

Inleiding in de theoretische informatica

(SV 1.1)

W. Oele

9 november 2016

Inhoudsopgave

1	De efficiëntie van programmatuur	3
1.1	Het algoritme	5
1.2	De fysische rekentijd	7
1.3	De herhalingsfrequentie	8
1.4	De O-notatie	9
1.5	De complexiteit van een iteratief algoritme	12
1.6	Het slechtste-, beste- en gemiddelde gedrag	14
1.7	Berekenbaarheid	15
1.8	De klasse van NP-problemen	16
2	Recursie	18
2.1	De torens van Hanoi	18
2.2	De verdeling van een cirkel	20
2.3	De Fibonacci getallen	22
2.4	Verdeel en heers	24
2.4.1	MergeSort	25
2.4.2	BinSearch	26
2.4.3	Vermenigvuldigen van grote integers	27
2.4.4	QuickSort	28
3	De grafentheorie	30
3.1	De Multigraaf	32
3.2	Normale grafen	33
3.3	Opspannende bomen	34
3.4	Implementaties van grafen	35

3.5	De bereikbaarheid van de knopen	38
3.6	De (kortste) afstandsmatrix	41
4	Graafalgoritmen	44
4.1	Het bepalen van de minimum opspannende boom	44
4.1.1	Het algoritme van Kruskal	45
4.1.2	Het algoritme van Prim	45
4.2	Dynamisch programmeren	46
4.2.1	De kortste afstand	47
4.2.2	De kortste afstandsvector (Dijkstra)	49
4.2.3	De bereikbaarheidsmatrix (Warshall)	50
4.2.4	De kortste afstandsmatrix (Warshall)	51
4.3	Het vertak- en begrensprincipe	52
4.3.1	Het DFS-algoritme	53
4.3.2	Het BFS-algoritme	54
4.3.3	Het PFS-algoritme	56
A	Literatuur	57

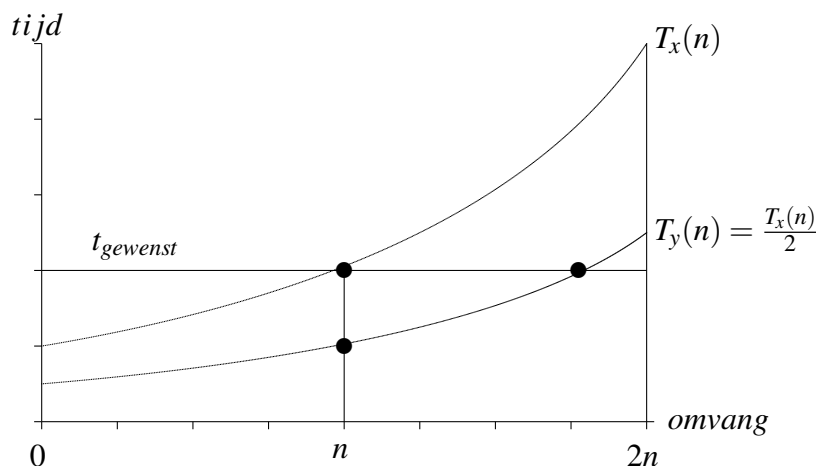
Hoofdstuk 1

De efficiëntie van programmatuur

Een programma dat het weer voor de volgende dag voorspelt, moet binnen 24 uur klaar zijn met rekenen. De pragmatische eis, dat een taak binnen een bepaalde tijd geklaard moet zijn, zou in principe aan elk programma gesteld kunnen worden. Vaak worden de consequenties die voortkomen uit het niet serieus nemen van deze eis, onderschat. De mening “*Al is de software nog zo snel, de hardware achterhaalt haar wel*” leidt in een later stadium, als het programma grotere taken moet verrichten, tot problemen.

Bij de meeste programmatuur blijkt de rekestijd en het geheugengebruik tijdens een grotere taak meer dan proportioneel toe te nemen. Bovendien blijkt dat het verhogen van de rekensnelheid en het vergroten geheugen van een computer niet altijd een proportionele verbetering van de *prestatie* van de programmatuur ten gevolge heeft. Wij kunnen dit effect demonstreren als wij de rekestijd T als functie van de taakomvang n uitzetten in een grafiek $T(n)$.

Iemand heeft de beschikking over een machine x waarop een programma een bepaalde taak met omvang n in $T_x(n)$ seconden uitvoert. Na verloop van tijd is de taakomvang verdubbeld naar $2n$. Zou een nieuwe machine y die twee keer zo snel is als machine x deze vergrote taak in de gewenste tijd kunnen uitvoeren? Om dit probleem te analyseren worden $T_x(n)$ en $T_y(n)$ samen uitgezet in een grafiek:



Bij de aanschaf van de nieuwe twee keer zo snelle computer blijkt dat de rekestijd bij een vaste taakomvang is gehalveerd. Echter de taak die binnen het gewenste tijdsbestek uitgevoerd kan worden, is niet 2 keer maar ongeveer $1\frac{3}{4}$ keer zo groot geworden. Dit nadelige effect wordt erger als de vorm van $T(n)$ erg steil is. Deze steilheid, ook wel de *complexiteit* genoemd, van de rekestijd is een maat voor de *inefficiëntie*, van de programmatuur.

Voorbeeld: De rekestijdfunctie van een bepaald type programma is een zeer steile $T(n) = 2^n$. Wat is de toename van de taakomvang met een twee keer zo snelle computer binnen een gewenst tijdsbestek? Antwoord:

$$T_x(n_x) = T_y(n_y) \quad \Rightarrow \quad 2^{n_x} = \frac{1}{2} \cdot 2^{n_y} \quad \Rightarrow \quad n_y = n_x + 1$$

Hieruit volgt dat de taakomvang met 1 is toegenomen. Op het vorige model computer kon men binnen de gewenste tijd 1000 eenheden verwerken, op het nieuwe model computer is dit amper toegenomen tot 1001 eenheden! Dit effect is nog vreemder als wij bedenken dat een 10^6 keer zo snelle computer maar 20 eenheden per tijdseenheid meer kan verwerken. Het blijkt dat de complexiteit van de programmatuur verantwoordelijk is voor dit effect. Een investering in efficiëntere programmatuur kan meer opleveren dan een investering in snellere apparatuur. De prestatie en efficiëntie kan men op verschillende manieren betrekken bij de aanschaf en ontwikkeling van programmatuur:

- Bij een bepaalde taakomvang kan men de prestatie van een programma meten met een stopwatch. Diverse programma's zijn op deze manier te vergelijken als de metingen plaats vinden op dezelfde apparatuur. Dit is een vorm van *benchmarktesten*.
- Men kan de efficiëntie van een programma bepalen bij verschillende waarden van de taakomvang. Met inter- en extrapolatietechnieken kan men de grafiek $T(n)$ vastleggen. Aan de hand van deze gegevens kan men beslissen of het programma ook in de toekomst geschikt is voor de toegenomen taakomvang.

Indien wij de programmatuur ontleden komen wij via de individuele programma's, de modulen, de objecten, de gegevensstructuren en de routines uiteindelijk op de *atomaire instructies*. Deze atomaire instructies worden in principe in een vaste tijd uitgevoerd, onafhankelijk van de taakomvang.

In het geheugen nemen atomaire instructies ieder afzonderlijk enkele bytes in beslag. De uitvoeringstijd van een atomaire instructie ligt in de orde van grootte van 10^{-9} tot 10^{-6} seconden. Per type atomaire instructie kan dit verschillen, een *optel-instructie* wordt meestal sneller uitgevoerd dan een *vermenigvuldig-instructie*.

Er zijn niet-atomaire instructies waarvan de uitvoeringstijd afhankelijk is van de instructiegegevens, de *operanden*. Bijvoorbeeld een instructie om een geheugenblok met variabele lengte te kopiëren naar een ander deel van het geheugen. Zulke instructies worden door de *vertaler* samengesteld uit atomaire instructies. Het geheugengebruik en de uitvoeringstijd van zo'n samengestelde instructie is niet alleen afhankelijk van de gebruikte vertaler, maak ook van de gebruikte apparatuur.

Indien wij de complexiteit van hogere structuren dan de atomaire instructies analyseren dan komen wij als eerste bij de procedures. Afhankelijk van de gebruikte computertaal worden procedures soms subroutines, functies of methoden genoemd. Wij vervangen deze termen voor het gemak door een omvattend begrip: het *deterministisch algoritme* of kortweg het *algoritme*.

Om een algoritme op tijdcomplexiteit te beoordelen, moeten wij de invloed van de apparatuur en de vertaler zoveel mogelijk elimineren uit de functie $T(n)$. Analoog kunnen wij een algoritme op geheugencomplexiteit beoordelen, waarbij de invloed van apparatuur en vertaler zoveel mogelijk verwijderd moet worden uit de functie $S(n)$ (S staat voor *Space*, zoals T voor *Time* staat).

1.1 Het algoritme

Wat is een algoritme? De Perzische schrijver *Abu Ja'far Mohammed ibn Musa al-Khowarizmi* schreef in de negende eeuw het boek "*Kitab al jabr w'al-muqabala*" ofwel: "*Regels voor restoratie en reductie*". Het woord "*algoritme*" is waarschijnlijk een verbastering van "*al-Khowarizmi*". De huidige betekenis van het woord algoritme komt overeen met woorden als recept, proces, methode, procedure, functie, subroutine, alles wat met instructies beschreven kan worden. Een algoritme heeft enkele belangrijke kenmerken:

Eindigheid: In het algemeen moet een algoritme stoppen na een eindig aantal instructies. Programma's die stoppen nadat hun taak is uitgevoerd, worden als algoritmen beschouwd. Hoewel een operating system is samengesteld uit algoritmen, is het zelf geen eindig algoritme;

Bepaaldheid: Een algoritme is bepaald of deterministisch als

1. elke instructie van het algoritme duidelijk aangeeft wat er gedaan moet worden. Het gebruik van atomaire instructies garandeert dat aan deze eis voldaan kan worden. De instructie om een getal van 2000 cijfers te ontbinden in priemfactoren is onbepaald of onvolledig bepaald. Zodra men in staat is deze instructie te vertalen in een aantal atomaire instructies die in eindige tijd de opdracht uitvoeren, is voldaan aan een deze eis van bepaaldheid.
2. Elke instructie van het algoritme moet een verandering bewerkstelligen, die verklaard kan worden vanuit de toestand van het algoritme en de eventuele invoergegevens. De instructie: “*pak een element uit de verzameling X*” is onbepaald. Wij kunnen deze instructie wel bepaald maken door te stellen dat het element het eerste element is van de verzameling, maar dan gaan wij ervan uit dat de verzameling een lijststructuur heeft. Dit is niet altijd het geval. Bij het ontwerpen van algoritmen maakt men regelmatig gebruik van niet-deterministische instructies. Men moet zich wel realiseren, dat zo’n niet-deterministische instructie uiteindelijk toch met behulp van atomaire instructies geïmplementeerd moet worden.

Een algoritme waarvan alle instructies bepaald zijn noemen wij een *deterministisch algoritme*. Indien wij de eis van bepaaldheid laat vallen, spreken wij over een *niet-deterministisch algoritme*;

Invoer: Elk algoritme heeft invoergegevens. Soms is dit een aantal getallen, een aantal karakters, een aantal regels of records. Meestal beschouwt men de invoer als de taakomvang. Er zijn echter algoritmen die geen echte invoer hebben. Bijvoorbeeld het berekenen van het getal π in een bepaald aantal decimalen nauwkeurig. In dit geval beschouwt men de nauwkeurigheid als de taakomvang.

Uitvoer: Elk algoritme moet een gespecificeerd resultaat opleveren. Dit resultaat mag ook een terechte foutmelding zijn. Het algoritme om de wortel van een getal te berekenen, moet bij invoer van een negatief getal een foutmelding opleveren;

Effectiviteit: Elk algoritme moet effectief zijn. Dit betekent dat alle instructies in het iets in de toestand van het programma op een relevante manier moeten veranderen. Instructies die niet bijdragen tot het gespecificeerde resultaat kunnen beter niet in het algoritme worden opgenomen;

Algoritmen worden geschreven in een taal. Meestal wordt gebruik gemaakt van een bestaande computertaal zoals C, C++ of Pascal. Soms wordt gebruik gemaakt van een pseudotaal zoals het in volgende deterministische algoritme om het maximum van een rij integer getallen te bepalen:

1. $\max(X:[\text{integer}]):\text{integer}$
2. **if** $X = []$ **then error** (“rij is leeg”) ;rij X is leeg
3. $\max \leftarrow \text{head}(X)$;neem het eerste element van X

4. $X \leftarrow \text{tail}(X)$;verwijder het eerste element van X
5. while $X \neq []$;rij X niet leeg
6. if $\text{max} < \text{head}(X)$ then $\text{max} \leftarrow \text{head}(X)$;max=maximum(head(X),max)
7. $X \leftarrow \text{tail}(X)$;verwijder volgende element
8. return (max)	;max = maximum rij X

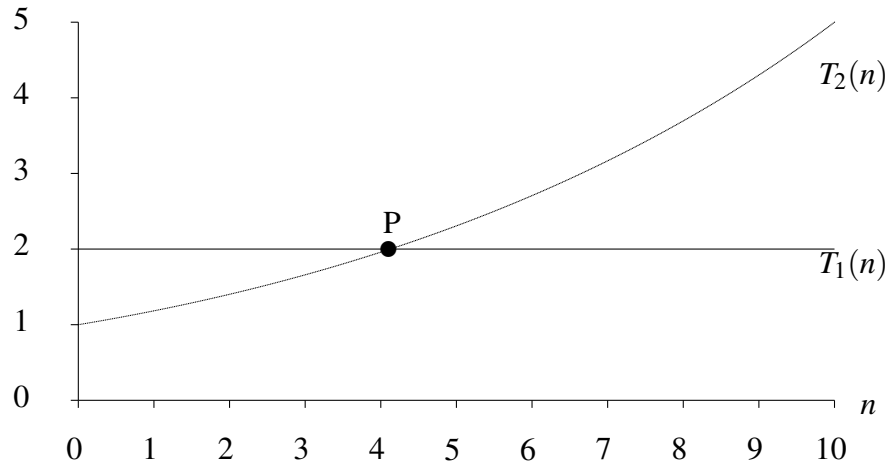
De invoer van dit algoritme is de rij integers X. De uitvoer van dit algoritme is integer **max**, die de maximale waarde uit de rij integers heeft. De foutmelding bij een lege rij integers mogen wij beschouwen als uitvoer. De **if**, **then**, **else**, \leftarrow en $=$ zijn in dit voorbeeld atomaire instructies. Er wordt ook gebruik gemaakt van enkele niet-atomaire instructies, zoals *tail()* en *head()*. Zodra van deze niet-atomaire instructies het tijd- en ruimtedrag bekend is, kunnen wij een analyse maken van het totale algoritme. Het tijdgedrag blijkt in dit voorbeeld grotendeels bepaald te worden door de afmeting van de rij X. Bovendien blijkt dat het algoritme alleen kan stoppen met een eindige rij getallen X.

Het volgende voorbeeld is een niet-deterministisch algoritme. Pas bij het beëindigen van het algoritme is de waarde *max* bepaald. Dit integenstelling met het vorige algoritme waar de waarde *max* bij elke stap bepaald wordt door de gegeven rij.

1. $\text{max}(X:\{\text{integer}\}):\text{integer}$	
2. if $X = \{ \}$ then error ("verzameling is leeg")	;verzameling X is leeg
3. $\text{max} \in X$;neem een element van X
4. $X \leftarrow X \setminus \{\text{max}\}$;verwijder het element max
5. while $X \neq \{ \}$;verzameling X niet leeg
6. $e \in X$;neem een element e uit X
7. if $\text{max} < e$ then $\text{max} \leftarrow e$;max=maximum(e,max)
8. $X \leftarrow X \setminus \{e\}$;verwijder het element e uit X
9. return (max)	;max = maximum verzameling X

1.2 De fysische rekestijd

In de praktijk van het ontwerpen van programmatuur moet men niet alleen rekening houden met de efficiëntie, de complexiteit, maar ook de met de *prestatie*, de *fysische rekestijd* van een algoritme. De fysische rekestijd is de rekestijd die wij met een stopwatch meten. Stel dat wij kunnen kiezen uit twee algoritmes. Beide algoritmes zijn geschikt voor een bepaalde taak. Het gedrag van de fysische rekestijd van beide algoritmes is verschillend. Het eerste algoritme heeft een fysische rekestijd $T_1(n) = 2$ gemeten in milliseconden. Zo'n tijdgedrag noemt men constant. Het tweede algoritme heeft een fysisch rekestijd van $T_2(n) = \frac{1}{25}n^2 + 1$ milliseconden. Dit laatste tijdgedrag wordt kwadratisch genoemd.



