HO
GENT

Classic Computer Science Problems

Lesnota's

Stijn Lievens

Professionele Bachelor in de Toegepaste Informatica

Inhoudsopgave

Voorwoord

Dit is de cursustekst voor het opleidingsonderdeel “Classic Computer Science Algorithms”, uit het tweede jaar professionele bachelor toegepaste informatica.

Dit opleidingsonderdeel tracht enerzijds om jou wat extra (theoretisch) inzicht bij te brengen in de werking van enkele belangrijke algoritmen en datastructuren. Anderzijds is het de bedoeling om te leren programmeren in Python. De tekst die nu voor je ligt wordt gebruikt in het “theoretische” deel van de cursus. Voor het praktische Python-deel wordt gebruikgemaakt van Dodona (`dodona.ugent.be`) een online leerplatform voor programmeren met ingebouwde automatische test- en feedbackmogelijkheden. Jullie maken eveneens gebruik van de (Engelstalige) officiële Python-documentatie om jezelf de syntax van en het programmeren in Python eigen te maken. We durven hopen dat dit laatste vlot zal verlopen omdat Python als een eenvoudig te leren programmeertaal wordt beschouwd. Bovendien hebben jullie, als tweedejaarsstudent, reeds een volledig jaar ervaring met programmeren in Java en Javascript.

In de volgende paragrafen worden de verschillende hoofdstukken in de cursus kort overlopen.

In het eerste hoofdstuk gebruiken we enkele algoritmen voor zoeken in en sorteren van arrays als aanknopingspunt voor een informele bespreking van tijdscomplexiteit van algoritmen. Het kennen (en eventueel verbeteren) van de tijdscomplexiteit van een algoritme is zeer belangrijk als je wenst dat je algoritme nog steeds performant werkt ook wanneer de invoer “groot” wordt, bv. wanneer de te sorteren array zeer lang wordt.

In het tweede hoofdstuk introduceren we een eerste dynamische datastructuur, nl. een gelinkte lijst. Het verschil met een “gewone” array is dat de verschillende elementen hier geen opeenvolgende plaatsen in het hoofdge-

heugen innemen, maar dat er wordt gewerkt met referenties (*pointers*) om naar het volgende element in de lijst te wijzen. We tonen hoe zo'n gelinkte lijst kan geïmplementeerd worden. Als toepassing van een gelinkte lijst wordt een stapel geïmplementeerd. Dit is een zeer eenvoudige datastructuur maar met krachtige toepassingen zoals het controleren van haakjes en het evalueren van uitdrukkingen in postfix notatie.

In het derde hoofdstuk bekijken we hashtabellen en tonen aan hoe gelinkte lijsten kunnen aangewend worden om deze te realiseren. Hashtabellen zijn een zeer krachtige datastructuur en kunnen gebruikt worden om dictionary of Map-datastructuren mee te implementeren. Dictionaries zijn een zeer vaak gebruikte datastructuur in Python.

In hoofdstuk 4 bestuderen we het concept van een "boom". We bekijken verschillende types van bomen, bv. gewortelde bomen en binaire (zoek)bomen, en bekijken hun toepassingen en implementatie. Deze implementatie gebeurt opnieuw vaak aan de hand van een dynamische datastructuur, maar nu zijn er typische meerdere wijzers (*pointers*) per top.

Hoofdstuk 5 veralgemeent in zekere zin het concept van een boom naar dat van een graaf. Grafen hebben een zeer ruim toepassingsgebied aangezien het in essentie gaat over het aangeven van relaties tussen verschillende objecten. Eens men een bepaald probleem als een graaf heeft gemodelleerd dan komen sommige vragen komen vaak terug, zoals "welke andere objecten kunnen we bereiken vanuit dit object? "of "wat is de kortste weg tussen dit object en de andere objecten?"of "kunnen we deze objecten zodanig ordenen dat we enkel maar vooruit gaan?". We tonen aan hoe deze vragen kunnen beantwoord worden a.d.h.v. enkele klassieke algoritmen zoals het algoritme van Dijkstra. In de marge van hoofdstuk 5 zien we ook dat er fundamentale verschillen zijn in de moeilijkheidsgraad van verschillende soorten van problemen. Dit is het onderwerp van de appendix m.b.t. complexiteitstheorie.

In hoofdstuk 6 bekijken we hoe je "zoeken" kan aanpakken wanneer het aantal toestanden (objecten) te groot wordt om in het hoofdgeheugen op te slaan. Je zal veel gelijkenissen ontdekken met hetgeen in hoofdstuk 5 werd gezien, maar toch zijn er enkele belangrijke verschillen. We bestuderen het A*-algoritme waar je een vuistregel (een heuristiek) kan gebruiken om het zoeken efficiënter te laten verlopen terwijl je onder bepaalde omstandigheden nog steeds kan garanderen dat de gevonden oplossing optimaal is.

In hoofdstuk 7 wordt het zoekproces bemoeilijkt door het feit dat er nu een tegenstander is die het spel ook wil winnen. We bekijken het minimax-algoritme dat de optimale spelstrategie aangeeft wanneer de tegenstander perfect rationaal is. Dit algoritme is echter veel te traag om in praktische spellen gebruikt te worden. Een cruciaal inzicht is echter dat bepaalde delen van de spelboom kunnen overgeslaan worden zonder het finale antwoord te wijzigen. Dit resulteert in een algoritme dat α-β-snoeien heet.

Hoofdstuk 8, tenslotte, geeft een korte introductie tot machinaal leren. De drie verschillende types van machinaal leren worden besproken. Voor ongesuperviseerd leren bekijken we één algoritme voor clustering, nl. het K-gemiddelden algoritme, in meer detail.

Zoals je waarschijnlijk al hebt gemerkt uit de voorgaande opsomming is het opleidingsonderdeel Classic Computer Science Algorithms relatief wiskundig en theoretisch. In deze tekst wordt de nadruk echter niet gelegd op de wiskundige bewijzen. Enkel wanneer deze kort zijn en helpen om de eigenschap beter te begrijpen (en te onthouden) worden deze vermeld. Wanneer de bewijzen een meer wiskundige of theoretische inslag hebben worden ze in deze grootte afgedrukt. Het belangrijkste is echter dat je de eigenschappen kan *toepassen*. Hiertoe zijn een heel aantal oefeningen verwerkt in deze tekst. In de lessen zal je de kans krijgen om een aantal van deze oefeningen samen met je medestudenten en met hulp van de lesgever op te lossen. Het is *zeer belangrijk* dat je zelf de overige oefeningen en opgaven tracht te maken. Dat is de beste (zelfs enige?) manier om voldoende vertrouwd te geraken met de leerstof.

Om te demonstreren dat de besproken algoritmen ook in de praktijk werken voorzien we een aantal oefeningensessies op PC. Hierbij maken we, zoals reeds hierboven vermeld, gebruik van de online programmeeromgeving Dodona en programmeren een aantal algoritmen in Python.

Alvorens af te sluiten wens ik nog eens heel uitdrukkelijk collega Noemie Slaats hartelijk te bedanken. Zij was zo vriendelijk me de broncode van haar curusnota's ter beschikking te stellen. De eerste drie hoofdstukken van deze curustekst zijn zeer sterk gebaseerd op bepaalde delen van de cursus Probleemoplossend Denken I (Slaats, 2019) die tot het academiejaar 2019–2020 werd gegeven in de eerste bachelor toegepaste informatica. Ik bedank eveneens collega's Pieter-Jan Maenhaut en Koen Mertens voor hun suggesties bij de tekst.

Dit is het eerste jaar dat dit opleidingsonderdeel wordt gegeven, en bijgevolg is deze cursustekst splinternieuw. Ik heb getracht de tekst met veel zorg te schrijven maar ongetwijfeld zijn er nog veel zaken voor verbetering vatbaar. Ik doe dan ook een warme oproep aan jullie, de studenten, om opmerkingen of aanmerkingen ter verbetering (bv. tikfouten, spellingsfouten, onduidelijkheden, enz.) aan te brengen. Ik zal jullie dankbaar zijn maar nog belangrijker: je opvolgers zullen je er eveneens dankbaar voor zijn. Voor het aangeven en bijhouden van errata zal er op Chamilo een forum worden aangemaakt.

Tenslotte wens ik jullie veel lees-, leer- en oefenplezier.

Stijn Lievens, september 2021

HOOFDSTUK 1

Zoeken en Sorteren

Zoeken in en sorteren van arrays zijn fundamentele algoritmen binnen de informatica. We starten met een bespreking van het meest eenvoudige zoekalgoritme, nl. LINEAIR ZOEKEN en zien vervolgens hoe het zoekproces in een gesorteerde array significant sneller kan verlopen m.b.v. BINAIR ZOEKEN. Hieruit vloeit de natuurlijke vraag voort hoe men een array kan sorteren. We zien een tweetal "eenvoudige" sorteeralgoritmen, nl. SORTEREN DOOR SELECTIE en SORTEREN DOOR TUSSENVOEGEN die allebei een kwadratische uitvoeringstijd hebben. Wanneer de te sorteren rij lang wordt zijn deze algoritmen niet langer praktisch. MERGESORT is een elegant sorteeralgoritme met een uitvoeringstijd van de orde $\mathcal{O}(n \lg n)$. Doorheen het hoofdstuk geven we eveneens een informele bespreking van de asymptotische uitvoeringstijden van de verschillende algoritmen.

1.1 Zoeken in een Array

Een zoekalgoritme wordt gebruikt om een bepaald element te zoeken in een gegeven array, m.a.w. om de positie van dat element in de array te bepalen. Indien de array bij aanvang van het zoekproces reeds gesorteerd is, zal dit invloed hebben op de te gebruiken methode. In een ongesorteerde rij kan men minder efficiënt zoeken dan in een gesorteerde rij.

Een bepaald woord opzoeken in een woordenboek gaat bijvoorbeeld snel en efficiënt aangezien een woordenboek een gesorteerde lijst van woorden is. Hetzelfde woord opzoeken in een willekeurig geordende lijst van woorden zal niet zo vlot verlopen.

Van een array veronderstellen we dat die de mogelijkheid heeft om elementen op willekeurige plaatsen in constante tijd op te halen. Dit wil zeggen dat we veronderstellen dat het even snel gaat om het voorste, het laatste of een element in het midden van de array op te halen. We bekijken nu twee verschillende algoritmen om elementen op te zoeken in een array.

1.1.1 Sequentieel Zoeken

Veronderstel dat je programma een lange array van namen bijhoudt, bijvoorbeeld de namen van 5000 studenten, waarbij de namen niet gesorteerd zijn. Als je wil weten of je eigen naam in de array voorkomt, dan is de meest voor de hand liggende manier om daarachter te komen het één voor één overlopen van alle namen uit de lijst (van voor naar achter) en kijken of jouw naam voorkomt. Deze manier van werken heet LINEAIR ZOEKEN of SEQUENTIEEL ZOEKEN.

Lineair zoeken is gemakkelijk te implementeren. In de pseudocode in Algoritme 1.1 gaan we op zoek naar één specifiek item *zoekItem* in een array van items. De elementen van de array worden één voor één vergeleken met de op te sporen waarde *zoekItem*. Het algoritme stopt als er een element wordt gevonden dat gelijk is aan *zoekItem* of als het einde van de array bereikt is. De methode ZOEKSEQUENTIEEL verwacht twee inputwaarden: de waarde *zoekItem* en een array *rij* van lengte n. Als retourwaarde wordt de index van *zoekItem* teruggegeven, indien *zoekItem* niet voorkomt in de array is de retourwaarde -1 . Meer specifiek vindt dit algoritme steeds het *eerste* voorkomen van de gezochte waarde *zoekItem* in de array.

Opmerking 1.1 "Zoeken" is een proces dat onafhankelijk is van het type van de elementen in de array. In deze code wordt er enkel verondersteld dat we twee elementen met elkaar kunnen vergelijken om te beslissen of ze al dan niet gelijk zijn aan elkaar. ∎

Opmerking 1.2 (Assignatie en vergelijking) In de pseudocode gebruiken we

$$a \leftarrow b$$

om de waarde van b toe te kennen aan a, m.a.w. het symbool $\leftarrow$ wordt gebruikt voor assignatie. Vergelijken gebeurt met

$$a = b$$

Algoritme 1.1 Sequentieel zoeken in een array.

Invoer Een item zoekItem dat moet gevonden worden, een array van items genaamd rij met lengte n.

Uitvoer de index van het eerste element in rij dat gelijk is aan zoekItem wordt teruggegeven of -1 indien zoekItem niet voorkomt in rij.

```
1:  function ZOEKSEQUENTIEEL(zoekItem, rij)
2:      i ← 0                                          ▷ overloopt de posities
3:      while i < n and rij[i] ≠ zoekItem do
4:          i ← i + 1
5:      end while
6:      if i = n then                                  ▷ niet gevonden
7:          index ← −1
8:      else
9:          index ← i                                  ▷ gevonden
10:     end if
11:     return index
12: end function
```

waarvan het resultaat een Booleanse waarde is. In het bijzonder is deze **true** wanneer a en b aan elkaar gelijk zijn en **false** wanneer dit niet het geval is.∎

1.1.2 Binair Zoeken

Lineair of sequentieel zoeken is eenvoudig te implementeren en werkt uiteraard ook als de array waarin wordt gezocht reeds gesorteerd is. Intuïtief is het echter duidelijk dat er een "betere" manier moet bestaan om te zoeken in een gesorteerde array.

Voorbeeld 1.3 (Woordenboek) Wanneer mensen een woord willen opzoeken in een woordenboek dan slaat men het boek (ongeveer) in het midden open, kijkt of men te ver is of niet en bladert terug of verder. Na een paar pogingen heeft men de juiste bladzijde voor zich liggen waarop het gezochte woord hopelijk voorkomt. Het probleem van het zoeken in een woordenboek van honderden bladzijden is al heel snel gereduceerd tot het veel kleinere probleem van het zoeken op één of twee bladzijden. Deze methode werkt natuurlijk alleen dankzij het feit dat de woorden in een woordenboek gesorteerd zijn. ∎

BINAIR ZOEKEN gaat op dezelfde manier tewerk als het zoeken in een woordenboek, maar dan heel systematisch. In een gesorteerde array wordt het

middelste item bekeken. Als dit item kleiner is dan het gezochte item, wordt er verder gezocht in de rechterhelft van de array. In het andere geval wordt er verder gezocht in de linkerhelft van de array. Essentieel voor de performantie van binair zoeken is dat de array bij elke stap in (bijna) twee gelijke delen wordt verdeeld. Op het einde van dit proces blijft er precies één element over waarna de iteratie stopt. Door het vergelijken van dit ene element met het gezochte element weten we of het element voorkomt in de array of niet.

Het idee dat het zoekproces in de oorspronkelijke array wordt herleid naar een zoekproces in een array half zo groot, laat toe dit probleem op te lossen met recursie, maar uiteraard kan het algoritme ook iteratief worden geïmplementeerd. De twee verschillende mogelijke oplossingsmethodes, nl. a.d.h.v. iteratie en a.d.h.v. recursie, worden allebei besproken.

Opmerking 1.4 Bij lineair zoeken was de enige vereiste aan het type van de elementen dat er kan beslist worden of ze gelijk zijn of niet. Bij binair zoeken is de array reeds gesorteerd (van klein naar groot). Dit betekent dat we voor twee elementen niet enkel moeten kunnen beslissen of ze gelijk zijn of niet, we moeten ook kunnen nagaan welke van de twee de grootste is als ze niet gelijk zijn. In de pseudocode schrijven we dit eenvoudigweg als:

$$x < y.$$ ∎

De in- en uitvoer van het algoritme voor binair zoeken is hetzelfde als de in- en uitvoer van het algoritme voor sequentieel zoeken.

Opmerking 1.5 Bij de implementatie van binair zullen we wel wat aandacht besteden aan het feit dat de index van het *eerste* voorkomen in de array wordt geretourneerd zodat we steeds dezelfde retourwaarde krijgen als bij lineair zoeken. Er zijn echter ook implementaties van binair zoeken mogelijk die deze garantie niet geven. ∎

Om bij te houden welk deel van de array onderzocht moet worden, introduceren we de gehele variabelen l voor de positie links en r voor de positie rechts. Als *zoekItem* in de array voorkomt, dan komt *zoekItem* voor onder de elementen

$$rij[l], rij[l+1], \ldots, rij[r].$$

Bij aanvang moet de volledige array onderzocht worden, dit betekent dat de variabelen l en r geïnitialiseerd moeten worden als volgt: $l \leftarrow 0$ en $r \leftarrow$

$n - 1$, waarbij n de lengte van de array voorstelt. Van het te onderzoeken deel wordt telkens het midden bepaald, de index van dit midden wordt bijgehouden in de gehele variabele m.

Wanneer het aantal elementen in een rij oneven is, dan is het duidelijk wat het middelste element is, bv. voor de elementen

$$rij[5], rij[6], rij[7], rij[8], rij[9].$$

is het middelste element duidelijk $rij[7]$. De index 7 wordt berekend als het gemiddelde van $l = 5$ en $r = 9$, want $7 = (5 + 9)/2$. Wanneer het aantal elementen in een rij even is, bv.

$$rij[5], rij[6], rij[7], rij[8], rij[9], rij[10]$$

dan zijn er twee "middelste" elementen. In het voorbeeld zijn dit $rij[7]$ en $rij[8]$. We kiezen er in dit geval voor om het kleinste van deze twee elementen als het middelste element aan te duiden. In dit geval is

$$7 = \lfloor \frac{5 + 10}{2} \rfloor.$$

Bijgevolgt kunnen we m in beide gevallen als volgt bepalen

$$m = \left\lfloor \frac{l + r}{2} \right\rfloor.$$

Door de manier waarop m werd gedefinieerd geldt steeds dat

$$l \leq m < r.$$

Het kan m.a.w. voorkomen dat m gelijk is aan l, maar m is steeds strikt kleiner dan r, wanneer $l < r$.

In Algoritme 1.2 vindt men de iteratieve implementatie van het algoritme voor binair zoeken.

Het binair zoekalgoritme kan ook recursief geïmplementeerd worden. Een recursief algoritme bevat steeds een selectiestructuur. Deze selectiestructuur beschrijft wat er moet gebeuren in het *basisgeval* en wat er moet gebeuren in de *recursieve* stap.

- Basisstap: de te onderzoeken array bevat slechts één element.
  Dit element wordt vergeleken met de inputwaarde *zoekItem*.

Algoritme 1.2 Binair zoeken in een array geïmplementeerd m.b.v. iteratie.

Invoer Een item zoekItem dat moet gevonden worden, een *gesorteerde* array genaamd rij van items van lengte n.

Uitvoer de index van het eerste element in rij dat gelijk is aan zoekItem wordt teruggegeven of -1 indien zoekItem niet voorkomt in rij.

```
1:  function ZOEKBINAIR(zoekItem, rij)
2:      l ← 0
3:      r ← n − 1
4:      while l ≠ r do          ▷ herhalen totdat slechts één element overblijft
5:          m ← ⌊(l+r)/2⌋
6:          if rij[m] < zoekItem then
7:              l ← m + 1                          ▷ in de rechterhelft zoeken
8:          else
9:              r ← m                              ▷ in de linkerhelft zoeken
10:         end if
11:     end while                                  ▷ l = r
12:     if rij[l] = zoekItem then
13:         index ← l                              ▷ gevonden
14:     else
15:         index ← −1                             ▷ niet gevonden
16:     end if
17:     return index
18: end function
```

- Recursieve stap: de rij wordt gehalveerd.
  De functie wordt opnieuw aangeroepen op een deelrij half zo groot als de oorspronkelijke array.

In Algoritme 1.3 vindt men de recursieve implementatie. Er is een "driver"-functie met de naam ZOEKBINAIR. Dit is de functie die wordt opgeroepen wanneer men een array wenst te sorteren. De functie ZOEKRECURSIEF wordt opgeroepen door deze driver-functie en is de eigenlijke recursieve functie die het zoekproces implementeert.

Opmerking 1.6 (Het eerste voorkomen) De gegeven implementatie zorgt ervoor dat het algoritme steeds het eerste voorkomen van een element vindt in de array. Deze garantie is niet langer geldig wanneer men de iteratie of recursie voortijdig zou afbreken op het moment dat men het gezochte element (voor de eerste maal) ontmoet. ∎

Algoritme 1.3 Recursieve implementatie van binair zoeken.

Invoer Een item zoekItem dat moet gevonden worden, een *gesorteerde* array genaamd rij van items van lengte n.

Uitvoer de index van het eerste element in rij dat gelijk is aan zoekItem wordt teruggegeven of -1 indien zoekItem niet voorkomt in rij.

```
1: function ZOEKBINAIR(zoekItem, rij)
2:     return ZOEKRECURSIEF(zoekItem, rij, 0, n − 1)
3: end function
```

Invoer Een item zoekItem dat moet gevonden worden, een *gesorteerde* array genaamd rij van items van lengte n, twee natuurlijke getallen l en r die het gedeelte van de array aangeven waarin gezocht moet worden.

Uitvoer de index van het eerste element in rij dat gelijk is aan zoekItem wordt teruggegeven indien het element voorkomt tussen rij$[l]$, rij$[l+1], \ldots,$ rij$[r]$. De teruggeven index ligt dan tussen l en r. Er wordt -1 teruggegeven indien zoekItem niet voorkomt tussen de elementen rij$[l]$, rij$[l+1], \ldots,$ rij$[r]$.

```
4: function ZOEKRECURSIEF(zoekItem, rij, l, r)
5:     if l = r then                          ▷ basisstap, rij van lengte 1
6:         if zoekItem = rij[l] then
7:             return l
8:         else
9:             return −1
10:        end if
11:    else                                   ▷ inductiestap
12:        m ← ⌊(l+r)/2⌋
13:        if rij[m] < zoekItem then
14:            return ZOEKRECURSIEF(zoekItem, rij, m + 1, r)   ▷ zoek rechts
15:        else
16:            return ZOEKRECURSIEF(zoekItem, rij, l, m)       ▷ zoek links
17:        end if
18:    end if
19: end function
```

1.1.3 Sequentieel vs. Binair Zoeken

Het algoritme voor binair zoeken is ingewikkelder om te implementeren dan het algoritme voor sequentieel zoeken. We hopen dat het algoritme voor binair zoeken dan ook "beter" is dan het algoritme voor sequentieel zoeken. Maar wat bedoelen we daar mee?

Wanneer men algoritmen gaat analyseren dan is men doorgaans geïnteresseerd in de volgende twee zaken:

1. De tijd die het algoritme nodig heeft wanneer het wordt uitgevoerd.
2. De hoeveelheid hoofdgeheugen (RAM) die het algoritme nodig heeft bij uitvoering.

In deze cursus concentreren we ons voornamelijk op het analyseren van de uitvoeringstijd. Hierbij is vooral van belang om op te merken dat het niet zeer zinvol is om de uitvoeringstijd van een algoritme exact te (trachten) bepalen. Immers,

- Verschillende computers werken op verschillende snelheden.
- Ook op dezelfde computer zal in het algemeen het uitvoeren van hetzelfde algoritme op dezelfde input (lichtjes) verschillende uitvoeringstijden opleveren, afhankelijk van welke andere processen er op dat moment nog draaien op de computer.
- Door het gebruik van steeds krachtigere programmeertalen is het soms moeilijk om te begrijpen welke en hoeveel instructies er precies zullen worden uitgevoerd.

Om al deze redenenen beperkt men zich bij het analyseren van de uitvoeringstijd van algoritmen tot een ASYMPTOTISCHE ANALYSE van de uitvoeringstijd. Dit betekent dat men zich afvraagt welk gedrag de uitvoeringstijd vertoont voor "grote" waarden van de input. Typisch gedrag dat men zou kunnen zien is:

- Wanneer de invoer dubbel zo groot wordt, dan duurt het algoritme dubbel zo lang. De uitvoeringstijd gedraagt zich m.a.w. als een lineaire functie: $T(n) = n$.

- Wanneer de invoer dubbel zo groot wordt, dan duurt het algoritme vier keer zo lang. De uitvoeringstijd gedraagt zich m.a.w. als een kwadratische functie: $T(n) = n^2$.
- Wanneer de invoer met één groter wordt gemaakt dan duurt het algoritme dubbel zo lang: $T(n) = 2^n$. Dit is een exponentiële uitvoeringstijd.
- We kunnen de invoer dubbel zo groot maken en toch duurt het algoritme maar een constante tijd langer: $T(n) = \log(n)$. Dit is een logaritmische uitvoeringstijd.
- ...

