, behoren $u$ en $v$ tot verschillende bomen in $F$. Beschouw de *cut* (snede) die de knopen van $u$'s boom scheidt van de rest van de graaf. $e$ kruist deze snede. Omdat we bogen stijgend op gewicht verwerken, is $e$ gegarandeerd de *lichtste* boog die deze snede kruist. Volgens de fundamentele wiskundige *Cut Property* behoort de lichtste boog over een willekeurige snede altijd tot de MST. Dus is de toevoeging van $e$ veilig en iteratief correct.

### 4.3.2 Prim–Jarník Algoritme (1930/1957)

**Prim–Jarník**

Greedy algoritme met een **priority queue**.

1. Start met één knoop; voeg alle buren toe aan de PQ met hun gewicht als prioriteit.
2. Serve de knoop met laagste prioriteit; voeg toe aan het bos.
3. Onderzoek buren: als een buur nog in de PQ zit met een hoger gewicht, `reschedule!` met het lagere gewicht.

Vereist een uitgebreide PQ-ADT met `reschedule!` en `priority-of` (via numerieke identiteiten in een node-indexed vector).

**Performantie:** $O(|E| \cdot \log |V|)$. Met Fibonacci heaps: $O(|E| + |V| \cdot \log |V|)$.

### 4.3.3 Borůvka's Algoritme (1926)

**Borůvka**

Laat meerdere bomen **parallel** groeien:

1. Voor elke boom in het huidige bos: selecteer de lichtste boog die het bos verbindt met een andere boom.
2. Voeg al deze geselecteerde bogen tegelijk toe.
3. Herhaal tot er één boom overblijft.

Het aantal bomen halveert per ronde $\Rightarrow O(\log |V|)$ rondes.

**Performantie:** $O(|E| \cdot \log |V| \cdot \alpha(|V|))$. Zonder disjoint sets: $O(|E| \cdot \log |V|)$.

**Let op:** werkt alleen correct als alle gewichten verschillend zijn, of als een consistente tie-breaking gebruikt wordt.

### 4.3.4 Samenvatting MST-algoritmen

| Algoritme | Performantie | Strategie |
|---|---|---|
| Kruskal | $O(\|E\| \log \|V\|)$ | sorteer bogen, union-find |
| Prim–Jarník | $O(\|E\| \log \|V\|)$ | PQ, greedy |
| Borůvka | $O(\|E\| \log \|V\| \cdot \alpha(\|V\|))$ | parallel groei |

Table 6: Vergelijking MST-algoritmen

**Greedy Algoritmen**

Zowel Kruskal als Prim–Jarník zijn **greedy**: ze nemen telkens de lokaal optimale keuze (lichtste veilige boog). Het kan wiskundig bewezen worden dat deze greedy strategie een *globaal* optimum oplevert voor het MST-probleem.

# 5 Hoofdstuk 12: Gerichte Grafalgoritmen

Zes delen: (1) DAG's & topologisch sorteren, (2) sterke samenhang, (3) padexistentie (Warshall), (4–5) SSSP-algoritmen, (6) APSP (Floyd-Warshall).

## 5.1 DAG's en Topologisch Sorteren

**DAG (Directed Acyclic Graph)**

Een gerichte graaf zonder cycli. DAG's modelleren partiële ordeningen (bv. prerequisite-structuren, compilatieafhankelijkheden).

**Topologisch Sorteren**

Het construeren van een **totale ordening** op de knopen die de partiële ordening (gegeven door de bogen) respecteert. Een DAG heeft minstens één topologische sortering; deze is in het algemeen niet uniek.

### 5.1.1 DFT-gebaseerd (Tarjan, 1976)

**DFT Topologisch Sorteren**

Laat `node-processed` knopen toevoegen aan de kop van een lijst ⇒ de lijst is in **omgekeerde post-order** = topologische ordening.

```
(define (dft-topological-sort g)
  (define series '())
  (dft g root-nop node-nop
    (lambda (node)
      (set! series (cons node series)))
    edge-nop edge-nop edge-nop)
  series)
```

**Performantie:** $O(|V| + |E|)$.

### 5.1.2 BFT-achtig (Kahn, 1962), Source Reduction

**BFT Topologisch Sorteren (Kahn)**

1. **Fase 1:** bereken ingraad van elke knoop met BFT. Verzamel alle *sources* (ingraad = 0).
2. **Fase 2:** start een BFT vanuit de sources. `edge-discovered`: verminder ingraad van `to`; enqueue `to` alleen als ingraad = 0 ⇒ retourneer `(= (vector-ref in-degrees to) 0)`.

**Let op:** de tweede fase is *geen* echte BFT (door het filteren in `edge-discovered`).
**Performantie:** $O(|V| + |E|)$.

## 5.2 Sterk Samenhangende Componenten (SCC)

**Sterk Samenhangend**

Een gerichte graaf is **sterk samenhangend** als er voor elk paar knopen $(u, v)$ een gericht pad van $u$ naar $v$ én van $v$ naar $u$ bestaat.
**Strongly Connected Components (SCC's):** maximale sterk samenhangende deelgrafen.
**Kernel-graaf:** vervang elke SCC door één knoop ⇒ altijd een DAG.