Bij een taakomvang van $0 \leq n \leq 4$ presteert in dit voorbeeld het kwadratisch algoritme iets beter: $T_2(n) < T_1(n)$. Bij een taakomvang $4 < n$ presteert het algoritme met het constant tijdgedrag beter. Soms worden twee algoritmen samengevoegd tot één algoritme dat het beste tijdgedrag van beide combineert.

Uit dit voorbeeld blijkt dat een kwadratisch algoritme vanaf een bepaald punt slechter gaat presteren. De plaats van dat “breakeven point” P wordt bepaald door de termen en factoren die afhankelijk zijn van de apparatuur en de vertaler (in dit geval de constanten 1, 2 en $\frac{1}{25}$) en het verschil in ‘steilheid’ van de twee functies. Deze steilheid is het gevolg van de aanwezigheid van herhalingslussen in de programmatuur.

1.3 De herhalingsfrequentie

De fysische rekestijd wordt bepaald door de atomaire instructies die op hun beurt weer afhankelijk zijn van de apparatuur en de vertaler. Omdat de *herhalingsfrequentie* onafhankelijk is van de uitvoeringstijd van de atomaire instructies is zij een betere maatstaf voor de complexiteit van algoritmen dan de fysieke rekestijd. De herhalingen in een algoritme zijn verantwoordelijk voor de steilheid van de kromme $T(n)$. In de volgende voorbeelden wordt de taakomvang voorgesteld door de variabele n . Indien de lusvariabelen i en j onaangetast blijven in het instructieblok $\{\dots\}$, kunnen wij op eenvoudige wijze de herhalingsfrequentie bepalen:

```
1. for(i=0; i<10; i++) {...}
```

De herhalingsfrequentie van $\{\dots\}$ is: 10.

2. `for(i=0; i<n; i++) { ... }`

De herhalingsfrequentie van $\{...\}$ is: n .

3. `for(i=0; i<(n*n); i++) { ... }`

De herhalingsfrequentie van $\{...\}$ is: n^2 .

4. `for(i=0; i<n ;i++)
{ for(j=0; j<n; j++) { ... } }`

De herhalingsfrequentie van $\{...\}$ is: n^2 .

5. `for(i=0; i<n; i++)
{ for(j=0; j<6; j++) { ... } }`

De herhalingsfrequentie van $\{...\}$ is: $6n$.

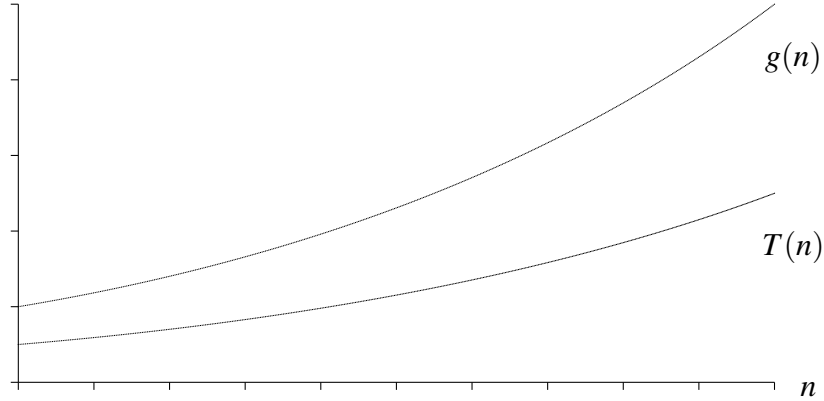
6. `for(i=0; i<n; i++)
{ for(j=i; j<n; j++) { ... } }`

De herhalingsfrequentie van $\{...\}$ is: $\sum_{i=0}^n i = \frac{1}{2}(n^2 + n)$.

Hoewel deze voorbeelden gebaseerd zijn op **for**-lussen kunnen wij op de zelfde manier de herhalingsfrequentie bepalen van de **while**- en de **repeat**-lussen. Uitzonderlijk zijn voorbeelden 1 en 5. waarin de herhalingsfrequentie niet volledig afhankelijk van de taakomvang n is. De factoren en termen van de herhalingsfrequentie die niet van de taakomvang n afhankelijk zijn, dragen niet bij in de steilheid van de functie $T(n)$. Wij hebben eigenlijk behoefte aan een notatiewijze voor de complexiteit waarin deze van de apparatuur en de vertaler afhankelijke factoren worden genegeerd.

1.4 De O-notatie

Een notatiewijze die geschikt is om de complexiteit van programmatuur aan te geven is de O-notatie. Deze notatiewijze zullen wij eerst verklaren aan de hand van twee functies $T(n)$ en $g(n)$. In de volgende grafiek stijgt functie $g(n)$ veel sterker dan functie $T(n)$.



Indien er een bepaalde waarde n_0 en een positieve constante c bestaat, zodanig dat geldt:

$$\forall n_0 < n : \quad \frac{T(n)}{g(n)} \leq c \quad (1.1)$$

dan mogen wij gebruik maken van de *O-notatie* van *Bachmann-Landau*:

$$T(n) \in O(g(n))$$

Wij spreken dit uit als “ $T(n)$ is van de grootte-orde $g(n)$ ”. In het geval dat $g(n)$ de functie $T(n)$ overstijgt, zal de limiet naar 0 naderen. Het is ook mogelijk dat de functie $T(n)$ asymptotisch gelijk blijft lopen met $c \cdot g(n)$. Het is niet mogelijk dat $T(n)$ asymptotisch meer gaat stijgen dan $g(n)$. Om de definitie van de O-notatie niet alleen voor natuurlijke getallen maar ook voor alle reële getallen geschikt te maken, wordt gebruik gemaakt van de absolute waarden van de functies. In de informatica waar wij meestal met positieve getallen voor de taakomvang werken, is deze toevoeging niet relevant.

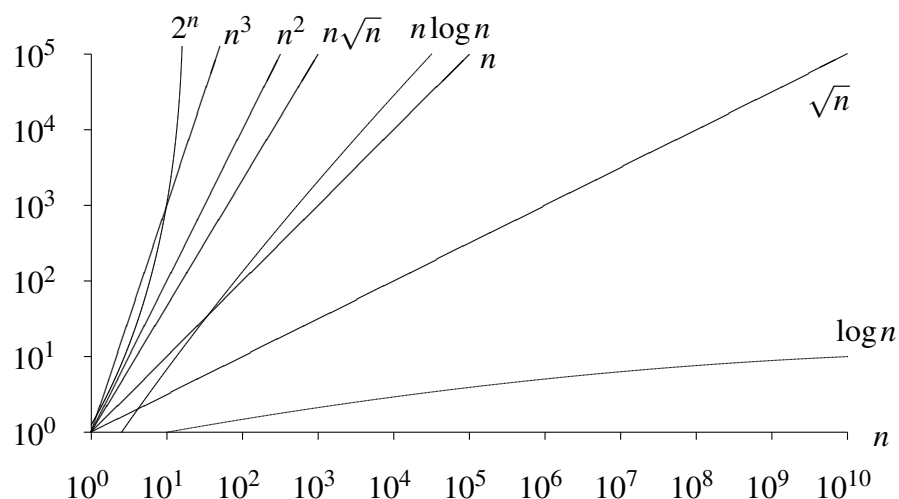
$O(g(n))$ is officieel een verzameling functies, gedefiniëerd als:

$$O(g(n)) = \{T(n) \mid \lim_{n \rightarrow \infty} \left| \frac{T(n)}{g(n)} \right| \leq c \quad 0 \leq c < \infty\} \quad (1.2)$$

Met de notatie $T(n) \in O(g(n))$ geven aan dat het asymptotische gedrag van $T(n)$ evenredig of minder stijgt dan de functie $g(n)$. De meest voorkomende functies $g(n)$ die gebruikt worden in de O-notatie voor het tijd- of geheugengedrag van algoritmen zijn:

$g(n)$	$n = 10$	$n = 100$	$n = 1000$	naam
1	1	1	1	constant
$\log n$	1	2	3	logaritmisch
\sqrt{n}	3	10	32	lineair
n	10	100	1000	
$n \log n$	10	200	3000	
$n\sqrt{n}$	32	1000	31623	
n^2	100	10000	10^6	kwadratisch
n^3	1000	10^6	10^9	kubisch
2^n	1024	10^{30}	10^{300}	exponentieel

Het asymptotisch gedrag van deze functies wordt duidelijk als wij ze uitzetten in een loglog-assenstelsel (machtsfuncties worden rechte lijnen):



De rangorde in asymptotisch gedrag van deze functies is uit te drukken als de relatie tussen de volgende deelverzamelingen:

$$O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n)$$

Als de functie $T(n)$ is een element van de verzameling functies $O(g(n))$ dan is de officiële notatie: $T(n) \in O(g(n))$. Men mag bij de kleinste mogelijke deelverzameling $O(g(n))$ noteren: $T(n) = O(g(n))$. Dit geschiedt als er geen andere functie $g(n)$ bekend is die minder stijgt en voldoet aan formule 1.1. De notatie $O(g(n)) = T(n)$ is niet toegestaan, dit veroorzaakt algebraïsche problemen. Bij het gebruik van de O-notatie kunnen wij bepaalde rekenregels toepassen. In de volgende rekenregels is c een constante en zijn $f(n)$ en $g(n)$ functies van de taakomvang n :

1. $c = O(1)$
2. $c \cdot f(n) = O(f(n))$
3. $f(n) + c = O(f(n))$
4. $g(n) + f(n) = O(f(n))$ als $O(g(n)) \subset O(f(n))$
5. $f(n) \cdot g(n) = O(f(n) \cdot g(n))$

Met de O-notatie is het mogelijk constanten weg te werken die afhankelijk zijn van de apparatuur en de gebruikte vertaler. Enkele voorbeelden (waarin $c_m, c_{m-1} \dots c_0$ constanten zijn):

- $T(n) = c_0 \quad \Rightarrow \quad T(n) = O(1)$
- $T(n) = c_1 \cdot n + c_0 \quad \Rightarrow \quad T(n) = O(n)$
- $T(n) = c_m \cdot n^m + c_{m-1} \cdot n^{m-1} + \dots c_0 \quad \Rightarrow \quad T(n) = O(n^m)$
- $T(n) = 5^n + n^3 + 3 \quad \Rightarrow \quad T(n) = O(2^n)$
- $T(n) = n \log_2 n + n \quad \Rightarrow \quad T(n) = O(n \log n)$
- $T(n) = 5 \cdot \log_e n + 24 \quad \Rightarrow \quad T(n) = O(\log n)$

De laatste drie voorbeelden laten zien dat de basis van een exponentiële functie en het grondtal van een logaritmische functie in de O-notatie niet relevant zijn. Men kiest als basis van een exponentiële functie gewoonlijk het getal 2. Bovendien laat men bij de notatie van een logaritmische functie het grondtal weg.

1.5 De complexiteit van een iteratief algoritme

Er zijn twee belangrijke methoden om een herhalingsstructuur in een algoritme aan te brengen. De meest gebruikte methode is de *iteratie*. Een iteratie maakt gebruik van instructies van het type **for**, **while**, **do ... while** en af en toe de **goto**. Indien de taak bekend is, kan men van een iteratief programma eenvoudig de complexiteit bepalen.

Als voorbeeld nemen wij de berekening van het product van twee rechthoekige matrices a en b , met respectievelijk $m \times n$ en $n \times p$ elementen. De resulterende c matrix heeft $m \times p$ elementen. De basis van een matrixvermenigvuldiging is het normale *inproduct* tussen de rijvector i van matrix a en de kolomvector j van matrix b :

$$c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}$$

Alle elementen van de c -matrix moeten berekend worden met dit vectorprodukt. Het vermenigvuldigen kan in drie geneste **for**-lussen. De binnenste **for**-lus, voorafgegaan door de initialisatie $c[i][j] = 0$, vormt het inproduct. De buitenste twee **for**-lussen selecteren de $m \times p$ elementen in de c -matrix.

```
#define m 15
#define n 10
#define p 4

void mul(matrix a, matrix b, matrix c)
{
    int i, j, k;

    for (i = 0; i < m; i++) {
        for (j = 0; j < p; j++) {
            c[i][j]=0;
            for (k = 0; k < n; k++)
                c[i][j] += a[i][k]*b[k][j];
        }
    }
}
```

Het vermenigvuldigen van twee rechthoekige matrices heeft een $O(m \cdot n \cdot p)$ complexiteit. De complexiteit wordt $O(n^3)$ als de matrices vierkant zijn, dus elk $n \times n$ elementen hebben.

Wij kunnen de volgende vuistregels hanteren voor het bepalen van de tijdcomplexiteit van iteratieve algoritmen:

- Bepaal de juiste definitie voor de taakomvang. Soms is de taakomvang niet de definiëren in één variabele (bijvoorbeeld de vermenigvuldiging van matrices met ongelijke afmetingen) maar bestaat uit meerdere factoren of termen. Anderzijds, het is meestal niet noodzakelijk om een taakomvang zeer gedetailleerd te definiëren, een hoge mate van nauwkeurigheid kan vertroebelen werken.
- Alleen lussen waarvan de herhalingsfrequentie afhankelijk is van de taakomvang doen mee bij de analyse. Indien een lus onafhankelijk is van de taakomvang, krijgt zij de status van een atomaire instructie met een $O(1)$ complexiteit;

- De totale complexiteit van niet-geneste lussen is gelijk aan de de lus met de “grootste” complexiteit. Zie rekenregel 4;
- Let op de variabelen, de functies en de procedures die op een of andere manier afhankelijk zijn van de taakomvang. Procedures of functies die onafhankelijk zijn van de taakomvang, krijgen de status van een atomaire instructie met een $O(1)$ complexiteit;
- Let op nestingen van lussen waarvan de herhalingsfrequentie van de binnenlus afhankelijk is van de lusvariabele van de buitenlus. Zie voorbeeld 6 in paragraaf 1.3.

1.6 Het slechtste-, beste- en gemiddelde gedrag

Bij de behandeling van de complexiteit zijn wij er van uit gegaan dat de steilheid van de functie $T(n)$ bepaald wordt door de herhalingen die afhankelijk zijn van de taakomvang. Deze vereenvoudigde voorstelling van complexiteit is niet volledig. De afhankelijkheid van de herhalingsfrequentie van de taakomvang kan tamelijk ingewikkeld zijn. Bijvoorbeeld, een algoritme heeft tot taak een string met ‘nullen’ en ‘enen’ te verwerken en maakt daarvoor gebruik van de procedure *verwerk()*. Stel dat de variabele $x[i]$ de waarde 0 of 1 kan aannemen:

```

1. verwerk(x:bitstring)
2. for  $i \leftarrow 1 \dots n$ 
3. if  $x[i] = 1$  then do lus1() ;lus1 heeft  $O(n)$  gedrag
4.   else do lus0()           ;lus0 heeft  $O(1)$  gedrag

```

Afhankelijk van de relatieve frequentie van de ‘nullen’ en ‘enen’ zal de procedure *verwerk* zich gedragen als een $T_0(n) = O(n)$ of een $T_1(n) = O(n^2)$ algoritme. Men noemt het slechtste gedrag (in dit geval $T_1(n) = O(n^2)$) het “*worst case*” gedrag. Het beste gedrag (in het voorbeeld $T_0 = O(n)$) wordt het “*best case*” gedrag genoemd. Het gedrag wat het meest optreedt, het gemiddelde gedrag, zal tussen die twee uitersten liggen en wordt “*average case*” gedrag genoemd. In dit voorbeeld kan het gemiddelde gedrag berekend worden met de formule voor het gewogen gemiddelde van het slechtste- en het beste gedrag als de relatieve frequentie van de ‘nullen’ en ‘enen’ bekend is:

$$E(T(n)) = \sum_{\forall x} T_x(n) \cdot P(\underline{x} = x) \quad (1.3)$$

Indien de verdeling van de ‘nullen’ en ‘enen’ bekend is, kunnen wij ook de variantie van het gedrag berekenen:

$$Var(T(n)) = \sum_{\forall x} T_x(n)^2 \cdot P(\underline{x} = x) - E(T(n))^2 \quad (1.4)$$

Als er niets bekend is over de verdeling van de invoer, neemt men een uniforme verdeling aan. In het algemeen zijn berekeningen met de formule 1.3 en de formule 1.4 niet erg gebruikelijk behalve bij het ontwerpen van kritieke *real time* programmatuur.