Bij de analyse van de zoekalgoritmen zien we dat de asymptotische uitvoeringstijd bepaald wordt door het aantal vergelijkingen dat wordt uitgevoerd.

Aantal vergelijkingen bij sequentieel zoeken

Bij sequentieel zoeken kunnen we ons afvragen hoeveel keer de vergelijking

$$\text{rij}[i] \neq \text{zoekItem}$$

op regel 3 van Algoritme 1.1 wordt uitgevoerd.

In het *beste geval* staat het gezochte element helemaal vooraan de array en is er één zo'n vergelijking nodig. In het *slechtste geval* behoort het gezochte element niet tot de array. In dit geval wordt het element vergeleken met alle elementen van de array alvorens te besluiten dat het niet tot de array behoort. Dit zijn m.a.w. n vergelijkingen. Wanneer het element wel tot de array behoort dan zal het *gemiddeld* gezien ergens in het midden staan, zodat we verwachten ongeveer $n/2$ vergelijkingen uit te voeren.

We zien m.a.w. dat het aantal vergelijkingen (en ook de uitvoeringstijd) in het slechtste geval ongeveer verdubbelt wanneer de lengte van de invoerrij wordt verdubbeld. De tijdscomplexiteit is lineair of van de orde n. We noteren dit als

$$T(n) = \mathcal{O}(n).$$

Opmerking 1.7 Bij de complexiteitsanalyse, en meer bepaald bij het gebruik van de "Big-Oh" notatie, wiskundig genoteerd als $\mathcal{O}$, wordt uitgegaan van het slechtst mogelijke geval om de uitvoeringstijd te beschrijven. ∎

Aantal vergelijkingen bij binair zoeken

Bij binair zoeken tellen we hoeveel keer de vergelijking

$$\text{rij}[m] < \text{zoekItem} \tag{1.1}$$

op regel 6 in Algoritme 1.2 wordt uitgevoerd. Voor de eenvoud van de analyse nemen we aan dat de lengte van de rij een macht van twee is, of anders gezegd $n = 2^k$, waarbij k een natuurlijk getal is.

Wanneer $k = 0$, dan is de lengte van de rij gelijk aan 1 en wordt de vergelijking hierboven geen enkele keer uitgevoerd[1].

Wanneer $k = 1$, dan is $l = 0$ en $r = 1$ bij de start van het algoritme en dus wordt $m = 0$. Als de vergelijking (1.1) evalueert naar **true**, dan wordt $l = 1$ en $r = 1$, in het andere geval wordt $l = 0$ en $r = 0$. In beide gevallen zal de hoofdlus van het algoritme stoppen. De vergelijking (1.1) wordt m.a.w. precies één keer uitgevoerd.

Wanneer $k = 2$, dan is $l = 0$ en $r = 3$ bij de start van het algoritme. De eerste maal dat m wordt berekend krijgt deze m.a.w. de waarde $m = 1$. Wanneer de vergelijking (1.1) evalueert naar **true** dan zoeken we verder in de rechterhelft met $l = 2$ en $r = 3$, in het andere geval zoeken we verder in de linkerhelft met $l = 0$ en $r = 1$. In beide gevallen is de lengte van de deelrij waarin wordt gezocht gelijk aan 2. Het aantal bijkomende uitvoeringen van vergelijking (1.1) is dan, zoals reeds vastgesteld, gelijk aan 1. In dit geval zijn er m.a.w. precies twee uitvoeringen van vergelijking (1.1).

Wanneer $k = 3$ (m.a.w. de lengte van de rij is 8) dan zal na de eerste uitvoering van vergelijking (1.1) de lengte van de deelrij gereduceerd worden tot 4. In dit geval zijn er dus $1 + 2 = 3$ uitvoeringen van de vergelijking (1.1).

Op deze manier zien we dat wanneer $n = 2^k$ er precies k uitvoeringen zullen zijn van de vergelijking (1.1). Het verband tussen k en n wordt gegeven door

$$n = 2^k \iff k = \log_2(n).$$

Dit betekent dat de uitvoeringstijd in dit geval logaritmisch groeit:

$$T(n) = \mathcal{O}\big(\log_2(n)\big). \tag{1.2}$$

[1] Er wordt wel een vergelijking uitgevoerd ná de lus om te verifiëren of het element gevonden werd maar deze vergelijking wordt steeds uitgevoerd en is onafhankelijk van de lengte van de rij.

Opmerking 1.8 (Willekeurige *n*) De redenering die we hierboven hebben gemaakt is enkel geldig wanneer $n = 2^k$. Wat als de lengte van de rij niet precies een macht van twee is? In dit geval is het resultaat (1.2) nog steeds geldig. Stel bv. dat je wil zoeken in een rij van lengte 20. In gedachten kan je de rij uitbreiden tot een rij van lengte 32 (i.e. de volgende macht van 2). Dan zie je dat er voor de rij van lengte 20 hoogstens 5 vergelijkingen nodig zijn. ∎

Een experiment om de uitvoeringstijd te meten

We voeren een experiment uit waarbij we de uitvoeringstijd van de algoritmen effectief meten. Dit gaat als volgt:

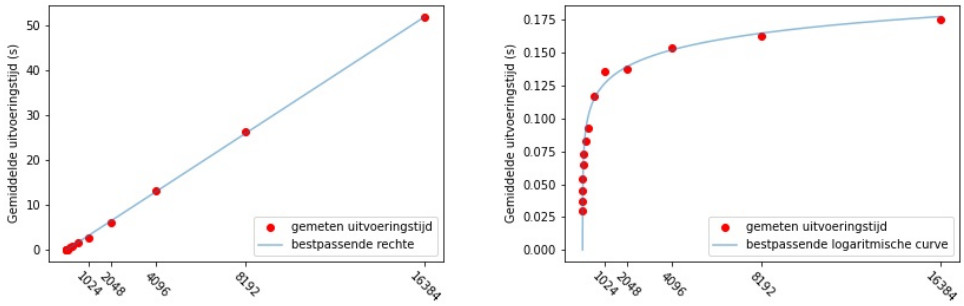
- Laat k de waardes 1 t.e.m. 14 doorlopen.
- Genereer een willekeurige rij van lengte $n = 2^k$ bestaande uit gehele getallen.
- Kies 30 random getallen uit deze rij.
- Voer voor elk van deze 30 random getallen de methode ZOEKSEQUENTIEEL en ZOEKBINAIR 100 000 keer uit en sla de uitvoeringstijd hiervan (apart) op.

Anders gezegd, na het uitvoeren van het experiment beschikken we voor elk algoritme en voor elke gekozen waarde van n over 30 meetpunten die elk het gevolg zijn van het 100 000 keer uitvoeren van een methode-aanroep.

Theoretisch verwachten we voor de methode ZOEKSEQUENTIEEL een lineaire uitvoeringstijd. In Figuur 1.1 tonen we in de linkerfiguur de rechte (door de oorsprong) die het beste[2] past bij de gemiddelde uitvoeringstijd van deze methode.

Voor de methode ZOEKBINAIR verwachten we daarentegen een logaritmische uitvoeringstijd. In Figuur 1.1 tonen we in de rechterfiguur de best passende logaritmische curve. Merk op dat de getallen op de verticale as van beide figuren van een volledig andere grootteorde zijn.

[2] In het opleidingsonderdeel "Data Science & AI" zal duidelijk worden wat hiermee wordt bedoeld.



Figuur 1.1: Uitvoeringstijd van lineair en binair zoeken, resp. in de linker- en rechterfiguur. Merk op dat de getallen op de verticale as van beide figuren van een volledig andere grootteorde zijn.

Merk ook op dat voor kleine waarden van n de uitvoeringstijden van de twee methoden niet significant van elkaar verschillen. Het verschil in gedrag manifesteert zich wanneer de rijen waarin moet gezocht worden iets langer zijn. Dit zie je in Tabel 1.1.

Verder merken we ook op dat de uitvoeringstijden van het algoritme van binair zoeken (voor een bepaalde waarde van n) zeer gelijkaardig zijn, onafhankelijk van het element dat wordt gezocht. Bij lineair zoeken zien we hier een grote variatie optreden, omdat lineair zoeken snel werkt wanneer het element zich vooraan in de rij bevindt en veel trager wanneer het element achter in de rij aanwezig is[3].

1.2 Sorteren van een Array

In de vorige sectie hebben we gezien dat het doorzoeken van een array veel efficiënter verloopt wanneer deze gesorteerd is. In deze sectie bekijken we een drietal algoritmen om arrays te sorteren. Twee daarvan worden gezien als "eenvoudige" sorteeralgoritmen. Het derde algoritme is iets minder voor de hand liggend, maar heeft een betere uitvoeringstijd.

[3]In het experiment werden geen testen gedaan met elementen die niet tot de array behoren. Bij zo'n experiment zullen de tijden voor binair zoeken zeer gelijkaardig blijven, terwijl voor lineair zoeken de tijden steeds aan de "grote" kant zullen liggen.

	Lineair Zoeken			Binair Zoeken		
n	gemiddeld	min	max	gemiddeld	min	max
2	0.023358	0.018782	0.028282	0.029508	0.027960	0.031835
4	0.030451	0.018702	0.044360	0.037046	0.034434	0.040373
8	0.045468	0.018790	0.064200	0.044859	0.041557	0.047688
16	0.070344	0.026154	0.111211	0.053811	0.051925	0.056694
32	0.116735	0.020733	0.226672	0.064633	0.060644	0.076644
64	0.161129	0.032630	0.372710	0.072495	0.068330	0.080041
128	0.439917	0.044428	0.746537	0.082789	0.076771	0.090164
256	0.706593	0.025980	1.438627	0.092677	0.085994	0.100332
512	1.686027	0.045711	3.502422	0.116800	0.101439	0.155817
1024	2.701336	0.211092	6.528466	0.136210	0.106221	0.289722
2048	6.096840	0.076119	13.212105	0.137714	0.121224	0.158811
4096	13.239109	0.221343	26.576251	0.153950	0.134502	0.163726
8192	26.328758	0.377525	51.927700	0.162872	0.149109	0.232349
16384	51.839257	5.057787	106.330041	0.175556	0.163862	0.269275

Tabel 1.1: Enkele uitvoeringstijden van lineair en binair zoeken. Per zoekmethode ziet men het gemiddelde, minimum en maximum van 30 tijdsmetingen. Elke tijdsmeting (aangeduid in seconden) bestaat op zijn beurt uit 100 000 herhalingen van dezelfde methodeoproep.

1.2.1 Sorteren door Selectie

Het basisidee van sorteren door selectie kan eenvoudig samengevat worden:

1. Zoek het grootste element en plaats het achteraan.
2. Sorteer de rest van de array.

We bekijken dit nu in iets meer detail.

In de te sorteren array a gaan we op zoek naar het grootste element. Indien dit element niet achteraan staat in de rij, moet dit element verwisseld worden met het element op de laatste plaats. Het grootste element staat nu achteraan in de rij; dat is de juiste plaats voor dit element. De $(n-1)$ overige elementen van de array moeten nog gesorteerd worden.

Voor de deelrij $a[0], \ldots, a[n-2]$ gaan we op analoge manier tewerk. Het grootste element in de rij wordt bepaald en achteraan geplaatst, dus op de

$(n-2)$-de positie.

Deze werkwijze wordt herhaald op steeds kortere deelrijen. De laatste keer zal de deelrij nog bestaan uit twee elementen. De implementatie wordt gegeven in Algoritme 1.4.

Algoritme 1.4 Sorteren door selectie.

Invoer De array a is gevuld met n elementen.
Uitvoer De array a is gesorteerd.

```
 1: function SELECTIONSORT(a)
 2:     for i = n − 1 ... 1 by −1 do                ▷ achteraan starten
 3:         positie ← i
 4:         max ← a[i]
 5:         for j = i − 1 ... 0 by −1 do            ▷ j doorloopt de deelrij
 6:             if a[j] > max then
 7:                 positie ← j
 8:                 max ← a[j]
 9:             end if
10:         end for
11:         a[positie] ← a[i]          ▷ grootste element wisselen met laatste
12:         a[i] ← max
13:     end for
14: end function
```

Voorbeeld 1.9 We passen het algoritme toe op een kleine voorbeeldrij. Het onderstreepte element is steeds het grootste element van het (mogelijks) nog niet gesorteerde deel van de array. Het deel achter de verticale streep is met zekerheid al correct. In elke iteratie groeit dit deel totdat finaal de volledige rij gesorteerd is.

44	55	12	42	<u>94</u>	18	06	67
44	55	12	42	<u>67</u>	18	06	|94
44	<u>55</u>	12	42	06	18	|67	94
<u>44</u>	18	12	42	06	|55	67	94
06	18	12	<u>42</u>	|44	55	67	94
06	<u>18</u>	12	|42	44	55	67	94
06	<u>12</u>	|18	42	44	55	67	94
06	12	18	42	44	55	67	94

Complexiteitsanalyse

De uitvoeringstijd wordt bepaald door het aantal keer dat de vergelijking

$$a[j] > \max$$

op regel 6 uitgevoerd wordt.

Dit aantal komt overeen met het aantal keer dat de teller j van waarde wijzigt.

	voor i	wordt j	aantal vergelijkingen
	$n-1$	$n-2, n-3, \ldots, 1, 0$	$n-1$
	$n-2$	$n-3, n-4, \ldots, 1, 0$	$n-2$
	$\vdots$		$\vdots$
	1	0	1
totaal			$\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$

De tabel geeft duidelijk weer dat de binnenste lus $\frac{n(n-1)}{2}$ keer wordt herhaald. Hieruit mogen we besluiten dat

$$T(n) = \mathcal{O}(n^2)\ .$$

In de volgende paragrafen tonen we aan dat

$$\sum_{k=1}^{n-1} k = 1 + 2 + \cdots + (n-2) + (n-1) = \frac{n(n-1)}{2}.$$

Opmerking 1.10 (Een belangrijke som) Om de tijdscomplexiteit van sorteren door selectie te bepalen hebben we gebruikgemaakt van de volgende eigenschap:

$$1 + 2 + \cdots + (n-1) + n = \frac{(n+1)n}{2}. \tag{1.3}$$

Dit kan eenvoudig op de volgende manier worden aangetoond, bv. voor $n = 10$. Noteer alle getallen van 1 t.e.m. 10 van klein naar groot en daaronder nog eens van groot naar klein. Neem nu de som van de getallen die onder elkaar staan. Deze som is steeds gelijk aan 11. Hoeveel keer hebben we nu een 11 staan? Dat is 10 keer. De som van alle getallen 11 is m.a.w. 110. Dit is echter *niet* het finale antwoord want we hebben elk getal van 1

t.e.m. 10 nu *twee keer* geteld, één keer in de bovenste rij en nog een keer in de onderste rij. De finale som is bijgevolg $110/2 = 55$. Dit proces wordt geïllustreerd in de onderstaande tabel.

stijgend	1	2	3	4	5	6	7	8	9	10
omgekeerd	10	9	8	7	6	5	4	3	2	1
som	11	11	11	11	11	11	11	11	11	11

Je kan dit proces uiteraard veralgemenen om zo het bewijs te vinden voor formule (1.3). Dit bewijs zou volgens de overlevering door de wiskundige Gauss reeds in de basisschool zijn ontdekt, al is het niet zeker dat dit verhaal ooit effectief gebeurd is[4]. ∎

De uitvoeringstijd van sorteren door selectie wordt bepaald door de waarde van de som:

$$1 + 2 + \cdots + (n-2) + (n-1). \tag{1.4}$$

We tonen nu een manier om de waarde van deze som te bepalen uit vergelijking (1.3). Het eerste wat we ons moeten realiseren is dat we in vergelijking (1.3) ook een andere letter mogen gebruiken dan n in zowel het linker- als rechterlid. Laat ons bv. eens de letter m gebruiken:

$$1 + 2 + \cdots + (m-1) + m = \frac{(m+1)m}{2}. \tag{1.5}$$

Als we nu het linkerlid hiervan vergelijken met uitdrukking (1.4) dan zien we dat deze overeenkomen als we voor m de uitdrukking $(n-1)$ invullen[5]. M.a.w. als we in het rechterlid van (1.5) de letter m vervangen door $(n-1)$ dan krijgen we de uitkomst van (1.4). Als we deze vervanging uitvoeren dan krijgen we:

$$\frac{(m+1)m}{2} \rightsquigarrow \frac{\big((n-1)+1\big)(n-1)}{2} = \frac{n(n-1)}{2}.$$

Dit is precies wat we wilden aantonen.

[4] https://nl.wikipedia.org/wiki/Carl_Friedrich_Gauss#Anekdotes

[5] Bij deze vervanging voeg je voor de veiligheid best haakjes toe rond $n-1$, vandaar dat we schrijven $(n-1)$.

Opmerking 1.11 (Een tweede belangrijke som) Een tweede som die nuttig zal zijn in deze cursus is de som van een aantal termen die afkomstig zijn uit een meetkundige (geometrische) rij:

$$1 + a + a^2 + \cdots + a^{n-1} + a^n = \frac{a^{n+1} - 1}{a - 1}. \tag{1.6}$$

In het linkerlid is de verhouding van twee opeenvolgende termen constant:

$$\frac{a^{k+1}}{a^k} = a.$$

De formule (1.6) kan eenvoudig bewezen worden. Stel

$$S_n = 1 + a + a^2 + \cdots + a^{n-1} + a^n, \tag{1.7}$$

dan zoeken we een gesloten formule voor S_n. Hiertoe vermenigvuldigen we S_n met a:

$$aS_n = a + a^2 + \cdots + a^n + a^{n+1}. \tag{1.8}$$

Als we nu (1.7) aftrekken van (1.8), dan krijgen we:

$$\begin{aligned} aS_n - S_n &= \left(a + a^2 + \cdots + a^n + a^{n+1}\right) \\ &\quad - \left(1 + a + a^2 + \cdots + a^{n-1} + a^n\right) \\ &= a^{n+1} - 1. \end{aligned}$$

Alle termen a t.e.m. a^n komen immers twee keer voor in het recherlid, één keer met een plusteken en één keer met een minteken. Op die manier zijn het enkel de termen a^{n+1} en 1 die overblijven in het rechterlid. Deze laatste vergelijking kan herschreven worden als:

$$(a - 1)S_n = a^{n+1} - 1,$$

wat equivalent is met de formule (1.6). ∎

1.2.2 Sorteren door Tussenvoegen

Het basisidee achter sorteren door tussenvoegen kan als volgt samengevat worden:

1. Veronderstel dat er reeds een deel vooraan de array gesorteerd is.

2. Neem het eerste element van het niet gesorteerde deel en voeg dit element toe op de juiste plaats in het gesorteerde deel. Op deze manier wordt het gesorteerde deel uitgebreid.

Sorteren door tussenvoegen of "card sort" kan m.a.w. het best vergeleken worden met het op volgorde steken van kaarten.
We beginnen met de tweede kaart. We kijken of deze voor de eerste moet komen of niet. Vervolgens nemen we de volgende kaart en deze plaatsen we dan direct op de juiste positie ten opzichte van de vorige kaarten. Zo doen we verder tot alle kaarten op de juiste plaats zitten.

In het algoritme is dit: indien de eerste i elementen reeds gesorteerd zijn dan gaan we kijken naar het $(i+1)$-ste element. Dit element wordt op de juiste plaats tussengevoegd. Indien nodig moeten de reeds gesorteerde grotere elementen allen één positie doorschuiven.

Voorbeeld 1.12 We passen dit principe nu toe op een eenvoudige rij. De verticale streep geeft aan welk deel van de array reeds is gesorteerd. In het bijzonder zijn de getallen voor de verticale streep steeds gesorteerd. Het onderstreepte getal is hetgene we gaan invoegen in het gesorteerde deel.

44|	<u>55</u>	12	42	94	18	06	67
44	55|	<u>12</u>	42	94	18	06	67
12	44	55|	<u>42</u>	94	18	06	67
12	42	44	55|	<u>94</u>	18	06	67
12	42	44	55	94|	<u>18</u>	06	67
12	18	42	44	55	94|	<u>06</u>	67
06	12	18	42	44	55	94|	<u>67</u>
06	12	18	42	44	55	67	94

In Algoritme 1.5 vindt men de pseudocode voor sorteren door tussenvoegen.

Complexiteitsanalyse

Anders dan bij sorteren door selectie hangt de uitvoeringstijd van het algoritme in dit geval niet enkel af van de inputgrootte. Ook de manier waarop de elementen bij aanvang geordend zijn speelt een rol.

Algoritme 1.5 Sorteren door tussenvoegen.

Invoer De array a is gevuld met n elementen.
Uitvoer De array a is gesorteerd.

```
 1: function CARDSORT(a)
 2:     for i = 1 ... n − 1 do
 3:         x ← a[i]                          ▷ x bevat het in te voegen element
 4:         j ← i                             ▷ j zoekt de juiste positie voor x
 5:         while j > 0 and x < a[j − 1] do   ▷ schuif grotere elementen op
 6:             a[j] ← a[j − 1]               ▷ schuif a[j − 1] eentje op
 7:             j ← j − 1
 8:         end while
 9:         a[j] ← x                          ▷ x wordt op de juiste positie tussengevoegd
10:     end for
11: end function
```

Indien de rij bij aanvang in omgekeerde volgorde gesorteerd is, zullen de opdrachten in de binnenste lus $\frac{n(n-1)}{2}$ keer uitgevoerd worden. Elke individuele uitvoering van zo'n opdracht neemt constante tijd. Het zonet genoemde aantal is het maximum aantal keer dat de lus kan doorlopen worden. Dit komt overeen met een uitvoeringstijd

$$T(n) = \mathcal{O}(n^2), \text{ (het slechtste geval).}$$

Als de rij bij aanvang van het algoritme reeds gesorteerd is, zal het stopcriterium voor de binnenste lus steeds voldaan zijn waardoor de lus niet wordt uitgevoerd. De buitenste lus wordt $(n-1)$ keer uitgevoerd. Dit resulteert in een lineaire uitvoeringstijd.

Men kan bewijzen dat ook in het gemiddelde geval, voor een willekeurige ordening van de inputrij, geldt dat het aantal vergelijkingen groeit zoals n^2. In het gemiddelde geval heeft sorteren door tussenvoegen dus, net als sorteren door selectie, een kwadratische uitvoeringstijd.

Omdat bij complexiteitsanalyse, zoals reeds eerder vermeld, steeds wordt uitgegaan van het slechtst mogelijke scenario kunnen we besluiten dat de tijdscomplexiteit van sorteren door tussenvoegen kwadratisch is:

$$T(n) = \mathcal{O}(n^2).$$

1.2.3 Sorteren door Mengen

Sorteren door mengen (*mergesort*) is een ingewikkelder algoritme dan de voorgaande twee eenvoudige sorteeralgoritmes maar is ook een heel stuk efficiënter. Het basisidee is het volgende:

1. Sorteer de eerste helft van de array.
2. Sorteer de tweede helft van de array.
3. Meng de twee gesorteerde deelrijen samen tot één gesorteerde array.

De eerste twee stappen in dit proces gebeuren op een *recursieve* manier.

Voorbeeld 1.13 Beschouw de rij

$$44 \quad 55 \quad 12 \quad 42 \quad 94 \quad 18 \quad 06 \quad 67\ .$$

Deze wordt opgesplitst in twee deelrijen

$$\begin{array}{cccc} 44 & 55 & 12 & 42 \\ 94 & 18 & 06 & 67. \end{array}$$

Beide deelrijen worden vervolgens recursief gesorteerd tot

$$\begin{array}{cccc} 12 & 42 & 44 & 55 \\ 06 & 18 & 67 & 94. \end{array}$$

Tot slot worden de gesorteerde deelrijen samengevoegd tot de gesorteerde rij

$$06 \quad 12 \quad 18 \quad 42 \quad 44 \quad 55 \quad 67 \quad 94.$$ ∎

We bekijken nu in iets meer detail hoe de mengstap precies verloopt. Voor het samenvoegen of mergen van de deelrijen wordt een hulprij aangemaakt. In deze hulprij worden de beide deelrijen samengevoegd.

Voor het samenvoegen van de deelrijen wordt in de beide deelrijen links gestart. Dit is voor beide deelrijen bij het kleinste element. Deze twee elementen worden vergeleken, het kleinste van de twee wordt opgeslagen in de hulprij. Zowel voor de beide deelrijen als voor de hulprij wordt er een teller

geïnitialiseerd die de huidige positie in die rij bijhoudt. Wanneer een element uit een deelrij is opgeslagen in de hulprij mag de teller van deze deelrij een positie opschuiven net als de teller van de hulprij. Dit proces wordt herhaald tot één van de deelrijen volledig is doorlopen. Vervolgens kunnen de resterende elementen van de andere deelrij achteraan toegevoegd worden in de hulprij.