### 5.2.1 Kosaraju's Algoritme (1978)

**Kosaraju**

1. **DFT 1** op $G$: verzamel knopen in *reverse post-order* (`node-processed` voegt toe aan de kop van een lijst).
2. Bereken $G^T$ (transpose: keer alle bogen om).
3. **DFT 2** op $G^T$, met als roots de lijst uit stap 1. Elke nieuwe root = nieuwe SCC.

**Correctheid:** $u$ en $v$ zijn wederzijds bereikbaar $\Leftrightarrow$ $u$ en $v$ zitten in dezelfde DFT-boom van $G^T$.
**Performantie:** $O(|V| + |E|)$ (twee DFT's + transpositie).

### 5.2.2 Tarjan's Algoritme (1972)

**Tarjan's SCC**

Één enkele DFT volstaat. Gebruikt dezelfde `highest-back-edge`-techniek als het bruggen/articulatiepuntenalgoritme, plus een **stack**.

1. `node-discovered`: push knoop op stack; zet `preorder[v] := hbe[v] := time++`.
2. `edge-processed`: `hbe[from] := min(hbe[from], hbe[to])` (alleen als `to` nog niet "included").
3. `edge-bumped`: idem, met `preorder[to]` i.p.v. `hbe[to]`.
4. `node-processed`: als `hbe[v] = preorder[v]` $\Rightarrow$ pop knopen van de stack tot en met $v$; dit zijn de leden van een nieuwe SCC.

**Subtiliteit:** de `included`-vector voorkomt dat informatie van reeds afgesloten SCC's terugvloeit via cross edges.
**Performantie:** $O(|V| + |E|)$ (één DFT).

**Examen: Tarjan's Algoritme Uitleg (Tarjan SCC)**

**Vraag:** Leg Tarjan's algoritme uit + voorbeeld.
**Antwoord:** Tarjan gebruikt één DFT met een expliciete `stack` om componentleden bij te houden. Elke knoop krijgt een `preorder`-nummer en een `highest-back-edge` (hbe) referentie. Bij het ontdekken van knopen worden ze op de stack gepusht en hbe is initieel gelijk aan preorder. Wanneer we via cross/back/down edges een kleinere hbe vinden, propageren we dit naar de ouders. Zodra we een knoop $v$ volledig verwerkt hebben (`node-processed`) en `hbe[v] == preorder[v]`, betekent dit dat er geen pad is vanuit $v$ of zijn afstammelingen naar een component 'hoger' in de boom. Dan worden alle knopen tot en met $v$ van de stack gehaald; zij vormen tezamen een nieuwe SCC.

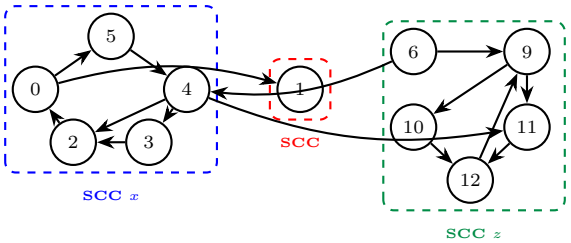



Figure 17: Gerichte graaf met SCC's gegroepeerd

## 5.3 Padexistentie: Warshall's Algoritme

**Transitieve Sluiting ($G^+$)**

De graaf $G^+ = (V, E^+)$ met $(u, v) \in E^+ \Leftrightarrow$ er bestaat een pad van $u$ naar $v$ in $G$.
Na berekening van $G^+$ (als adjacentiematrix): padexistentie in $O(1)$.

### Warshall's Algoritme (1962)

```
(define (traclo-warshall g)
  (for-each-node g (lambda (via)
    (for-each-node g (lambda (from)
      (for-each-node g (lambda (to)
        (if (and (adjacent? g from via)
                 (adjacent? g via to))
          (add-edge! g from to)))))))))
```

**Invariant:** na iteratie `via` bevat $g$ een boog $(u, v) \Leftrightarrow$ er een pad $u \to v$ bestaat dat alleen tussenliggende knopen $\leq$ `via` gebruikt.
**Performantie:** $\Theta(|V|^3)$.

### Naïeve methoden

- Herhaalde matrixvermenigvuldiging: $O(|V|^4)$.
- Logaritmische machtsverheffen: $O(|V|^3 \log |V|)$.
- Warshall is optimaal: $O(|V|^3)$.

### Examen: Verband met matrixvermenigvuldiging (Warshall)

**Vraag:** Leg het algoritme van Warshall uit en het verband met matrixvermenigvuldiging.
**Antwoord:** De transitieve sluiting $G^+$ komt overeen met de limiet van $(I \vee A)^k$. Naïeve matrixvermenigvuldiging $(A \times A)$ over de booleaanse algebra ($\vee$ en $\wedge$) berekent paden van lengte 2 in $O(|V|^3)$, en paden van lengte $|V|$ in $O(|V|^4)$ of $O(|V|^3 \log |V|)$. Warshall berekent dit direct in $\Theta(|V|^3)$ door knopen één voor één als *tussenknoop* (`via`) toe te staan, in plaats van paden per *lengte* te vergroten.

## 5.4 SSSP: Single Source Shortest Path

### Kortste Pad in Gewogen Grafen

De **lengte** van een pad $p = v_0, v_1, \ldots, v_k$ is $w_G(p) = \sum_{i=1}^{k} w_G(v_{i-1}, v_i)$.