Een bekend algoritme *LinSearch*, het zoeken van een bepaald element in een ongesorteerde rij, onderzoekt elk element één voor één. In dit algoritme komt het slechtste en het beste gedrag duidelijk naar voren. In het slechtste geval moeten alle elementen onderzocht worden omdat het gezochte element niet aanwezig is: $O(n)$. In het beste geval wordt het gezochte element als eerste gevonden: $O(1)$. Het gemiddelde gedrag van *LinSearch* is gelijk aan: $T(n) \approx \frac{1}{2}(n+1) = O(n)$

Het zoeken in een gesorteerde rij gaat in het algemeen sneller met het *BinSearch* algoritme: $O(\log n)$. Maar om een rij te sorteren zal men op een extra verwerkingstijd moeten rekenen: $O(n \log n)$.

Een zeer goede sorteermethode zoals *QuickSort* geeft in het slechtste geval nogal wat rekentijd: $O(n^2)$. Gemiddeld doet *QuickSort* het in het algemeen goed: $O(n \log n)$.

1.7 Berekenbaarheid

Een bekend voorbeeld van een onberekenbaar probleem is een algoritme om te bepalen of twee willekeurige functies aan elkaar gelijk zijn.

Een ander bekend voorbeeld van *berekenbaarheid*, of liever gezegd onberekenbaarheid, is het "*eindigheidsprobleem*". Is het mogelijk een algoritme te construeren dat kan bepalen of een algoritme eindig is? Dit is een bijzonder interessant algoritme omdat er veel programma's zijn die blijven 'hangen'. Vooral bij *real time* programmatuur is dit fataal. Een vliegtuig kan neerstorten als het besturingsprogramma blijft hangen. Stel dat zo'n algoritme bestaat; wij noemen dit algoritme *eindig()*. Het algoritme *eindig(a)* dat bij invoer van een algoritme *a* als antwoord "waar" geeft als het algoritme *a* eindig is en "onwaar" geeft als het oneindig is. Nu gaan wij een nieuw algoritme construeren, het algoritme *geinig()* dat zichzelf gaat analyseren:

1. *geinig()*
2. **if** *eindig(geinig)* **then do forever** ;geinig wordt niet beëindigd
3. **else return** ;geinig wordt beëindigd

Dit betekent dat *geinig* eindig is als *geinig* oneindig is. Omgekeerd, *geinig* is oneindig als *geinig* eindig is. Met andere woorden, het algoritme *geinig* kan niet bestaan. Hieruit volgt dat het algoritme *eindig* ook niet kan bestaan ($(eindig \rightarrow geinig) \leftrightarrow (\neg geinig \rightarrow \neg eindig)$). Het ‘eindigheidsprobleem’ is niet berekenbaar. In de praktijk worden programma’s die niet mogen hangen, bewaakt door een *watchdog*. Dit is een eenvoudig maar uiterst betrouwbaar elektronisch apparaatje, dat ingrijpt als er niet regelmatig op een soort ‘dodemansknop’ wordt gedrukt.

Een minder bekend berekenbaarheidsprobleem is het volgende algoritme:

1. `vreemd(x:integer)`
2. **while** $x \neq 1$
3. **if** *even*(x) **then** $x \leftarrow x/2$
4. **else** $x \leftarrow 3 \cdot x + 1$
5. **return**

De stappen van het algoritme *vreemd*(3) zijn:

$$3 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

Men is er nog niet in geslaagd om de eindigheid van dit algoritme voor een willekeurige startwaarde aan te tonen.

1.8 De klasse van NP-problemen

Algoritmen die wij in een van de volgende verzamelingen kunnen onderbrengen, worden *polynomiaal* genoemd. Indien ($0 < \varepsilon < 1 < c$) dan geldt de volgende pikorde in de relaties tussen deze verzamelingen:

$$O(1) \subset O(\log \log n) \subset O(\log n) \subset O(n^\varepsilon) \subset O(n) \subset O(n \log n) \subset O(n^c)$$

Algoritmen die niet in een van deze deelverzamelingen ondergebracht kunnen worden, noemen wij *niet-polynomiaal*. Zulke algoritmen kunnen soms ondergebracht worden in een van de volgende verzamelingen:

$$O(n^{\log n}) \subset O(c^n) \subset O(n^n) \subset O(c^{c^n})$$

In de praktijk worden problemen die polynomiaal opgelost worden, *doenlijke problemen* genoemd. Dit betekent dat deze problemen in het algemeen binnen een redelijk tijdsbestek oplosbaar zijn. De zogenaamde *ondoenlijke problemen* zijn problemen die niet binnen een *polynomiaal* tijdsbestek opgelost kunnen worden.

Er is een klasse problemen die misschien polynomiaal opgelost kunnen worden als wij over een ‘*orakel*’ beschikken. Een orakel zou tijdens een zoektocht naar een schat in een doolhof tijd besparen. Men zou het orakel met behulp van parallellisme kunnen nabootsen door bij elk kruispunt een gekloond zoekproces af te starten zodat er gelijktijdig gezocht kan worden. De kloon die het eerste de schat vindt, stopt het gehele proces. Deze problemen die misschien in polynomiale tijd opgelost kunnen worden, noemen wij de klasse van *NP-problemen*. Hoewel de naam NP suggereert dat deze afkorting staat voor niet-polynomiaal, betekent het eigenlijk *niet-deterministisch polynomiaal*. Algoritmen die werken met orakels of kloonprocessen zijn in principe niet-deterministisch. Het is niet bewezen of deze NP-problemen opgelost kunnen worden door normale deterministische polynomiale algoritmen. Men neemt voorlopig aan dat dit niet mogelijk is. Anderzijds, er is nog geen bewijs gevonden dat NP-problemen niet opgelost kunnen worden door normale deterministische polynomiale algoritmen. Enkele voorbeelden uit een lange lijst van NP-problemen zijn:

Het handelreizigersprobleem: Gegeven een aantal steden en de afstanden daartussen. Bepaal de kortste route die langs deze steden gaat.

Hamiltonse cykel: Gegeven een aantal steden. Bepaal de route zodanig dat elke stad precies één keer bezocht wordt. Men mag er van uit gaan dat niet alle steden rechtstreeks met elkaar verbonden zijn.

Het rugzakprobleem: Plaats voorwerpen met verschillend gewicht en waarde in een rugzak met een beperkte capaciteit, zodanig dat de totaal vervoerde waarde zo groot mogelijk is;

Het voldoeningsprobleem: Het bepalen of een in een normaalvorm geformuleerde uitspraak uit de propositielogica waar is. Zo’n uitspraak waarin n logische variabelen aanwezig zijn, wordt met bruto rekenwerk in $O(2^n)$ stappen uitgevoerd.

Diverse verbindingsproblemen: Hoe ontwerp je een elektronische schakeling met de kortst mogelijke verbindingen tussen de componenten. Men kan ook streven naar zo weinig mogelijk verbindingen. Dit laatste probleem speelt ook bij het ontwerpen van oo-programmatuur waar men de objecten zoveel mogelijk onafhankelijk van elkaar wil maken.

De laatste jaren wordt veel onderzoek gedaan naar alternatieve algoritmen voor NP-problemen die binnen polynomiale tijd acceptabele oplossingen geven. Vaak zijn dit algoritmen voorzien van ‘orakels’ gebaseerd op toevalseffecten. Soms vinden deze algoritmen snel niet-optimale maar wel acceptabele resultaten. Anderzijds, blijken deze algoritmen soms te verzuipen in extreem lange rekentijden.

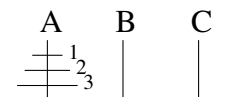
Hoofdstuk 2

Recursie

In het vorige hoofdstuk is de efficiëntie van iteratieve algoritmen behandeld, In dit hoofdstuk zullen wij de efficiëntie van recursieve algoritmen behandelen. Algoritmen worden vaak recursief ontworpen met strategieën zoals het *verdeel- en heersprincipe*. Het is vaak interessant de recursieve versie van zo'n recursief ontworpen algoritme te transformeren naar een iteratieve versie omdat een *iteratief algoritme* vaak een betere prestatie en efficiëntie heeft.

2.1 De torens van Hanoi

Het probleem van de *torens van Hanoi* werd voor het eerst geformuleerd door de Franse wiskundige E. Lucas in 1883. Het betreft een legende over een groep monniken die een toren moeten verplaatsen. Zo'n toren bestaat uit 64 schijfvormige ringen die over een paal *A* geschoven zijn. Er is een extra paal *B* aanwezig om tijdens de verplaatsingen schijven te bewaren. Zodra alle schijven zijn verplaatst naar paal *C*, zal de wereld vergaan. Deze voorspelling lijkt over een zeer korte tijd in vervulling zal gaan. Het verplaatsen van de toren kost gelukkig meer tijd dan men denkt. Ten gevolge van de voorwaarde dat een schijf alleen op een grotere schijf geplaatst mag worden, blijkt per schijf het aantal handelingen exponentieel toe te nemen.

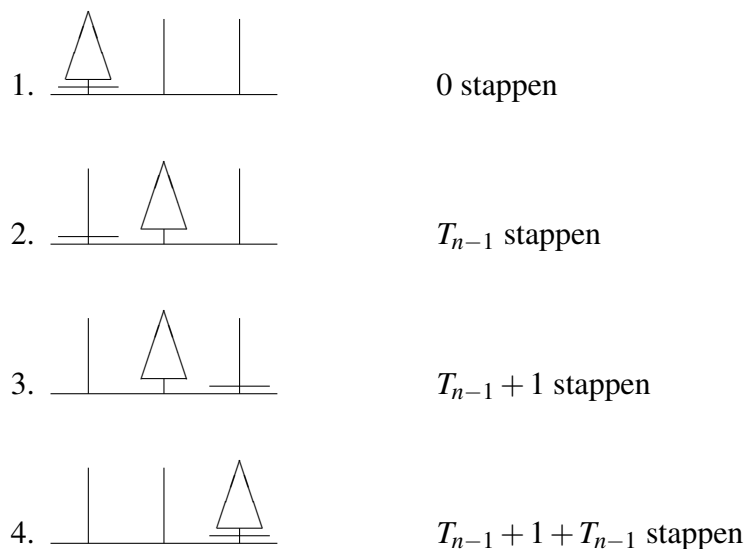


Laten wij eens gaan kijken in hoeveel stappen wij de schijven 1, 2 en 3 van paal *A* naar paal *C* kunnen verplaatsen. Paal *B* gebruiken wij als tijdelijke opslag:

1. 1 van *A* naar *C*;

2. 2 van A naar B;
3. 1 van C naar B;
4. 3 van A naar C;
5. 1 van B naar A;
6. 2 van B naar C;
7. 1 van A naar C.

Het is dus mogelijk om 3 schijven op de voorgeschreven manier te verplaatsen van paal A naar paal C in 7 stappen. Wij kunnen ons afvragen hoeveel stappen T_n nodig zijn voor het verplaatsen van n schijven. Om dit probleem op te lossen gaan wij gebruik maken van het begrip *recurrentie*. Aan de hand van de volgende afbeeldingen wordt misschien duidelijk wat wij daaronder moeten verstaan.



Oppervlakkig bekeken, lijkt het erop of er maar 3 verplaatsingen nodig zijn. Echter de reeks afbeeldingen geven niet elke individuele verplaatsing weer. Na de overgang van afbeelding 1 naar afbeelding 2 zijn er T_{n-1} stappen uitgevoerd om $n - 1$ schijven te verplaatsen. Uiteindelijk blijken er $T_n = 2 \cdot T_{n-1} + 1$ stappen nodig zijn om n schijven te verplaatsen. Wij weten ook dat er 0 stappen nodig zijn om 0 schijven te verplaatsen. Wij zijn nu in staat om een *recurrente betrekking* op te stellen:

$$\begin{aligned} T_0 &= 0 \\ T_n &= 2 \cdot T_{n-1} + 1 \quad (0 < n) \end{aligned}$$

Hoewel er in de wiskunde genoeg recepten zijn om dit probleem op te lossen, zullen wij eerst proberen de oplossing te vinden met een eenvoudige redenering. Introduceer een nieuwe functie $F_n = T_n + 1$, dan kan men de recurrente betrekking herschrijven als:

$$\begin{aligned} F_0 &= 1 \\ F_n &= 2 \cdot F_{n-1} \quad (0 < n) \end{aligned}$$

Deze recurrente betrekking is zeer gemakkelijk op te lossen:

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 2 \\ F_2 &= 4 \\ F_3 &= 8 \\ &\dots \\ F_n &= 2^n \end{aligned}$$

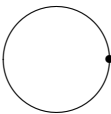
Uit $T_n = F_n - 1$ volgt dat $T_n = 2^n - 1$. Dit resultaat conflicteert niet met het eerder gevonden aantal stappen om 3 schijven te verplaatsen: $T_3 = 2^3 - 1 = 7$. Programma's voor computeralgebra zoals Maple zijn in staat zulke recurrente betrekkingen op te lossen:

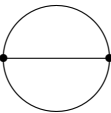
```
> T(n) = rsolve({T(n) = 2*T(n-1) + 1, T(0)=0}, T(n));
```

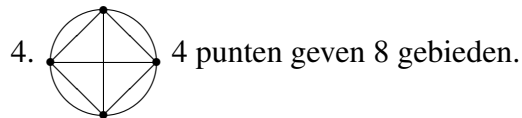
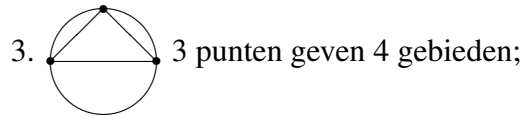
$$T(n) = 2^n - 1$$

2.2 De verdeling van een cirkel

Teken een cirkel met een willekeurig punt op de rand, de cirkel bestaat uit één gebied. Kies een ander punt op de rand. Trek een lijn tussen twee punten op de rand van een cirkel. De cirkel wordt in twee gebieden verdeeld, herhaal dit een aantal keren. In hoeveel gebieden is de cirkel maximaal verdeeld bij n punten?

1.  1 punt geeft 1 gebied;

2.  2 punten geven 2 gebieden;



Het lijkt erop dat met n punten de cirkel in 2^{n-1} gebieden verdeeld wordt. Laten wij deze formule controleren voor 5 punten: $2^{5-1} = 16$ gebieden. Wij trekken de conclusie dat het aantal gebieden T_n gelijk is aan:

$$T_n = 2^{n-1}$$

Indien wij de moeite nemen een stap verder te gaan dan komen wij in de problemen met 6 punten. Ook als wij er voor zorgen dat er zoveel mogelijk snijpunten zijn, komen wij op niet meer dan 31 gebieden. De voor de hand liggende conclusie 2^{n-1} blijkt niet geldig. Hoe moet het dan wel?

Als wij uitgaan dat de rand- en snijpunten in de cirkel niet op elkaar vallen, dan mogen wij concluderen dat:

1. de cirkel met 1 randpunt maar 1 gebied bevat;
2. elk tweetal randpunten een koorde introduceert en dus een extra gebied;
3. elk viertal randpunten een snijpunt introduceert en dus een extra gebied.

Het aantal gebieden g kan nu berekend worden voor n randpunten:

$$g = \binom{n}{4} + \binom{n}{2} + 1 = \frac{1}{24} \cdot (n^4 - 6n^3 + 23n^2 - 18n + 24)$$

n	=	1	2	3	4	5	6	7	8	9	10
g	=	1	2	4	8	16	31	57	99	163	256

Dit voorbeeld is gegeven als een waarschuwing tegen het te snel accepteren van een patroon van een beperkt aantal getallen als de oplossing van het probleem, zonder dit te controleren aan de hand van de recursieve vergelijkingen die het probleem beschrijven.

2.3 De Fibonacci getallen

Er is een patroon getallen dat in de natuur regelmatig herkend kan worden. Dit is de rij getallen 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ... de zogenaamde *Fibonacci getallen*. Bijvoorbeeld een zonnebloem is opgebouwd uit spiralen die bestaan uit zonnebloempitten. Meestal met 34 pitten in een linksdraaiende spiraal en 55 pitten in een rechtsdraaiende spiraal. Sommige kleine zonnebloemen hebben 21 en 34 of 13 en 21 pitten. Een ander voorbeeld zijn konijnen die zich voortplanten. Na elke vruchtbare periode zijn er afstammelingen bij gekomen. Indien er geen konijnen sterven, lopen de aantallen op met de steilheid van de rij *Fibonacci getallen*.