Voorbeeld 1.14 (Mengen van twee gesorteerde (deel)rijen) Het mengen van twee deelrijen wordt stap voor stap uitgewerkt in het volgende voorbeeld. De eerste twee kolommen geven de respectievelijke gesorteerde (deel)rijen weer. Het getal in het vetjes is het element waarnaar op dat moment wordt verwezen. De derde kolom geeft de hulprij weer: deze wordt bij elke iteratie met één element uitgebreid. Dit getal is steeds het kleinste van de twee getallen waarnaar op dat moment wordt verwezen in de gesorteerde (deel)rijen.

In dit voorbeeld werd de eerste deelrij als eerste volledig gekopieerd naar de hulprij. Alle resterende getallen uit de tweede deelrij worden dan gekopieerd (zonder verdere controles) naar de hulprij. Dit gebeurt op de onderste lijn in onderstaande tabel.

12	42	44	55	**06**	18	67	94	06							
12	42	44	55	06	**18**	67	94	06	12						
12	**42**	44	55	06	**18**	67	94	06	12	18					
12	**42**	44	55	06	18	**67**	94	06	12	18	42				
12	42	**44**	55	06	18	**67**	94	06	12	18	42	44			
12	42	44	**55**	06	18	**67**	94	06	12	18	42	44	55		
12	42	44	55	06	18	**67**	94	06	12	18	42	44	55	67	94

Algoritme 1.6 beschrijft de methode MERGESORT. Door de recursie zullen we opnieuw gebruik moeten maken van een hulpmethode waarin de eigenlijke recursie plaatsvindt.

Algoritme 1.6 De hoofdmethode voor sorteren door mengen

Invoer de array a is gevuld met n elementen.
Uitvoer de array a is gesorteerd

```
1: function MERGESORT(a)
2:     MERGESORTRECURSIVE(a, 0, n − 1)
3: end function
```

De functie MERGESORT roept de recursieve functie MERGESORTRECURSIVE aan. Deze functie wordt uitgewerkt in Algoritme 1.7. In deze methode wordt een deel van de rij gesorteerd door het te sorteren deel op te splitsen in twee deelrijen van halve lengte. Vervolgens worden de gesorteerde deelrijen gemengd. Het samenvoegen van de beide deelrijen gebeurt in de functie MERGE. Deze functie wordt beschreven in Algoritme 1.8.

Algoritme 1.7 Het algoritme mergeSortRecursive sorteert een deel van de array a.

Invoer de array a is gevuld met n elementen, *begin* en *einde* wijzen naar geldige posities in de array a.

Uitvoer de elementen met index *begin* tot en met index *einde* werden gesorteerd.

```
1: function MERGESORTRECURSIVE((a, begin, einde)
2:     if begin < einde then
3:         midden ← ⌊(begin + einde)/2)⌋
4:         MERGESORTRECURSIVE(a, begin, midden)
5:         MERGESORTRECURSIVE(a, midden + 1, einde)
6:         MERGE(a, begin, midden, einde)
7:     end if
8: end function
```

Complexiteitsanalyse

We starten met een analyse van de functie om twee deelrijen van lengte $n/2$ te mengen tot een gesorteerde rij van lengte n. In de methode MERGE wordt elk element van de twee halve deelrijen één keer gekopieerd naar de hulprij, en vervolgens wordt elk element nog eens teruggekopieerd naar zijn uiteindelijke plaats in de array a. Per kopieerbewerking gebeurt er een hoeveelheid werk dat niet afhangt van de lengte van array a. Dit betekent dat de tijdscomplexiteit van de methode MERGE lineair is. Als de te mengen deelrijen dubbel zo lang worden, dan zal de uitvoeringstijd (ongeveer) verdubbelen. Voor de eenvoud (en omdat constante factoren in het algemeen toch niet van belang zijn wanneer men algoritmische complexiteit bestudeert vanuit het standpunt van asymptotische analyse) stellen we deze hoeveelheid werk voor door n.

Hoeveel werk zal sorteren door mengen verzetten? We berekenen dit nu (voor de eenvoud) voor rijen waarvan de lengte een macht van 2 is, m.a.w. $n = 2^k$ waarbij k een natuurlijk getal voorstelt.

Algoritme 1.8 De methode MERGE voegt twee gesorteerde deelrijen samen.

Invoer de array a is gevuld met n elementen; de elementen van de deelrij gaande van de *begin*-positie tot en met de *midden*-positie zijn gesorteerd; de elementen van de deelrij gaande van de (*midden*+1)-positie tot en met de *eind*-positie zijn gesorteerd.

Uitvoer de elementen met index *begin* t.e.m. index *einde* werden gesorteerd.

```
1:  function MERGE(a, begin, midden, einde)
2:      i ← begin                          ▷ de teller i doorloopt de linkse deelrij
3:      j ← midden + 1                     ▷ de teller j doorloopt de rechtse deelrij
4:      k ← i                              ▷ de teller k doorloopt de hulparray hulpa
5:      hulpa ← nieuwe array[n]            ▷ tijdelijke hulpopslag
6:      while i ≤ midden and j ≤ einde do  ▷ totdat een deelrij leeg is
7:          if a[i] ≤ a[j] then            ▷ het kleinste element komt eerst
8:              hulpa[k] ← a[i]
9:              i ← i + 1
10:         else
11:             hulpa[k] ← a[j]
12:             j ← j + 1
13:         end if
14:         k ← k + 1                      ▷ er is een element in hulpa opgeslagen
15:     end while
16:     if i > midden then                 ▷ de 2de deelrij moet leeggemaakt worden
17:         while j ≤ einde do
18:             hulpa[k] ← a[j]
19:             j ← j + 1
20:             k ← k + 1
21:         end while
22:     else                               ▷ de 1ste deelrij moet leeggemaakt worden
23:         while i ≤ midden do
24:             hulpa[k] ← a[i]
25:             i ← i + 1
26:             k ← k + 1
27:         end while
28:     end if
29:     for k = begin ... einde do         ▷ kopieer gesorteerde deelrij naar a
30:         a[k] ← hulpa[k]
31:     end for
32: end function
```

Als de lengte van de rij gelijk is aan 1 dan is

$$T(1) = 1,$$

want er moet enkel gecontroleerd worden dat de lengte van de rij gelijk is aan 1.

Wanneer de lengte van de rij gelijk is aan 2, dan is de totale hoeveelheid werk

$$T(2) = T(1) + T(1) + 2,$$

want we moeten de linkerdeelrij sorteren (eerste term $T(1)$), we moeten de rechterdeelrij sorteren (tweede term $T(1)$) en tenslotte moeten de twee gesorteerde deelrijen worden gemengd (term 2). Samen geeft dit

$$T(2) = 2T(1) + 2 = 2 \times 1 + 2 = 4.$$

Voor een rij van lengte $4 = 2^2$ wordt dit op een gelijkaardige manier:

$$T(4) = 2T(2) + 4 = 2 \times 4 + 4 = 12,$$

en voor een rij van lengte $8 = 2^3$ krijgen we:

$$T(8) = 2T(4) + 8 = 2 \times 12 + 8 = 32.$$

Welk patroon kunnen we hierin herkennen? Merk op dat

$$\begin{aligned}
T(1) &= T(2^0) = 1 = 1 \times (0 + 1) \\
T(2) &= T(2^1) = 4 = 2 \times (1 + 1) \\
T(4) &= T(2^2) = 12 = 4 \times (2 + 1) \\
T(8) &= T(2^3) = 32 = 8 \times (3 + 1)
\end{aligned}$$

Hieruit kunnen we herkennen dat het algemene patroon is:

$$T(n) = T(2^k) = n \times (k + 1).$$

Hieruit kunnen we besluiten dat de tijdscomplexiteit $T(n)$ van sorteren door mengen wordt gegeven door:

$$T(n) = \mathcal{O}\big(n \log_2(n)\big).$$

Opmerking 1.15 (Geheugengebruik) Hoewel mergesort op het niveau van de uitvoeringstijd efficiënter is dan de voorgaande methodes, is deze methode minder efficiënt op het niveau van de benodigde geheugenruimte. Het MERGE-algoritme maakt immers gebruik van een extra array *hulpa* waarin de gesorteerde rij tijdelijk wordt opgeslagen. Dit komt de geheugencomplexiteit niet ten goede. ∎

1.3 Oefeningen

1. Ga in de array (1 2 3 4 6 7 8 9 10) achtereenvolgens op zoek naar de waarden 3, 5 en 10. Maak hierbij gebruik van het algoritme ZOEKBINAIR. Houd tijdens je simulatie de waarden bij van de variabelen l, r en m.

2. (**Dodona**) Implementeer de iteratieve methode voor binair zoeken.

3. (**Dodona**) Herwerk en implementeer de oplossingsmethode voor sorteren door selectie zodat niet langer het grootste element naar achteren wordt gebracht maar het kleinste element naar voor. Het reeds gesorteerde deel van de array bevindt zich m.a.w. steeds vooraan.

4. Pas de oplossingsmethode beschreven in het opgegeven algoritme aan zodat de elementen niet langer van klein naar groot worden gesorteerd maar van groot naar klein.

   a) sorteren door tussenvoegen

   b) sorteren door mengen

5. (**Dodona**) Bubble sort is een oplossingsmethode voor het sorteren van een array a van lengte n. Algoritme 1.9 beschrijft deze methode in pseudo-code.

Algoritme 1.9 De methode bubbleSort.

Invoer De array a is gevuld met n elementen.
Uitvoer De array a is gesorteerd.

```
1: function BUBBLESORT(a)
2:     for i = 0 ... n − 2 do
3:         for j = n − 1 ... i + 1 by −1 do
4:             if a[j − 1] > a[j] then
5:                 wissel de elementen a[j − 1] en a[j]
6:             end if
7:         end for
8:     end for
9: end function
```

   a) Implementeer deze methode in Python.

b) Tel het exact aantal keer, in functie van n, dat de controle van regel 4, nl. $a[j-1] > a[j]$ wordt uitgevoerd. Gebruik een gesloten formule om je resultaat weer te geven. Leid hieruit de $\mathcal{O}$-notatie voor de uitvoeringstijd van het algoritme bubbleSort af.

c) Stel $a = (13, 7, 8, 1)$. Volg in een overzichtelijke tabel de uitwerking van BUBBLESORT(a). Noteer in de tabel minstens de verschillende waarden die i, j en a doorlopen.

6. Hieronder volgen een aantal codefragmenten f_i. Bepaal voor elk van deze fragmenten de waarde van x in functie van de invoerwaarde n. Leid hieruit de asymptotische uifvoeringstijd, t.t.z. de $\mathcal{O}$-notatie, af van deze verschillende codefragmenten. Tracht je resultaten experimenteel te verifiëren door een programma te schrijven dat de verhouding berekent van de $f_i(n)$ over je theoretisch voorspelde uitvoeringstijd. Wanneer je de juiste uitvoeringstijd hebt dan zou deze verhouding zo goed als constant moeten zijn (voor grote waarden van n).

a)
```
1: function SOM1(n)
2:     x ← 0
3:     for i = 1 ... n do
4:         x ← x + 1
5:     end for
6:     return (x)
7: end function
```

b)
```
1: function SOM2(n)
2:     x ← 0
3:     for i = 1 ... 2n do
4:         for j = 1 ... n do
5:             x ← x + 1
6:         end for
7:     end for
8:     return (x)
9: end function
```

c)
```
1: function SOM3(n)
2:     x ← 0
3:     for i = 1 ... n do
4:         for j = 1 ... i do
5:             for k = 1 ... j do
6:                 x ← x + 1
```

```
            end for
        end for
    end for
    return (x)
end function
```

d)
```
function SOM4(n)
    x ← 0
    for i = 1 . . . n do
        for j = 1 . . . i do
            for k = 1 . . . i do
                x ← x + 1
            end for
        end for
    end for
    return (x)
end function
```

e)
```
function SOM5(n)
    x ← 0
    i ← n
    while i ≥ 1 do
        x ← x + 1
        i ← ⌊i/2⌋
    end while
    return (x)
end function
```

f)
```
function SOM6(n)
    x ← 0
    i ← n
    while i ≥ 1 do
        for j = 1 . . . i do
            x ← x + 1
        end for
        i ← ⌊i/2⌋
    end while
    return (x)
end function
```

g)
```
function SOM7(n)
    x ← 0
```

```
3:      i ← n
4:      while i ≥ 1 do
5:          for j = 1 . . . n do
6:              x ← x + 1
7:          end for
8:          i ← ⌊i/2⌋
9:      end while
10:     return (x)
11: end function
```

HOOFDSTUK **2**

Gelinkte lijsten

De elementen van een array bezetten steeds opeenvolgende geheugenplaatsen wat ervoor zorgt dat men de elementen kan aanspreken in constante tijd. Anderzijds zorgt dit ervoor dat het minder eenvoudig is om elementen ergens "in het midden" toe te voegen. GELINKTE LIJSTEN zijn een voorbeeld van een dynamische datastructuur waarbij de samenhang tussen de verschillende componenten gerealiseerd wordt door wijzers (*pointers*) naar de elementen in de datastructuur waardoor toevoegen "in het midden" eenvoudiger wordt. We bestuderen ENKELVOUDIG en DUBBEL GELINKTE LIJSTEN. We gebruiken de gelinkte lijsten om een STAPEL te realiseren en we zien enkele problemen die gemakkelijk met een stapel kunnen opgelost worden. In de oefeningen zien we tenslotte ook hoe een WACHTRIJ kan gerealiseerd worden m.b.v. een gelinkte lijst.

Bij sorteren door tussenvoegen hebben we gezien dat wanneer we een element willen tussenvoegen op een bepaalde plaats in de array dat we dan alle achterliggende elementen één plaats moeten opschuiven. Het is eenvoudig in te zien dat dit eveneens het geval is wanneer we een element willen verwijderen uit een array. Dit zorgt ervoor dat dit soort bewerkingen in een array in het slechtste geval een lineaire tijdscomplexiteit hebben: alle elementen moeten immers worden verplaatst. We bekijken nu een nieuwe datastructuur, nl. een lineaire gelinkte lijst of kortweg gelinkte lijst, waarbij dit probleem wordt opgelost.

Het voordeel van een gelinkte lijst is dat het uitvoeren van een basisfunctie

zoals toevoegen of verwijderen van elementen in een constante tijd gebeurt, op voorwaarde dat men reeds weet waar het element moet worden toegevoegd of verwijderd. Het opzoeken van een element in een gelinkte lijst vraagt echter nog steeds een lineaire tijd.

Een tweede voordeel van een gelinkte lijst is dat deze a priori geen beperking oplegt aan het aantal elementen dat aan de lijst kan toegevoegd worden. Gaandeweg wordt de lijst verder uitgebreid, een gelinkte lijst is m.a.w. een dynamische structuur.

2.1 Specificatie

Een gelinkte lijst bestaat uit een aantal *knopen* die via een *kettingstructuur* aan elkaar geschakeld zijn.

Een knoop bestaat uit twee velden:

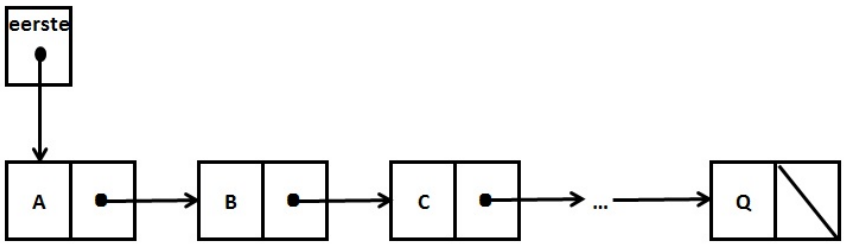
- een data-veld *data* waarin de eigenlijke data wordt opgeslagen;
- een veld *volgende* dat verwijst naar de volgende knoop in de lijst.

De laatste knoop bevat een wijzer *null*; op figuren wordt die aangeduid met een schuine streep.
Voor de eerste knoop moet een referentie *eerste* bijgehouden worden. In een lege lijst is de referentie *eerste* gelijk aan null.

Figuur 2.1 illustreert deze structuur.

De belangrijkste basisbewerkingen voor een enkelvoudig gelinkte lijst zijn:

- *zoek*( ): zoekt de positie van de knoop met als data-veld het argument;
- *verwijder*( ): verwijdert de knoop die volgt na de opgegeven knoop en geeft de waarde van het data-veld van de verwijderde knoop weer;
- *voegToe*( ): voegt een knoop toe na een opgegeven knoop, het data-veld krijgt de waarde van het argument.



Figuur 2.1: Een enkelvoudig gelinkte lijst.

2.2 Implementatie van een Gelinkte Lijst

Een gelinkte lijst wordt voorgesteld door een ketting van aaneengeschakelde knopen. Elke knoop bevat een data-veld *data* en een veld *volgende* dat de referentie naar de volgende knoop van de lijst bijhoudt.
De referentie *eerste* verwijst naar de eerste knoop van de lijst. De referentie *null* geeft aan dat het einde van de gelinkte lijst bereikt is.

We starten met een implementatie voor een knoop.

2.2.1 Implementatie van een Knoop

We schrijven de klasse Knoop in UML-notatie. De constructor wordt in pseudocode beschreven.

De klasse Knoop in UML

Knoop
- *data* : Element - *volgende* : Knoop
+ Knoop( )

De gedefinieerde Knoop is een datastructuur die een element van een niet nader gedefinieerde klasse Element bevat.

Algoritme voor de constructor

De constructor Knoop maakt een nieuw object van de klasse Knoop aan.

Algoritme 2.1 De constructor.

Invoer /
Uitvoer er werd een nieuwe knoop aangemaakt

```
1: function KNOOP
2:     data ← null
3:     volgende ← null
4: end function
```

2.2.2 Implementatie van een gelinkte lijst

De klasse Knoop wordt als inwendige klasse (inner class) van de klasse GelinkteLijst geïmplementeerd. Dit betekent dat methodes van de klasse GelinkteLijst toegang hebben tot de velden van Knoop.

We schrijven de klasse GelinkteLijst uit in UML-notatie. Alle methodes worden in pseudocode beschreven.

De klasse GelinkteLijst in UML

GelinkteLijst
- *eerste* : Knoop
+ GelinkteLijst( ) + zoek(*x*: Element) : Knoop + verwijder(*ref*: Knoop) : Element + voegToe(*ref*: Knoop, *x*: Element) : /

Algoritme voor de constructor

De constructor GelinkteLijst maakt een nieuw object van de klasse GelinkteLijst aan. Deze methode wordt in Algoritme 2.2 beschreven in pseudocode.

Algoritme voor het opzoeken van een element x

Het opzoeken van een knoop met dataveld x in een gelinkte lijst l gebeurt door de knopen van de lijst één voor één te overlopen.

Algoritme 2.2 De constructor van een lineair gelinkte lijst.

Invoer /
Uitvoer er werd een nieuwe (lege) gelinkte lijst aangemaakt

```
1: function GELINKTELIJST
2:     eerste ← null
3: end function
```

Als eerste knoop wordt de knoop waar de referentie *eerste* naar verwijst bekeken. Indien de waarde van het dataveld van deze knoop niet overeenstemt met x wordt de volgende knoop bekeken. Deze knoop is eenvoudig te vinden door de ketting te volgen via het referentieveld. Dit proces wordt herhaald tot x wordt gevonden of het einde van de gelinkte lijst wordt bereikt.

Het eerste voorkomen van het element x wordt bijgehouden. De referentie naar de knoop met dataveld x wordt geretourneerd.

Deze methode wordt in Algoritme 2.3 beschreven in pseudocode.

Algoritme 2.3 Een element x opzoeken in een gelinkte lijst. De returnwaarde is een instantie van een Knoop.

Invoer de gelinkte lijst werd aangemaakt, x is het te zoeken element
Uitvoer de referentie naar de eerste knoop met dataveld gelijk aan x werd geretourneerd, indien x niet voorkomt in de lijst werd de referentie null geretourneerd.

```
1: function ZOEK(x)
2:     ref ← eerste
3:     while ref ≠ null and ref.data ≠ x do
4:         ref ← ref.volgende
5:     end while
6:     return ref
7: end function
```

De uitvoeringstijd voor de methode *zoek* is afhankelijk van de positie van x in de lijst.

- In het slechtste geval wordt elke knoop van de lijst overlopen en wordt x helemaal niet gevonden. Dit komt overeen met een lineaire uitvoeringstijd.

- In het beste geval vindt men de gezochte referentie onmiddellijk. Dit komt overeen met een constante uitvoeringstijd.
- In het gemiddelde geval moet ongeveer de helft van de knopen overlopen worden. Dit komt opnieuw overeen met een lineaire uitvoeringstijd.

Het zoeken van een element x in een gelinkte lijst gebeurt niet efficiënter dan in een *gewone* array. Integendeel, in de praktijk is het volgen van referenties (*pointers*) trager dan het doorlopen van een array.

De overige basisfuncties voor een gelinkte lijst zullen wel allemaal uitvoerbaar zijn in een constante tijd, op voorwaarde dat de referentie reeds gekend is.

Algoritme voor het verwijderen van een knoop

De functie VERWIJDER verwacht als inputwaarde een referentie *ref*. Deze referentie *ref* verwijst naar de knoop in de gelinkte lijst die de te verwijderen knoop voorafgaat. De waarde van het data-veld van de verwijderde knoop wordt geretourneerd. De reden dat we de voorafgaande knoop moeten meegeven is omdat de referentie naar de volgende dient aangepast te worden in de knoop voorafgaand aan de te verwijderen knoop. Er is echter geen efficiënte manier om in een lineair gelinkte lijst de *voorganger* van een knoop te vinden.

De methode VERWIJDER wordt geïllustreerd in Figuur 2.2.

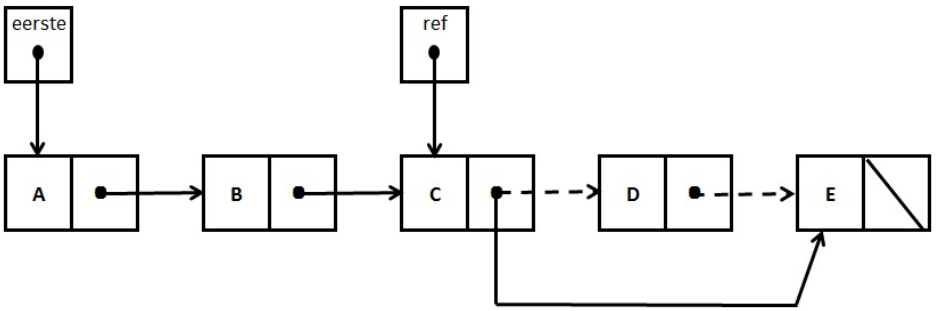
Algoritme 2.4 beschrijft de methode VERWIJDER in pseudocode.

Algoritme 2.4 De knoop na de gegeven knoop verwijderen in een gelinkte lijst l.

Invoer de gelinkte lijst l bestaat, de knoop *ref* is niet de laatste knoop in de lijst.

Uitvoer de knoop die volgt na de knoop met referentie *ref* werd verwijderd uit de lijst, het data-veld van de verwijderde knoop werd geretourneerd.

```
1: function VERWIJDER(ref)
2:     x ← ref.volgende.data
3:     ref.volgende ← ref.volgende.volgende
4:     return x
5: end function
```



Figuur 2.2: Verwijderen van de knoop na de gegeven knoop.

Opmerking 2.1 (Vooraan verwijderen) Aangezien de methode *verwijder* de knoop verwijdert die volgt ná de knoop die wordt aangewezen door de gegeven referentie, is het niet mogelijk de eerste knoop van de gelinkte lijst te verwijderen m.b.v. deze methode. Dit probleem wordt aangepakt in Paragraaf 2.2.3. ∎

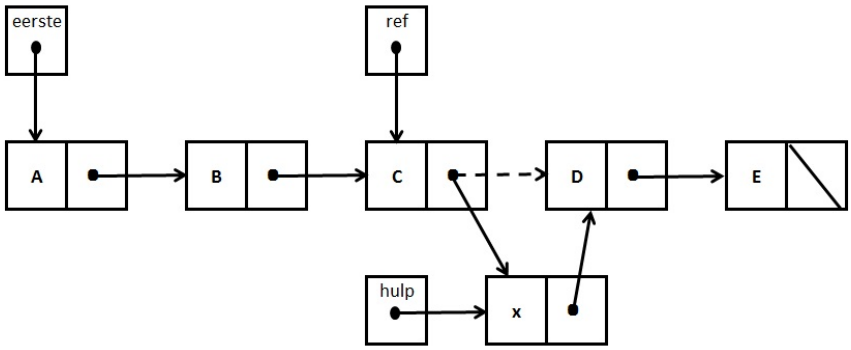
Algoritme voor het toevoegen van een knoop

De methode VOEGTOE voegt een nieuwe knoop met data-veld x toe na de knoop die wordt aangeduid door een gegeven referentie *ref*.