De **afstand** $\delta(u, v) = \begin{cases} w_G(p) & \text{als } p \text{ een kortste pad is} \\ +\infty & \text{als geen (kortste) pad bestaat} \end{cases}$

### Twee Eigenschappen van Kortste Paden

1. **Geen cycli:** als er een kortste pad bestaat, bestaat er ook een zonder cycli. Negatieve cycli $\Rightarrow$ geen kortste pad. Positieve cycli $\Rightarrow$ weglaten geeft korter pad. Nulcycli $\Rightarrow$ gewoon weglaten. Maximaal $|V| - 1$ bogen.
2. **Optimale substructuur:** elk deelpad van een kortste pad is zelf een kortste pad.

### Relaxatie

Centraal mechanisme voor alle SSSP-algoritmen:

```
(define (relax! distances how-to-reach u v weight)
  (when (< (+ (vector-ref distances u) weight)
           (vector-ref distances v))
    (vector-set! distances v
      (+ (vector-ref distances u) weight))
    (vector-set! how-to-reach v u)))
```

### 5.4.1 Bellman-Ford

**Bellman-Ford**

- Werkt voor **alle grafen zonder negatieve cycli**.
- Relaxeer *alle* bogen $|V| - 1$ keer (brute-force).
- **Fase 2:** relaxeer alle bogen nog eens; als er iets verandert $\Rightarrow$ negatieve cyclus gedetecteerd.

**Performantie:** $O(|E| \cdot |V|)$.

### 5.4.2 Dijkstra

**Dijkstra (1959)**

**Beperking:** alleen voor grafen met **niet-negatieve gewichten**.
Greedy met een priority queue:

1. Initialiseer PQ met alle knopen; `distances[source] := 0`, rest $+\infty$.
2. Serve de knoop met kleinste afstand.
3. Relaxeer alle uitgaande bogen; `reschedule!` buren met verbeterde afstand.

**Performantie:** $O(|E| \cdot \log |V|)$.

**Examen: Correctheidsbewijs Dijkstra**

**Vraag:** Dijkstra algoritme uitleggen + bewijs.
**Antwoord:** Dijkstra zoekt greedy naar het kortste pad vanuit een bronknoop en gebruikt een PQ. Het steunt cruciaal op de aanname dat *gewichten niet-negatief* zijn. **Bewijs (inductie):** Stel dat op het moment van serveren van knoop $u$, de toegekende afstand $d(u)$ de echte kortste afstand $\delta(s, u)$ is. Als we $u$ serveren, relaxeren we zijn buren. Kan er toch een korter pad naar $u$ bestaan via een nog ongeserveerde knoop $v$? Omdat we knopen in stijgende volgorde van afstand serveren, geldt op dit moment dat $d(v) \geq d(u)$. Een eventueel alternatief pad naar $u$ via $v$ zou een totale lengte $d(v) + w(v \dots u)$ hebben. Omdat alle bogen niet-negatief zijn, is $w(v \dots u) \geq 0$, waardoor dit alternatieve pad minstens $d(v) \geq d(u)$ lang is. Dus $d(u)$ kan nooit meer verbeteren nadat $u$ uit de PQ is gehaald.

**Dijkstra faalt bij negatieve gewichten**

Voorbeeld: graaf $0 \xrightarrow{2} 1 \xrightarrow{1} 2$ en $0 \xrightarrow{3} 3 \xrightarrow{-3} 2$. Dijkstra kiest greedy via $0 \to 1 \to 2$ (afstand 3), maar het pad $0 \to 3 \to 2$ (afstand 1) is korter. Greedy werkt niet met negatieve gewichten!

### 5.4.3 Lawler (1976), Alleen voor DAG's

**Lawler**

1. Sorteer topologisch.
2. Overloop knopen in topologische volgorde; relaxeer alle uitgaande bogen.

Één pass volstaat (geen backtracking nodig door afwezigheid van cycli).
**Performantie:** $O(|V| + |E|)$.
**Toepassing:** PERT-diagrammen (kritisch pad = langste pad; gebruik $-\infty$ en $>$ i.p.v. $+\infty$ en $<$).

| Algoritme | Performantie | Beperking |
|---|---|---|
| Bellman-Ford | $O(\|E\| \cdot \|V\|)$ | Geen negatieve cycli |
| Dijkstra | $O(\|E\| \cdot \log \|V\|)$ | Geen negatieve gewichten |
| Lawler | $O(\|V\| + \|E\|)$ | Alleen DAG's |

Table 7: Vergelijking SSSP-algoritmen (adjacentielijst)

### 5.4.4 Samenvatting SSSP

## 5.5 APSP: Floyd-Warshall

**Floyd-Warshall (1962)**

Generaliseert Warshall naar **gewogen** grafen: berekent kortste afstand tussen *elk* paar knopen.

```
(define (floyd-warshall g)
  (define D ...)  ; |V| x |V| matrix, init met weight(i,j)
  (for-each-node g (lambda (via)
    (for-each-node g (lambda (from)
      (for-each-node g (lambda (to)
        (ij! D from to
          (min (ij? D from to)
               (+ (ij? D from via)
                  (ij? D via to))))))))))
```

**Invariant:** na iteratie `via`: $D$[`from`][`to`] bevat het gewicht van het kortste pad dat alleen tussenliggende knopen $\leq$ `via` gebruikt.
**Performantie:** $\Theta(|V|^3)$. Werkt niet bij negatieve cycli (tenzij detectie via diagonaal: $D[v][v] < 0$).