De rij van Fibonacci getallen wordt gegenereerd door de volgende recurrente betrekking:

$$\begin{aligned} f_0 &= 0 \\ f_1 &= 1 \\ f_n &= f_{n-1} + f_{n-2} \quad n > 1 \end{aligned} \tag{2.1}$$

Deze recurrente definitie van de rij getallen kan zonder veel problemen in een *recursief algoritme fibrec* vertaald worden:

1. `fibrec(n:integer):integer`
2. **if** `n > 1` **then return**(`fibrec(n - 1) + fibrec(n - 2)`)
3. **else return**(`n`)

Erg efficiënt is *fibrec* niet, de berekeningen worden overbodig herhaald; F_{n-2} wordt tweemaal herberekend, F_{n-3} wordt driemaal herberekend etc. De **if**-instructie in regel 2 kost $O(1)$. Als *fibrec* totaal met $T(n)$ evenredig is, dan is $fibrec(n - 1) + fibrec(n - 2)$ evenredig met $T(n - 1) + T(n - 2)$. Wij krijgen voor de complexiteit van *fibrec* de volgende recurrente betrekking:

$$\begin{aligned} T(0) &= O(1) \\ T(1) &= O(1) \\ T(n) &= T(n - 1) + T(n - 2) + O(1) \end{aligned}$$

Maple geeft als oplossing (voor het gemak hebben wij voor $O(1)$ de waarde 1 gesubstitueerd):

```
rsolve({T(n) = T(n-1) + T(n-2) + 1, T(0)=1, T(1)=1}, T(n));
```

$$T(n) = \frac{4}{5} \frac{\sqrt{5} \left(2 \frac{1}{-1+\sqrt{5}}\right)^n}{-1+\sqrt{5}} + \frac{4}{5} \frac{\sqrt{5} \left(-2 \frac{1}{1+\sqrt{5}}\right)^n}{1+\sqrt{5}} - 1$$

De herhalingsfrequentie $T(n)$ van *fibrec* heeft de volgende waarden:

n	=	0	1	2	3	4	5	6	7	8	9	10
$T(n)$	=	1	1	3	5	9	15	25	41	67	109	177

De herhalingsfrequentie $T(n)$ van *fibrec* is exponentieel $O(2^n)$. Dit is niet zo'n best resultaat. Het is niet moeilijk de iteratieve versie *fibit* van het algoritme te ontwerpen:

1. *fibit*(*n*:integer):integer
2. $x \leftarrow 0; y \leftarrow 1$
3. **while** $n > 0$
4. $n \leftarrow n - 1$
5. $z \leftarrow x; x \leftarrow x + y; y \leftarrow z$
6. **return**(x)

De herhalingsfrequentie $T(n)$ van de iteratieve versie *fibit* wordt bepaald door één **while**-lus, dit geeft een lineaire complexiteit $O(n)$. Bovendien kan *fibit* gebruik maken van de resultaten uit de vorige lus. De iteratieve versie is duidelijk efficiënter dan recursieve versie.

Indien wij het n^{de} Fibonacci-getal willen berekenen, kunnen wij dat getal op nog efficiëntere manier verkrijgen dan met de iteratieve versie van het algoritme. Lossen wij de recurrente betrekking 2.1 met Maple op, dan krijgen wij:

```
f(n)=rsolve({f(n) = f(n-1) + f(n-2), f(0)=0, f(1)=1}, f(n));
```

$$f(n) = \frac{1}{5} \sqrt{5} \left(2 \frac{1}{-1+\sqrt{5}}\right)^n - \frac{1}{5} \sqrt{5} \left(-2 \frac{1}{1+\sqrt{5}}\right)^n$$

Omdat machtsverheffen, vermenigvuldigen, delen, worteltrekken, optellen en aftrekken van 80-bits *floating point getallen* bij de huidige generatie processoren als atomaire instructies aanwezig zijn, heeft het uitrekenen van de functie $f(n)$ uiteindelijk een $O(1)$ complexiteit.

2.4 Verdeel en heers

Er zijn algoritmen die volgens een bepaald principe of strategie zijn ontworpen. Onder een principe of strategie verstaan wij een *meta-algoritme*, een algoritme om algoritmen te ontwerpen. Het volgende recursieve algoritme *verwerk* maakt gebruik van het *verdeel- en heersprincipe* ook wel de *Divide and Conquer*-strategie genoemd:

1. `verwerk(t:taak):resultaat`
2. **if** *te_groot*(*t*) **then**
3. $(t_1, t_2 \dots t_\alpha) \leftarrow \textit{splijt}(t)$
4. **return**(*combineer*(*verwerk*(*t*₁), *verwerk*(*t*₂) ... *verwerk*(*t*_α)))
5. **else return**(*eenvoudige_verwerking*(*t*))

Hoe kunnen wij de complexiteit van een verdeel- en heersalgoritme bepalen? Als eerste wordt de taak met de omvang n getoetst op grootte binnen $O(n)$ tijd. Is de taak te groot dan wordt zij in α deeltaken gesplitst, het splitsen geschiedt met een complexiteit binnen $O(n)$ tijd. Elke deeltaak heeft een omvang n/β , vaak is β gelijk aan α . Vervolgens worden de resultaten gecombineerd, dit kan met een complexiteit niet slechter dan $O(n)$. Indien een taak niet te groot is, kan zij met een eenvoudig algoritme worden verwerkt. Het is belangrijk dat deze taakomvang zodanig klein gekozen wordt dat dit kan geschieden binnen $O(1)$. Resultierend kunnen wij de strategie beschrijven met de recurrente betrekking:

$$\begin{aligned} T(1) &= O(1) \\ T(n) &= \alpha \cdot T(n/\beta) + O(n) \quad n > 1 \end{aligned}$$

Deze recurrente betrekking heeft de volgende oplossingen:

$$\begin{array}{ll} \alpha > \beta & T(n) = O(n^{\frac{\log \alpha}{\log \beta}}) \\ \alpha = \beta & T(n) = O(n \log n) \\ \alpha < \beta & T(n) = O(n) \end{array}$$

Dit betekent dat het verdeel- en heersprincipe, met inachtnaam van de beperkingen voor de complexiteit van het splitsen en het combineren, leidt tot een polynomiaal gedrag. De tijdcomplexiteit die ontstaat indien wij de taak in twee even grote deeltaken verdelen, wordt beschreven met de volgende recurrente betrekking:

$$\begin{aligned} T(1) &= O(1) \\ T(n) &= 2 \cdot T(n/2) + O(n) \end{aligned}$$

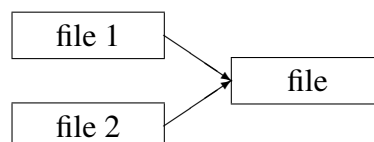
Dit geeft een redelijke, bijna lineaire, complexiteit $O(n \log n)$. Een aantal bekende algoritmen zoals *MergeSort* en (het gemiddelde gedrag van) *QuickSort* voldoen aan deze voorwaarde. De taak wordt meestal in twee gelijke delen verdeeld. Deze verdeling heeft een goede reden. Als er ongelijk verdeeld wordt, gaat dat ten koste van de efficiëntie van het algoritme. Stel dat een taak verdeeld wordt in een deeltaak met een omvang $n - 1$ en een deeltaak met een omvang 1. In dat geval geldt voor de tijdcomplexiteit de volgende recurrente betrekking:

$$\begin{aligned} T(1) &= O(1) \\ T(n) &= T(n-1) + T(1) + O(n) \end{aligned}$$

Deze recurrente betrekking geeft een polynomiaal $O(n^2)$ gedrag. Dit is minder efficiënt dan $O(n \log n)$.

2.4.1 MergeSort

In de praktijk wordt *MergeSort* toegepast als men files wil sorteren. Men deelt een file van n records in twee deelfiles met de halve omvang, dit geschiedt met een complexiteit $O(n)$. Het splitsen wordt herhaald totdat de files uiteindelijk een omvang hebben van één record per file. Het sorteren van files met één record is een triviale sorteeractie, zij zijn al gesorteerd. De resultaatfiles worden in groepen van twee 'gemerged' tot één gesorteerde file.



Het mergeproces leest van de twee invoer files het eerste record. Het record met de kleinste waarde wordt weggeschreven naar een uitvoerfile. Daarna worden de volgende records gelezen en vergeleken. Het mergeproces stopt als beide invoerfiles leeg zijn. De n files zijn nu gemergd in $n/2$ gesorteerde files met ieder 2 records. Vervolgens worden deze $n/2$ files gemergd tot $n/4$ files etc. Het 'mergen' van twee files met een maximale omvang van n records kan dus binnen een complexiteit van $O(n)$.

1. mergesort(t:[element]):[element]
2. **if** $tail(t) \neq []$ **then**
3. $(t_1, t_2) \leftarrow splits(t)$
4. $merge(mergesort(t_1), mergesort(t_2))$
5. **else return**(t)

Omdat $\alpha = \beta$ is de complexiteit MergeSort gelijk aan $O(n \log n)$. In de praktijk wordt MergeSort veel toegepast voor het sorteren van grote files. Het is een erg *stabiel algoritme*, het slechtste en beste gedrag zijn beide gelijk aan $O(n \log n)$.

2.4.2 BinSearch

Een ontaarde vorm van het verdeel- en heersprincipe is het zoekalgoritme *BinSearch*. Dit algoritme zoekt in een gesorteerde rij [elementen] naar een element e . Als het eerste element van de rij gelijk is aan het gezochte element dan stopt het algoritme met de logische waarde 'true'. In het andere geval splitst het de rij door midden in de twee deelrijen t_1 en t_2 . Als het eerste element van tweede deelrij $head(t_2)$ groter is dan het gezochte element, wordt in de eerste deelrij verder gezocht. Als het kleiner is, wordt in de tweede deelrij verder gezocht. Het zoekproces in een lege rij stopt met de logische waarde 'false'.

1. `binsearch(e:element, t:[element]):boolean`
2. **if** $t = []$ **then return** (false)
3. **if** $head(t) = e$ **then return** (true)
4. **else** $(t_1, t_2) \leftarrow split(t)$
5. **if** $head(t_2) > e$ **then** `binsearch(t_1 , e)` **else** `binsearch(t_2 , e)`

Het combineren van de deelresultaten is bij BinSearch ontaard in een simpele **if then else** test met een $O(1)$ gedrag. Als het splitsproces met een $O(1)$ complexiteit kan plaats vinden, bijvoorbeeld bij implementaties met een lineair array of random acces files , dan geldt:

$$\begin{aligned} T(1) &= O(1) \\ T(n) &= T(n/2) + O(1) \end{aligned}$$

BinSearch geeft in deze gevallen een logaritmische complexiteit $O(\log n)$.

Als het splitsproces alleen plaats kan vinden met een $O(n)$ complexiteit, bijvoorbeeld bij een gelinkte lijst of een sequentiële file, dan geldt:

$$\begin{aligned} T(1) &= O(1) \\ T(n) &= T(n/2) + O(n) \end{aligned}$$

In deze gevallen heeft BinSearch een $O(n)$ complexiteit. In dit geval is het niet beter dan een lineaire zoekactie in een ongesorteerde rij elementen.

2.4.3 Vermenigvuldigen van grote integers

Als wij twee getallen U en M ieder met een lengte van n bits met elkaar vermenigvuldigen door herhaald op te tellen dan is de complexiteit van zo'n vermenigvuldiging gelijk aan $n \cdot O(n) = O(n^2)$. Wij zijn er vanuit gegaan dat een optel instructie $O(n)$ maximaal n keer herhaald moet worden.

Wij kunnen gebruik maken van het feit dat het vermenigvuldigen van een getal met de waarde 2 op een snelle manier kan geschieden: het getal wordt binair 1 bit naar links geschoven door een 0 aan de rechter kant aan te schuiven:

$$0000010_2 \cdot 00000101_2 = 00001010_2 \quad (2_{10} \cdot 5_{10} = 10_{10})$$

Vermenigvuldigen met 2^m betekent m keer een 0-bit van rechts aanschuiven. Omgekeerd, delen door 2^m betekent m keer een 0-bit van links aanschuiven, de laagste bits schuiven eruit. Het schuiven en optellen van getallen met n bits kan binnen een complexiteit gelijk aan $O(n)$. Wij kunnen gebruik maken van het verdeel- en heersprincipe om het vermenigvuldigen een betere complexiteit te geven als het aantal bits een macht van twee is: $n = 2^k$. Bij getallen met een aantal bits, ongelijk aan een macht van twee, plaatsen wij eerst extra 0-bits links van de getallen, zonder de waarde te veranderen. Wij verdelen het $n = 2^k$ bit getal U in in twee even grote getallen U_h en U_l met ieder $\frac{1}{2}n = 2^{k-1}$ bits. Op dezelfde manier verdelen wij het getal V in twee halve getallen V_h en V_l .

Als voorbeeld nemen wij het getal $U = 1001001_2 = 73_{10}$. Dit getal heeft 7 bits. Wij verlengen U met een extra 0-bit: $U = 01001001_2 = 73_{10}$ en verdelen het in twee even grote stukken: $U_h = 0100_2 = 4_{10}$ en $U_l = 1001_2 = 9_{10}$. Nu geldt $73 = 4 \cdot 2^4 + 9$.

Wij kunnen in principe de getallen U en V als volgt verdelen:

$$\begin{aligned} U &= 2^{\frac{1}{2}n} \cdot U_h + U_l \\ V &= 2^{\frac{1}{2}n} \cdot V_h + V_l \end{aligned}$$

Daaruit volgt dat:

$$U \cdot V = (2^n + 2^{\frac{1}{2}n})U_h \cdot V_h + 2^{\frac{1}{2}n}(U_h - U_l) \cdot (V_l - V_h) + (2^{\frac{1}{2}n} + 1)U_l \cdot V_l$$

Wij kunnen volstaan met 3 vermenigvuldigingen van $\frac{1}{2}n$ getallen. Er wordt bovendien nog gebruik gemaakt van $4 - 1$ schuifoperaties (gebruikmakend van het tussenresultaat $2^{\frac{1}{2}n}$ bij de berekening van 2^n), 4 opteloperaties en 2 aftrekoperaties op n bits getallen.

$$\begin{aligned} T(1) &= O(1) \\ T(n) &= 3 \cdot T(n/2) + O(n) \end{aligned}$$

Vergeleken met vermenigvuldiging die gebaseerd is op herhaald optellen $O(n^2)$, geeft het verdeel- en heersprincipe een duidelijke verbetering ($\alpha = 3, \beta = 2$) van de complexiteit: $\alpha = 3, \beta = 2 \Rightarrow T(n) = O(n^{1.59})$.

2.4.4 QuickSort

Het sorteren van een rij elementen (array, lijst of file) met het *QuickSort* algoritme gaat als volgt:

1. `quicksort(t:[element]):[element]`
2. **if** $|t| > \text{minimale_omvang}$ **then**
3. $(t_1, t_2) \leftarrow \text{partitioneer}(t)$
4. `combineer(quicksort(t_1),quicksort(t_2))`
5. **else return**(eenvoudig_sorteer algoritme(t))

Bij QuickSort wordt de lijst zodanig gesplitst dat de sorteerde deellijsten bij het samenvoegen simpel aan elkaar geplaatst kunnen worden, de complexiteit van de functie *combineer* is $O(1)$. De splitsing van de rij element in twee partities t_1 en t_2 gebeurt op de volgende manier:

Kies een willekeurig element $m \in t$. Voor het gemak neemt men voor m het eerste of het laatste element in de rij t . Deze m wordt het scheidend element, de ‘drempelwaarde’. Alles wat kleiner is dan m komt in de eerste deellijst t_1 , alles wat groter of gelijk is in de tweede deellijst t_2 . Het partitioneren heeft een complexiteit gelijk aan $O(n)$. Als m de *mediaan* is en daardoor beide partities even groot worden, mogen wij de complexiteit berekenen met de volgende recurrente betrekking:

$$\begin{aligned} T(1) &= O(1) \\ T(n) &= 2 \cdot T(n/2) + O(n) \end{aligned}$$

Het Quicksort-algoritme heeft in dit geval een $O(n \log n)$ complexiteit. Echter bij een ongunstige keuze van een drempel m kan een partitie met 1 element en een partitie met $n - 1$ elementen ontstaan. In dit geval is geldt:

$$\begin{aligned} T(1) &= O(1) \\ T(n) &= T(n-1) + T(1) + O(n) \end{aligned}$$

Het krijgt een $O(n^2)$ gedrag waardoor het ontaardt in “*SlowSort*”. Het gemiddelde gedrag van QuickSort geeft de beste prestatie van alle sorteeralgoritmen die gebaseerd zijn op verwisselingen van elementen. De betere uitwerkingen van QuickSort bevatten maatregelen om het slechtste gedrag te voorkomen:

- Een van deze maatregelen is gebaseerd op het feit dat QuickSort in zijn onaangepaste vorm relatief veel tijd kwijt is aan kleine rijen. Daarom wordt bij een bepaalde omvang van de rij t , bijvoorbeeld 32 of minder elementen, overgegaan op een kwadratisch sorteeralgoritme met een hogere prestatie bij kleine rijen (zoals bijvoorbeeld *SelectionSort*);

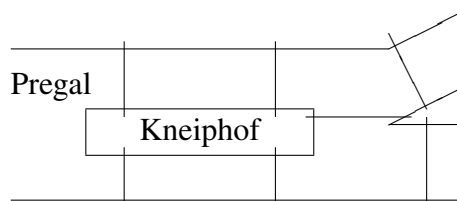
- Om zoveel mogelijk de mediaan te benaderen, worden 3 elementen gekozen, waarvan de waarde van het middelste element de drempelwaarde wordt;
- De realisaties van QuickSort met de kortste fysische rekentijd zijn vaak de iteratieve versies.