De werkwijze wordt geïllustreerd in Figuur 2.3 en omvat de volgende stappen:

- creëer een nieuwe knoop *hulp*;
- stockeer de waarde x in het data-veld van de knoop *hulp*;
- laat *hulp.volgende* verwijzen naar *ref.volgende*;
- laat *ref.volgende* verwijzen naar *hulp*.

De vertaling van deze werkwijze naar pseudocode wordt gegeven in Algoritme 2.5.



Figuur 2.3: Toevoegen van een knoop na de gegeven knoop.

Algoritme 2.5 Een knoop met dataveld x toevoegen na de gegeven knoop.

Invoer de gelinkte lijst bestaat, en *ref* is niet null.
Uitvoer na de knoop, waarnaar gerefereerd wordt door de referentie *ref*, werd een nieuwe knoop met data-veld x toegevoegd.

```
1: function VOEGTOE(ref, x)
2:     hulp ← nieuwe Knoop( )
3:     hulp.data ← x
4:     hulp.volgende ← ref.volgende
5:     ref.volgende ← hulp
6: end function
```

Opmerking 2.2 (Vooraan toevoegen) De methode VOEGTOE laat enkel toe een knoop toe te voegen ná de knoop die wordt aangewezen door de gegeven referentie. Deze methode laat niet toe een knoop toe te voegen helemaal vooraan in de gelinkte lijst. Eveneens is het met deze methode niet mogelijk om een knoop toe te voegen aan een lege gelinkte lijst. Dit kan verholpen worden door gebruik te maken van een *ankercomponent*. ∎

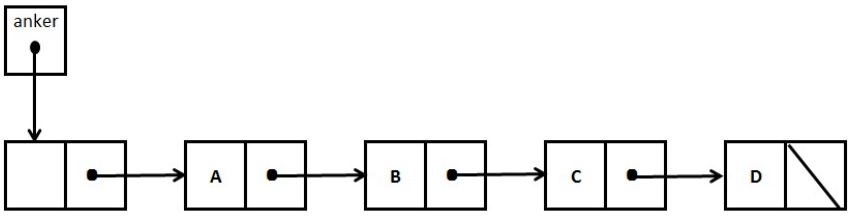
2.2.3 Het gebruik van ankercomponenten

De methoden VERWIJDER en VOEGTOE uit de voorgaande paragraaf zijn enkel bruikbaar voor algemene gevallen. Het speciale geval om een element toe te voegen aan een lege lijst of een element toe te voegen vooraan in de

lijst is niet mogelijk met de besproken methode.[1] Het is eveneens niet mogelijk het eerste element van een gelinkte lijst te verwijderen met de methode VERWIJDER volgens de besproken specificatie.

Een eenvoudige en elegante oplossing voor deze bijzondere gevallen is om een *ankercomponent* toe te voegen aan de gelinkte lijst. De ankercomponent is een extra knoop die helemaal vooraan aan de gelinkte lijst wordt toegevoegd. Het data-veld van deze ankercomponent is leeg. Het referentie-veld van de ankercomponent verwijst naar de eerste knoop van de gelinkte lijst. De ankercomponent maakt dus logisch gezien geen deel uit van de eigenlijke lijst maar dient enkel om de implementatie te vereenvoudigen.

Figuur 2.4 is een illustratie van een gelinkte lijst met ankercomponent.

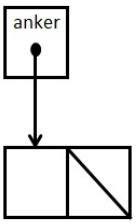


Figuur 2.4: Een gelinkte lijst met ankercomponent.

Door deze ankercomponent toe te voegen heeft elke knoop een voorganger in de lijst. Dit betekent dat nu ook de eerste knoop kan verwijderd worden of dat op de eerste positie een nieuwe knoop kan toegevoegd worden.
Een lege gelinkte lijst bestaat nu uit één enkele knoop, nl. de ankercomponent, en de referentie naar deze component. Figuur 2.5 is een voorstelling van een lege gelinkte lijst wanneer een ankercomponent wordt gebruikt.

Het gebruik van een ankercomponent vereenvoudigt de basisbewerkingen voor een gelinkte lijst. Er moet immers niet steeds op bijzondere situaties gecontroleerd worden.

[1] Dit betekent dat het eigenlijk niet mogelijk is om deze implementatie effectief aan de praat te krijgen aangezien het niet mogelijk is om een knoop toe te voegen aan een lege lijst!

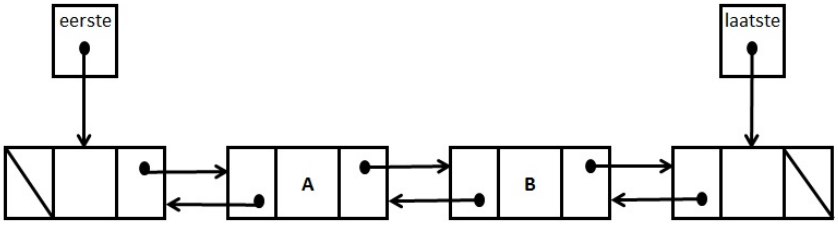


Figuur 2.5: Een lege gelinkte lijst met gebruik van een ankercomponent.

2.3 Dubbelgelinkte lijsten

In een enkelvoudig gelinkte lijst is de eerste knoop van de lijst vlot bereikbaar. De laatste knoop bereiken kost echter tijd lineair in de lengte van de lijst.
Een mogelijke oplossing hiervoor is om in elke knoop van de gelinkte lijst twee referenties bij te houden: een referentie *volgende* en een referentie *vorige*. Naast de referentie *eerste* naar de eerste knoop in de lijst, is het dan logisch om ook een referentie *laatste* naar de laatste knoop in de lijst bij te houden.

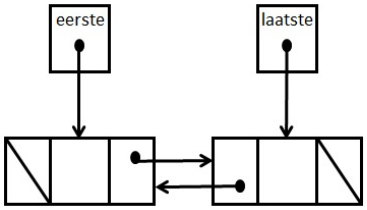
Een dergelijke structuur wordt een *dubbelgelinkte lijst* genoemd. Figuur 2.6 is een illustratie van een dergelijke lijst.



Figuur 2.6: Een dubbelgelinkte lijst met twee ankercomponenten.

In een dubbelgelinkte lijst wordt meestal gewerkt met twee ankercompo-

nenten. Figuur 2.7 stelt een lege dubbelgelinkte lijst voor.



Figuur 2.7: Een lege dubbelgelinkte lijst met twee ankercomponenten.

Testen of de lijst leeg is, kan als volgt: als *eerste.volgende* = *laatste* of als *laatste.vorige* = *eerste*.

2.4 Beschrijving en Implementatie van Stapels

2.4.1 Beschrijving van een Stapel

De naam *stapel* of *stack* is gekozen naar analogie met een stapel boeken.

Indien we een boek van de stapel nodig hebben dan is het best bereikbare boek hetgeen bovenaan ligt. Een boek dat zich op een andere plaats bevindt, is slechts bereikbaar als we alle bovenliggende boeken verwijderd hebben.

Ditzelfde principe wordt toegepast voor het datatype stack:

- Een element dat we willen toevoegen aan een stapel, komt steeds bovenop de reeds bestaande stapel te liggen.
- Enkel het bovenste element van de stapel kan verwijderd worden. Dit element wordt de *top* van de stapel genoemd.

Een stapel is m.a.w. een *LIFO*- of *Last-In-First-Out*-structuur.

De voorgaande beschrijving legt de structuur van een stapel vast. Voor de implementatie van deze structuur zullen we gebruik maken van een klasse Stack. In deze klasse worden een aantal basisbewerkingen gedefinieerd. Deze zijn:

- *Stack*( ): constructor, maakt een nieuwe stapel aan waarna de stapel bestaat als lege stapel;
- *empty*( ): controleert of een stapel al dan niet leeg is;
- *push*( ): voegt een nieuw element toe bovenaan een stapel, het toegevoegde element wordt de nieuwe top van de stapel;
- *pop*( ): verwijdert het bovenste element van een stapel en retourneert het verwijderde element;
- *peek*( ): geeft het bovenste element van de stapel terug, zonder het te verwijderen.

De omschrijvingen van deze bewerkingen moeten vertaald worden naar algoritmen die hun werking ondubbelzinnig vastleggen. Indien de implementatie efficiënt gebeurt zal de uitvoeringstijd van de algoritmen niet afhankelijk zijn van de grootte van de stapel.

2.4.2 Implementatie van een Stapel

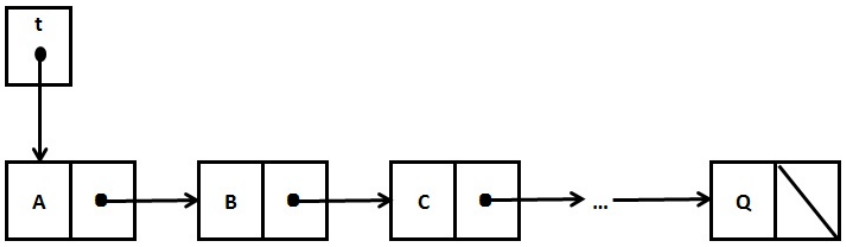
We tonen nu hoe een stapel kan geïmplementeerd worden als een gelinkte lijst waarbij enkel de top bereikbaar is. In het bijzonder wordt een stapel geïmplementeerd door een referentie t naar de eerste knoop in de lijst, die de top van de stapel bevat.

Figuur 2.8 is een illustratie van een stapel geïmplementeerd als gelinkte lijst.

De klasse Stack in UML

Stack
- t : Knoop
+ Stack( ) + empty( ) : boolean + push(x: Element) : / + pop( ) : Element + peek( ) : Element

De klasse Knoop wordt als inwendige klasse (inner class) van de klasse Stack geïmplementeerd. Dit betekent dat methodes van de klasse Stack toe-



Figuur 2.8: Een stapel geïmplementeerd als gelinkte lijst. Het gebruik van een ankercomponent is hier niet nodig. De top van de stapel bevat het element A. Het element Q werd als eerste element op de stapel geplaatst.

gang hebben tot de velden van Knoop. De implementatie van de klasse Knoop is zoals voorheen.

De constructor

Een lege stapel bevat nog geen elementen, m.a.w. de referentie t is *null*.

Algoritme 2.6 De constructor.

Invoer /
Uitvoer er werd een nieuwe stapel aangemaakt, deze stapel bestaat als lege stapel.

```
1: function STACK
2:     t ← null
3: end function
```

Algoritme ter controle of een stapel al dan niet leeg is

De methode EMPTY controleert of een stapel al dan niet leeg is. Het resultaat van de methode is een boolean. Algoritme 2.7 beschrijft deze methode. Het enige wat er gebeurt is nagaan of de referentie naar de top al dan niet gelijk is aan null.

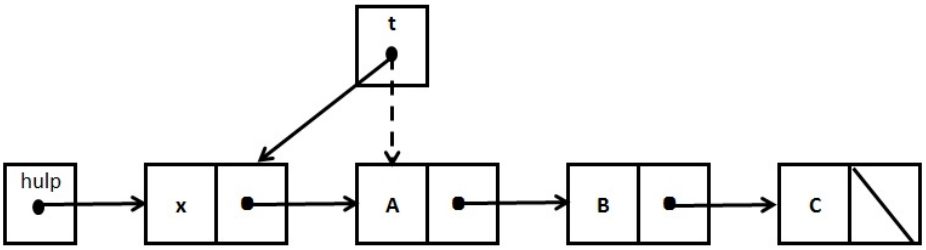
Algoritme 2.7 Controleren of een stapel s al dan niet leeg is.

Invoer de stapel s bestaat
Uitvoer de waarde **true** of **false** werd afgeleverd, afhankelijk van het feit of de stapel s leeg is of niet.

```
1: function EMPTY
2:     return t = null
3: end function
```

Algoritme voor het toevoegen van een element aan een stapel

De methode PUSH wordt geïllustreerd in Figuur 2.9.



Figuur 2.9: Een element toevoegen aan de top van de stapel.

Het element x wordt bovenop de stapel geduwd. Algoritme 2.8 beschrijft de methode PUSH in pseudocode.

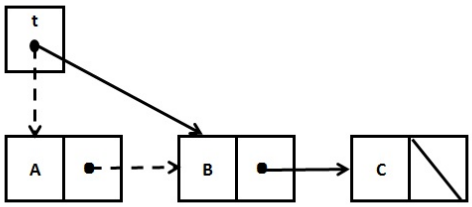
Algoritme 2.8 Een element x toevoegen aan de top van een stapel s.

Invoer de stapel s bestaat, x moet op de stapel worden geplaatst
Uitvoer het element x werd als top-element op de stapel s geplaatst.

```
1: function PUSH(x)
2:     hulp ← nieuwe Knoop( )
3:     hulp.data ← x
4:     hulp.volgende ← t
5:     t ← hulp
6: end function
```

Algoritme voor het verwijderen van een element van een stapel

In dit algoritme wordt de top van de stapel verwijderd en geretourneerd. Er wordt verondersteld dat de stapel niet leeg is. De methode POP wordt geïllustreerd in Figuur 2.10 en Algoritme 2.9 beschrijft deze methode in pseudocode.



Figuur 2.10: De top van de stapel verwijderen.

Algoritme 2.9 De top verwijderen en de waarde ervan retourneren.

Invoer de stapel s bestaat en is niet leeg
Uitvoer de top werd verwijderd en de waarde van de top werd geretourneerd.

```
1: function POP
2:     x ← t.data
3:     t ← t.volgende
4:     return x
5: end function
```

Algoritme voor het retourneren van de waarde van de top van een stapel

Er wordt opnieuw verondersteld dat de stapel niet leeg is.

Algoritme 2.10 De waarde van de top retourneren.

Invoer de stapel s bestaat en is niet leeg
Uitvoer de waarde van de top werd geretourneerd, s werd niet gewijzigd.

```
1: function PEEK
2:     return t.data
3: end function
```

2.5 Toepassingen van Stapels

2.5.1 Controleren van haakjes

Elke programmeur kent het verschijnsel dat een programma niet werkt omdat ergens een haakje ontbreekt. Compilers controleren steeds of de haakjes allemaal correct zijn en genereren een foutboodschap wanneer dit niet het geval is.

Basisidee

Voor de controle van de haakjes worden twee zaken onderzocht:

1. Er wordt gecontroleerd of elk openend haakje juist één overeenkomstig sluitend haakje heeft, bv. voor elk '('-symbool moeten we verder in het programma één ')'-symbool terugvinden. Andere mogelijke koppels zijn: [en], { en }. M.a.w. de symbolen moeten in paren voorkomen.

2. De volgorderegels moeten in acht genomen worden. De haakjes mogen genest zijn, dit wil zeggen dat een paar haakjes zich tussen een ander paar mag bevinden, maar paren haakjes mogen elkaar niet overlappen.
   Bijvoorbeeld, ([ ]) is een geldige volgorde van symbolen, terwijl ([ )] niet geldig is.

We willen met behulp van een stapel een algoritme ontwerpen dat de vereffening van symbolen verifieert.

Dit kan als volgt:

- Maak een lege stapel aan.
- Doorloop alle symbolen één voor één. Indien het symbool een haakje is, moet er een opdracht uitgevoerd worden:
  - als het een open-symbool is, plaats het dan op de stapel (*push*-bewerking);
  - als het een sluit-symbool is onderscheiden we twee mogelijke situaties:

    - de stapel is leeg: er wordt een foutmelding gegenereerd aangezien er geen corresponderend open-symbool aan is vooraf gegaan;
    - de stapel is niet leeg: het top-element wordt er afgehaald (*pop*-bewerking). Dit symbool wordt vergeleken met het net ingelezen symbool, als beide symbolen niet corresponderen wordt een fout gemeld.

- Als alle karakters zijn ingelezen en de stapel is niet leeg, dan wordt een fout gemeld. Dit betekent immers dat er nog open-symbolen zijn waarvoor geen sluit-symbool is gevonden.

Algoritme

Algoritme 2.11 geeft pseudocode voor het controleren van de haakjes in een array van strings. De pseudocode om te controleren of het om een openend haakje gaat en om na te gaan of haakjes van dezelfde soort zijn wordt niet getoond omdat deze niet tot de essentie van het algoritme behoren.

2.5.2 Waardebepaling van een rekenkundige uitdrukking

Binnen de wiskunde worden in een rekenkundige uitdrukking de binaire operatoren zoals $+$ en $\times$ normaalgezien tussen de operanden geschreven.

Bijvoorbeeld:

$$1 + 2 \times 3.$$

Hierbij staan de operatoren $+$ en $\times$ *tussen* de operanden 1, 2 en 3. Een dergelijke notatie wordt een *infix-notatie* genoemd. In de meeste programmeertalen wordt deze notatie overgenomen.

De rekenkundige uitdrukkingen in infix-notatie kunnen echter niet rechtstreeks door de compiler naar machine-code vertaald worden. De *postfix-notatie* is beter geschikt voor de vertaling naar machine-code.

De postfixnotatie staat ook bekend onder de benaming *RPN* (Reverse Polish Notation). Bij sommige modellen van zakrekenmachines wordt de postfixnotatie gebruikt, bv. bij een aantal rekenmachines van het merk HP.

In een postfix-uitdrukking worden de operatoren *na* hun operanden geschreven.

Algoritme 2.11 Controle op correct gebruik van haakjes.

Invoer *uitdrukking* is een array van Strings met de tokens van een uitdrukking waarin eventueel haakjes voorkomen.

Uitvoer indien alle open haakjes correct worden afgesloten werd er geen foutmelding gegenereerd. In het andere geval wordt er een boodschap getoond op het scherm

```
 1: function CONTROLEERHAAKJES(uitdrukking)
 2:     s ← STACK()
 3:     for i = 0 ... uitdrukking.lengte − 1 do
 4:         symbool ← uitdrukking[i]
 5:         if symbool is openend haakje then
 6:             s.PUSH(symbool)
 7:         else
 8:             if symbool is sluitend haakje then
 9:                 if s.EMPTY() then
10:                     PRINT("Te veel sluit symbolen")
11:                 else
12:                     voorgaand ← s.POP()
13:                     if symbool en voorgaand niet corresponderend then
14:                         PRINT( "Fout symbool:", symbool)
15:                     end if
16:                 end if
17:             end if
18:         end if
19:     end for
20:     if not s.EMPTY() then
21:         PRINT("Te veel open symbolen." )
22:     end if
23: end function
```

Bijvoorbeeld:

$$3\ 4\ \times .$$

De binaire vermenigvuldigingsoperator staat na zijn operanden en werkt in op de twee argumenten ervoor nl. 3 en 4. De betekenis van deze postfix-uitdrukking is dus 3×4 en de waarde ervan is 12.

Verder moet bij het verwerken van rekenkundige uitdrukkingen eveneens rekening gehouden worden met de prioriteitsregels. Zo moet de vermenigvuldiging uitgevoerd worden vóór de optelling, tenzij haakjes anders aangeven. Ook dit moet correct verwerkt worden door de compiler. De waarde

Tabel 2.1: Infix-notatie versus postfix-notatie. Haakjes zijn overbodig in de postfix-notatie.

Infix-notatie	Postfix-notatie
2×3	$2\ 3\ \times$
$2 \times 3 + 5 = (2 \times 3) + 5$	$2\ 3\ \times\ 5\ +$
$2 \times 3 + 5/7$	$2\ 3\ \times\ 5\ 7\ /\ +$
$2 \times 3/5 \times 7$	$2\ 3\ \times\ 5\ /\ 7\ \times$
$2 + 3 \times 5 + 7$	$2\ 3\ 5\ \times\ +7\ +$
$(2 + 3) \times (5 + 7)$	$2\ 3\ +\ 5\ 7\ +\ \times$
$(2 + (3 - 5) \times 4)/((6 - 7) \times 9)$	$2\ 3\ 5\ -\ 4\ \times\ +6\ 7\ -\ 9\ \times\ /$

van

$$1 + 2 \times 3$$

is m.a.w. 7 en niet 9. Om de waarde 9 te bekomen schrijft men

$$(1 + 2) \times 3.$$

Hierbij werden haakjes gebruikt die aangeven dat de optelling eerst moet worden uitgevoerd.

Standaard wordt er gewerkt volgens de volgende afspraken:

- de operatoren $\times$ en / hebben een hogere prioriteit dan $+$ en $-$;
- prioriteiten kunnen door gebruik van haakjes (in de meeste programmeertalen zijn alleen ronde haakjes hiervoor toegelaten) worden aangepast;
- de prioriteit van $\times$ en / is gelijk. Als ze beide in een uitdrukking staan, worden ze uitgevoerd in de volgorde waarin je ze tegenkomt, dus van links naar rechts. Hetzelfde geldt voor $+$ en $-$.

In de postfix-notatie wordt het gebruik van haakjes overbodig. Dit wordt geïllustreerd in Tabel 2.1.

Voor de evaluatie van een postfix-uitdrukking hoeven eveneens geen prioriteitsregels meer in acht genomen te worden. De uitdrukking wordt steeds

eenvoudigweg van links naar rechts doorlopen. Wanneer een binaire operator ontmoet wordt, wordt deze uitgevoerd op de beide voorgaande operanden. In de postfix-uitdrukking wordt vervolgens de operator samen met zijn operanden vervangen door het resultaat van de bewerking.

Het bepalen van de waarde van een postfix-uitdrukking

Een machine kan de waarde van een postfix-uitdrukking gemakkelijk bepalen met behulp van een stapel.

Stel dat we een postfix-uitdrukking invoeren als een reeks van symbolen. De berekening van de postfix-uitdrukking met behulp van een stapel gaat als volgt:

- De uitdrukking wordt van links naar rechts doorlopen.
- Als een operand (i.e. een getal) wordt ontmoet dan wordt dit op de stapel geplaatst.
- Als een (binaire) operator wordt ontmoet dan worden de twee laatst gestapelde operanden van de stapel gehaald en de bewerking met die twee operanden wordt uitgevoerd. Het resultaat van de bewerking wordt op de stapel geplaatst.
- Deze stappen worden herhaald totdat de volledige uitdrukking is doorlopen.
- Als de volledige uitdrukking is doorlopen bestaat de stapel nog slechts uit één element, nl. de waarde van de postfix-uitdrukking.

Dit algoritme is uitvoerbaar in lineaire tijd, aangezien voor elk symbool uit de uitdrukking maximaal een constant aantal stapel-bewerkingen moet uitgevoerd worden, waarbij elke stapel-bewerking constante tijd vraagt. Een implementatie van dit algoritme wordt in de oefeningen uitgewerkt.

Toch worden in de meeste programmeertalen rekenkundige uitdrukkingen in infix-notatie geschreven omdat dit voor de programmeur handiger is. Gelukkig bestaat er een eenvoudig algoritme om een infix-uitdrukking om te zetten naar een equivalente postfix-uitdrukking.

Omzetten van een infix-uitdrukking naar postfix-notatie

De conversie van een (gewone) infix-uitdrukking als $3 + 4 \times 5$ naar zijn overeenkomstige postfix-uitdrukking $3\ 4\ 5\ \times +$ kan eveneens met behulp van een stapel gebeuren.

Uit de voorbeelden van Tabel 2.1 leren we dat in een postfix-uitdrukking de operanden steeds in precies dezelfde volgorde staan als in de equivalente infix-uitdrukking. De volgorde van de operatoren kan echter wijzigen.

Voor de conversie van infix naar postfix moeten de volgende stappen ondernomen worden:

1. Lees de invoertekst van links naar rechts.
2. Als een operand wordt gelezen, dan wordt deze rechtstreeks naar de uitvoer geschreven.
3. Als een operator of een haakje wordt gelezen, dan wordt die tijdelijk op de stapel bewaard als één van de volgende situaties zich voordoet:
   - de stapel is leeg;
   - de gelezen operator heeft een hogere prioriteit dan de operator die bovenaan de stapel ligt;
   - het gelezen haakje is een openend haakje. Verder wordt een openend haakje beschouwd als een operator van de laagste prioriteit, elke andere operator kan er dus bovenop gelegd worden.