**Examen: Verschil Warshall en Floyd-Warshall**

**Vraag:** Gelijkenissen en verschillen tussen Warshall en Floyd-Warshall.
**Antwoord:** Beide algoritmen gebruiken exact dezelfde $\Theta(|V|^3)$ dynamisch-programmeren structuur met drie geneste lussen (`via`, `from`, `to`) waarbij knopen één voor één als tussenknoop worden toegelaten. Het grote verschil: **Warshall** werkt op ongewogen grafen (booleaanse matrix) met logische operatoren (`or` en `and`) om *padexistentie* te berekenen. **Floyd-Warshall** werkt op gewogen grafen met numerieke operatoren (`min` en `+`) om de *kortste afstand* (APSP) te berekenen.

# 6 Hoofdstuk 13: Dynamisch Programmeren

**Wat is Dynamisch Programmeren?**

Een techniek om optimalisatieproblemen op te lossen. **Voorwaarden:**

- **Optimale substructuur:** De optimale oplossing van het probleem bevat de optimale oplossingen van de deelproblemen (cut-and-paste argument).
- **Overlappende deelproblemen:** In tegenstelling tot Divide & Conquer, hergebruiken we dezelfde deelproblemen meerdere keren. DP berekent ze slechts één keer en onthoudt het resultaat.

**Strategie (4 stappen):**

1. Karakteriseer de structuur van een optimale oplossing.
2. Definieer de optimale waarde recursief.
3. Bereken de optimale waarde (typisch bottom-up, of top-down met memoizatie).
4. Extraheer de optimale oplossing op basis van extra bijgehouden info.

**Onafhankelijkheid van Deelproblemen & Cut-and-Paste**

Optimale substructuur geldt enkel als de deelproblemen **onafhankelijk** zijn. Een *cut-and-paste* bewijs toont dit aan: als een deeloplossing niet optimaal was, konden we deze letterlijk "uitknippen" en vervangen door de optimale, wat de totale oplossing nog beter zou maken. **Tegenvoorbeeld:** Het vinden van het *kortste simpele pad* heeft optimale substructuur. Het vinden van het *langste simpele pad* heeft dit **niet**, omdat langere paden bezochte knopen "verbruiken", waardoor de deelproblemen overlappen en elkaar beperken (ze zijn niet onafhankelijk).

**Examen: Bottom-up vs Top-down**

**Vraag:** Wat is het verschil tussen bottom-up en top-down + memoizatie?
**Antwoord: Top-down met memoizatie** evalueert de recursie op een natuurlijke manier, maar slaat elke berekende waarde op in een tabel. Bij overlappende deelproblemen wordt de opgeslagen waarde meteen geretourneerd (vermijdt exponentieel dubbel werk). **Bottom-up** berekent en tabelleert de oplossingen expliciet van klein naar groot op basis van de afhankelijkheidsgraaf.
**Vuistregel:** Als *alle* mogelijke deelproblemen bezocht moeten worden, is Bottom-Up typisch marginaal sneller door de afwezigheid van recursie-overhead. Als slechts een relatief kleine fractie van de deelproblemen daadwerkelijk nodig is om de uiteindelijke top te bereiken, is Top-Down efficiënter omdat deze onnodige deelproblemen overslaat.

## 6.1 Rod Cutting

**Rod Cutting Probleem**

Optimaal snijden van een metalen staaf van lengte $n$ met prijzen $p_i$ voor stukken van lengte $i$. Recursieve vergelijking:

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

waarbij $r_0 = 0$.
Zonder DP (pure recursie): $\Theta(2^n)$.
Met DP (Bottom-up of Top-down memoizatie): $\Theta(n^2)$.

## 6.2 Matrix Chain Multiplication (MCM)

**Matrix Chain Multiplication**

Gegeven een keten van matrices $A_1 \cdot A_2 \cdots A_n$, zoek de optimale plaatsing van haakjes om het aantal scalaire vermenigvuldigingen te minimaliseren. (Een matrix $p \times q$ maal $q \times r$ kost $pqr$ operaties).
**Recursieve vergelijking:** Definieer $m[i,j]$ als de minimale kost om $A_i \dots A_j$ te vermenigvuldigen.

$$m[i,j] = \begin{cases} 0 & \text{als } i = j \\ \min_{i \le k < j} \left( m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \right) & \text{als } i < j \end{cases}$$

**Top-Down met Memoizatie (LOOKUP-CHAIN):** Initialiseer een tabel $m$ met $\infty$. Voor elke sub-aanroep: check of $m[i,j] < \infty$ is. Zo ja, retourneer direct (dit was al berekend). Zo nee, bereken recursief met de formule, sla op in de tabel, en retourneer. Dit vermijdt de exponentiële overhead van pure recursie!
Performantie met DP: $\Theta(n^3)$.

## 6.3 Langste Gemeenschappelijke Deelrij (LCS)

**Longest Common Subsequence (LCS)**

Gegeven twee strings $X = x_1 \dots x_m$ en $Y = y_1 \dots y_n$, zoek de langste gemeenschappelijke deelrij (niet noodzakelijk aaneengesloten).
**Optimale Substructuur:** Laat $Z = z_1 \dots z_k$ een LCS zijn van $X$ en $Y$.