Hoofdstuk 3

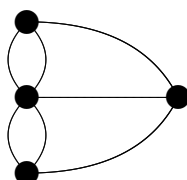
De grafentheorie

Een *graaf* is een verzameling van *knopen*, al dan niet verbonden door *takken*. Een graaf wordt soms een *netwerk* genoemd. In principe is een graaf een abstracte weergave van objecten en hun relaties. De knopen stellen de objecten voor, de takken de relaties. Wij kunnen bij objecten denken aan concrete zaken zoals plaatsen, personen en dingen. Bij relaties kunnen wij denken aan wegen, familieverbanden en verbindingen. In de abstracte wereld kunnen de objecten ook niet-concrete zaken voorstellen, zoals punten, lijnen en verzamelingen. Bovendien wordt in de abstracte wereld de boel vaak op zijn kop gezet doordat objecten relaties kunnen worden en omgekeerd relaties objecten kunnen worden.

Het *Koningsberger bruggenprobleem* is een van de bekendste problemen uit geschiedenis van de *grafentheorie*. De rivier de Pregal stroomt door de stad Koningsberg rond het eiland Kneiphof en splitst zich stroomafwaarts in twee zijtakken. Het eiland Kneiphof is met twee bruggen met de noordoever en met twee bruggen met de zuidoever verbonden. Bovendien zijn er stroomafwaarts nog twee bruggen over de zijtakken van de rivier de Pregal. Is het mogelijk een wandeling door Koningsberg te maken zodanig dat elke brug precies één keer bewandeld wordt en je bovendien weer op het vertrekpunt uitkomt?

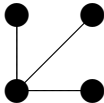
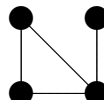


De wiskundige *L. Euler* (1707-1783) toonde aan dat zo'n wandeling in Koningsberg niet mogelijk is. Bovendien gaf hij een algemene regel voor zo'n soort wandeling in een willekeurig netwerk van wegen. Hij begon met het probleem voor te stellen als een graaf. Hij tekende de graaf met balletjes als knopen en verbindingslijnen als takken. De takken stellen bruggen voor, de balletjes de kruispunten.

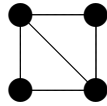


Bij de analyse van het *Koningsberger bruggenprobleem* kwam Euler tot de definitie van de *graad* $\mu(x)$ van een knoop x . De graad van een knoop is het aantal takken dat daarmee incident is. In de grafentheorie zegt men voor “*knoop x zit aan tak a* ” ook wel “*knoop x is incident met tak a* ”. In het Koningsberger bruggenprobleem zijn drie knopen van de graad 3 en één knoop van de graad 5. De verklaring is nu heel eenvoudig. Een knoop met een even graad kan zonder problemen te veroorzaken in een wandeling opgenomen worden. Alleen in een graaf waar alle knopen een even graad hebben is een rondgaande *Eulerse wandeling* mogelijk. Zulke grafen noemen wij *Eulerse grafen*. In een graaf waar twee knopen een oneven *graad* hebben, is een Eulerse wandeling mogelijk als deze beide knopen gebruikt worden als vertrek- en eindpunt. Zo’n soort graaf wordt *semi-Eulerse graaf* genoemd

Een ander bekend probleem uit de grafentheorie is het *Hamiltonse circuit*. Dit probleem werd voor het eerst geformuleerd door *W. R. Hamilton* (1705-1865) bij zijn studie van regelmatige n -hoeken. Is het mogelijk in een netwerk een wandeling te maken zodanig dat elke knoop precies één keer wordt gepasseerd en waarbij vertrekpunt gelijk is aan het eindpunt? Hoewel niet elke tak in de graaf voor de wandeling noodzakelijk is, is het Hamiltons circuitprobleem complexer dan het Eulerse wandelprobleem. Soms bevat de graaf een Hamiltons circuit. Het wordt dan een *Hamiltonse graaf* genoemd. Soms is er wel een Hamiltonse wandeling mogelijk, maar is het vertrekpunt niet gelijk aan het eindpunt. Zo’n graaf wordt *semi-Hamiltonse graaf* genoemd. Een graaf kan alleen een Hamiltons circuit bevatten als alle knopen in een *cykel* zijn opgenomen. Een knoop is opgenomen in een cykel als er een wandeling mogelijk is via een tak, waarbij men uiteindelijk weer terug kan keren via een andere tak. Met andere woorden, een cykel is een wandeling die begint en eindigt in dezelfde knoop. Hoewel het voor bijzondere grafen mogelijk is eenvoudige antwoorden te formuleren, is de vraag of een willekeurige graaf Hamiltons is, een NP-probleem.

1. Een niet-Hamiltonse graaf: 
2. Een semi-Hamiltonse graaf: 

3. Een Hamiltonse graaf:



Het Hamiltonse circuitprobleem komt in vele vormen voor: Is het mogelijk op een schaakbord van $a \times b$ velden, met een paardensprong alle velden af te lopen, zodanig dat het paard weer terugkomt op zijn eerste positie? Is het mogelijk dat een handelsreiziger een aantal steden op een route precies één keer bezoekt?

De grafentheorie kent nog vele andere problemen en toepassingen o.a:

- Hoe bepalen wij de ‘gelijkheid’ van twee grafen?
- Kan een graaf op een plat vlak getekend worden zonder dat de takken elkaar snijden?
- Hoeveel kleuren zijn er maximaal nodig om alle knopen te kleuren zodanig dat elke knoop een andere kleur heeft dan zijn buurknopen?

3.1 De Multigraaf

In het voorbeeld van de Koningsbergerbruggen is gebruik gemaakt van een graaf met bijzondere eigenschappen, de *multigraaf*. Tussen twee verschillende knopen van een multigraaf kan meer dan één tak aanwezig zijn. Multigrafen worden vooral onderzocht in de *topologie*, een gebied uit de wiskunde.

Definitie multigraaf: Een *multigraaf* G is een paar (V, E) , waar V (voor Vertices) de eindige niet-lege verzameling knopen heet. De familie ¹ van takken E (voor Edges) zijn paren uit V . Met andere woorden, er zijn één of meer knopen en nul of meer takken. Elke tak wordt bepaald door een tweetal knopen. Tussen twee knopen kunnen meer takken zitten. Een tak aan twee kanten aan dezelfde knoop verbonden zitten. Zo’n tak heet een *zelf-loop*.

Definitie graad: De graad $\mu(x)$ van een knoop x is het aantal takken dat *incident* is aan die knoop. Bij de bepaling van de graad telt de zelf-loop niet mee.

- Een multigraaf met zelf-loop:

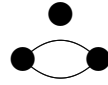


- Een samenhangende multigraaf:



¹De lijst kent aanwezigheid, aantal en volgorde van elementen: $[b, c, a, a] \neq [b, c, a] \neq [a, b, c]$. Een familie of ‘bag’ kent de aanwezigheid en het aantal elementen: $\langle b, c, a, a \rangle \neq \langle b, c, a \rangle = \langle a, b, c \rangle$. Een verzameling kent alleen de aanwezigheid van elementen: $\{b, c, a, a\} = \{b, c, a\} = \{a, b, c\}$.

- Een niet-samenhangende multigraaf;



3.2 Normale grafen

Een graaf wordt meestal in de informatica gebruikt om *binaire relaties* tussen objecten te modelleren. Tussen twee objecten is maximaal één binaire relatie mogelijk. Dit betekent dat tussen twee knopen maximaal één tak aanwezig mag zijn. Meestal zijn zelf-loops niet relevant. Indien zelf-loops noodzakelijk zijn, zullen wij dat expliciet vermelden. Al met al is de definitie van de multigraaf is niet optimaal. Daarom definiëren wij de *gewone graaf*, geschikt voor binaire relaties:

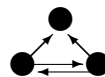
Definitie gewone graaf: Een gewone graaf G is een paar (V, E) , waar V de eindige niet-lege verzameling knopen heet. De verzameling takken E zijn paren knopen (u, v) waarbij $u, v \in V$.

Omdat E een verzameling is, kan tussen twee knopen kan niet meer dan één tak zitten. Dit is wel toegestaan bij de multigraaf, daar vormen de takken een *familie*. Als wij in het vervolg het over een graaf hebben, dan bedoelen wij de gewone graaf, zonder zelf-loops. Als het een multigraaf betreft, dan zullen wij expliciet de naam multigraaf gebruiken.

Definitie ongerichte graaf: Een ongerichte graaf G is een gewone graaf waarbij de takken een verzameling *ongeordende* paren uit V is.

Definitie gerichte graaf: Een gerichte graaf G is een gewone graaf waarbij de takken een verzameling *geordende* paren uit V is.

In een gerichte graaf mogen tussen twee verschillende knopen twee takken in verschillende richting aanwezig zijn. Van gerichte graaf maken wij een figuur door de knopen als een stip te tekenen en de takken als gerichte pijlen.



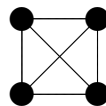
Het begrip graad $\mu(x)$ van een knoop in een gerichte graaf x is niet zo handig meer. Daarom spreekt men bij gerichte grafen liever over de *ingraad* $\mu_i(x)$ en de *uitgraad* $\mu_o(x)$ van een knoop x . De ingraad is het aantal binnenkomende pijlen, de uitgraad is het aantal uitgaande pijlen van een knoop. Een Eulerse wandeling in een gerichte graaf is alleen mogelijk als bij alle knopen de ingraad gelijk is aan de uitgraad: $\forall_{v \in V} : \mu_i(v) = \mu_o(v)$.

Definitie gewogen graaf: Een (ongerichte of gerichte) gewogen graaf, is een drietal $G = (V, E, W)$ waar V en E een gewone graaf vormen en de gewichtsfunctie $W : E \rightarrow R$ de takken E afbeeldt op de reële getallen.

Deze gewichtsfunctie stelt vaak de kosten voor van een tak of weg in afstand, tijd of geld. Men noemt zulke grafen *takgewogen grafen*. Er zijn gewogen grafen waarin een gewichtsfunctie gedefiniëerd is op de knopen. Deze grafen noemt men *knoopgewogen grafen*.

Het aantal knopen noemen wij meestal n ; de knopen zelf worden genummerd of krijgen een letter. Het aantal takken noemen wij m . Als takken ongericht zijn is uv hetzelfde als vu . Indien in een graaf alle mogelijke takken aanwezig zijn, dan noemen wij deze graaf *volledig*.

In een volledige ongerichte graaf zijn tussen de n knopen, zonder de n zelf-loops mee te tellen, maximaal $m = \binom{n}{2} = \frac{1}{2}(n^2 - n)$ takken te plaatsen:



In een volledige gerichte graaf zijn twee keer zoveel takken $m = n^2 - n$. Als wij de zelf-loops erbij tellen zijn er $m = n^2$ takken.

Als er een tak uv is dan noemen wij u en v *adjacent* of *buren*. Er zit precies één tak tussen de twee knopen. Men zegt ook wel “*knoop u is direct verbonden met knoop v* ” ook wel “*knoop u is adjacent met knoop v* ”. *Adjacency* betekent ‘aangrenzend’. Let op het verschil met incident: adjacentie is een relatie tussen knopen onderling, incidentie is een relatie tussen een knoop en een tak. Het begrip buren van een knoop u buurknoop krijgt bij de gerichte graaf uitbreiding met de begrippen *opvolgers* en *voorgangers* van een knoop. Er zijn takken van de voorgangers naar de knoop u en er zijn takken van de knoop u naar de opvolgers.

3.3 Opspannende bomen

Als men in een graaf nul of meer knopen en incidentie takken verwijdert dan krijgt men één of meer deelgrafen. Een graaf waar niet alle knopen vanuit een willekeurige knoop via een wandeling bereikbaar zijn, noemen wij een *niet-samenhangende graaf*. De deelgrafen van knopen die wel met elkaar verbonden zijn noemen wij de *componenten* van de niet-samenhangende graaf. Een graaf die maar uit één component bestaat, noemen wij een *samenhangende graaf*. Indien wij uit een samenhangende graaf alleen takken die cycli veroorzaken, verwijderen dan hebben wij een bijzondere deelgraaf geconstrueerd, de *opspannende boom*. In een graaf zijn vaak meer opspannende bomen aanwezig.

Definitie opspannende boom: Een opspannende boom is een samenhangende deelgraaf zonder cykels met n knopen en $n - 1$ takken. Uit elke samenhangende graaf zijn één of meer opspannende bomen te construeren.

Definitie minimum opspannende boom: Dit is een boom uit de verzameling opspannende bomen van een samenhangende takgewogen graaf. De som van de takgewichten van deze opspannende boom heeft de laagste waarde die mogelijk is.

Er zijn veel toepassingen van opspannende bomen. Men kan zich een netwerk van wegen voorstellen door een takgewogen graaf. Hierbij kunnen we het gewicht $W(u, v)$ van tak uv interpreteren als de lengte van de weg tussen stad u en stad v . Wij zouden bijvoorbeeld de volgende vragen kunnen stellen:

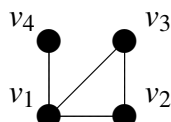
1. Welke wegen zijn voldoende om alle steden met elkaar te kunnen verbinden?
2. Zijn er meer antwoorden mogelijk op vraag 1?
3. Welke wegen zijn voldoende en geven de minste onderhoudskosten als de onderhoudskosten evenredig zijn met de lengte van het wegennetwerk?

Het eerste probleem komt overeen met het bepalen van een element uit de verzameling opspannende bomen van de graaf. Het antwoord op het tweede probleem is het aantal opspannende bomen in een graaf met n onderscheidbare knopen. Het derde probleem is het bepalen van een element uit de verzameling van de zogenaamde *minimum opspannende bomen* van een takgewogen graaf.

3.4 Implementaties van grafen

Een bepaalde graafimplementatie wordt gekozen indien de meest voorkomende operaties op een graaf efficiënter worden uitgevoerd dan bij andere implementaties. Daarbij moeten wij niet alleen rekening houden met de rekentijd maar ook met het geheugengebruik. Het wandelen of zoeken in een graaf blijkt de implementatiekeuze sterk te beïnvloeden. Wandelingen bestaan grotendeels uit de keuzen die gemaakt worden bij het passeren van een knoop. Deze keuzen zijn gebaseerd op eigenschappen, gewichten en aantallen van burens, voorgangers, opvolgers, incidentie takken en afgelegde wegen.

Er zijn veel implementaties van een graaf. In het algemeen kunnen wij de implementaties in twee groepen verdelen: de lijstimplementaties en de matriximplementaties.