   Als de gelezen operator gelijke of lagere prioriteit heeft dan de operator op de top van de stapel, dan worden alle operatoren van de stapel met gelijke of hogere prioriteit van de stapel gehaald en worden deze toegevoegd aan de uitvoer. Op dit moment is de stapel ofwel leeg of bevat de top van de stapel een operator met lagere prioriteit. Vervolgens wordt de ingelezen operator op de stapel geplaatst.

   Als een sluitend haakje wordt ingelezen dan worden alle operatoren van de stapel gehaald en toegevoegd aan de uitvoer totdat een openhaakje wordt bereikt. Het haakje wordt eveneens van de stapel gehaald maar niet aan de uitvoer toegevoegd.
4. Als het einde van de invoertekst bereikt is, worden alle operatoren van de stapel gehaald en aan de uitvoer toegevoegd totdat de stapel leeg is.

We illustreren deze methode a.d.h.v. een voorbeeld.

Voorbeeld

Beschouw de infix-uitdrukking:

$$a + (b + c) \times d \times (e + f \times g).$$

Om deze infix-notatie om te zetten naar de overeenkomstige postfix-notatie moet de invoertekst van links naar rechts gelezen worden.

Er wordt een lege stapel *stack* aangemaakt om de operanden, die niet onmiddellijk naar de uitvoer mogen, tijdelijk in op te slaan.

Het eerste ingelezen symbool is a. Dit is een operand dus a mag rechtstreeks naar de uitvoer.
Het volgende symbool $+$ is een operator. Bij aanvang is de stapel leeg, dus komt de operator op de stapel te liggen.
De toestand van de stapel en de uitvoertekst is op dit moment:

stack	Uitvoertekst
$+$	a

Na het symbool $+$ volgt een haakje. Een openend haakje wordt steeds op de stapel geplaatst. Na het haakje volgt de operand b. Een operand gaat onmiddellijk naar de uitvoertekst.

stack	Uitvoertekst
(	
$+$	$a\ \ b$

Het volgende symbool in de invoertekst is $+$. Deze operator heeft een hogere prioriteit dan het symbool (dat bovenaan de stapel ligt, dus $+$ wordt op de stapel geplaatst. Daarna komt de operand c die rechtstreeks naar de uitvoertekst gaat.

stack	Uitvoertekst
$+$	
(	
$+$	$a\ \ b\ \ c$

Het volgende symbool is). Als een sluitend haakje wordt bereikt, worden alle operatoren van de stapel gehaald totdat een openend haakje wordt

bereikt. In dit geval zal de + die overeenkomt met de top van de stapel verwijderd worden en aan de uitvoertekst toegevoegd worden.
Vervolgens wordt het openend haakje van de stapel gehaald. Dit haakje wordt niet aan de uitvoertekst toegevoegd. De toestand van de stapel en uitvoertekst ziet er nu als volgt uit:

stack	Uitvoertekst
$+$	$a\ b\ c\ +$

Het volgend symbool is $\times$. De prioriteit van $\times$ is hoger dan deze van $+$ dus $\times$ wordt op de stapel geplaatst. Hierna volgt de operand d die onmiddellijk naar de uitvoertekst wordt geschreven.
De toestand is nu:

stack	Uitvoertekst
$\times$	
$+$	$a\ b\ c\ +\ d$

Het symbool $\times$ wordt ingelezen. Het topelement $\times$ van de stapel heeft gelijke prioriteit als het ingelezen symbool. Dit betekent dat alle bewerkingen met gelijke of hogere prioriteit dan het ingelezen symbool van de stapel worden verwijderd en aan de uitvoertekst worden toegevoegd. In dit geval is dit enkel voldaan voor het topelement. Vervolgens wordt het ingelezen symbool, $\times$, op de stapel geplaatst.

stack	Uitvoertekst
$\times$	
$+$	$a\ b\ c\ +\ d\ \times$

In de invoertekst is het volgende symbool een openend haakje. Een openend haakje wordt altijd op de stapel geplaatst. Hierna volgt de operand e, deze wordt onmiddellijk aan de uitvoertekst toegevoegd.
Het volgende symbool $+$ heeft hogere prioriteit dan $($ en mag dus op de stapel geplaatst worden.
De toestand is op dit moment:

stack	Uitvoertekst
$+$	
$($	
$\times$	
$+$	$a\ b\ c\ +\ d\ \times\ e$

In de invoertekst volgt de operand f. Deze wordt onmiddellijk aan de uitvoertekst toegevoegd.
Het volgende symbool de operator $\times$ heeft hogere prioriteit dan het topelement van de stapel. De operator $\times$ wordt op de stapel geplaatst.
Vervolgens wordt de operand g aan de uitvoertekst toegevoegd.
Op dit moment is de toestand van de stapel en de uitvoertekst:

stack	Uitvoertekst
$\times$	
$+$	
$($	
$\times$	
$+$	$a\ b\ c\ +\ d\ \times\ e\ f\ g$

Als laatste symbool wordt een sluitend haakje ingelezen. Wanneer een sluitend haakje wordt ingelezen moeten alle operatoren van de stapel gehaald worden tot een openend haakje bereikt wordt. Deze operatoren worden aan de tekst toegevoegd. Dit betekent dat achtereenvolgens de operator $\times$ en de operator $+$ aan de uitvoertekst wordt toegevoegd. Het openend haakje wordt eveneens van de stapel gehaald maar dit symbool wordt niet aan de uitvoertekst toegevoegd.

stack	Uitvoertekst
$\times$	
$+$	$a\ b\ c\ +\ d\ \times\ e\ f\ g\ \times\ +$

We zijn aan het einde gekomen van de invoertekst. Alle symbolen nog aanwezig op *stack* worden nu van de stapel gehaald en één voor één toegevoegd aan de uitvoertekst. De postfix-uitdrukking is nu volledig opgebouwd:

stack	Uitvoertekst
	$a\ b\ c\ +\ d\ \times\ e\ f\ g\ \times\ +\ \times\ +$

Deze methode voor de conversie van infix-notatie naar postfix heeft een uitvoeringstijd die lineair is in n, met n de lengte van de uitdrukking.

De implementatie van dit algoritme wordt uitgewerkt in de oefeningen.

2.6 Oefeningen

1. Breid de klasse GelinkteLijst uit met een methode *size*. De methode heeft als resultaat het aantal elementen van een gelinkte lijst.

2. a) Pas in de klasse GelinkteLijst de constructor GelinkteLijst aan zodat de methode een lege gelinkte lijst met ankercomponent aanmaakt.

   b) Breid deze klasse uit met een methode invert. Deze methode heeft als resultaat een gelinkte lijst l_2. De gelinkte lijst l_2 bevat dezelfde elementen als een bestaande gelinkte lijst l_1 maar de elementen komen voor in omgekeerde volgorde. De gelinkte lijst l_1 is na afloop ongewijzigd.

3. Pas de klasse Knoop aan zodat met de objecten van deze klasse een dubbelgelinkte lijst kan aangemaakt worden. Noem de nieuwe klasse KnoopDubbel.
   Schrijf vervolgens voor dubbelgelinkte lijsten met twee ankercomponenten de basisfuncties van de klasse DubbelGelinkteLijst in pseudocode uit.

DubbelGelinkteLijst
- *eerste* : KnoopDubbel - *laatste*: KnoopDubbel
+ DubbelGelinkteLijst( ) + verwijder(*ref*: KnoopDubbel) : Element + voegToeVoor(*ref*: KnoopDubbel, *x*: Element) : / + voegToeNa(*ref*: KnoopDubbel, *x*: Element) : / + zoek(*x*: Element) : *ref*: KnoopDubbel

   De klasse KnoopDubbel wordt als inwendige klasse (inner class) van de klasse DubbelGelinkteLijst geïmplementeerd.

4. Een wachtrij is een FIFO-datastructuur. Het element dat het eerst op de wachtrij werd geplaatst is ook het eerste dat wordt verwijderd. Een wachtrij heeft m.a.w. twee uiteinden: een *kop* en een *staart*. Elementen worden toegevoegd aan de kant van de staart en verwijderd aan de kant van de kop.

a) Een wachtrij kan geïmplementeerd worden door twee referenties k en s. De knoop k refereert naar de kop van de wachtrij, de knoop s refereert naar de staart van de wachtrij.
Herschrijf alle basisbewerkingen voor een wachtrij corresponderend met deze implementatie. Deze zijn

- *Queue*( ): constructor, maakt een nieuwe wachtrij aan waarna de wachtrij bestaat als lege wachtrij;
- *empty*( ): controleert of een wachtrij al dan niet leeg is;
- *enqueue*( ): voegt een gegeven element toe aan de staart van een wachtrij;
- *dequeue*( ): verwijdert het element aan de kop in een wachtrij en retourneert het verwijderde element;
- *front*( ): retourneert het voorste element, m.a.w. de kop, van een wachtrij, zonder het te verwijderen.

b) Breid deze klasse uit met een methode *invert*. Deze methode plaatst de elementen van een bestaande wachtrij in omgekeerde volgorde.
Je algoritme mag geen nieuwe knopen alloceren. Je code mag tevens het veld 'data' van geen enkele knoop wijzigen.

5. Implementeer de methodes om

- een postfix uitdrukking te evalueren
- een infix uitdrukking om te zetten naar een postfix uitdrukking
- een infix uitdrukking te evalueren. Dit is dan een soort van eenvoudige rekenmachine.

HOOFDSTUK 3

Hashtabellen

In dit relatief korte hoofdstuk bespreken we HASHTABELLEN, een datastructuur die in veel programmeertalen wordt gebruikt om een *map*, *dict* of *associatieve array* te implementeren. We introduceren de ideeën van een HASHCODE, die willekeurige objecten afbeeldt op (mogelijks zeer grote) getallen en een HASHFUNCTIE die de waarde van de hashcode reduceert tot een vooraf bepaald bereik. Mogelijke OVERLAPPINGEN of BOTSINGEN kunnen worden verwerkt m.b.v. GESLOTEN of OPEN HASHING. We eindigen dit hoofdstuk met enkele raadgevingen voor een goede keuze van hashcode en hashfunctie.

Een element opzoeken in een gelinkte lijst vraagt in het algemeen nog steeds lineaire tijd. De overige basisfuncties zijn uitvoerbaar in een constante tijd, op voorwaarde dat men reeds de positie heeft bepaald en de referentie hiernaar kent.

Wanneer we echter bv. een digitaal woordenboek wensen te implementeren dan is het vooral belangrijk dat het opzoeken van elementen efficiënt gebeurt, aangezien een woordenboek voornamelijk wordt gebruikt voor het opzoeken van woorden.
Voor dergelijke toepassingen bieden *hashtabellen* een oplossing. Typisch zal in een hashtabel het opzoeken, toevoegen en het verwijderen van elementen gemiddeld in constante tijd gebeuren. Een hashtabel is echter *niet* in staat om de elementen gesorteerd terug te geven, wat bv. bij een binaire zoekboom (zie Sectie 4.4) wel het geval is.

Voor de eenvoud veronderstellen we in dit hoofdstuk dat we een *set*-interface

wensen te implementeren, waarbij we enkel *sleutels* opslaan, eventuele bijhorende *waarden* worden in deze tekst niet vermeld. De besproken technieken kunnen echter eenvoudig worden aangepast om ook met zo'n additionele informatie rekening te houden.

3.1 Hashtabellen

Veronderstel, bij wijze van eenvoudig voorbeeld, dat we een verzameling willen implementeren waarbij de sleutelwaarden de natuurlijke getallen tussen 0 en 999 zijn. In dit geval kan men eenvoudig weg een array a van grootte 1000 alloceren, en we spreken af dat

$$a[i] = i$$

enkel en alleen als het getal i tot de verzameling hoort. We gebruiken een bijzondere waarde (bv. "null" of -1) om aan te geven dat i niet tot de verzameling behoort. In dit geval kunnen de drie basisbewerkingen, nl. toevoegen, opzoeken en verwijderen, in constante tijd uitgevoerd worden. Het gebruik van een array op deze manier noemen we *directe adressering*.

Voorbeeld 3.1 (Directe adressering) Veronderstel dat de elementen in de verzameling natuurlijke getallen zijn tussen 0 en 9. Dan komt de array

Index	0	1	2	3	4	5	6	7	8	9
Sleutel	-1	1	-1	3	-1	-1	-1	7	8	9

overeen met de verzameling $\{1, 3, 7, 8, 9\}$. ∎

In veel gevallen zijn de objecten die we willen opslaan echter geen natuurlijke getallen. Veronderstel bv. dat de mogelijke objecten die we willen opslaan strings zijn van precies twee karakters lang, m.a.w. strings van de vorm "aa" t.e.m. "zz". Dit zijn geen natuurlijke getallen. Het is echter wel eenvoudig een afbeelding te construeren die iedere tweeletterige string afbeeldt op een uniek natuurlijk getal tussen 0 en 675. Hiertoe laten we de letter 'a' overeenkomen met de waarde 0, 'b' met de waarde 1 enzovoort tot en met de letter 'z' die de waarde 25 krijgt. Daarna interpreteren we deze twee cijfers in basis 26. De index voor het woord "je" is dan gelijk aan

$$9 \times 26^1 + 4 \times 26^0 = 238.$$

Deelverzamelingen van alle tweeletterige strings kunnen m.a.w. eenvoudig geïmplementeerd worden m.b.v. een array van lengte $26 \times 26 = 676$. Opnieuw moet afgesproken worden welke speciale waarde wordt gebruikt om aan te geven dat een element niet tot de verzameling behoort.

Veronderstel vervolgens dat we een (deel)verzameling van woorden met een willekeurige lengte willen opslaan. Het is nog steeds mogelijk om op een analoge manier als hierboven elk woord te associëren met een uniek natuurlijk getal. Het woord "hashtabel" krijgt bv. de volgende waarde:

$$7 \times 26^8 + 0 \times 26^7 + 18 \times 26^6 + 7 \times 26^5 + 19 \times 26^4 \\ + 0 \times 26^3 + 1 \times 26^2 + 4 \times 26^1 + 11 \times 26^0 = 1467\,441\,788\,967 \approx 1.46 \times 10^{12}.$$

Hier ziet men onmiddellijk dat de getallen die hier gegenereerd worden veel te groot zijn. Het is immers in de praktijk onmogelijk om arrays met miljarden elementen te alloceren, en zelfs als het zou kunnen dan nog zou het overgrote deel van de array bestaan uit elementen "null"of iets dergelijks. De meeste combinaties van letters zijn immers geen geldige woorden.

In plaats van het alloceren van een ondoenbaar grote array beperken we ons tot een array van een bepaalde vooraf gekozen en doenbare grootte N. Om de positie van een woord in deze tabel te bepalen berekenen we eerst zijn getalwaarde zoals hierboven geïllustreerd. Meer in het algemeen gaan we er van uit dat de objecten die we willen opslaan beschikken over een methode om hun HASHCODE te berekenen. De hashcode beeldt objecten af op unieke (en mogelijks zeer grote) getallen.

Om die hashcodes dan te transformeren naar het bereik tussen 0 en $N - 1$ gebruikt men een HASHFUNCTIE. Een zeer vaak gebruikte hashfunctie is het werken met modulo N, i.e. men neemt de rest na deling door N. Formeel uitgedrukt, waarbij w een woord voorstelt, vinden we

$$h(w) = w.\text{hashCode}(\,) \pmod N.$$

Deze hashfunctie zorgt er inderdaad voor dat we voor elk woord (of object) w een positie bepalen in de array.

Aangezien er meer hashcodes zijn dan posities in de tabel is het niet uitgesloten dat een aantal woorden op dezelfde positie moeten opgeslagen worden. De opbouw van de *hashtabel* zal afhangen van de manier waarop wordt omgegaan met deze *botsingen*.

Algemeen kunnen we stellen dat hashing kan opgesplitst worden in twee luiken:

1. De keuze van een hashfunctie h die alle mogelijke items afbeeldt op een positie uit de hashtabel.
2. Het selecteren van een methode om de waarden die overlappen of botsen (*collisions*) te verwerken.

3.2 Verwerken van de overlappingen

Er zijn verschillende oplossingsmethodes voor het opvangen van de *collisions*. We bespreken twee methodes.

De eerste methode is een voorbeeld van *gesloten hashing*. In dit geval wordt er in de tabel zelf gezocht naar een goede positie om een element, dat botst met een ander element uit de tabel, op te slaan.

De tweede methode is een voorbeeld van *open hashing*. In dit geval wordt er buiten de tabel gezocht naar een plaats om de overlappende elementen op te slaan.

3.2.1 Gesloten hashing

In het geval van GESLOTEN HASHING worden alle woorden in de array zelf opgeslagen. Er doet zich een BOTSING voor wanneer de positie die correspondeert met een bepaald woord reeds ingenomen is, of anders gezegd $h(w)$ is reeds bezet. Het woord w kan dan niet op die positie opgeslagen worden zonder informatie te verliezen, en er moet gezocht worden naar een alternatieve positie om het woord w op te slaan. Als alternatieve positie om het woord op te slaan kan gekozen worden voor de eerstvolgende vrije positie in de tabel, waarbij de tabel als een circulaire structuur wordt opgevat. Dit noemt men LINEAIRE PEILING.

Voorbeeld 3.2 (Toevoegen elementen lineaire peiling) We bouwen een hashtabel op van lengte $N = 10$. In de tabel moeten een aantal gehele getallen opgeslagen worden. Als bijhorende hashcode kiezen we voor het getal zelf. Als hashfunctie kiezen we voor

$$h(w) = w \pmod{N}.$$

We voegen één voor één de elementen 10, 15, 29, 100, 115 en 129 toe. Indien een positie reeds ingenomen is, zoeken we naar de eerste vrije positie om het nieuwe element op te slaan. Hierbij beschouwen we de tabel als een circulaire structuur: de eerste positie volgt op de laatste positie.

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	opmerkingen:
10										waarde 10 toegevoegd op positie 0
10					15					waarde 15 toegevoegd op positie 5
10					15				29	waarde 29 toegevoegd op de positie 9
10	100				15				29	100 schuift door naar eerste vrije positie
10	100				15	115			29	115 schuift door naar eerste vrije positie
10	100	129			15	115			29	129 schuift door naar eerste vrije positie

Men kan in een hashtabel met lineaire peiling ook eenvoudig zoeken. Men berekent de waarde van de hashfunctie voor het gezochte element. Wanneer die positie leeg is dan weet men dat het element niet voorkomt. Wanneer die positie bezet is vergelijkt men het element op deze positie met het gezochte element. Wanneer deze twee elementen gelijk zijn dan heeft men het element gevonden en kan het zoekproces stoppen. Wanneer deze twee elementen echter verschillend zijn dan bekijkt men de volgende positie in de array. Dit proces herhaalt zich tot men ofwel het gezochte element ontmoet (het element is dan gevonden), of totdat men een lege positie in de array tegenkomt (het gezochte element behoort dan niet tot de hashtabel). Wanneer alle posities bezet zijn zou men ook kunnen terugkeren naar de eerste bezochte positie. In dit geval moet men ook stoppen, en men weet dan dat het gezochte element niet aanwezig is.

Voorbeeld 3.3 (Opzoeken in hashtabel opgebouwd met lineaire peiling) De waarde 10 opzoeken gaat relatief snel. Gelet op de gekozen hashfunctie zou 10 op de 0-de positie moeten staan. Na een eenvoudige controle vinden we 10 effectief terug op de 0-de positie in de tabel. We mogen besluiten dat 10 voorkomt in de tabel op de 0-de positie.
De waarde 115 opzoeken gaat iets moeizamer. De hashfunctie beeldt 115 af op 5. Dus 115 zou op de 5-de positie moeten opgeslagen worden. Wanneer we dit controleren vinden we echter niet 115 maar 15 op de 5-de positie. Dit wil niet noodzakelijk zeggen dat 115 niet voorkomt in de tabel. Aangezien bij een botsing een element wordt opgeslagen op de eerstvolgende vrije po-

sitie, is het perfect mogelijk dat 115 op de volgende, dus de 6-de positie, staat. Dit is inderdaad het geval.
Als laatste voorbeeld gaan we in de tabel op zoek naar de waarde 149. We verwachten 149 terug te vinden op de 9-de positie. Op de 9-de positie staat echter 29. Het is mogelijk dat er sprake was van een botsing waardoor 149 is moeten doorschuiven naar de eerstvolgende vrije positie. Wanneer we de volgende positie, d.i. de 0-de positie, bekijken vinden we opnieuw niet 149 maar wel 10. Analoog voor de eerste positie en de tweede positie. Dit betekent dat we 149 nog steeds niet gevonden hebben en de derde positie is vrij. Aangezien de derde positie vrij is, zal 149 niet voorkomen in de tabel. We hebben immers op geen enkele positie volgend op de 9-de positie 149 teruggevonden. ∎

Uit dit voorbeeld blijkt dat het efficiënt opzoeken van een willekeurig element in een dergelijke tabel afhankelijk is van het aantal lege posities in de tabel. Wanneer het aantal elementen in de tabel toeneemt hebben deze bij lineaire peiling de neiging om zich te groeperen. Dit fenomeen wordt *primaire clustering* genoemd.

In een dergelijke tabel kan ook niet zomaar een element verwijderd worden. Dit zou immers een vrije positie creëeren op een plaats die mogelijks deel uitmaakt van een sequentie van posities die noodzakelijk is om een bepaalde elementen te kunnen bereiken vanaf de positie aangeduid door de waarde van hun hashfunctie.

Voorbeeld 3.4 (Verwijderen uit hashtabel met lineaire peiling) We hernemen even het voorbeeld. Stel dat we het element 10 uit de tabel zouden verwijderen dan wordt de 0-de positie leeg. Wanneer we dan vervolgens op zoek zouden gaan naar 100 dan lijkt het alsof 100 niet voorkomt in de tabel aangezien de 0-de positie leeg is. ∎

Om dergelijke fouten te vermijden mag een element niet effectief verwijderd worden. Een mogelijke oplossing is om bij het element een *vlag* te plaatsen die bijhoudt dat het element verwijderd is.
Indien we deze werkwijze toepassen in de tabel zal na het verwijderen van de waarde 10 de 0-de positie niet effectief leeg zijn. Hierdoor leiden we af dat de mogelijkheid bestaat dat 100 op een volgende positie is opgeslagen, zoals inderdaad het geval is.

3.2.2 Open hashing

In het geval van OPEN HASHING wordt er buiten de tabel gezocht naar een goede positie om elementen op te slaan. Een mogelijke oplossing is om alle elementen die op dezelfde positie moeten staan in de hashtabel op te slaan in een gelinkte lijst. Anders gezegd, elementen met dezelfde waarde voor de hashfunctie worden geschakeld in dezelfde lineair gelinkte lijst.

Wanneer gelinkte lijsten worden gebruikt om de overlappingen op te vangen bestaat een hashtabel uit een array van lengte N en op elke positie van de array wordt een referentie opgeslagen naar de gelinkte lijst waarin alle waarden worden opgeslagen die afgebeeld worden op die positie.

Wanneer we op zoek willen gaan naar de definitie van een woord[1] in een dergelijke tabel vinden we een gelinkte lijst op de positie die correspondeert met dat specifiek woord. Om te kunnen achterhalen welke knoop van de gelinkte lijst bij welk woord hoort, zal in de knopen van de gelinkte lijst naast de definitie ook typisch het woord zelf moeten bijgehouden worden. Figuur 3.1 is een illustratie van een dergelijke hashtabel.

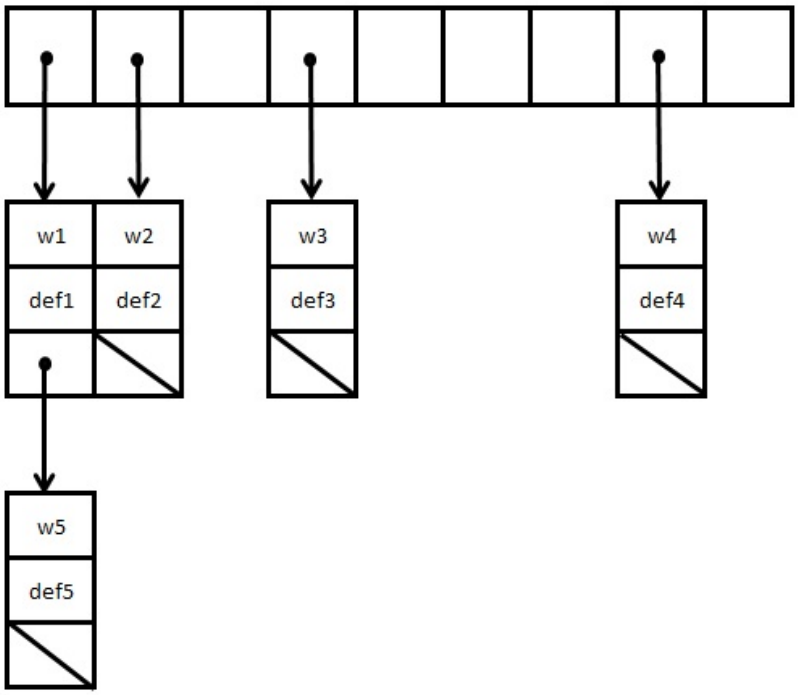
Stappen bij implementatie van open hashing

Veronderstel dat we een woordenboek willen opbouwen waarbij we met een groot aantal woorden w een definitie willen associëren.