1. Als $x_m = y_n$, dan is $z_k = x_m = y_n$ en is $Z_{k-1}$ een LCS van $X_{m-1}$ en $Y_{n-1}$.
2. Als $x_m \neq y_n$ en $z_k \neq x_m$, dan is $Z$ een LCS van $X_{m-1}$ en $Y$.
3. Als $x_m \neq y_n$ en $z_k \neq y_n$, dan is $Z$ een LCS van $X$ en $Y_{n-1}$.

**Recursieve vergelijking:** Lengte $c[i,j]$ van de LCS van $X_i$ en $Y_j$:

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

**Reconstructie van de LCS:** We kunnen de werkelijke string reconstrueren door via een hulplabel-matrix ($b[i,j]$ met pijlen $\nwarrow, \uparrow, \leftarrow$) terug te wandelen vanaf $c[m,n]$, of door simpelweg de waarden in de $c$-matrix lokaal opnieuw te evalueren om te zien via welke tak de sprong werd gemaakt.
Performantie: $\Theta(m \cdot n)$.

**Examen: LCS Substructuur**

**Vraag:** Leg LCS substructuur uit en pas toe.
**Antwoord:** De optimale substructuur van LCS (zie stelling hierboven) stelt dat als de laatste karakters van twee prefixes overeenkomen ($x_i = y_j$), ze gegarandeerd deel uitmaken van de LCS, waardoor we het probleem reduceren tot de LCS van beide kortere prefixes (diagonale sprong). Als ze *niet* overeenkomen, moet de LCS zich bevinden in het weglaten van ofwel het laatste karakter van string $X$, ofwel van string $Y$. We nemen dan simpelweg het maximum van die twee opties. In DP tabelleren we dit in een 2D matrix ($m \times n$).

## 6.4 Optimale Binaire Zoekbomen (OBST)

**Optimal Binary Search Tree**

Gegeven gesorteerde keys $k_1 < k_2 \cdots < k_n$ met probabiliteiten $p_i$ om gezocht te worden, en dummy keys $d_0 \dots d_n$ voor niet-bestaande waarden met probabiliteiten $q_i$. Bouw een BST met minimale verwachte zoek-kost.
**Verwachte zoek-kost:**

$$1 + \sum_{i=1}^{n} \text{depth}(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}(d_i) \cdot q_i$$

**Recursieve vergelijking:** Zij $e[i, j]$ de verwachte zoek-kost voor een optimale subboom over keys $k_i \dots k_j$ (en hun dummy's). Zij $w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$ de totale probabiliteit. Als we $k_r$ als root kiezen voor het interval $[i, j]$, wordt $k_i \dots k_{r-1}$ de linkersubboom en $k_{r+1} \dots k_j$ de rechtersubboom. Het feit dat alle knopen in de subbomen 1 niveau dieper liggen t.o.v. root $k_r$, voegt precies $w(i, j)$ extra toe aan de totale kost.

$$e[i, j] = \begin{cases} q_{i-1} & \text{als } j = i - 1 \\ \min_{i \le r \le j} \left( e[i, r-1] + e[r+1, j] \right) + w(i, j) & \text{als } i \le j \end{cases}$$

**Examen: OBST Substructuur**

**Vraag:** Heeft een optimale binaire zoekboom optimale substructuur? Geef recursievergelijking.
**Antwoord:** Ja. Als een bepaalde boom $T$ optimaal is voor de set keys en we beschouwen een willekeurige subboom van $T$ die de keys $k_i \dots k_j$ bevat, dan moet die subboom zélf de optimale structuur hebben voor dat specifieke interval (cut-and-paste bewijs: als er een betere subboom was, konden we die in $T$ plaatsen voor een nog lagere totale kost). De recursieve vergelijking is: $e[i, j] = \min_{i \le r \le j} \left( e[i, r-1] + e[r+1, j] \right) + w(i, j)$, met basisgeval $e[i, i-1] = q_{i-1}$.

# 7 Hoofdstuk 19: Conflict-Driven Clause Learning (CDCL)

NP-harde problemen zijn alomtegenwoordig (k-kleurbaarheid, TSP, bin packing, ...). SAT is het prototypische NP-complete probleem en vormt de basis voor moderne solvers.

## 7.1 Propositielogica, Basis

**CNF (Conjunctive Normal Form)**

- **Literal:** een atoom $x$ of zijn negatie $\overline{x}$.
- **Clause:** disjunctie van literalen: $l_1 \vee l_2 \vee \cdots \vee l_m$.
- **Formule (CNF):** conjunctie van clauses: $c_1 \wedge c_2 \wedge \cdots \wedge c_k$.

Een clause wordt ook behandeld als een *verzameling* van literalen; een formule als een verzameling van clauses.

**Interpretatie**

Een **interpretatie** $I$ is een consistente verzameling van literalen (bevat nooit zowel $x$ als $\overline{x}$). Een literal/clause/formule kan in $I$ de waarde `t` (true), `f` (false) of `u` (unknown) hebben:

- $\ell^I = \texttt{t}$ als $\ell \in I$; `f` als $\overline{\ell} \in I$; `u` anders.
- $c^I = \texttt{t}$ als $\exists \ell \in c : \ell^I = \texttt{t}$; `f` als $\forall \ell \in c : \ell^I = \texttt{f}$; `u` anders.
- $F^I = \texttt{t}$ als $\forall c \in F : c^I = \texttt{t}$; `f` als $\exists c \in F : c^I = \texttt{f}$; `u` anders.