De *binaire adjacenciematrix* slaat de graaf op als een binaire $n \times n$ matrix waarbij bit α_{rk} de logische waarde 1 of 0 heeft, afhankelijk of rk een tak is of niet. Uit de ongerichtheid volgt $\alpha_{rk} = \alpha_{kr}$. De matrix is symmetrisch rond de hoofddiagonaal. Daarom slaat men de vaak alleen de boven- of onderdriehoeksmatrix op. Het is niet mogelijk een multigraaf te implementeren als een binaire adjacenciematrix. Bij een multigraaf kiezen wij voor een *adjacencie matrix* van natuurlijke getallen. Elk element α_{rk} van zo'n adjacenciematrix is gelijk aan het aantal takken tussen knoop r en knoop k . Iets soortgelijks geschiedt ook bij de adjacencielijst voor een multigraaf.

$$Adjacenciematrix = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

De *adjacencielijst* bestaat uit n lijsten, voor elke knoop één. Deze lijst slaat voor elke knoop zijn burens op.

$$\begin{aligned} v_1 &: [v_2, v_3, v_4] \\ v_2 &: [v_1, v_3] \\ v_3 &: [v_1, v_2] \\ v_4 &: [v_1] \end{aligned}$$

De *incidentielijst* bestaat uit m lijsten, elke tak (alle takken moeten genummerd zijn) is incident aan twee knopen. Deze manier van representeren wordt meestal voor multigrafen gebruikt:

$$\begin{aligned} t_1 &: [v_1, v_2] \\ t_2 &: [v_2, v_3] \\ t_3 &: [v_1, v_4] \\ t_4 &: [v_1, v_3] \end{aligned}$$

Men kan de incidentiestructuur ook in matrixvorm (twee lineaire arrays) noteren, de *incidentiematrix*:

t_1	t_2	t_3	t_4
v_1	v_2	v_1	v_1
v_2	v_3	v_4	v_3

Maple maakt gebruik van adjacentielijsten. Ongerichte takken worden tussen verzamelingshaakjes ‘{ }’ geplaatst, gerichte takken worden tussen de lijsthaakjes ‘[]’ geplaatst. Wij zullen het vorige voorbeeld met Maple invoeren, waarna wij de graad van knoop v_1 zullen bepalen:

```
> with(networks);
> new(G);
> addvertex(1,2,3,4,G);

2,3,1,4

> addedge({{1,2},{1,3},{1,4},{2,3}},G);

e4,e5,e6,e7

> vdegree(1,G);

3
```

In de lijsten is de symmetrie te zien: in principe komt i voor in lijst j alleen als j voorkomt in lijst i . Er is natuurlijk een variant mogelijk waarbij deze redundantie wordt voorkomen. De lijst kan gesorteerd zijn, maar dit hoeft niet; hangt van de toepassing af, en meestal is het niet van belang. Wij noemen deze implementatie een adjacentielijst maar er is natuurlijk niets op tegen, de burens, opvolgers en voorgangers van een knoop op te slaan in een gebalanceerde boom, hash-tabel of andere verzamelingsstructuur. In het algemeen zijn graaf implementaties *niet-deterministisch*, vaak omdat de nummering van takken of knopen vrij te kiezen is. *Als gevolg hiervan geven sommige algoritmen die afhankelijk zijn van deze willekeurig gekozen volgorde van knopen of takken verschillende correcte uitkomsten.*

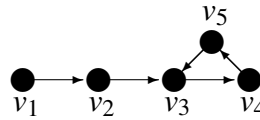
De meeste implementaties hebben voor- en nadelen:

	adj. matrix	adj. lijst	inc. lijst/matrix
Geheugengebruik	$O(n^2)$	$O(m)$	$O(\log m)$
Is er een tak x naar y ?	$O(1)$	$O(\mu_o(x))$	$O(\log m)$
Voor alle opvolgers van x ?	$O(n)$	$O(\mu_i(x))$	$O(\log m)$
Voor alle voorgangers van x ?	$O(n)$	$O(n+m)$	$O(\log m)$

Wij noemen het een graaf *dichtbezet* of *dense* als het aantal takken meer is dan het aantal knopen: $m > n$. Als dat niet zo is, dan noemen wij de graaf *dunbezet* of *sparse*. De meeste graafalgoritmen werken met dunbezette grafen, de adjacentielijst is dan het efficiëntst. Dichtbezette grafen worden meestal in een adjacentiematrix geïmplementeerd. Multigrafen worden meestal met een incidentielijst geïmplementeerd. Het is eenvoudig om van een lijst een matrix te maken en omgekeerd.

3.5 De bereikbaarheid van de knopen

In de binaire adjacentiematrix geven de elementen $\alpha_{x,y}$ met een 1 of een 0 aan of een tak van knoop x naar y aanwezig is of niet. Wij kunnen deze adjacentiematrix voorstellen als de matrix met alle wandelingen die in één “stap” te maken zijn. Bijvoorbeeld de volgende graaf:



heeft de volgende adjacentiematrix A :

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Laten wij deze binaire adjacentiematrix eens nader bekijken. Een element x in de rijvector r van de adjacentie matrix is 1 als er een tak is van knoop r naar knoop x . Indien deze tak er niet is, is het element 0. Een element x in de kolomvector k is 1 als er een tak is van knoop x naar knoop k , anders is het element 0. Wij definiëren het *binair inproduct* tussen de rijvector r en de kolomvector k als:

$$\begin{pmatrix} a_{r,1} & a_{r,2} & \dots & a_{r,n} \end{pmatrix} \oplus \begin{pmatrix} b_{1,k} \\ b_{2,k} \\ \dots \\ b_{n,k} \end{pmatrix} = (a_{r,1} \wedge b_{1,k}) \vee (a_{r,2} \wedge b_{2,k}) \dots (a_{r,n} \wedge b_{n,k})$$

Wij kunnen ons afvragen wat dit te maken heeft met de bereikbaarheid van knopen. Een 1 als resultaat van de binaire vermenigvuldiging tussen rijvector r en kolomvector k geeft aan of er een verbinding is tussen r en k via een willekeurige tussenknoop. Daarentegen geeft een 0 als resultaat aan dat er geen verbinding is via een tussenknoop.

Bijvoorbeeld: In rij 1 en kolom 2 uit de adjacentiematrix van de voorbeeld graaf heeft knoop 1 een tak naar knoop 2, knoop 2 heeft alleen een tak uit knoop 1. Er is geen 2-staps wandeling mogelijk van knoop 1 naar knoop 2:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = (0 \wedge 1) \vee (1 \wedge 0) \vee (0 \wedge 0) \vee (0 \wedge 0) \vee (0 \wedge 0) = 0$$

Bij een normale matrixvermenigvuldiging $C = A \times B$ wordt elk element $c_{r,k}$ bepaald door het *normale inproduct* van rijvector r en kolomvector k . Wij kunnen analoog aan de normale matrixvermenigvuldiging een matrixvermenigvuldiging maken met het binaire inproduct \otimes . De operatie $A \otimes B = C$ is een binaire matrixvermenigvuldiging tussen matrix A en B . Wat stellen nu de elementen $\alpha_{x,y}$ in de matrix $A^2 = A \otimes A$ voor:

- $\alpha_{x,y} = 1$: Er is een wandeling in 2 stappen van x naar y ;
- $\alpha_{x,y} = 0$: Er is geen wandeling in 2 stappen van x naar y .

Of bij: $A^i = \overbrace{A \otimes A \otimes \dots \otimes A}^i$

- $\alpha_{x,y} = 1$: Er is een wandeling in i stappen van x naar y ;
- $\alpha_{x,y} = 0$: Er is geen wandeling in i stappen van x naar y .

Wat is de langste wandeling gemeten naar het aantal gepasseerde takken in een graaf met n knopen? In elke graaf met n knopen kunnen wij altijd in $n - 1$ stappen naar een willekeurige knoop die indirect verbonden is. Dit ligt voor de hand, denk maar aan de opspannende boom. Is de graaf niet-samenhangend, dan is het maximaal aantal stappen in principe nog minder dan $n - 1$. Men kan eerder stoppen met het berekenen van matrix A^{i+1} als $A^i = A^{i-1}$ $1 < i < n - 1$.

Omdat wandelingen met oneindig aantal stappen mogelijk zijn, geeft men de bereikbaarheidsmatrix aan met de macht ∞ . Elke cykel kan oneindig maal bewandeld worden. Voor elk element $\gamma_{x,y}$ in de bereikbaarheidsmatrix A^∞ geldt:

- $\gamma_{x,y} = 1$: Er is een wandeling van x naar y ;
- $\gamma_{x,y} = 0$: Er is geen wandeling van x naar y .

Wij kunnen nu de *bereikbaarheidsmatrix* A^∞ met de elementen bepalen, door alle wandelingen met i of minder stappen te combineren:

$$A^\infty = \sum_{i=1}^{n-1} A^i \quad (3.1)$$

Om de bereikbaarheidsmatrix A^∞ te berekenen, definiëren wij eerst de operator \oplus op twee binaire matrices. Deze operator kan men beschouwen als een *binaire optelling* van matrices:

$$\begin{aligned}
& \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \dots & \dots & \dots & \dots \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{pmatrix} \oplus \begin{pmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,n} \\ b_{2,1} & b_{2,2} & \dots & b_{2,n} \\ \dots & \dots & \dots & \dots \\ b_{n,1} & b_{n,2} & \dots & b_{n,n} \end{pmatrix} \\
&= \begin{pmatrix} a_{1,1} \vee b_{1,1} & a_{1,2} \vee b_{1,2} & \dots & a_{1,n} \vee b_{1,n} \\ a_{2,1} \vee b_{2,1} & a_{2,2} \vee b_{2,2} & \dots & a_{2,n} \vee b_{2,n} \\ \dots & \dots & \dots & \dots \\ a_{n,1} \vee b_{n,1} & a_{n,2} \vee b_{n,2} & \dots & a_{n,n} \vee b_{n,n} \end{pmatrix} \tag{3.2}
\end{aligned}$$

De bereikbaarheidsmatrix A^∞ wordt nu berekend door alle A^i matrices binair op te tellen met behulp van de operator \oplus :

$$A^\infty = A \oplus A^2 \oplus A^3 \oplus \dots \oplus A^{n-1} \tag{3.3}$$

De voorbeeldgraaf heeft uiteindelijk de volgende bereikbaarheidsmatrix:

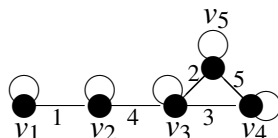
$$A^\infty = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Uit de bereikbaarheidsmatrix A^∞ blijkt onder andere dat de knopen v_1 en v_2 geen onderdeel zijn van een cykel (de *diagonaalelementen* in de bereikbaarheidsmatrix van deze knopen zijn beide nul: $\gamma_{1,1} = \gamma_{2,2} = 0$).

Deze berekening van de bereikbaarheidsmatrix bestaat uit n keer een matrixvermenigvuldiging $O(n^3)$ en heeft daarom een tijdcomplexiteit van $O(n^4)$. Later zullen wij het *algoritme van Warshall* behandelen dat de bereikbaarheidsmatrix efficiënter berekent met een $O(n^3)$ complexiteit.

3.6 De (kortste) afstandsmatrix

In een netwerk (zie volgende schema) zijn dorpen ($v_1 \dots v_5$) verbonden met wegen, elke weg heeft een bepaalde afstand in kilometers. Bij elke tak is de gewichtswaarde aangegeven, de zelf-loops hebben een gewichtswaarde 0.



Wij kunnen van dit netwerk van dorpen een directe afstandsmatrix DA maken. In DA wordt elke tak van x naar y aangegeven met zijn gewichtswaarde. De zelf-loops hebben in dit voorbeeld allemaal de gewichtswaarde 0. De gewichtswaarde wordt oneindig als er geen directe weg aanwezig is tussen knoop x en y . Omdat in ons voorbeeld sprake is van een ongerichte graaf is de DA matrix symmetrisch rond de hoofddiagonaal.

$$DA = \begin{pmatrix} 0 & 1 & \infty & \infty & \infty \\ 1 & 0 & 4 & \infty & \infty \\ \infty & 4 & 0 & 3 & 2 \\ \infty & \infty & 3 & 0 & 5 \\ \infty & \infty & 2 & 5 & 0 \end{pmatrix}$$

Een bekend probleem uit de grafentheorie is het probleem om van een directe afstandsmatrix DA een kortste afstandsmatrix KA te maken. Wij behandelen binnenkort een algoritme voor dit probleem, het algoritme van Warshall. Gelukkig is in ons voorbeeld de KA matrix eenvoudig uit het netwerkschema te bepalen.

$$KA = \begin{pmatrix} 0 & 1 & 5 & 8 & 7 \\ 1 & 0 & 4 & 7 & 6 \\ 5 & 4 & 0 & 3 & 2 \\ 8 & 7 & 3 & 0 & 5 \\ 7 & 6 & 2 & 5 & 0 \end{pmatrix}$$

Welk dorp komt het meest in aanmerking als wij in een van de vijf dorpen een politiepost willen vestigen? Wij kunnen als criterium de meest centrale ligging nemen, het dorp dat de kortste afstand heeft tot alle andere dorpen. Dit doen wij door de som van elke rijvector te berekenen met het normaal inproduct met de *eenheidskolomvector* :

$$\begin{pmatrix} 0 & 1 & 5 & 8 & 7 \\ 1 & 0 & 4 & 7 & 6 \\ 5 & 4 & 0 & 3 & 2 \\ 8 & 7 & 3 & 0 & 5 \\ 7 & 6 & 2 & 5 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 21 & 18 & 14 & 23 & 20 \end{pmatrix}$$

Het blijkt dat dorp v_3 met de kleinste rijsum voor de politiepost de meest centrale ligging heeft. Wij kunnen het probleem verder uitbreiden. Stel dat elk dorp een verschillend aantal inwoners heeft, aangegeven met de kolomvector b :

$$b = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{pmatrix} = \begin{pmatrix} 150 \\ 80 \\ 70 \\ 90 \\ 60 \end{pmatrix}$$

Deze kolomvector b maakt van de graaf een knooppegwogen graaf. Wat is nu de beste vestigingsplaats voor een politiepost, indien het inwoneraantal van elk dorp meetelt?

$$\begin{pmatrix} 0 & 1 & 5 & 8 & 7 \\ 1 & 0 & 4 & 7 & 6 \\ 5 & 4 & 0 & 3 & 2 \\ 8 & 7 & 3 & 0 & 5 \\ 7 & 6 & 2 & 5 & 0 \end{pmatrix} \cdot \begin{pmatrix} 150 \\ 80 \\ 70 \\ 90 \\ 60 \end{pmatrix} = \begin{pmatrix} 1570 & 1420 & 1460 & 2270 & 2120 \end{pmatrix}$$

Uit het normale product van de KA matrix met de bevolkingsvector b blijkt dat dorp v_2 voor de politiepost de beste vestigingsplaats is. In de informatica bestaat een soortgelijk probleem bij het plaatsen van een server in een client-server netwerk. De transmissietijden komen overeen met de afstanden, de verkeersintensiteit met de bevolking.

Bij grote matrices is het met de hand uitrekenen van vermenigvuldigingen een tijdrovende bezigheid. Gelukkig kan dit sneller met Maple:

```
> with(linalg):

> KA:=matrix([[0,1,5,8,7],
               [1,0,4,7,6],
               [5,4,0,3,2],
               [8,7,3,0,5],
               [7,6,2,5,0]]);
```

$$KA := \begin{bmatrix} 0 & 1 & 5 & 8 & 7 \\ 1 & 0 & 4 & 7 & 6 \\ 5 & 4 & 0 & 3 & 2 \\ 8 & 7 & 3 & 0 & 5 \\ 7 & 6 & 2 & 5 & 0 \end{bmatrix}$$

```
> b := vector( [150,80,70,90,60] );
```

$$b := [150 \quad 80 \quad 70 \quad 90 \quad 60]$$

```
> multiply(KA, b);
```

$$[1570 \quad 1420 \quad 1460 \quad 2270 \quad 2120]$$

Hoofdstuk 4

Graafalgoritmen

Wij beginnen met enkele algoritmen om de minimum opspannende boom van een graaf te bepalen. Deze twee algoritmen zijn het algoritme van *Kruskal* en het algoritme van *Prim*. Vervolgens behandelen wij het *dynamisch programmeren*, een *bottom-up* principe dat in veel algoritmen gebruikt wordt. Evenals het verdeel- en heersprincipe leidt het dynamisch programmeren tot recurrente betrekkingen die zeer gemakkelijk in recursieve algoritmen zijn om te zetten. Bovendien geeft het dynamisch programmeren aanleiding tot het gebruik van tussenresultaten die leiden tot efficiënte iteratieve versies. Tenslotte behandelen wij enkele graafalgoritmen om in een graaf te wandelen, waarvan wij gebruik maken bij het *vertak- en begrensprincipe*.

Wij gaan er in dit hoofdstuk vanuit dat de student bekend is met de definities, toepassingen en implementaties van de volgende abstracte datastructuren: de *stack*, de *queue* en de *priority-queue*. In het bijzonder verwijzen wij naar de belangrijkste implementatie van de priority-queue: de *heap*.¹

4.1 Het bepalen van de minimum opspannende boom

Voor opspannende bomen gelden de volgende regels:

- Elke opspannende boom van een samenhangende graaf heeft altijd $n - 1$ takken. Dit is eenvoudig aan te tonen.
- Elke opspannende boom van een niet-samenhangende graaf met c componenten heeft altijd $n - c$ takken. Ook dit is eenvoudig aan te tonen.

¹Uit een priority-queue wordt altijd het element met de hoogste prioriteit het eerst verwijderd. Deze prioriteit wordt door de gebruiker gedefiniëerd. Een priority-queue met een heap-implementatie heeft een $O(\log n)$ complexiteit voor het toevoegen en verwijderen van een element. Het vullen en leegmaken van een heap met n elementen kan met een efficiëntie van $O(n \log n)$.

- In een samenhangende takgewogen graaf zijn opspannende bomen met een minimaal gewicht. Een boom met een minimaal gewicht noemen een *minimum opspannende boom*.