In het geval van open hashing kunnen de basisfuncties voor een hashtabel als volgt beschreven worden:

- *hashFunctie*(w) voert achtereenvolgens uit:
  - bereken de hashcode van het woord w;
  - bereken de bijhorende positie;
  - geef de berekende positie terug.
- *voegToe*$(w, definitie)$ voert achtereenvolgens uit:
  - bereken de juiste positie voor het woord w met de hashfunctie;
  - maak een nieuwe knoop aan met drie velden:
    - een veld voor het woord,

[1] Dit is een voorbeeld van de *map* interface



Figuur 3.1: Een voorbeeld van een hashtabel: open hashing.

    - een veld voor de definitie,
    - een veld volgende;
  - voeg de knoop toe aan de gelinkte lijst die start op de gevonden positie. De plaats van de knoop in deze gelinkte lijst hangt af van de implementatie, zo kan de knoop altijd vooraan worden toegevoegd of er kan voor gekozen worden om de gelinkte lijst steeds gesorteerd te houden.

- *zoekOp*(w) werkt als volgt:
  - bereken de juiste positie van het woord w met de hashfunctie;
  - zoek in de gelinkte lijst, waar de gevonden positie naar refereert, naar de knoop met het woord w;

  - geef de definitie van die knoop terug of geef aan dat het gegeven woord niet tot de hashtabel behoort.

- *verwijder*(w) werkt als volgt:
  - bereken de juiste positie van het woord w met de hashfunctie;
  - zoek in de gelinkte lijst, waar de gevonden positie naar refereert, naar de knoop met het woord w;
  - verwijder de gevonden knoop uit de ketting;
  - geef de verwijderde definitie terug.

Om een efficiënte hashtabel op te bouwen is het niet alleen belangrijk dat de overlappingen goed worden opgevangen maar is het vooral belangrijk een efficiënte hashcode en hashfunctie te kiezen.

3.3 Keuze van hashcode en hashfunctie

Een goede combinatie van hashcode en hashfunctie moet aan twee voorwaarden voldoen. Ten eerste moeten ze snel te berekenen zijn en ten tweede moeten ze zorgen voor een zo groot mogelijke spreiding van de elementen over de verschillende posities. Dit lijkt voor de hand liggend maar het gebeurt in de praktijk heel vaak dat er tegen de tweede voorwaarde wordt gezondigd. Dit leidt dan tot een teleurstellende performantie.

Voorbeeld 3.5 (Problematische hashcode) Veronderstel dat we hashcode voor een string definiëren als de som van de waarden van de verschillende letters. Bv. het woord "hashtabel" zou in dit geval de volgende waarde krijgen:

$$\begin{aligned} &7 + 0 + 18 + 7 + 19 \\ &\quad + 0 + 1 + 4 + 11 = 67. \end{aligned}$$

Deze hashcode heeft twee problemen. Ten eerste zijn de waarden die nu worden gegenereerd (te) klein. Als we ons bv. beperken tot woorden die hoogstens 10 karakters lang zijn dan is de maximumwaarde $25 \times 10 = 250$. Zelfs als we een array alloceren met 10 000 posities dan zullen enkel de eerste 250 posities gebruikt kunnen worden. Ten tweede worden alle permutaties van dezelfde letters op dezelfde hashcode afgebeeld, zo zal het Engelstalige woord "hashtable" ook hashcode 67 hebben. Dit zorgt voor onnodig veel botsingen. ∎

Ook de grootte van de hashtabel, i.e. de waarde van N kan een grote invloed hebben op de performantie van de hashtabel.

Voorbeeld 3.6 (Problematische tabelgrootte) Veronderstel dat we gegevens over studenten aan HOGENT willen opslaan in een hashtabel. Als sleutelwaarde gebruiken we een uniek studentennummer voor elke student. Veronderstel dat dit studentennummer als volgt is opgebouwd. Eerst komen er 4 cijfers (een volgnummer) en daarna komt het jaar waarin deze student zich voor het eerst inschreef[2]. We gebruiken deze nummers onmiddellijk ook als hashcode. Het nummer 1234-2019 correspondeert met een student die zich voor de eerste keer inschreef in 2019. Het nummer 5467-2020 correspondeert met een student die zich voor de eerste keer inschreef in 2020. Volgens deze werkwijze kan er jaarlijks aan maximaal $10^4 = 10\,000$ studenten een nummer toegekend worden. We veronderstellen dat dit voldoende is aangezien er zich gemiddeld 5000 nieuwe studenten inschrijven[3].

We wensen gegevens bij te houden voor de laatste 10 jaar, dus van ongeveer 50 000 studenten, en we gebruiken hiervoor een hashtabel met open hashing. Wanneer de grootte van deze hashtabel gelijk aan 10 000 wordt gekozen dan zou, bij een ideale verdeling, elke lineair gelinkte lijst ongeveer 5 elementen bevatten.

Echter, door de gekozen tabelgrootte bestaat de hashfunctie uit het werken modulo 10 000 en krijgen we het volgende:

$$\begin{aligned} h(\text{1234-2019}) &= 2019 \\ h(\text{5489-2019}) &= 2019 \\ h(\text{0358-2019}) &= 2019 \\ h(\text{9786-2020}) &= 2020. \end{aligned}$$

Dit betekent dat alle studenten die gestart zijn in hetzelfde academiejaar op dezelfde positie terechtkomen in de tabel. Al deze overlappingen worden bijgehouden in een gelinkte lijst, maar deze gelinkte lijst zal wel bijzonder lang worden. Deze gelinkte lijst zal in ons geval dus zo'n 5000 studenten bevatten. Het probleem is hier dat we de tabelgrootte op zo'n manier hebben gekozen dat een aantal posities van de hashcode (het studentennummer in ons geval) niet relevant zijn voor het bepalen van de positie.

[2] Een echt studentennummer is iets anders opgebouwd.

[3] Fictieve cijfers.

Dit probleem kan vermeden worden door niet modulo 10 000 te rekenen maar door te werken modulo een priemgetal. Het liefst een priemgetal dichtbij de gewenste grootte. Hoe groter hoe meer verschillende beeldwaarden de hashfunctie zal hebben.
Aangezien er gekozen is voor een tabel van lengte $\pm 10\,000$ moet het priemgetal voor de opbouw van de hashfunctie zo dicht mogelijk bij 10 000 liggen. Het getal 10 007 is het priemgetal het dichtste gelegen bij 10 000. Wanneer we elke hashcode uitrekenen (mod 10 007) dan zijn er 10 007 verschillende antwoorden mogelijk. De hashfunctie wordt nu gegeven door

$$h : \textit{student} \mapsto \textit{student}.\text{hashcode } (\text{mod } 10\,007).$$

Deze hashfunctie zorgt ervoor dat de verschillende studenten beter verdeeld worden over alle posities van de tabel. Bijvoorbeeld:

$$\begin{aligned} h(\text{1234-2019}) &= 3388 \\ h(\text{5489-2019}) &= 3624 \\ h(\text{0358-2019}) &= 9520 \\ h(\text{9786-2020}) &= 3567. \end{aligned}$$

De studentengegevens worden op de corresponderende positie ingevuld in de tabel. ∎

3.4 Oefeningen

1. Voeg alle (verschillende) letters uit het woord 'DEMOCRATISCH' toe aan een hashtabel. De grootte van de hashtabel wordt bepaald door $N = 5$, dus er zijn vijf plaatsen (buckets) in de hashtabel. In de hashtabel wordt er geen waarde opgeslagen corresponderend met de letters. De te gebruiken hashcode is $11 \times k$ met k de positie van de letter in het alfabet.

   Hoe evolueert de hashtabel?

2. Beschouw het volgende probleem: gegeven een array a bestaande uit gehele getallen en een geheel getal t. Retourneer twee verschillende indices i en j zodanig dat $a_i + a_j = t$.

   a) De meest voor de hand liggende oplossing is om alle koppels (i, j) met $i < j$ te overlopen en te verifiëren of $a_i + a_j = t$. Schrijf deze oplossing uit in pseudocode en analyseer de tijdscomplexiteit.

b) Op welke manier kan je gebruikmaken van een hashtabel om deze tijdscomplexiteit te verbeteren? Je mag er hierbij van uitgaan dat toevoegen aan en opzoeken in de hashtabel kan gebeuren in constante tijd. Vergelijk de hoeveelheid extra geheugen die deze oplossing nodig heeft met de oplossing van vorig puntje.

HOOFDSTUK 4

Bomen

We starten met de recursieve definitie van GEWORTELDE BOMEN samen met hun basisbegrippen. We bespreken kort twee DATASTRUCTUREN om zo'n bomen voor te stellen in het computergeheugen. Aangezien gewortelde bomen op een natuurlijke wijze recursief gedefinieerd zijn volgt hieruit dat een aantal bewerkingen en berekeningen ook gemakkelijk recursief kunnen geïmplementeerd worden. Vervolgens geven we de definitie van een BINAIRE BOOM samen met een aantal basisbewerkingen. BINAIRE ZOEKBOMEN laten, dankzij hun ordeningseigenschap, toe om (meestal) snel elementen op te zoeken, toe te voegen en te verwijderen. BINAIRE HOPEN worden gebruikt om op een efficiënte en eenvoudige manier een PRIORITEITSWACHTRIJ te implementeren.

4.1 Terminologie m.b.t. bomen

Er zijn veel situaties waarin informatie geordend is volgens één of andere hiërarchische structuur. Denken we bv. maar aan bestandsystemen, de structuur van XML-bestanden, familiestambomen, organisatorische structuren enzovoort. De abstractie die zulke situaties modelleert is een boom, een fundamenteel begrip in de informatica.

Aangezien bomen een recursieve structuur vertonen, is een *recursieve definitie* aangewezen.

Definitie 4.1 Een GEWORTELDE BOOM T is een verzameling van TOPPEN die aan de volgende eigenschappen voldoet:

1. Er is één speciale top t die de WORTEL van de boom wordt genoemd.
2. De andere toppen zijn verdeeld in $m \geq 0$ disjuncte verzamelingen $T_1, \ldots, T_m$ die op hun beurt elk weer een gewortelde boom zijn. ∎

Opmerking 4.2 Merk op dat één enkele top ook steeds een boom is aangezien het toegelaten is dat het aantal verzamelingen m nul is. Dit is het eindgeval van de recursie. ∎

We zeggen dat de bomen T_1 t.e.m. T_m de DEELBOMEN zijn van T. De wortels $t_1, \ldots, t_m$ van de deelbomen T_1 t.e.m. T_m worden de KINDEREN van de wortel t genoemd. Omgekeerd is t de OUDER van $t_1, \ldots, t_m$. De toppen $t_1, \ldots, t_m$ zijn BROERS van elkaar. De termen AFSTAMMELING en VOOROUDER zijn logische uitbreidingen van de kind/ouder terminologie. De afstammelingen van een top t vormen een verzameling die bestaat uit de kinderen van t, en alle afstammelingen van de kinderen. Een top t is een voorouder van een top t' als en slechts als t' een afstammeling is van t.

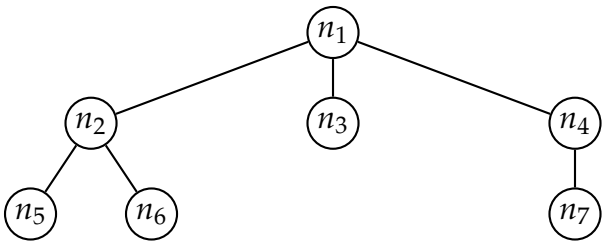
Uit de recursieve definitie van een gewortelde boom volgt dat elke top in de boom uiteindelijk de wortel is van een deelboom die bevat is in de boom. Het aantal kinderen van een top wordt de GRAAD van die top genoemd. Een BLAD is een top met graad nul. Een top die geen blad is wordt INTERN genoemd. De GRAAD van een *boom* wordt gedefinieerd als het maximum van de graden van zijn toppen.

Bomen hebben ook een grafische voorstelling. De toppen worden meestal als kleine cirkeltjes getekend, eventueel met een label erin om duidelijk te maken over welke top het gaat. De wortel van een boom wordt bovenaan getekend, en we tekenen een lijn tussen de wortel en zijn kinderen. Figuur 4.1 geeft een voorbeeld van een boom.

Voorbeeld 4.3 De wortel van de boom T in Figuur 4.1 is de top met label n_1. De andere toppen zijn verdeeld in 3 disjuncte verzamelingen, nl.

$$T_1 = \{n_2, n_5, n_6\}, \quad T_2 = \{n_3\} \quad \text{en} \quad T_3 = \{n_4, n_7\}.$$

De toppen n_2, n_3 en n_4, zijn de kinderen van de top n_1. Omgekeerd is n_1 dus de ouder van n_2, n_3 en n_4. De toppen n_2, n_3 en n_4 zijn broers van elkaar want ze hebben dezelfde ouder.



Figuur 4.1: Een eerste voorbeeld van een boom.

De verzameling T_1 is op zijn beurt weer een boom met de top n_2 als wortel. De toppen n_5 en n_6 zijn de kinderen van n_2. Omgekeerd is de top n_2 de ouder van n_5 en n_6. De toppen n_5 en n_6 zijn broers.

Aangezien n_5 en n_6 kinderen zijn van n_2, die op zijn beurt weer een kind is van n_1, zijn n_5 en n_6 afstammelingen van n_1. In dit voorbeeld bestaat de verzameling van de afstammelingen van n_1 uit alle toppen van T behalve n_1.

Uit de figuur lezen we ook gemakkelijk de graad van de verschillende toppen af. De graad van n_1 is 3, want n_1 heeft 3 kinderen. De graad van n_2 is 2, want deze top heeft 2 kinderen, enzovoort.

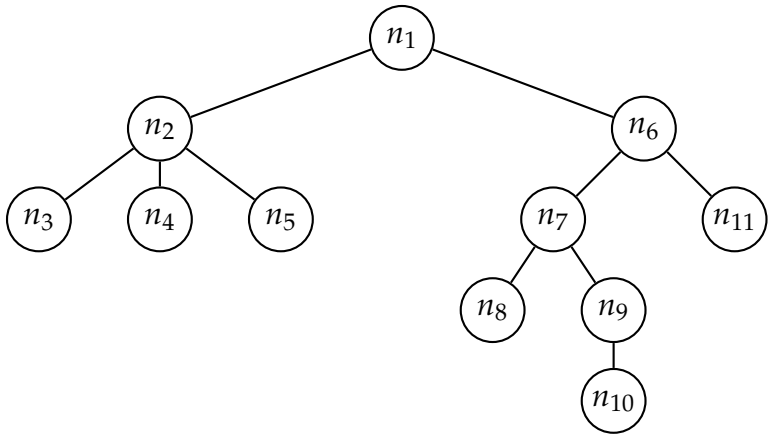
De bladeren van de boom zijn de toppen n_5, n_6, n_3 en n_7, want deze toppen hebben allen graad nul.

De graad van de boom is 3 want dit is het maximum van de graden van zijn toppen. Merk op dat in dit geval de graad van de boom gelijk is aan de graad van zijn wortel maar dat is zeker niet altijd het geval. ∎

De DIEPTE van een top n m.b.t. een boom T wordt als volgt (recursief) gedefinieerd: de diepte van de wortel van T is nul terwijl de diepte van elke andere top één meer is dan de diepte van zijn ouder. Op deze manier krijgt elke top een unieke diepte. De diepte van de *boom* T is de maximale diepte van zijn toppen. De HOOGTE van een top is de diepte van de deelboom met die top als wortel. De diepte van een top geeft m.a.w. de 'afstand' tot de wortel terwijl de hoogte de 'afstand' geeft tot het verste blad. De hoogte van een *boom* wordt gedefinieerd als de hoogte van zijn wortel. Het is eenvoudig in te zien dat de hoogte en de diepte van een *boom* steeds gelijk zijn aan elkaar.

top	diepte	hoogte
n_1	0	2
n_2	1	1
n_3	1	0
n_4	1	1
n_5	2	0
n_6	2	0
n_7	2	0

Tabel 4.1: Diepte en hoogte van de toppen van de boom in Figuur 4.1.



Figuur 4.2: Een voorbeeldboom.

Voorbeeld 4.4 In Tabel 4.1 staat voor elke top van de boom in Figuur 4.1 de diepte en de hoogte opgesomd. Men kan deze tabel gemakkelijk zelf verifiëren. Start met de wortel n_1 de diepte 0 toe te kennen. Zijn kinderen krijgen dan diepte 1. De kinderen van de kinderen krijgen dan diepte 2.

De diepte van de boom T is 2 aangezien dit de maximale diepte is van de toppen in de boom. De hoogte van de top n_1 is dus 2.

Als we de diepte van de boom T_1 bekijken, dan zien we dat daar de maximale diepte gelijk is aan 1, en dus is de hoogte van de top n_2 (de wortel van T_1) gelijk aan 1. ∎

4.1.1 Oefeningen

1. Bekijk de boom in Figuur 4.2. Beantwoord de volgende vragen:

   a) Geef de wortel van de boom.

   b) Geef de verzamelingen T_1 t.e.m. T_m volgens Definitie 4.1.

   c) Geef de kinderen van elke top in de boom.

   d) Geef de graad van elke top in de boom.

   e) Geef de graad van de boom T.

   f) Welke toppen zijn broers van elkaar?

   g) Geef de bladeren van de boom.

   h) Geef de afstammelingen van n_6.

   i) Geef de voorouders van n_{10}.

   j) Maak een tabel waarin voor elke top zijn hoogte en diepte wordt gegeven.

4.2 Datastructuren voor bomen

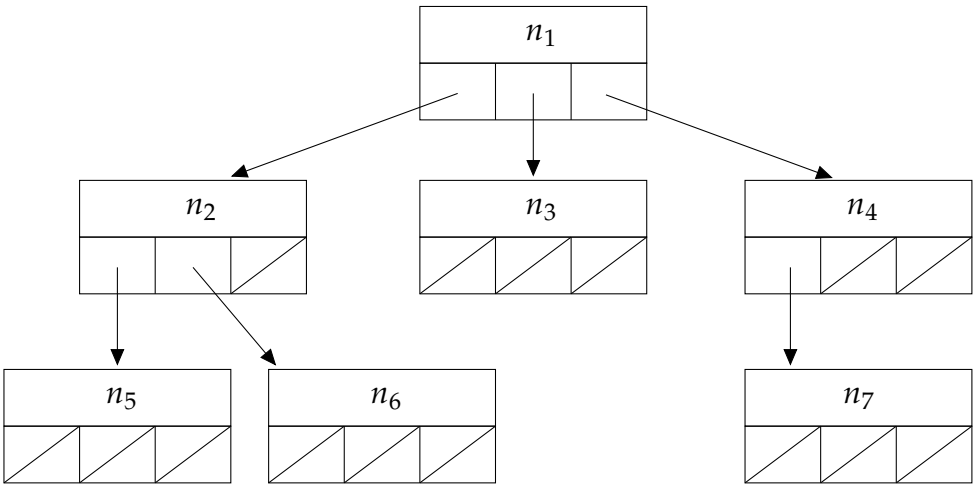
Er zijn verschillende manieren om bomen voor te stellen in het geheugen van een computer. Wij bespreken hier twee voorstellingen.

Opmerking 4.5 Wanneer men bomen voorstelt in het computergeheugen dan legt men vaak een (willekeurige) volgorde op aan de deelbomen, daar waar dat bij de wiskundige definitie niet het geval is. ∎

4.2.1 Array-van-kinderen voorstelling

De eenvoudigste manier om een boom voor te stellen is door rechtstreeks de vader-kind relatie te implementeren. Dit betekent dat we een structuur `Top` definiëren die een veld heeft om de data van de top bij te houden, alsook een array van referenties naar de kinderen van die top. Wanneer een kind niet bestaat dan wordt dit voorgesteld door de referentie `null`. De boom zelf bestaat uit een referentie naar zijn wortel.

Voorbeeld 4.6 In Figuur 4.3 staat een voorstelling getekend van hoe de boom in Figuur 4.1 zou kunnen opgeslaan worden in het computergeheugen. Aangezien de graad van de boom 3 is, zijn er in elke structuur voor een top drie



Figuur 4.3: Array-van-kinderen voorstelling voor de boom in Figuur 4.1. De referentie naar de wortel wordt niet explictiet getoond.

posities voorzien die een referentie naar een andere top kunnen bijhouden. Wanneer er geen kind is om naar te verwijzen dan wordt dit aangeduid door een schuine streep. ∎

We kunnen in Figuur 4.3 zien dat er veel referenties niet gebruikt worden. Inderdaad, we hebben geheugenruimte voorzien voor $7 \times 3 = 21$ referenties, waarvan er slechts 6 effectief gebruikt worden. Merk op dat het aantal gebruikte referenties juist één minder is dan het aantal toppen van de boom.

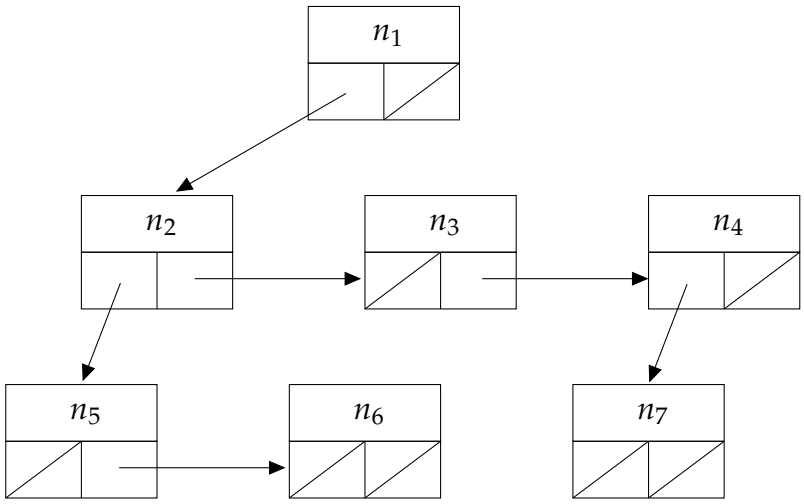
We kunnen de observatie uit de figuur gemakkelijk veralgemenen: inderdaad, stel dat n het aantal toppen van de boom T is en dat de graad van T gelijk is aan k. Wanneer we de array-van-kinderen voorstelling gebruiken dan zijn er in totaal $n \times k$ referenties. Echter, van deze referenties zijn er slechts $n - 1$ verschillend van `null` (want elke top, behalve de wortel, is kind van juist één andere top). Dit betekent dat slechts een fractie

$$\frac{n-1}{nk} \approx \frac{1}{k}$$

van de referenties verschillend is van `null` en dus effectief gebruikt wordt.

Wanneer bv. de graad van de boom gelijk is aan 3 wordt dus slechts ongeveer $1/3$ van de referenties gebruikt.

Een nadeel van de array-van-kinderen voorstelling is dus het inefficiënt gebruik van geheugen, zeker wanneer de graad van de boom groot is.



Figuur 4.4: De eerste-kind-volgende-broer voorstelling van de boom in Figuur 4.1.

4.2.2 Eerste-kind-volgende-broer voorstelling

We kunnen een boom voorstellen op een manier die efficiënter met het geheugen omgaat. In plaats van in elke top referenties naar al zijn kinderen op te slaan, houden we altijd juist twee referenties bij: een referentie naar zijn eerste kind, en een referentie naar zijn volgende broer.

Voorbeeld 4.7 In Figuur 4.4 is een voorstelling getekend van de eerste-kind-volgende-broer voorstelling van de voorbeeldboom in Figuur 4.1. De eerste referentie in de structuur die een top voorstelt is een referentie naar het "eerste" kind. De tweede referentie is er een naar de "volgende" broer van de top. ∎

Uit Figuur 4.4 leiden we af dat er nu heel wat minder `null` referenties zijn dan bij de array-van-kinderen voorstelling. Inderdaad, we hebben geheugenruimte voorzien voor $2 \times 7 = 14$ referenties waarvan er opnieuw 6 effectief gebruikt worden.