**SAT-probleem**

**Input:** CNF-formule $F$. **Output:** "YES" als er een interpretatie $I$ bestaat met $F^I = \texttt{t}$; anders "NO".
SAT is **NP-compleet** (Cook–Levin).

## 7.2 DPLL (Davis–Putnam–Logemann–Loveland)

**DPLL, Kernalgoritme**

Depth-first search door een (impliciete, exponentieel grote) zoekboom van interpretaties, versterkt met twee pruningstechnieken:

1. **Unit Propagation:** als een clause $c$ *unit* is (= alle literalen false behalve één), maak de laatste literal true. Herhaal tot fixpoint.
2. **Pure Literal Elimination** (optioneel): als een literal $\ell$ nooit in zijn negatie $\overline{\ell}$ voorkomt in een onvervulde clause, maak $\ell$ true.

```
dpll(T, I):
  I := unit-prop(T, I)
  I := pure-lits(T, I)           (* optioneel *)
  if some c in T is conflicting: return false
  if all variables assigned:     return true
  x := select-unassigned-variable(I)
  return dpll(T, I + {x}) or dpll(T, I + {~x})
```

**Unit & Conflicting Clause**

- Clause $c$ is **unit** in $I$ als alle literalen behalve één false zijn.
- Clause $c$ is **conflicting** in $I$ als *alle* literalen false zijn.

**Examen: Voorbeeld Unit Propagation**

**Vraag:** Leg Unit Propagation uit a.d.h.v. voorbeeld (bijv. $\neg a$), en leg DPLL uit.
**Antwoord:** DPLL is een backtracking search algoritme dat iteratief beslissingen (gokjes) maakt en die propageert via deterministische regels. De belangrijkste regel is Unit Propagation. Stel we hebben een clause $C = (a \vee \neg b \vee c)$ en we wijzen $\neg a$ (oftewel $a = \texttt{f}$) en $b = \texttt{t}$ toe. De clause evalueert dan tot $(\texttt{f} \vee \texttt{f} \vee c)$. Omdat $c$ de enige overblijvende ongedefinieerde literal is die de clause nog kan vervullen, is de clause **unit**. Unit Propagation dwingt ons nu om deductief te besluiten dat $c = \texttt{t}$ (literal $c$ is waar) om een conflict te vermijden. Dit proces wordt doorgezet (cascadering) tot alle unit clauses zijn weggewerkt.

## 7.3 CDCL, Uitbreidingen van DPLL

CDCL = DPLL + drie cruciale verbeteringen:

1. **Two-Watched Literal Scheme:** efficiënte datastructuur voor unit propagation.
2. **Clause Learning:** bij een conflict wordt de oorzaak geanalyseerd en een nieuwe clause geleerd.
3. **Non-chronological Backtracking (Backjumping):** spring terug naar het relevante decisieniveau.

### 7.3.1 Two-Watched Literal Scheme

**Two-Watched Literals**

Voor elke clause: houd twee *non-false* literalen bij ("watches"). Zolang beide watches non-false zijn, kan de clause niet unit of conflicting zijn.

Wanneer een watch $\ell$ false wordt:

- Zoek een andere non-false literal $\ell'$ in de clause $\Rightarrow$ vervang de watch.
- Geen andere non-false literal gevonden:
  - Andere watch is $\texttt{t}$ $\Rightarrow$ clause is satisfied, niets te doen.
  - Andere watch is $\texttt{u}$ $\Rightarrow$ clause is **unit**, propageer!
  - Andere watch is $\texttt{f}$ $\Rightarrow$ clause is **conflicting**, backtrack!

**Cruciaal voordeel:** géén onderhoud nodig bij backtracking! Na backtracking wijzen watches opnieuw naar non-false literalen.

**Examen: Two-Watched Literals (Voor- en Nadelen)**

**Vraag:** Leg Two Watched Literals uit. Wat zijn de voor- en nadelen t.o.v. gewone DPLL?
**Antwoord:** In plaats van constant de status van *alle* literalen in een clause te evalueren of counters bij te houden, volgt dit schema per clause slechts twee 'non-false' literalen (*watches*). Zolang deze twee literalen niet false worden, is de clause gegarandeerd niet unit of conflicting.
**Voordeel:** Enorme snelheidswinst. Bij propagatie van $x$ hoeven we enkel clauses te inspecteren waar $\neg x$ toevallig een *watch* is. Daarnaast is er **geen werk bij backtracking** nodig: we hoeven geen counters of watches te herstellen, want bij het terugdraaien van variabelen worden de watches vanzelf weer 'non-false' of 'unassigned'.
**Nadeel:** Er is specifieke overhead nodig voor het initialiseren en bijhouden van pointerlijsten (memory indirectie), wat voor heel kleine of simpele formules potentieel trager is dan een naïeve datastructuur.

**Administratie per literal**

Elke literal $\ell$ heeft een lijst $\ell.\texttt{watchers}$ van clauses die $\overline{\ell}$ als watch hebben. Wanneer $\ell$ true wordt (dus $\overline{\ell}$ false), doorloop $\overline{\ell}.\texttt{watchers}$.