4.1.1 Het algoritme van Kruskal

Een bekend algoritme voor het bepalen van een minimum opspannende boom G' in een samenhangende takgewogen graaf G werd in 1956 geformuleerd door *J. B. Kruskal*. Het *algoritme van Kruskal* begint met de verzamelingen knopen uit de originele samenhangende graaf te kopiëren naar een verzameling knopen waarin de minimum opspannende boom G' geconstrueerd wordt: $V' \leftarrow V$. De takken worden indien zij geen cycli veroorzaken in toenemend gewicht geplaatst in de verzameling E' . Bij het plaatsingsproces vormen deze takken deelgrafen die zelf ook bomen zijn. Deze deelgrafen groeien uiteindelijk aan elkaar tot een minimum opspannende boom (V', E') . Indien een tak in een deelgraaf een cykel veroorzaakt, dan komen er in deze deelgraaf met n_d knopen meer dan $n_d - 1$ takken. In dat geval wordt zo'n tak genegeerd.

```

1. kruskal( $G(V,E)$ :graaf): $G'(V',E')$ :minimum opspannende boom
2.  $E' \cup \{\}$ ;  $V' \leftarrow V$ 
3. forall  $e \in E$ :
4.   forsome  $m \in E \wedge \text{gewicht}(m) = \min(\text{gewicht}(e))$  :
5.      $E \leftarrow E \setminus \{m\}$ 
6.     if  $m$  geen cycli introduceert then  $E' \leftarrow E' \cup \{e\}$ 
7. return  $(V', E')$ 

```

De efficiëntie van het algoritme van Kruskal wordt vooral bepaald door de “sorteeractie” in regel 3 en 4. De constructie **forall** en **forsome** wordt vaak met een *heap* uitgevoerd. De tijdcomplexiteit voor dit algoritme op een samenhangende takgewogen graaf met m takken komt uit op $O(m \log m)$.

4.1.2 Het algoritme van Prim

Geheel onkundig van het algoritme van Kruskal bedacht *R.C. Prim* in 1957 een algoritme om minimum opspannende bomen te vinden. Dit algoritme bleek achteraf in 1930 al eerder bedacht door *V. Jarnik*. Toch is het algoritme nog steeds bekend als het *algoritme van Prim*.

In het algoritme van Prim wordt een opspannende boom $G'(V', E')$ geconstrueerd uit de samenhangende gewogen graaf $G(V, E)$. Het begint met het plaatsen van een willekeurige knoop v_0 in de verzameling knopen V' van de te construeren boom. Daarna worden

alle takken die met deze boom in constructie verbonden zijn, beoordeeld op het takgewicht. De tak m met het lichtste gewicht wordt opgenomen in de boom (inclusief zijn incidenten knoop v). De boom in constructie groeit per stap aan met één tak en één knoop. De boom stroomt via de lichtste takken als een watervlek over de graaf uit. Deze opspannende boom is, in tegenstelling met de bomen die ontstaan tijdens het Kruskal algoritme, samenhangend. Het groeiproces stopt als er geen takken meer te plaatsen zijn.

Het algoritme van Prim is gebaseerd op het *dynamisch programmeerprincipe* en het vertoont verwantschap met het *algoritme van Dijkstra*. Hierover later meer.

1. $\text{prim}(G(V,E):\text{graaf}):G'(V',E'):\text{minimum opspannende boom}$
2. $V' \cup v_0; \quad V \leftarrow V \setminus v_0$
3. **while** $V \neq \{\}$:
4. **forall** $v' \in V'$:
5. **forsome** $m = (v \in V, v' \in V') \in E \wedge \text{gewicht}(m) = \min(\text{gewicht}((v, v')))$:
6. $V' \leftarrow V' \cup v; \quad V \leftarrow V \setminus v$
7. $E' \leftarrow E' \cup m$
8. **return** (V', E')

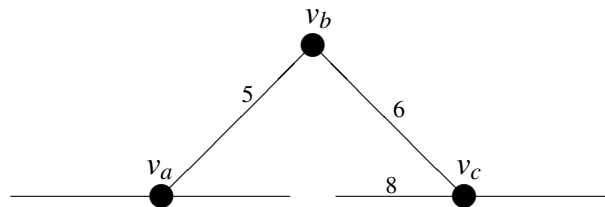
Regel 3, de hoofdloop, wordt n maal doorlopen. Regels 4, 5, 6 en 7 worden maximaal n keer uitgevoerd. De totale complexiteit van het algoritme van Prim komt op $O(n^2)$.

4.2 Dynamisch programmeren

Het verdeel- en heersprincipe maakt gebruik van een *top-down* aanpak van het probleem. De top-down aanpak blijkt uit het verdelen van een probleem in kleinere deelproblemen. De totale oplossing is op simpele wijze te combineren uit de deeloplossingen. Dit leidt tot relatief eenvoudige recursieve algoritmen. Niet alle problemen zijn geschikt voor zo'n top-down aanpak. Soms werkt een *bottom-up* aanpak beter. Bij een bottom-up aanpak ontbreekt de verdeling in kleinere problemen, de totale oplossing wordt uit deeloplossingen geconstrueerd. Een voordeel van het bottom-up principe is het feit dat wij kunnen profiteren van de tussenresultaten. De iteratieve versie van Fibonacci in paragraaf 2.3 is een goed voorbeeld van een bottom-up aanpak, waarbij gebruik gemaakt wordt van tussenresultaten.

Bij het *dynamisch programmeren* speelt naast het bottom-up principe ook het begrip *optimale oplossing* een rol. Het combineren van deeloplossingen is bij dynamisch programmeren ingewikkelder dan bij het verdeel- en heersprincipe. Welke deeloplossingen moeten gecombineerd worden en welke niet? Gelukkig is er bij het dynamisch programmeren sprake van optimale oplossingen. Optimale oplossingen bestaan uit optimale deeloplossingen. Dit wordt het *optimaliteitsprincipe* genoemd.

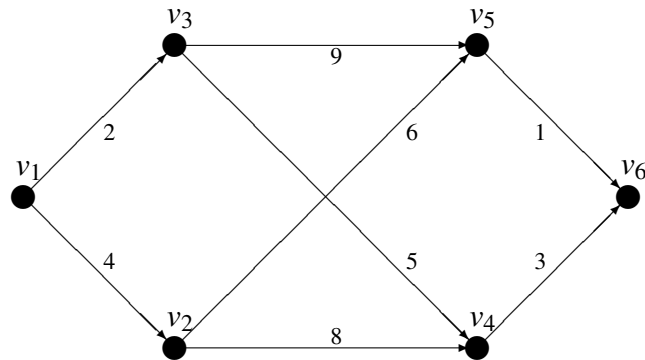
Neem als voorbeeld twee willekeurige punten op de kortste weg tussen een begin- en eindpunt in een netwerk. Deze twee tussenpunten zijn via deze weg zelf ook op de kortste manier met elkaar verbonden. Het lijkt erop dat de totale oplossing, de kortste weg, bestaat uit gecombineerde optimale deeloplossingen. Toch zit er een addertje onder het gras. De kortste weg van v_a naar v_c hoeft geen gebruik te maken van de kortste weg van v_a naar v_b en de kortste weg van v_b naar v_c . Er kan altijd nog andere kortere weg bestaan tussen v_a naar v_c , rechtstreeks of via andere knopen.



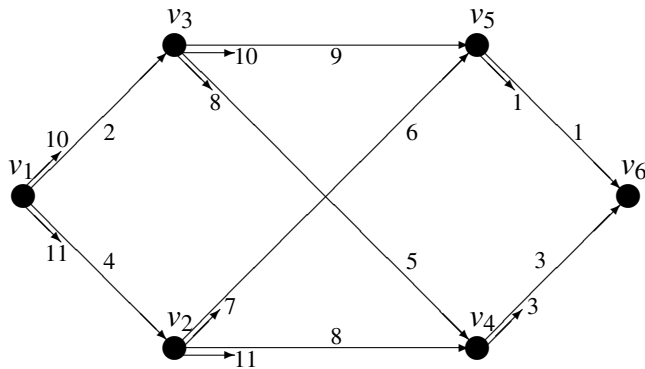
Bij het dynamisch programmeren worden verschillende combinaties van deeloplossingen over grotere probleemdelen berekend. Om het aantal combinaties te beperken, mogen alleen de meest optimale oplossingen blijven bestaan. Pas bij de verwerking van het gehele probleem blijkt welke combinatie deeloplossingen de meest optimale oplossing is.

4.2.1 De kortste afstand

Het dynamisch programmeren kunnen wij het beste demonstreren aan de hand van een eenvoudig voorbeeld. Wij willen een wandeling maken vanaf startpunt v_{start} naar eindpunt v_{eind} in een netwerk. Het is niet de bedoeling dat wij terecht komen in een cykel. Daarom maken wij ons netwerk voor met een gerichte *acyclische graaf*, een graaf zonder cyclen. De gewichtswaarde van een tak stelt de afstand voor. De volgende graaf voldoet aan de gestelde voorwaarden, Het is de bedoeling de kortste wandeling tussen de startknoop v_1 en de eindknoop v_6 te bepalen:



Het dynamisch programmeren gedraagt zich als een gemeenteambtenaar die op elk kruispunt richtingsborden plaatst. Elk bord geeft de kortste afstand naar $v_{eind} = v_6$ in die richting. De gemeenteambtenaar begint achteraan bij v_{eind} en werkt vanaf daar naar voren.



Indien wij de werkzaamheden van de gemeenteambtenaar in een algoritme vertalen, dan herkennen wij het bottom-up principe: De berekening van het kleinste deelprobleem wordt het eerst gedaan, de kortste afstanden van de knopen v_5 en v_4 naar v_6 . De gemeenteambtenaar kan nog geen gebruik maken van tussenresultaten. Pas bij knopen v_3 en v_2 kan hij gebruik maken van de berekende resultaten van v_5 en v_4 . Het *optimaliteitsprincipe* komt pas in knoop v_1 tot zijn recht. De kortste afstand in knoop v_1 is de kortste afstand via v_3 of v_2 . De minder optimale oplossingen in knopen v_3 en v_2 doen niet meer mee. Als wij de geplaatste richtingsborden volgen dan blijkt de kortste weg met de totale afstand 10 vanaf v_1 via v_3 en v_4 naar v_6 te lopen.

Wij kunnen het algoritme van de gemeenteambtenaar voor berekening van de kortste afstand $ka(v, v_{eind})$ in een gerichte graaf zonder cykels tussen een willekeurige knoop v en een verder gelegen knoop v_{eind} opstellen in de volgende recurrente betrekking:

$$ka(v_{eind}, v_{eind}) = 0$$

$$ka(v, v_{eind}) = \forall v_{opv} \in opv(v) : \min(da(v, v_{opv}) + ka(v_{opv}, v_{eind}))$$

De opvolgers van v zijn rechtstreeks uit de rijvector van v in de adjacentiematrix te lezen. De directe afstand $da(v, v_{opv})$ kunnen wij in de $DA[v, v_{opv}]$ matrix lezen (zie paragraaf 3.6). Hoewel deze recurrente betrekking rechtstreeks in een algoritme is te vertalen, is het nog niet erg efficiënt. Het algoritme rekent bij elke recursieve aanroep de reeds berekende waarden $ka(v_{opv}, v_{eind})$ opnieuw uit:

1. $ka(v, v_{eind}) : afstand$
2. **if** $v = v_{eind}$ **then return**(0)
3. **else return** (**forall** $v_{opv} \in opv(v) : \min(DA[v, v_{opv}] + ka(v_{opv}, v_{eind}))$)

In een iteratieve uitvoering van dit algoritme zal men de tussenresultaten $ka(v_{opv}, v_{eind})$ niet herberekenen, maar opslaan in de kolomvector v_{eind} van de kortste afstandsmatrix $KA[v_{opv}, v_{eind}]$. Het tijdgedrag van dit algoritme wordt bepaald door de enkele lus met een herhalingsfrequentie die evenredig is met het aantal knopen n . De rijvectoren van de adjacentiematrix worden doorzocht op opvolgers n . De totale tijdcomplexiteit komt op $O(n^2)$.

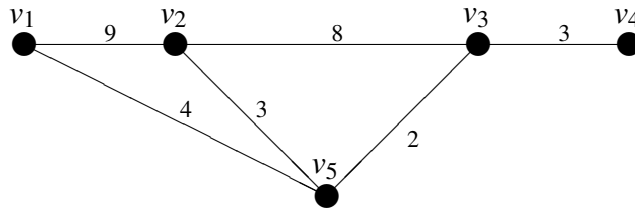
4.2.2 De kortste afstandsvector (Dijkstra)

Soms willen wij de kortste afstand berekenen van een bepaalde knoop v_{start} naar alle andere knopen. Dit komt overeen met het berekenen van de rijvector v_{start} in de KA -matrix. Met het *algoritme van Dijkstra*, wordt de kortste afstand tussen een knoop v_{start} en alle andere knopen in gerichte- of ongerichte graaf berekend, ongeacht of deze graaf cyclisch bevat of niet.

Het algoritme van Dijkstra maakt gebruik van een deelgraaf W . Deze deelgraaf W bestaat in eerste instantie uit de knoop v_{start} zelf. Daarna wordt van de direct verbonden knopen de meest dichtbijgelegen knoop v aan W in de deelgraaf W opgenomen. Van elke opgenomen knoop in deze deelgraaf W wordt de kortste afstand naar v_{start} opgenomen in de vector RA . De deelgraaf W bevat dus alleen knopen v waarvan de kortste afstand van v_{start} tot v bekend is. Knopen v_n uit V mogen alleen in W opgenomen worden als zij direct verbonden zijn met een van de knopen in W , waarbij bovendien geldt dat zij de kleinste afstand hebben tot v_{start} . Het lijkt of de verzameling W zich gedraagt als water dat zich verspreidt over de oppervlakte van de verzameling knopen V van de graaf G .

1. $dijkstra(G(V, E) : graaf, v_{start} \in V,)$
2. **forall** $v \in V : RA[v] \leftarrow DA[v_{start}, v]$
3. $W \leftarrow W \cup \{v_{start}\} \quad V \leftarrow V \setminus \{v_{start}\}$
4. **while** $V \neq \{\}$:
5. **forall** $v \in V$:
6. **forsome** $w \in V \wedge RA[w] = \min(RA[v]) : W \leftarrow W \cup \{w\}; V \leftarrow V \setminus \{w\}$
7. **forall** $v \in V : RA[v] \leftarrow \min(DA[v, w] + RA[w])$

De directe afstanden tussen de knopen van de graaf zijn gegeven in de matrix DA . Het resultaat komt in de vector RA . Wij zullen het algoritme van Dijkstra op de volgende graaf uitvoeren:



Het algoritme van *Dijkstra* maakt de volgende stappen ($v_{start} = v_2$) in de **while**-lus:

i	V_i	W_i	RA_i
1	$\{v_1, v_3, v_4, v_5\}$	$\{v_2\}$	$[9, 0, 8, \infty, 3]$
2	$\{v_1, v_3, v_4\}$	$\{v_2, v_5\}$	$[7, 0, 5, \infty, 3]$
3	$\{v_1, v_4\}$	$\{v_2, v_5, v_3\}$	$[7, 0, 5, 8, 3]$
4	$\{v_4\}$	$\{v_2, v_5, v_3, v_1\}$	$[7, 0, 5, 8, 3]$
5	$\{\}$	$\{v_2, v_5, v_3, v_1, v_4\}$	$[7, 0, 5, 8, 3]$

De tijdcomplexiteit wordt bepaald door een nesting van een **while**-lus (regel 4) en een **for**-lus (regel 5 en 6). Regel 6 bevat een **forsome** instructie. Hiermee wordt een knoop v_n gevonden die de kortste afstand heeft tot v_{start} en direct verbonden is met een knoop in de deelgraaf W . Het gehele *algoritme van Dijkstra* komt op een tijdcomplexiteit van $O(n^2)$.

4.2.3 De bereikbaarheidsmatrix (Warshall)

In paragraaf 3.5 hebben wij een $O(n^4)$ algoritme behandeld. Met dynamisch programmeren is het mogelijk een betere complexiteit te verkrijgen. *S. Warshall* gaf in 1962 een dynamisch programmeeralgoritme met een $O(n^3)$ complexiteit voor de berekening van de bereikbaarheidsmatrix A^∞ . Hij ging uit van het volgende optimaliteitsprincipe:

Als er een verbinding is van knoop x naar knoop y en er is een verbinding tussen y naar z , dan is er een verbinding tussen x en z .

1. **for** $y \leftarrow 1 \dots n$
2. **for** $x \leftarrow 1 \dots n$
3. **if** $A[x, y] = \text{true}$ **then**
4. **for** $z \leftarrow 1 \dots n$
5. **if** $A[y, z] = \text{true}$ **then** $A[x, z] \leftarrow \text{true}$

Het algoritme gaat uit van een normale binaire adjacentiematrix A , waarin de knopen genummerd zijn van $1 \dots n$. Het algoritme transformeert de binaire adjacentiematrix A in de binaire bereikbaarheidsmatrix A^∞ .