Met deze voorstelling is de referentie naar het eerste kind `null` als en slechts als de top een blad is. Wanneer de top geen blad is, dan zijn de kinderen van de top geschakeld in een lineaire lijst structuur. Dit betekent dat we hier dus geen rechtstreekse toegang hebben tot bv. het derde kind: om de kinderen van een bepaalde top te vinden moeten we in deze voorstelling dus eerst

de referentie volgen naar zijn eerste kind. Vervolgens moet er geïtereerd worden over de (gelinkte lijst) van volgende-broer referenties.

De eerste-kind-volgende-broer voorstelling is dus efficiënter qua geheugengebruik maar navigeren in de boom wordt iets moeilijker.

4.2.3 Oefeningen

1. Bereken hoeveel null-referenties er zullen zijn bij de array-van-kinderen voorstelling van de boom in Figuur 4.2. Wat is de verhouding van het aantal effectief gebruikte referenties tot het aantal voorziene referenties?

2. Teken de array-van-kinderen voorstelling van de boom in Figuur 4.2.

3. Bereken hoeveel null-referenties er zullen zijn bij eerste-kind-volgende-broer voorstelling van de boom in Figuur 4.2. Wat is de verhouding van het aantal effectief gebruikte referenties tot het aantal voorziene referenties?

4. Wat is de verhouding van het aantal effectief gebruikte referenties tot het aantal voorziene referenties voor een willekeurige gewortelde boom van graad k met n toppen.

5. Teken de eerste-kind-volgende-broer voorstelling van de boom in Figuur 4.2.

4.3 Recursie op bomen

4.3.1 Alle toppen van een boom bezoeken

Om alle toppen van een boom te bezoeken zijn er meerdere mogelijkheden. We bespreken er hier twee van. Een eerste mogelijkheid is om eerst de wortel van de boom te bezoeken en daarna (recursief en op dezelfde manier) de toppen van zijn deelbomen te doorlopen. De tweede mogelijkheid is om eerst (recursief) de toppen van de deelbomen te doorlopen en daarna pas de wortel van de boom te bezoeken. Beide mogelijkheden worden respectievelijk PREORDE en POSTORDE doorlopen van de boom genoemd.

Om een boom te doorlopen in preorde gaat men als volgt tewerk:

1. Bezoek de wortel van de boom.
2. Doorloop de deelbomen van de wortel in preorde.

Om een boom te doorlopen in postorde doet men het volgende:

1. Doorloop de deelbomen van de wortel in postorde.
2. Bezoek de wortel van de boom.

Dit proces zal eindigen want wanneer de boom slechts uit één top bestaat (m.a.w. als die top een blad is), dan moet er niets gedaan worden in de tweede stap wanneer preorde wordt gebruikt (of de eerste stap in het geval van postorde). Hier eindigt de recursie dus. Bovendien hebben de deelbomen steeds (strikt) minder toppen dan de originele boom, zodanig dat men in een eindig aantal stappen het basisgeval zal bereiken. Men komt dus niet in een oneindig diepe recursie terecht.

In Algoritme 4.1 wordt de pseudocode gegeven voor het in preorde doorlopen van een boom.

Algoritme 4.1 Doorlopen van een boom in preorde

Invoer Een gewortelde boom T, en een visit functie.
Uitvoer De visit functie is aangeroepen voor elke top van de boom.

```
1: function PREORDE(T,visit)
2:     PreOrdeRecursief(T.wortel, visit)          ▷ start met de wortel
3: end function
4: function PREORDERECURSIEF(v, visit)
5:     visit(v)
6:     for all w ∈ kinderen(v) do                 ▷ implementatie-onafhankelijk
7:         PreOrdeRecursief(w, visit)
8:     end for
9: end function
```

Voorbeeld 4.8 Beschouw opnieuw de boom uit Figuur 4.1. Veronderstel nu dat we tijdens elk 'bezoek' aan een top enkel het label van die top afdrukken (print).

In Tabel 4.2 kunnen we zien in welke volgorde de verschillende methodes worden opgeroepen. Als we kijken in welke volgorde de print-opdrachten

```
PreOrdeRecursief(n1)
  print(n1)
  PreOrdeRecursief(n2)
    print(n2)
    PreOrdeRecursief(n5)
      print(n5)
    PreOrdeRecursief(n6)
      print(n6)
  PreOrdeRecursief(n3)
    print(n3)
  PreOrdeRecursief(n4)
    print(n4)
    PreOrdeRecursief(n7)
      print(n7)
```

```
PostOrdeRecursief(n1)
  PostOrdeRecursief(n2)
    PostOrdeRecursief(n5)
      print(n5)
    PostOrdeRecursief(n6)
      print(n6)
    print(n2)
  PostOrdeRecursief(n3)
    print(n3)
  PostOrdeRecursief(n4)
    PostOrdeRecursief(n7)
      print(n7)
    print(n4)
  print(n1)
```

Tabel 4.2: Traceren van de methode-oproepen wanneer de voorbeeldboom in Figuur 4.1 in preorde resp. postorde doorlopen wordt.

voorkomen dan zien we dat de labels in deze volgorde afgedrukt worden wanneer de boom in preorde doorlopen wordt:

$$n_1, n_2, n_5, n_6, n_3, n_4, n_7.$$

Wanneer de boom postorde doorlopen en de labels afdrukken dan krijgen we:

$$n_5, n_6, n_2, n_3, n_7, n_4, n_1.$$

Dit kan je eveneens aflezen uit Tabel 4.2. ∎

4.3.2 Eenvoudige berekeningen op bomen

Veel eenvoudige berekeningen op bomen kunnen op een natuurlijke manier recursief geformuleerd worden op een manier die doet denken aan het pre- of postorde doorlopen van de toppen. Vaak moet er voor en na de recursieve oproep ook nog wat werk worden gedaan, zoals het initialiseren en bijwerken van variabelen.

Veronderstel dat we het aantal toppen van een boom wensen te berekenen, maar dat de voorstelling van een boom dit niet rechtstreeks toelaat[1]. Uit

[1]Dit betekent dat we geen veld bijhouden voor de grootte.

de definitie van een boom kunnen we zien dat het aantal toppen $\#(T)$ in de boom T gelijk is aan

$$\#(T) = 1 + \#(T_1) + \#(T_2) + \cdots + \#(T_m). \tag{4.1}$$

Deze formule drukt uit dat het aantal toppen in een boom gelijk is aan de som van het aantal toppen in elk van de deelbomen van de wortel, vermeerderd met 1 (we mogen de wortel zelf niet vergeten). Formule (4.1) kan eenvoudig vertaald worden in een recursief algoritme, zie Algoritme 4.2.

Algoritme 4.2 Berekenen van aantal toppen van een boom.

Invoer Een gewortelde boom T
Uitvoer Het aantal toppen van de boom.

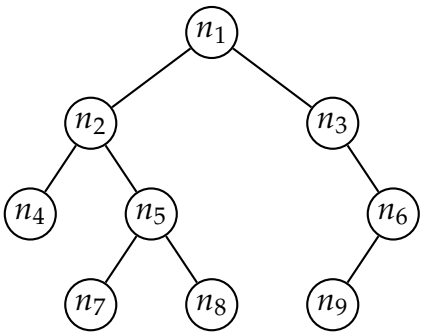
```
1:  function AANTAL(T)
2:      return AantalRecursief(T.wortel)
3:  end function
4:  function AANTALRECURSIEF(v)
5:      n ← 1                                   ▷ Top zelf niet vergeten
6:      for all w ∈ kinderen(v) do
7:          n ← n + AantalRecursief(w)
8:      end for
9:      return n
10: end function
```

Een andere berekening die eenvoudig kan geïmplementeerd worden is het berekenen van de hoogte van een boom. Inderdaad, de hoogte $h(T)$ van een boom is in het algemeen één meer dan het maximum van de hoogtes van zijn deelbomen:

$$h(T) = \begin{cases} 0 & \text{als } m = 0 \\ 1 + \max\big(h(T_1), h(T_2), \ldots, h(T_m)\big) & \text{als } m > 0. \end{cases} \tag{4.2}$$

4.3.3 Oefeningen

1. Geef de volgorde waarin de toppen worden bezocht wanneer de boom in Figuur 4.2 respectievelijk in preorde en postorde wordt doorlopen.
2. Geef code analoog aan Algoritme 4.1 om een gewortelde boom in postorde te doorlopen.
3. Geef code analoog aan Algoritme 4.2 om de hoogte van een gewortelde boom te berekenen. Baseer je op formule (4.2).



Figuur 4.5: Een binaire boom bestaande uit 9 toppen.

4.4 Binaire bomen

4.4.1 Definitie en eigenschappen

We beschouwen nu bomen die zeer vaak voorkomen in de praktijk, nl. binaire bomen. We geven opnieuw een recursieve definitie:

Definitie 4.9 Een BINAIRE BOOM is een verzameling toppen die

1. ofwel leeg is,
2. ofwel bestaat uit een *wortel* en twee disjuncte verzamelingen T_l en T_r, die op hun beurt ook een binaire boom zijn. We noemen T_l en T_r respectievelijk de *linker-* en *rechterdeelboom* van de wortel. ∎

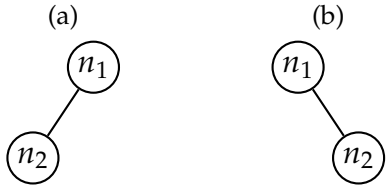
Voorbeeld 4.10 In Figuur 4.5 zien we een voorbeeld van een binaire boom T met

$$T = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9\}.$$

De linker- en rechterdeelboom van de wortel n_1 zijn respectievelijk:

$$T_l = \{n_2, n_4, n_5, n_7, n_8\} \quad \text{en} \quad T_r = \{n_3, n_6, n_9\}.$$

De verzamelingen T_l en T_r zijn op hun beurt weer binaire bomen. De wortel van T_r is de top n_3. De linkerdeelboom van n_3 is leeg, terwijl de rechterdeelboom bestaat uit de toppen n_6 en n_9. Zoals je ziet worden lege bomen eenvoudigweg niet getekend. ∎



Figuur 4.6: Twee verschillende binaire bomen die gelijk zijn als "gewone" gewortelde bomen.

Opmerking 4.11 Een eerste verschil tussen de "gewone" gewortelde bomen uit Definitie 4.1 en binaire bomen is dat een binaire boom geen enkele top kan bevatten. Een tweede verschil is dat er bij binaire bomen altijd een ordening aan de deelbomen wordt opgelegd. Inderdaad, bekijk bv. de twee bomen in Figuur 4.6. Deze twee bomen zijn identiek als we ze beschouwen als "gewone" gewortelde bomen, maar ze zijn verschillend als binaire bomen. Inderdaad, bij de ene boom is de rechterdeelboom leeg, terwijl bij de andere boom de linkerdeelboom leeg is. ∎

In een binaire boom is het gemakkelijk om een bovengrens te geven voor het aantal toppen met een bepaalde diepte k. De diepte van een top in een binaire boom wordt trouwens op dezelfde manier gedefinieerd als in een "gewone" gewortelde boom.

Eigenschap 4.12 In een binaire boom is het aantal toppen met diepte k hoogstens 2^k. ∎

Bewijs In een niet-lege binaire boom is er steeds juist één top met diepte 0, nl. de wortel. Deze wortel heeft hoogstens twee niet-lege deelbomen, dus zijn er hoogstens 2 toppen met diepte 1. Elk van de toppen met diepte 1 heeft opnieuw hoogstens twee niet-lege deelbomen, dus het aantal toppen met diepte twee is hoogstens $2 \times 2 = 4$. Zo verdergaand zien we (door inductie) dat het aantal toppen met diepte k hoogstens 2^k is. ♢

Stel dat we weten dat de diepte van een binaire boom T gelijk is aan d. Dan kunnen we m.b.v. de vorige eigenschap een bovengrens geven voor het aantal toppen dat die boom kan bevatten. Een ondergrens volgt gemakkelijk uit het feit dat er minstens één top moet zijn op elke diepte.

Eigenschap 4.13 Voor een (niet-lege) binaire boom T met diepte $d \geq 0$ geldt dat:

$$d + 1 \leq \#(T) \leq 2^{d+1} - 1. \tag{4.3}$$

∎

Bewijs We tellen het aantal toppen van een boom T door het aantal toppen met een bepaalde diepte k te tellen en dit voor alle mogelijke dieptes. In formulevorm:

$$\begin{aligned}
\#(T) &= \sum_{k=0}^{d} \text{aantal toppen met diepte } k \\
&\leq \sum_{k=0}^{d} 2^k && \text{wegens Eigenschap 4.12} \\
&= 1 + 2 + 4 + \cdots + 2^d \\
&= 2^{d+1} - 1.
\end{aligned}$$

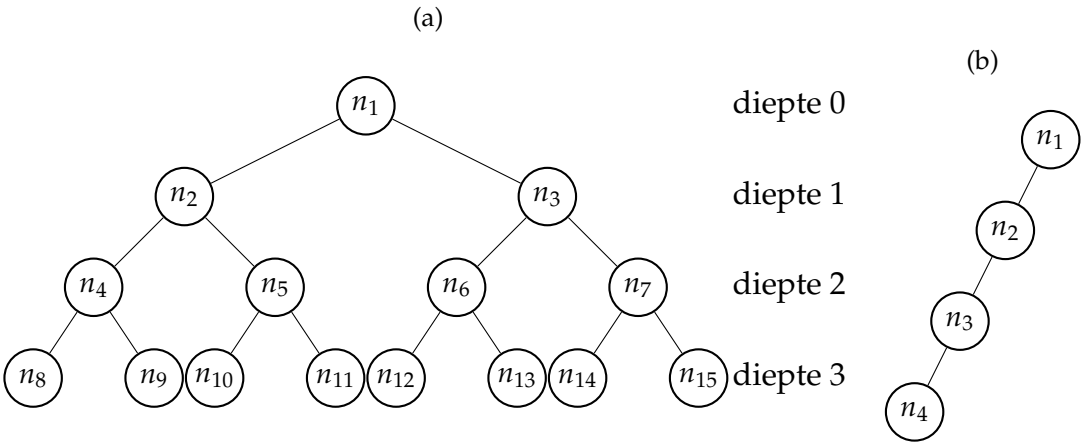
Dit bewijst de bovengrens voor $\#(T)$.

Een ondergrens voor het aantal toppen in een binaire boom met diepte d is ook gemakkelijk te bepalen. De boom zal het kleinste aantal toppen hebben wanneer er juist één top is op elke diepte. Het minimum aantal toppen in een binaire boom van diepte d is dus $d + 1$. ♢

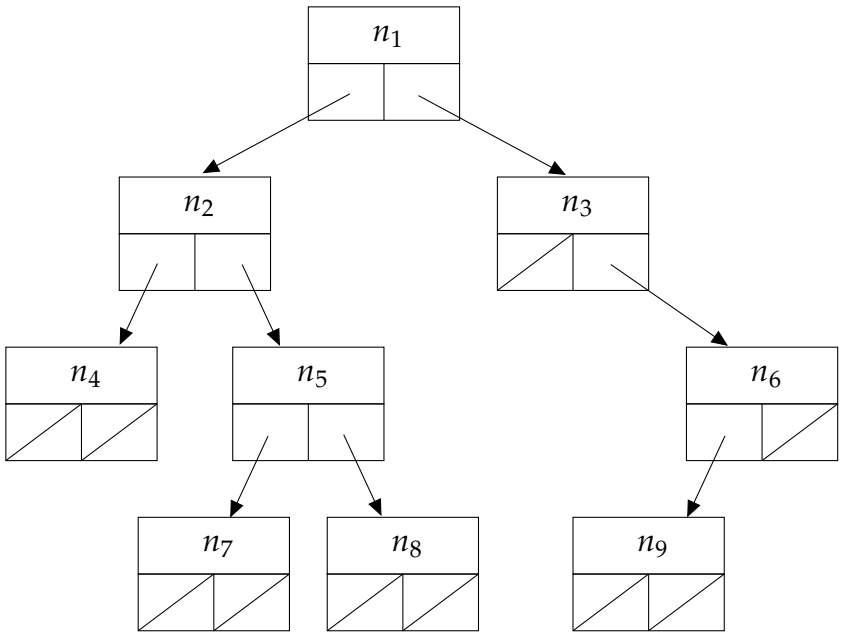
Voorbeeld 4.14 In Figuur 4.7 zie je aan de linkerkant de binaire boom van diepte 3 met het maximaal aantal toppen, nl. $2^{3+1} - 1 = 15$. Aan de rechterkant zie je een binaire boom van diepte 3 met het minimum aantal toppen. Merk op ook dat er slechts één binaire boom is met het maximaal aantal toppen voor een gegeven diepte, terwijl er *veel* verschillende binaire bomen zijn met het minimum aantal toppen voor een gegeven diepte. ∎

4.4.2 Voorstelling van een binaire boom

Elke top van een binaire boom wordt voorgesteld door een structuur die, behalve de eigenlijke data voor die top, ook twee referenties bijhoudt naar respectievelijk de linker- en rechterdeelboom. Wanneer de linker- of rechterdeelboom leeg is, dan is de overeenkomstige referentie `null`. De binaire boom zelf wordt voorgesteld door een referentie naar zijn wortel.



Figuur 4.7: De binaire boom met diepte 3 die het maximum aantal toppen bevat versus een binaire boom van diepte 3 die het minimum aantal toppen bevat.



Figuur 4.8: Interne voorstelling van de binaire boom in Figuur 4.5.

Voorbeeld 4.15 Figuur 4.8 geeft de interne voorstelling van de binaire boom in Figuur 4.5. De `null` referenties worden aangeduid met een schuine streep. ∎

4.4.3 Alle toppen van een binaire boom bezoeken

We hebben in Sectie 4.3 reeds gezien hoe we alle toppen van een gewortelde boom kunnen bezoeken, door de boom te doorlopen in ofwel pre- of postorde. Als we spreken over pre- of postorde doorlopen van binaire bomen dan is het (bijna) altijd zo dat de linkerdeelboom steeds doorlopen wordt voor de rechterdeelboom.

Om een binaire boom in PREORDE te doorlopen gaan we dus als volgt tewerk:

1. Bezoek de wortel van boom.
2. Als de linkerdeelboom niet leeg is, doorloop de linkerdeelboom dan recursief in preorde.
3. Als de rechterdeelboom niet leeg is, doorloop de rechterdeelboom dan recursief in preorde.

Om een binaire boom in POSTORDE te doorlopen gaan we dus als volgt tewerk:

1. Als de linkerdeelboom niet leeg is, doorloop de linkerdeelboom dan recursief in postorde.
2. Als de rechterdeelboom niet leeg is, doorloop de rechterdeelboom dan recursief in postorde.
3. Bezoek de wortel van boom.

Omdat elke top nu steeds twee deelbomen heeft (die eventueel leeg kunnen zijn), is er nog een derde natuurlijke mogelijkheid om een binaire boom te doorlopen. We kunnen namelijk de wortel bezoeken *tussen* het doorlopen van de linker- en rechterdeelboom. Bij bomen die meer dan twee kinderen kunnen hebben heeft deze optie weinig zin omdat er geen "natuurlijke" manier is om de wortel tussen de kinderen te plaatsen. Deze derde manier

om een binaire boom te doorlopen noemt men het INORDE doorlopen van de boom.

Om een binaire boom in INORDE te doorlopen gaan we dus als volgt tewerk:

1. Als de linkerdeelboom niet leeg is, doorloop de linkerdeelboom dan recursief in inorde.
2. Bezoek de wortel van boom.
3. Als de rechterdeelboom niet leeg is, doorloop de rechterdeelboom dan recursief in inorde.

Algoritme 4.3 Doorlopen van een binaire boom in inorde

Invoer Een binaire boom T en een visit functie.
Uitvoer De visit functie is aangeroepen voor elke top van T.

```
1:  function INORDE(T,visit)
2:      if T ≠ ∅ then                          ▷ controleer dat boom niet leeg is
3:          InOrdeRecursief(T.wortel, visit)   ▷ start met de wortel
4:      end if
5:  end function
6:  function INORDERECURSIEF(v, visit)
7:      if v.links ≠ ∅ then
8:          InOrdeRecursief(v.links, visit)
9:      end if
10:     visit(v)                   ▷ visit aanroepen tussen recursieve oproepen
11:     if v.rechts ≠ ∅ then
12:         InOrdeRecursief(v.rechts, visit)
13:     end if
14: end function
```

Voorbeeld 4.16 Tijdens een bezoek aan een top drukken we zijn label af. Wanneer we de toppen van de boom in Figuur 4.5 in preorde doorlopen dan worden de labels afgedrukt in de volgorde:

$$n_1, n_2, n_4, n_5, n_7, n_8, n_3, n_6, n_9.$$

Dit kunnen we afleiden uit Tabel 4.3 die de oproepen traceert voor de verschillende manieren om een binaire boom te doorlopen. Voor postorde lezen we af uit dezelfde tabel dat de volgorde van afdrukken de volgende is:

$$n_4, n_7, n_8, n_5, n_2, n_9, n_6, n_3, n_1.$$

preorde(n_1)
  print(n_1)
  preorde(n_2)
  print(n_2)
  preorde(n_4)
  print(n_4)
  preorde(n_5)
  print(n_5)
  preorde(n_7)
  print(n_7)
  preorde(n_8)
  print(n_8)
  preorde(n_3)
  print(n_3)
  preorde(n_6)
  print(n_6)
  preorde(n_9)
  print(n_9)
	postorde(n_1)
	  postorde(n_2)
	  postorde(n_4)
	  print(n_4)
	  postorde(n_5)
	  postorde(n_7)
	  print(n_7)
	  postorde(n_8)
	  print(n_8)
	  print(n_5)
	  print(n_2)
	  postorde(n_3)
	  postorde(n_6)
	  postorde(n_9)
	  print(n_9)
	  print(n_6)
	  print(n_3)
	  print(n_1)
		inorde(n_1)
		  inorde(n_2)
		  inorde(n_4)
		  print(n_4)
		  print(n_2)
		  inorde(n_5)
		  inorde(n_7)
		  print(n_7)
		  print(n_5)
		  inorde(n_8)
		  print(n_8)
		  print(n_1)
		  inorde(n_3)
		  print(n_3)
		  inorde(n_6)
		  inorde(n_9)
		  print(n_9)
		  print(n_6)

Tabel 4.3: Trace van de oproepen wanneer men de binaire boom in Figuur 4.5 in pre-, post- en inorde doorloopt.

Bij het in inorde doorlopen zullen de labels als volgt afgedrukt worden:

$$n_4, n_2, n_7, n_5, n_8, n_1, n_3, n_9, n_6.$$ ∎

4.4.4 Oefeningen

1. a) Teken de binaire boom met labels 0 t.e.m. 9 waarvoor de inorde sequentie

   $$9, 3, 1, 0, 4, 2, 7, 6, 8, 5$$

   is terwijl de postorde sequentie

   $$9, 1, 4, 0, 3, 6, 7, 5, 8, 2$$

   is.

   b) Doe nu hetzelfde voor de volgende sequenties, of leg uit waarom zo'n binaire boom niet bestaat:

   inorde: $9, 3, 1, 0, 4, 2, 7, 6, 8, 5$

en

postorde: $9, 1, 4, 0, 3, 6, 5, 7, 8, 2$

4.5 Binaire zoekbomen

Een vaak voorkomende activiteit in een computerprogramma is het bijhouden van een verzameling waarden waarbij we wensen:

1. gegevens toe te voegen aan de verzameling,
2. gegevens te verwijderen uit de verzameling, en
3. te controleren of een element aanwezig is in de verzameling.

Er zijn verschillende gegevensstructuren die kunnen gebruikt worden om deze functionaliteit te implementeren, zoals bv. (lineair gelinkte) lijsten en hash-tabellen.

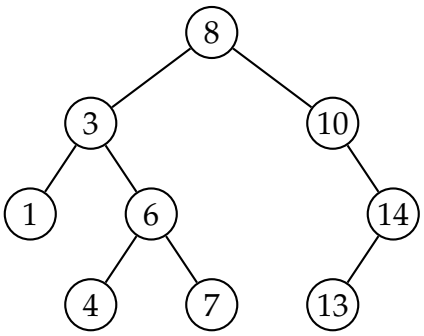
Een andere datastructuur die deze bewerkingen kan implementeren is de binaire zoekboom. Een noodzakelijke voorwaarde om een binaire zoekboom te kunnen gebruiken is dat de labels een *totaal geordende verzameling* vormen. Dit wil zeggen dat er voor elk paar (mogelijke) labels x en y geldt dat

$$x < y \quad \text{of} \quad y < x \quad \text{of} \quad x = y.$$

Anders gezegd: wanneer x en y verschillend zijn dan kunnen we zeggen welk element het kleinste is; er zijn geen onvergelijkbare elementen. De labels zouden bv. gehele getallen kunnen zijn, of karakterreeksen (strings) die lexicografisch (i.e. alfabetisch) vergeleken worden. De labels van een binaire zoekboom worden dikwijls SLEUTELS genoemd.