### 7.3.2 Conflict-Analyse en Clause Learning

**Motivatie**

DPLL kan dezelfde fout exponentieel vaak herhalen. Voorbeeld: als $x_1 = \texttt{t}$ een conflict veroorzaakt bij $x_{99}, x_{100}$, herhaalt DPLL dit voor alle $2^{97}$ combinaties van $x_2, \ldots, x_{98}$.

**Resolutie**

Als $c_1 \ni \ell$ en $c_2 \ni \overline{\ell}$, dan is de **resolvent** $c_3 = (c_1 \cup c_2) \setminus \{\ell, \overline{\ell}\}$ een logisch gevolg van $c_1 \wedge c_2$.

Notatie: $\dfrac{c_1 \cup \{\ell\} \qquad c_2 \cup \{\overline{\ell}\}}{c_1 \cup c_2}$

Het resolutiesysteem is **volledig**: herhaalde resolutie leidt tot de lege clause $\Leftrightarrow$ de theorie is onvervulbaar.

**Conflict-Analyse (1-UIP)**

Bij een conflict:

1. Start met de conflicting clause $c_{\text{confl}}$.
2. Resolveer herhaaldelijk met de *reason*-clause van de laatst gepropageerde literal op het huidige decisieniveau.
3. Stop zodra de resulterende clause precies **één literal van het huidige decisieniveau** bevat (*1st Unique Implication Point*).

De geleerde clause $c_{\text{new}}$ is:

- Conflicting op het huidige niveau,
- Bevat precies één literal van het huidige niveau $\Rightarrow$ **propageert** op het niveau van de op-één-na-jongste literal.

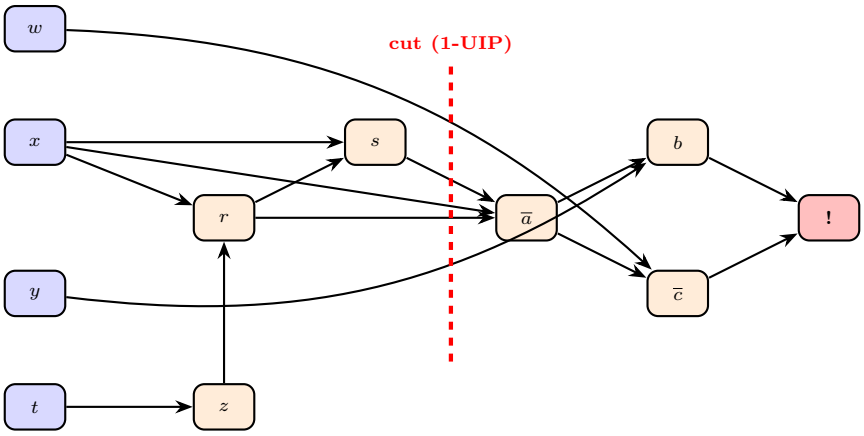



Figure 18: Implication graph met conflict en 1-UIP cut $\Rightarrow$ geleerde clause $\overline{y} \vee \overline{w} \vee a$

### 7.3.3 Non-Chronological Backtracking (Backjumping)

**Backjumping**

Na het leren van $c_{\text{new}}$:

1. Bepaal het **backjump-niveau**: het decisieniveau van de op-één-na-jongste literal in $c_{\text{new}}$.
2. Maak alle beslissingen ongedaan tot en met dit niveau.
3. De geleerde clause is nu **unit** en propageert onmiddellijk.

**Voordeel t.o.v. chronologisch backtracking:** springt potentieel vele niveaus terug, vermijdt het herhalen van irrelevante beslissingen.

**Geen noodzaak om bij te houden welke branches al verkend zijn:** de geleerde clauses "onthouden" dit en voorkomen via propagatie dat de solver dezelfde fouten maakt.

### 7.3.4 Verdere Extensies

**Moderne CDCL-extensies**

- **Search restarts:** periodiek herstarten om vast te lopen in onproductieve delen van de zoekruimte te voorkomen. Geleerde clauses blijven behouden.
- **Conflict-guided branching (VSIDS):** kies variabelen die recent bij conflicten betrokken waren. "Activity score" wordt verhoogd bij elk conflict en periodiek afgebouwd.
- **Clause deletion:** verwijder "nutteloze" geleerde clauses om geheugen te besparen.
- **Clause minimization (Self-subsumption):** resoluties na de 1-UIP voortzetten als dit resulteert in een nog kortere/sterkere clause.
- **Binary clause propagation:** speciale datastructuren die prioriteit geven aan snelle propagatie van lengte-2 clauses.
- **Proof logging:** bij een UNSAT-resultaat genereert de solver een traceerbaar bewijs dat door een onafhankelijke verifier gecontroleerd kan worden op correctheid.

## 7.4 CDCL: Samenvatting Algoritme

**CDCL Pseudocode**

```
cdcl(T):
  currentDecisionLevel := 0
  while true:
    c_confl := unit-prop()          (* uses two-watched literals *)
    if c_confl != nil:
      if c_confl = empty clause:  return false
      c_new := analyzeConflict(c_confl)   (* resolution *)
      backJump(c_new)
      T := T + {c_new}
    else if some literal l is unassigned:
      currentDecisionLevel++
      makeTrue(l, dec)
    else:
      return true
```

**Kerndata per variabele**

In een CDCL-solver wordt per atoom $x$ bijgehouden:

- `x.value`: `t`, `f`, of `u`.
- `x.level`: het decisieniveau waarop $x$ zijn waarde kreeg.
- `x.reason`: `dec` (beslissing) of een pointer naar de clause die $x$ propageerde.