4.2.4 De kortste afstandsmatrix (Warshall)

Voor de berekening van de kortste afstand tussen alle knopen in een gerichte graaf kunnen wij, net zoals bij het bereikbaarheidsalgoritme van Warshall, gebruik maken van het optimaliteitsprincipe. Warshall ging in eerste instantie uit van het volgende principe:

De kortste afstand van knoop x naar knoop z waarbij gebruik gemaakt wordt van directe opvolgers y van x , is de kortste afstand van knoop x naar knoop z of de kortste afstand van x naar zijn directe opvolger y plus de kortste afstand van y naar z .

Deze formulering van het optimaliteitsprincipe kan worden afgezwakt door de directe opvolgers y te vervangen door tussengelegen knopen $1 \dots y < x$ waarbij de berekeningen van de tussenresultaten y plaats vinden in volgorde van de nummering van de knopen:

De kortste weg van knoop x naar knoop z waarbij gebruik gemaakt kan worden van de tussengelegen knopen $y : 1 \leq y < x$, is de kortste weg van knoop x naar knoop z of de kortste afstand van x naar y plus de kortste afstand van y naar z .

Het optimaliteitsprincipe in zijn laatste vorm werd door *S. Warshall* geformuleerd. Het resulterende algoritme heeft een $O(n^3)$ complexiteit voor de berekening van de kortste afstandsmatrix KA uit de directe afstandsmatrix DA .

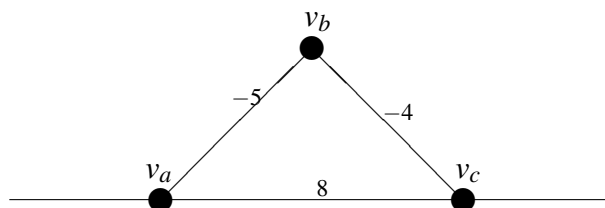
```

1. for  $y \leftarrow 1 \dots n$ 
2.   for  $x \leftarrow 1 \dots n$ 
3.     if  $DA[x, y] = \infty$  then
4.       for  $z \leftarrow 1 \dots n$ 
5.         if  $DA[x, y] > DA[x, z] + DA[z, y]$  then
6.            $DA[x, y] \leftarrow DA[x, z] + DA[z, y]$ 

```

Opmerkingen:

- Het algoritme van Warshall is geschikt om de kortste afstanden voor alle knopen in gerichte en ongerichte grafen met of zonder cyclen te bepalen. Het optimaliteitsprincipe gaat echter niet op bij cyclen met een negatieve som van de takgewichten. Bij het bepalen van de kortste afstand in een graaf gaat een *negatieve cykel* zich gedragen als een soort zwart gat:



Negatieve cyclen kunnen vermeden worden bij fysische afstanden omdat die altijd positief zijn ten opzichte van een bepaald referentiepunt. Sommige fysische grootheden hebben een *conservatief vectorveld*. Deze grootheden hebben altijd cyclen waarvan de taksom gelijk aan nul is. Er zijn echter situaties en problemen te verzinnen waarin negatieve cyclen niet te vermijden zijn. Men kan denken aan takgewichten die een winst of een verlies voorstellen. Het algoritme van Warshall is wel geschikt om negatieve cyclen te herkennen. Op het moment dat een negatieve cykel gedetecteerd wordt, wordt de waarde $DA[x, z]$ negatief.

- Als wij niet alleen de kortste afstand willen weten, maar ook de volgorde van de te bewandelen knopen, dan is het algoritme van Warshall eenvoudig uit te breiden.

4.3 Het vertak- en begrensprincipe

Talrijk zijn de problemen die zich als graaf laten voorstellen, zoals relaties, schakelingen, organisatiestructuren en programma's. Een oplossing van het probleem is meestal een bijzondere knoop of pad in deze graaf. Vaak is een oplossing van een probleem pas te vinden na een wandeling door een boom of graaf. In een samenhangende graaf kunnen we de knopen één voor één bezoeken. De volgorde tussen de knopen die we bezoeken wordt bepaald door de structuur van de graaf en de wandeldiscipline. Soms wordt op een simpele manier door de graaf gewandeld, met het risico van wegen in te slaan die bij voorbaat niet bijdragen tot een oplossing. Men zal tijdens het wandelen bepaalde wegen willen uitsluiten. Omdat het wandelen vaak bedoeld is om een oplossing te vinden, noemt men de wandeldisciplines ook wel zoekdisciplines.

Met het *vertak- en begrensprincipe* worden de wandelingen beperkt tot paden die bij kunnen dragen tot de oplossing. Men noemt het begrip vertak- en begrens ook wel *Branch and Bound*. Voordat wij het vertak- en begrensprincipe behandelen geven wij een overzicht van drie belangrijke wandel- of zoekdisciplines in een graaf:

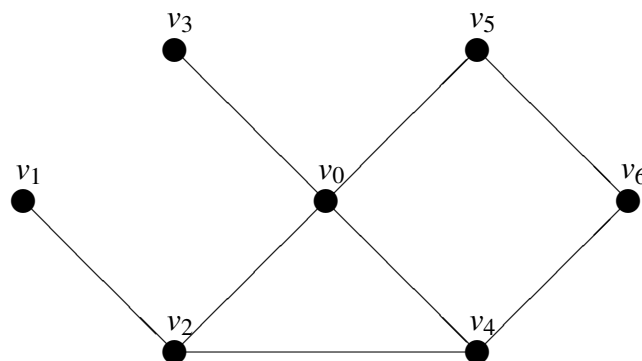
1. Depth First Search (DFS) . Grofweg vertaalt met “eerst zoeken in de diepte”;
2. Breadth First Search (BFS) , “eerst zoeken in de breedte”;
3. Priority First Search (PFS) “eerst zoeken via de wegen met de hoogste prioriteit”. Dit wandelalgoritme is bijzonder geschikt voor het *vertak- en begrensprincipe*.

Wij zullen eerst de eenvoudige wandelalgoritmen behandelen, het *DFS*- en het *BFS*-*algoritme*. Vervolgens behandelen de generalisatie van deze twee zoekdisciplines, het voor het vertak- en begrensprincipe geschikte *PFS*-*algoritme*.

4.3.1 Het DFS-algoritme

Het DFS-algoritme bezoekt een samenhangende graaf in een stervormig pad, er wordt vanuit de beginknoop zeer snel naar de verst verwijderde knopen gesprongen. Men wandelt van de startknoop in een kort aantal stappen naar de randknopen. Zodra men niet verder kan, komt men terug op een vorige knoop. Het lijkt het meest op een browser die steeds verder de WWW-pagina's bezoekt. Uiteindelijk keren wij via herhaald indrukken van de 'backtoets' van de browser weer op beginpagina terug. De terugkeerknopen zijn opgeslagen in een stack. De instructie $stack \downarrow v$ plaatst het element v in de stack. De instructie $v \uparrow stack$ haalt het jongste element uit de stack, waarbij het de naam v krijgt. Om het rondlopen in een cykel te voorkomen, wordt gebruik gemaakt van een binaire vector $bezocht[]$, die aan het begin van het algoritme eerst "false" gemaakt wordt.

1. $DFS(G(V, E) : \text{graaf}, v_0 \in V : \text{knoop})$
2. **forall** $v \in V : \quad bezocht[v] \leftarrow false$
3. $stack \downarrow v_0$
4. **while** $stack \neq []$
5. $v \uparrow stack$
6. **if** $bezocht[v] = false$ **then**
7. $bezocht[v] \leftarrow true$
8. **forall** $w \in \{adjacent(v)\} \wedge bezocht[w] = false : \quad stack \downarrow w$



Als wij met het DFS-algoritme vanuit knoop v_0 in deze graaf gaan wandelen dan bezoeken de volgende knopen tijdens de normale stappen (\rightarrow is een normale stap, \hookrightarrow is een

terugstap naar een bezochte knoop) in de volgorde:

$$v_0 \rightarrow v_5 \rightarrow v_6 \rightarrow v_4 \rightarrow v_2 \rightarrow v_1 (\hookrightarrow v_4) \rightarrow v_3 (\hookrightarrow v_2)$$

Het DFS-algoritme maakt de volgende stappen in de **while**-lus:

i	v_i	$stack_i$	v_{terug}
1	v_0	$[v_2, v_3, v_4, v_5]$	
2	v_5	$[v_2, v_3, v_4, v_6]$	
3	v_6	$[v_2, v_3, v_4, v_4]$	
4	v_4	$[v_2, v_3, v_4, v_2]$	
5	v_2	$[v_2, v_3, v_4, v_1]$	
6	v_1	$[v_2, v_3, v_4]$	v_4
7	v_3	$[v_2]$	v_2

Het DFS-algoritme wordt op elke knoop exact 1 keer aangeroepen. In de n aanroepen worden alle adjacentielijsten doorlopen, hun totale lengte is m (het aantal takken). Hieruit volgt dat het DFS-algoritme een tijdcomplexiteit $O(n + m)$ heeft.

Een toepassing van het DFS-algoritme is het ontdekken van cyclen. Een terugstap bezoekt een reeds bezochte knoop. Zo'n knoop v_{terug} is in een cykel opgenomen. Er zijn geen terugstappen in grafen met een boomstructuur (een boom is per definitie een samenhangende graaf zonder cyclen).

4.3.2 Het BFS-algoritme

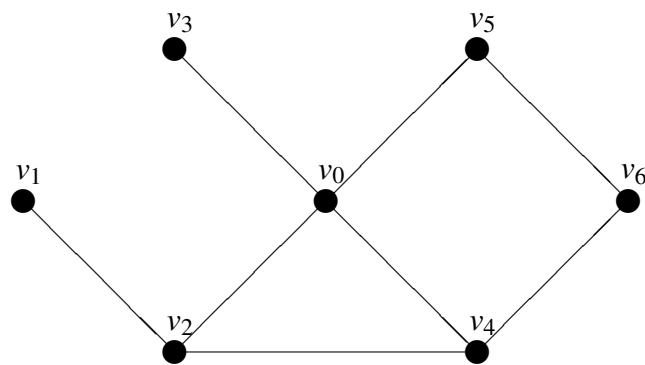
Het *BFS-algoritme* wordt soms het “ui-algoritme” genoemd, omdat het een graaf als een ui in laagjes afpelt. Er is wel een belangrijk verschil met het afpellen van een ui, het BFS-algoritme begint van binnen naar buiten. Het BFS-algoritme maakt gebruik van een *wachtrij* of queue. Als wij in het DFS-algoritme de stack (LIFO) vervangen door een wachtrij (FIFO) hebben wij het BFS-algoritme. Door het gebruik van een wachtrij worden de dichtst bij gelegen knopen het eerst bezocht. De startknoop het eerst, vervolgens de knopen die in een stap bereikbaar zijn, vervolgens de knopen die in twee stappen bereikbaar zijn etc. De graaf wordt, als een ui, van binnen uit schil voor schil bezocht. Het verschil tussen het DFS- en het BFS-algoritme komen we ook in de praktijk tegen. Een schaker die een BFS-algoritme gebruikt, denkt eerst voor alle stukken één zet vooruit. Een schaker die een DFS-algoritme gebruikt, denkt over één stuk vele zetten vooruit. De terugkeerknopen zijn opgeslagen in een queue. De instructie $queue \downarrow v$ plaatst het element v in de queue. De instructie $v \uparrow queue$ haalt het oudste element uit de queue, waarbij het de naam v krijgt.

1. $BFS(G(V, E) : graaf, v_0 \in V : knoop)$
2. **forall** $v \in V : bezocht[v] \leftarrow false$

```

3.  $queue \downarrow v_0$ 
4. while  $queue \neq []$ 
5.    $v \uparrow queue$ 
6.   if  $bezocht[v] = false$  then
7.      $bezocht[v] \leftarrow true$ 
8.     forall  $w \in \{adjacent(v)\} \wedge bezocht[w] = false$  :  $queue \downarrow w$ 

```



Als wij met het BFS-algoritme vanuit knoop v_0 deze graaf gaan bewandelen dan worden de knopen in de volgende volgorde bezocht:

$v_0 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_1 \rightarrow v_6$.

Het BFS-algoritme maakt de volgende stappen in de **while**-lus:

i	v_i	$queue_i$
1	v_0	$[v_2, v_3, v_4, v_5]$
2	v_2	$[v_3, v_4, v_5, v_1]$
3	v_3	$[v_4, v_5, v_1]$
4	v_4	$[v_5, v_1, v_6]$
5	v_5	$[v_1, v_6]$
6	v_1	$[v_6]$
7	v_6	$[]$

Voor de bepaling van de tijdcomplexiteit geldt:

- Elke knoop wordt hoogstens één keer in de queue gestopt;
- Elke knoop wordt er hoogstens één keer uit gehaald;
- Van elke knoop wordt dus ook hoogstens één keer de adjacentielijst doorzocht.

Hieruit volgt dat de tijdcomplexiteit voor het BFS-algoritme gelijk is aan $O(n + m)$.

4.3.3 Het PFS-algoritme

Een derde variant die we veel tegenkomen is het wandelen in een graaf waarbij de stack of de queue vervangen is door een priority-queue. De wandeling wordt volledig bepaald door de prioriteitswaarde van de knopen in de priority-queue. Een priority-queue geeft altijd de knoop met de hoogste prioriteit, ongeacht de volgorde waarin de priority-queue gevuld wordt. Zoals de priority-queue een generalisatie is van de stack en de queue, is het PFS-algoritme een generalisatie van het DFS- en het BFS-algoritme. De instructie $prior_queue \downarrow v$ plaatst het element v in de priority-queue. De instructie $v \uparrow prior_queue$ haalt het element met de hoogste prioriteit uit de priority-queue, waarbij het de naam v krijgt.

1. $PFS(G(V,E) : graaf, v_0 \in V : knoop)$
2. **forall** $v \in G : bezocht[v] \leftarrow false$
3. $prior_queue \downarrow v_0$
4. **while** $prior_queue \neq \langle \rangle :$
5. $v \uparrow prior_queue$
6. **if** $bezocht[v] = false$ **then**
7. $bezocht[v] \leftarrow true$
8. **forall** $w \in \{adjacent(v)\} \wedge bezocht[w] = false : prior_queue \downarrow w$

De tijdcomplexiteit $T(n)$ van het PFS algoritme wordt evenals het DFS- en het BFS-algoritme bepaald door het aantal knopen en takken plus de prioriteit die aan een knoop wordt toegekend. Dit betekent dat als de prioriteitsbepaling in $O(1)$ begrensd is, de totale complexiteit begrensd is met $O(n + m)$.

Met het PFS-algoritme kunnen wij enkele bekende algoritmen implementeren door de prioriteitsgewichten van de knopen voor elk algoritme te anders te definiëren. Enkele voorbeelden zijn:

- De prioriteit is evenredig met de volgorde van het opnemen in de priority-queue. Indien de gewichtswaarde van de knopen evenredig is met een toenemende volgorde, kunnen wij het PFS-algoritme beschouwen als een DFS-algoritme.
- De prioriteit is omgekeerd evenredig met de volgorde van het opnemen in de priority-queue. Indien de gewichtswaarde van de knopen evenredig is met een afnemende volgorde, kunnen wij het PFS-algoritme beschouwen als een BFS-algoritme.
- Indien het prioriteitsgewicht van een knoop omgekeerd evenredig is met de lengte van de tak waarmee zij met een reeds bezochte knoop verbonden is, dan krijgen wij het algoritme van Prim voor de minimum opspannende boom.
- Indien het prioriteitsgewicht van een knoop evenredig is met de afstand ten opzichte van de startknoop, dan krijgen wij het algoritme van Dijkstra voor de bepaling van de kortste afstand.

Bijlage A

Literatuur

- [Sedge84] Sedgewick, R., *Algorithms*, Addison-Wesley, Amsterdam, 1984.
- [Grim89] Grimaldi, R.P., *Discrete and Combinatorial Mathematics*, Addison-Wesley, Amsterdam, 1989.
- [Hor78] Horowitz, E., Sahni, S. , *Fundamentals of Computer Algorithms*, Computer Science Press, Rockville, Maryland, 1978.
- [Sedge96] Sedgewick, R., Flajolet, F., *An Introduction to the Analysis of Algorithms*, Addison-Wesley, Amsterdam, 1996.
- [GKP92] Graham, R.L., Knuth, D.E., Patashnik, O., *Concrete Mathematics*, Addison-Wesley, Amsterdam, 1992.
- [WIL85] Wilson, R.J., *Introduction to Graph Theory*, Longman House, Essex, 1985.
- [WEST96] West, D.B., *Introduction to Graph Theory*, Prentice Hall, New Jersey, 1996.