Daarnaast: een **trail**-stack met alle toegekende literalen in volgorde van toekenning.

## 7.5 MaxSAT

**MaxSAT Probleem**

In sommige problemen is het onmogelijk om aan alle clauses te voldoen. We verdelen de clauses dan onder in:

- **Hard clauses** hard($F$): Moeten voldaan zijn. (Kost $= \infty$ indien onvoldaan).
- **Soft clauses** soft($F$): Optioneel, elk met een bijbehorend **gewicht** $w(c) \in \mathbb{Z}_{>0}$ dat dient als straf/kost indien de clause *niet* voldaan is.

**Doel:** Vind een interpretatie $I$ die hard($F$) voldoet en de totale kost $\text{cost}(F, I) = \sum_{c \in \text{soft}(F), c^I = \text{f}} w(c)$ **minimaliseert**.

**Blocking Variable Transformation (BVT)**

Om het algoritme simpeler te maken, transformeren we de weging op clauses naar weging op *individuele literalen*. Voor elke soft clause $c_i \in \text{soft}(F)$:

1. Introduceer een nieuwe **blocking variabele** (relaxatievariabele) $b_i$.
2. Maak $c_i$ een hard clause door de relaxatievariabele toe te voegen: $(c_i \vee b_i) \in \text{hard}(F^b)$. (Dit is de clausificatie van $b_i \to c_i$).
3. De nieuwe soft clause wordt simpelweg de unit clause $(b_i)$ met het originele gewicht $w(c_i)$.

Dit leidt tot een lineaire pseudo-Booleaanse kostfunctie: $\sum_i w_i \overline{b_i}$ (waarbij $\overline{b_i}$ 1 is indien vals en 0 indien waar).

**Pseudo-Boolean Constraint Layer & LSU**

Een pseudo-Booleaanse restrictie $\sum_i w_i b_i \leq k$ kan via een circuit $D(F, k)$ (zie BDD's) gecodeerd worden in propositielogica. De **constraint layer** is de zuivere propositielogica formule: $F_k = \text{hard}(F^b) \wedge D(F, k)$. Als $F_k$ SAT is, weten we dat er een oplossing is met kost $\leq k$.

**Linear SAT/UNSAT (LSU) Algoritme:**

1. Los $\text{hard}(F)$ op. Als UNSAT, stop (geen enkele oplossing mogelijk).
2. Bereken de huidige kost $k$ van de gevonden interpretatie $I_{\text{best}}$.
3. Construeer $F_{k-1}$ (kunnen we een oplossing vinden met strict lagere kost?).
4. Los $F_{k-1}$ op via de SAT solver.
5. Indien SAT: update $I_{\text{best}}$ en ga terug naar stap 2 met de nieuwe verlaagde $k$.
6. Indien UNSAT: we kunnen de kost niet verder verlagen; $I_{\text{best}}$ (met kost $k$) is optimaal!

## 7.6 Pseudo-Boolean Constraints Encoding via BDDs

**Binary Decision Diagrams (BDD)**

Een gerichte acyclische graaf (DAG) die een Booleaanse functie voorstelt.

- Interne knopen (geordend per variabele-niveau) beslissen over een literal $l_i$.
- Linkeruitgaande boog = `f`, rechteruitgaande boog = `t`.
- **Sinks:** twee speciale bladeren, `t` en `f`.

Een BDD kan **gereduceerd** worden (RBDD) via twee regels:

1. **Omit:** verwijder knopen met slechts 1 unieke kindknoop (de waarde van de variabele maakt niet uit, skip het level).
2. **Consolidate:** voeg knopen op hetzelfde niveau met identieke linker- en rechterkinderen samen (isomorf).

**Van BDD naar SAT (Encodering)**

Elke node $v$ in de (gereduceerde) BDD krijgt een nieuwe hulpliteral $r_v$ met de betekenis "de constraint voorstellen door deze node is voldaan". Voor een node op variabele-niveau $l_i$, met linkerkind $v_L$ en rechterkind $v_R$, genereren we de volgende drie implicaties (die omgezet kunnen worden naar CNF-clauses van lengte 3):

$$\begin{aligned}
\varphi_{vL} &:= \overline{l_i} \to (r_v \leftrightarrow r_{vL}) \\
\varphi_{vR} &:= l_i \to (r_v \leftrightarrow r_{vR}) \\
\varphi_v &:= ((r_{vL} \wedge r_{vR}) \to r_v) \wedge ((\overline{r_{vL}} \wedge \overline{r_{vR}}) \to \overline{r_v})
\end{aligned}$$

Voeg tenslotte de drie base-clauses $r_{\text{root}}$, $r_t$ en $\overline{r_f}$ toe aan de theorie. Dit vertaalt een lineaire ongelijkheid ($\sum w_i b_i \leq k$) volledig naar CNF-vorm, waarbij de RBDD-reductiestap het exponentiële aantal gegenereerde variabelen en clauses minimaliseert!