Definitie 4.17 Een BINAIRE ZOEKBOOM is een gelabelde binaire boom die aan een bijzondere voorwaarde, de *binaire zoekboomeigenschap*, voldoet. ∎

Definitie 4.18 De BINAIRE ZOEKBOOMEIGENSCHAP is de volgende: voor *elke* top x van de binaire zoekboom geldt dat *alle* toppen in de linkerdeelboom van x een label hebben dat kleiner is dan het label van x, terwijl voor *alle* toppen in de rechterdeelboom van x geldt dat hun label groter is dan het label van x. ∎



Figuur 4.9: Een binaire zoekboom met 9 toppen.

Voorbeeld 4.19 In Figuur 4.9 zien we een binaire boom waarvan de labels gehele getallen zijn. We controleren dat de binaire zoekboomeigenschap voldaan is. Hiertoe moeten we voor elke top nagaan dat alle labels in de linkerdeelboom (resp. rechterdeelboom) van die top kleiner (resp. groter) zijn dan het label van de top. We controleren dit m.b.v. onderstaande tabel:

sleutel top	sleutels links	sleutels rechts
8	$\{1,3,4,6,7\}$	$\{10,13,14\}$
3	$\{1\}$	$\{4,6,7\}$
1	$\emptyset$	$\emptyset$
6	$\{4\}$	$\{7\}$
4	$\emptyset$	$\emptyset$
7	$\emptyset$	$\emptyset$
10	$\emptyset$	$\{13,14\}$
14	$\{13\}$	$\emptyset$
13	$\emptyset$	$\emptyset$

Hieruit blijkt dat de binaire zoekboomeigenschap voor deze boom voldaan is. ∎

Opmerking 4.20 Wanneer we de boom in Figuur 4.9 in inorde doorlopen dan worden de toppen in de volgende volgorde bezocht:

$$1,3,4,6,7,8,10,13,14.$$

Merk op dat deze rij gesorteerd is van klein naar groot. Een binaire zoekboom houdt zijn waarden dus ook *gesorteerd* bij, al is dat op het eerste zicht misschien niet zo duidelijk als bij een gesorteerde array. ∎

4.5.1 Opzoeken van een sleutel in een binaire zoekboom

Wanneer we een sleutel x willen opzoeken in een binaire zoekboom dan kunnen we gebruik maken van de binaire zoekboomeigenschap om (snel) uit te maken of x aanwezig is of niet.

Inderdaad, wanneer de binaire zoekboom T leeg is, dan is x uiteraard niet aanwezig. Wanneer T niet leeg is, en x is kleiner dan het label van de wortel van T, dan moet x (indien aanwezig in T) zich in de linkerdeelboom van de wortel bevinden. Analoog moet x zich in de rechterdeelboom van de wortel bevinden wanneer x groter is dan de sleutel van de wortel. In het laatste geval (wanneer x gelijk is aan het label van de wortel) bevindt x zich in de wortel van de boom.

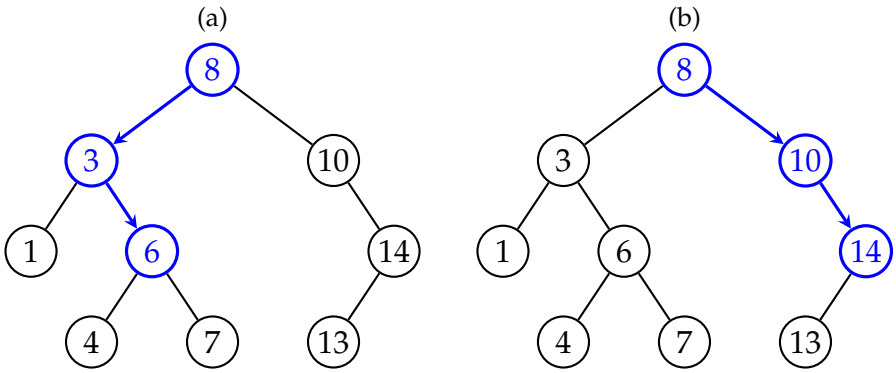
Opmerking 4.21 Merk op dat dit zoekproces lijkt op de manier waarop er binair gezocht wordt in een gesorteerde array. Alleen is het zo dat bij een binaire zoekboom het aantal te onderzoeken waarden niet steeds exact in twee wordt verdeeld zoals dat bij binair zoeken wel het geval is. ∎

Om dus (recursief) te zoeken naar een bepaalde waarde x in een binaire zoekboom T gaat men als volgt tewerk:

1. Wanneer de boom leeg is, geef dan 'niet gevonden' terug.
2. Vergelijk x met de sleutel van de wortel.
    a) Wanneer x kleiner is dan dit label, zoek dan (recursief) in de linkerdeelboom.
    b) Wanneer x groter is dan dit label, zoek dan (recursief) in de rechterdeelboom.
    c) Geef de wortel van de boom terug (x werd gevonden).

Voorbeeld 4.22 Veronderstel dat we op zoek gaan naar de sleutel 6 in de voorbeeldboom van Figuur 4.9.

We starten aan de wortel van de boom, en we zien dat de boom niet leeg is. We vergelijken 6 met de sleutel van de wortel (dit is 8) en we zien dat 6 kleiner is dan 8. We gaan nu verder in de linkerdeelboom met als wortel de top met label 3. De boom is dus opnieuw niet leeg, en we vergelijken 6 met het label 3 van deze top. We zien dat 6 groter is dan 3, en we gaan recursief



Figuur 4.10: In (a) wordt succesvol gezocht naar de sleutel 6 in de binaire zoekboom. In (b) zoeken we tevergeefs naar de sleutel 15.

verder in de rechterdeelboom. Wanneer we nu 6 vergelijken met de sleutel van de wortel van deze deelboom, dan zien we dat we de gewenste top gevonden hebben. ∎

Opmerking 4.23 Om de top 6 te vinden hebben we drie vergelijkingen uitgevoerd; dit is één meer dan de diepte van deze top. ∎

Voorbeeld 4.24 Veronderstel dat we op zoek gaan naar het element met de waarde 15 in de voorbeeldboom van Figuur 4.9.

We starten opnieuw aan de wortel van de boom, en we vergelijken 15 met 8 (de sleutel van de wortel). Aangezien 15 groter is, gaan we recursief verder in de rechterdeelboom. We vergelijken vervolgens 15 met het label 10, en we gaan opnieuw recursief verder in de rechterdeelboom. We zoeken dus opnieuw recursief in de rechterdeelboom, en we vergelijken het label 14 met 15. Aangezien 15 groter is gaan we opnieuw recursief verder in de rechterdeelboom. We zien nu dat deze boom leeg is, en dus kunnen we besluiten dat het element met de waarde 15 niet tot de boom behoort. ∎

Soms kan het ook nuttig zijn om de kleinste of grootste sleutel van een boom te kennen. Door de binaire zoekboomeigenschap weten we dat het kleinste element van een binaire zoekboom steeds tot de linkerdeelboom van de wortel behoort (wanneer die niet leeg is). Om het kleinste element van een (niet-lege) binaire zoekboom op te zoeken gaan we dus als volgt tewerk:

1. Wanneer de linkerdeelboom van de wortel leeg is, geef dan (de sleutel van) de wortel terug.

2. In het andere geval zoek je recursief naar het kleinste element van de linkerdeelboom.

Om het grootste element te vinden gaan we natuurlijk analoog tewerk.

Voorbeeld 4.25 We zoeken het kleinste element van de binaire zoekboom in Figuur 4.9.

Aangezien de linkerdeelboom van de wortel niet leeg is, gaan we recursief op zoek naar het kleinste element in de deelboom met als wortel de top 3. Hier is de linkerdeelboom opnieuw niet leeg, dus zoeken we naar het kleinste element van de deelboom met als wortel de top 1. Nu is de linkerdeelboom van de top met als label 1 leeg, en dus is het kleinste element van de binaire zoekboom gelijk aan 1. ∎

4.5.2 Toevoegen van een sleutel aan een binaire zoekboom

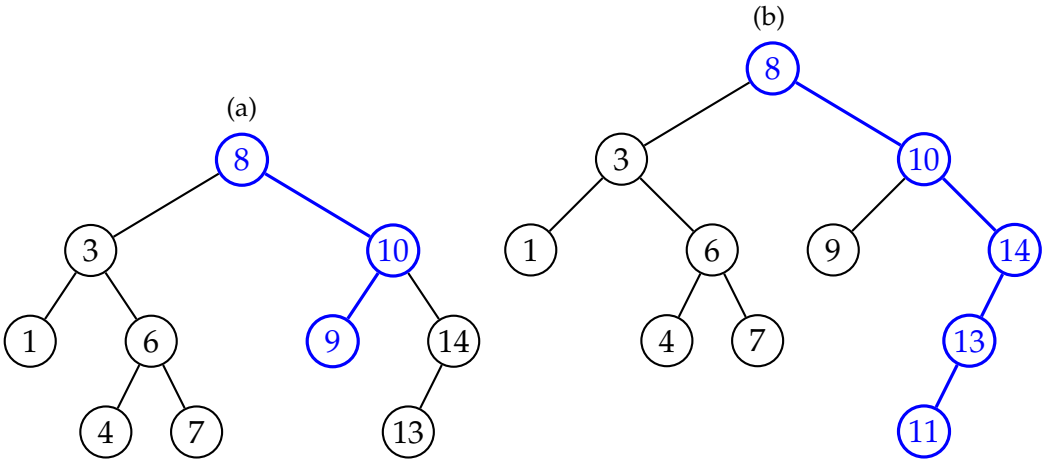
Wanneer we een sleutel x toevoegen aan een binaire zoekboom T, dan is het van belang dat de nieuwe boom T' eveneens een binaire zoekboom is, m.a.w. dat de nieuwe boom ook aan de binaire zoekboomeigenschap voldoet.

Toevoegen van een element aan een binaire zoekboom is eenvoudig en kan recursief worden geformuleerd. Wanneer we x wensen toe te voegen aan een lege boom T, dan vervangen we T eenvoudigweg door een boom bestaande uit één top met x als zijn sleutel. Wanneer de boom T niet leeg is en x bevat in de wortel dan moeten we niets doen (want x behoort reeds tot de boom). In het andere geval voegen we x toe aan de linker- of rechterdeelboom al naargelang het label van x kleiner of groter is dan het label van de wortel.

We kunnen het toevoegen van een sleutel x aan een (niet-lege) binaire zoekboom dan ook als volgt recursief formuleren:

1. Vergelijk x met het label van de wortel.

    a) Wanneer x kleiner is dan het label van de wortel, voeg dan x toe aan de linkerdeelboom wanneer die niet leeg is. Wanneer de lin-



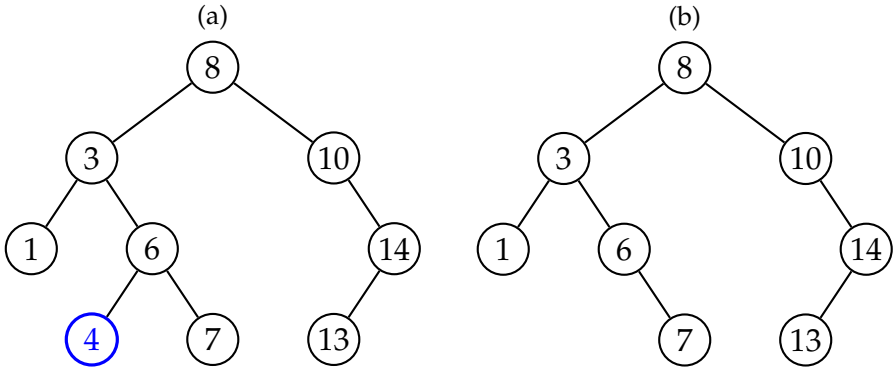
Figuur 4.11: In (a) zie je de resulterende boom na toevoegen van de sleutel 9 aan de boom in Figuur 4.9. In (b) wordt dan nog eens de sleutel 11 toegevoegd.

kerdeelboom leeg is vervang dan de (`null`)-referentie naar de linkerdeelboom door de referentie naar een nieuwe top met x als label.

b) Wanneer x groter is dan het label van de wortel, voeg dan x toe aan de rechterdeelboom wanneer die niet leeg is. Wanneer de rechterdeelboom leeg is vervang dan de (`null`)-referentie naar de rechterdeelboom door de referentie naar een nieuwe top met x als label.

c) Doe niets, want x behoort reeds tot de boom.

Voorbeeld 4.26 Veronderstel dat we de sleutel 9 willen toevoegen aan de binaire zoekboom in Figuur 4.9. We starten in de wortel en zien dat 9 groter is dan 8, het label van de wortel. We gaan dus verder in de rechterdeelboom. Aangezien 10 (het label van de wortel van de rechterdeelboom) groter is dan 9, moet het nieuwe element aan de linkerdeelboom van 10 toegevoegd worden. Aangezien deze linkerdeelboom leeg is, hebben we de juiste plaats waar 9 moet worden toegevoegd gevonden. We creëren een nieuwe top en maken deze de wortel van de linkerdeelboom van de top met als label 10. Je ziet de resulterende boom in Figuur 4.11(a). ∎



Figuur 4.12: Links zie je de originele zoekboom. Rechts zie je de binaire zoekboom na het verwijderen van de sleutel 4. De sleutel 4 bevindt zich in een blad en dus is verwijderen eenvoudig.

4.5.3 Verwijderen van een sleutel uit een binaire zoekboom

Het verwijderen van een sleutel uit een binaire zoekboom is iets moeilijker dan het opzoeken of het toevoegen van een element.

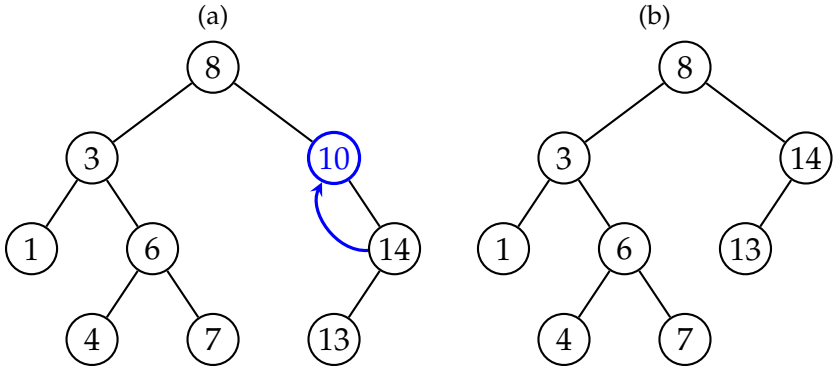
Het verwijderen van een sleutel x start met het opzoeken van deze sleutel in de boom. Er kunnen zich nu drie gevallen voordien:

1. De sleutel x bevindt zich in een blad.
2. De sleutel x bevindt zich in een top met één kind.
3. De sleutel x bevindt zich in een top met twee kinderen.

We bekijken nu deze gevallen één voor één.

Sleutel bevindt zich in een blad
Wanneer x zich in een blad bevindt, dan kunnen we eenvoudigweg dit blad verwijderen.

Voorbeeld 4.27 In Figuur 4.12 zie je de resulterende zoekboom wanneer het element 4 uit de boom wordt verwijderd. Aangezien de top die het element 4 bevat een blad is, kan deze top zonder problemen uit de boom worden verwijderd. ∎



Figuur 4.13: Binaire zoekboom na het verwijderen van het element 10 uit de boom in Figuur 4.9.

Sleutel bevindt zich in een top met één kind

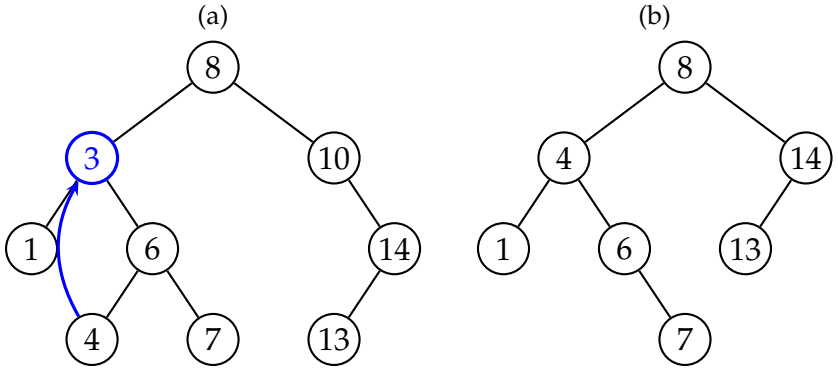
Wanneer de top n die x bevat slechts één kind heeft, dan kunnen we n vervangen door dit ene kind. Anders gezegd: het nieuwe linker- of rechterkind van de ouder van n is zijn vroegere kleinkind[2]. De binaire zoekboomeigenschap zal in dit geval nog steeds geldig zijn.

Voorbeeld 4.28 Wanneer we de sleutel 10 wensen te verwijderen uit de boom in Figuur 4.9 dan zien we dat 10 slechts één kind heeft (de lege linkerdeelboom wordt niet geteld als een kind). Dit betekent dat we in de boom de top met label 10 eenvoudigweg kunnen vervangen door de top met label 14. Dit komt neer op het vervangen van de rechterwijzer in de top met label 8. De resulterende boom is getoond in Figuur 4.13(b). ∎

Sleutel bevindt zich in een top met twee kinderen

Wanneer tenslotte de top n die x bevat twee kinderen heeft dan kunnen we eerst x vervangen door het kleinste element y van de rechterdeelboom. Dit element is de *opvolger* van x. In een volgende stap kunnen we dit element y gaan verwijderen uit de rechterdeelboom. Merk op dat de top m die dit element y bevat hoogstens één kind kan hebben. In het bijzonder moet de linkerdeelboom van m de lege boom zijn. Dit betekent dat de top m eenvoudig te verwijderen is. We hebben het probleem dus gereduceerd tot het verwijderen van een top met hoogstens één kind.

[2]Wanneer n de wortel van de boom is, dan krijgt de boom een nieuwe wortel.



Figuur 4.14: Binaire zoekboom na het verwijderen van het element 3 uit de boom in Figuur 4.9.

Voorbeeld 4.29 Wanneer we de sleutel 3 wensen te verwijderen uit de boom, dan zien we dat de top 3 twee kinderen heeft. We gaan dus eerst op zoek naar het kleinste element groter dan 3. Dit element bevindt zich in de rechterdeelboom van 3 en zal steeds hoogstens 1 kind hebben, aangezien de linkerdeelboom leeg moet zijn. In dit geval is de opvolger van 3 de sleutel 4. We vervangen dus eerst in de boom het label 3 door het label 4. Vervolgens verwijderen we de top die 4 bevatte. De resulterende boom zie je in Figuur 4.14(b). ∎

4.5.4 Tijdscomplexiteit van de bewerkingen

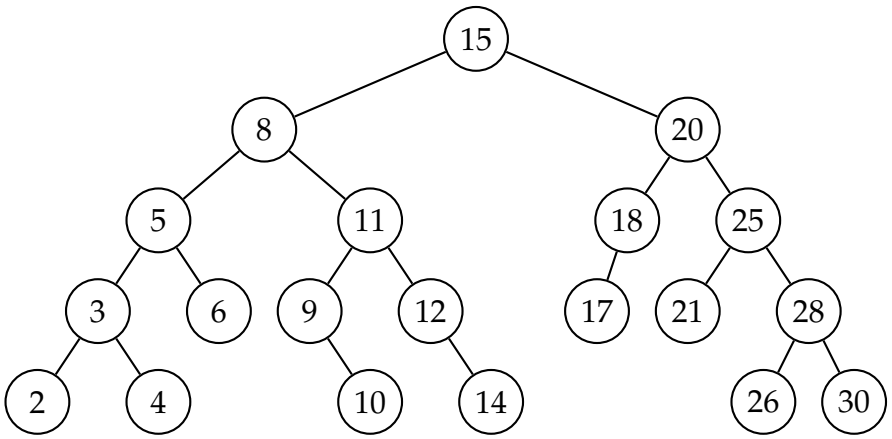
Het is intuïtief duidelijk dat alle bewerkingen op een binaire zoekboom in het slechtste geval een uitvoeringstijd hebben die evenredig is met de diepte van de boom, i.e. de uitvoeringstijd van de bewerkingen is in het slechtste geval $\mathcal{O}(d)$.

Wanneer de boom n toppen bevat, weten we uit Eigenschap 4.13 dat

$$n \leq 2^{d+1} - 1.$$

Door van beide leden de logaritme met grondtal twee te nemen zien we dat voor een totaal gebalanceerde boom geldt dat $\mathcal{O}(d) = \mathcal{O}(\lg(n))$. Voor een totaal gebalanceerde boom hebben alle bewerkingen dus een tijdscomplexiteit $\mathcal{O}(\lg(n))$, met n het aantal toppen in de zoekboom.

Echter, uit Eigenschap 4.13 zien we ook dat voor een totaal ongebalanceerde



Figuur 4.15: Een binaire zoekboom.

boom

$$d + 1 \leq n,$$

zodat $\mathcal{O}(d) = \mathcal{O}(n)$.

Men kan aantonen dat in het gemiddeld geval (wanneer elementen random worden toegevoegd aan een boom) de resulterende boom een diepte zal hebben die $\mathcal{O}(\lg(n))$ is.

4.5.5 Oefeningen

1. Geef de binaire zoekboom die opgebouwd wordt door de volgende sleutels

   $$4, 7, 5, 8, 11, 3, 2, 9, 10, 6$$

   één voor één aan de zoekboom toe te voegen in de gegeven volgorde.

2. Veronderstel dat men een binaire zoekboom opbouwt door sleutels één voor één toe te voegen aan een initieel lege boom.

   a) Geef een rij van lengte 7 die een binaire zoekboom van minimale diepte oplevert.

   b) Geef een rij van lengte 15 die een binaire zoekboom van minimale diepte oplevert.

   c) Geef een rij van lengte 5 die een binaire zoekboom van maximale diepte oplevert. Wanneer krijg je een over het algemeen een slecht gebalanceerde binaire zoekboom?

3. Beschouw de binaire zoekboom in Figuur 4.15. Voer de volgende opdrachten één na één uit.

   a) Welke toppen worden er bezocht bij het zoeken naar de top 12?

   b) Welke toppen worden er bezocht bij het zoeken naar de top 27?

   c) Voeg een top met sleutelwaarde 23 toe aan de zoekboom. Teken de resulterende zoekboom.

   d) Voeg vervolgens een top met sleutelwaarde 22 toe aan de zoekboom. Teken de resulterende zoekboom.

   e) Verwijder de top met waarde 4 uit de zoekboom. Teken de resulterende zoekboom.

   f) Verwijder vervolgens de top met waarde 18 uit de zoekboom. Teken de resulterende zoekboom.

   g) Verwijder vervolgens de top met waarde 20 uit de zoekboom. Teken de resulterende zoekboom.

4.6 Binaire hopen

4.6.1 Prioriteitswachtrij

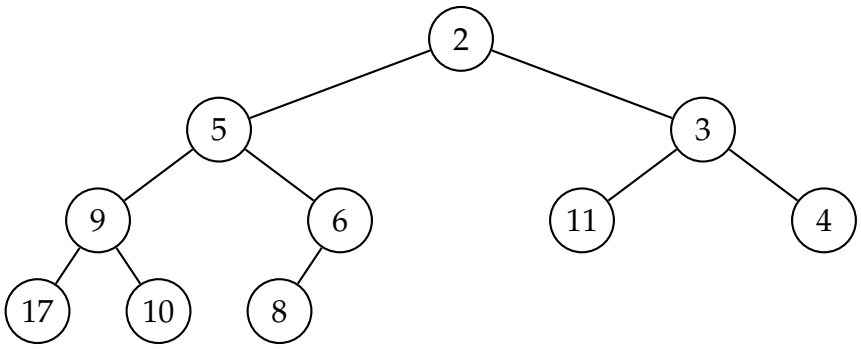
Binaire zoekbomen worden vaak gebruikt als implementatie van het abstract datatype "Verzameling". Een ander abstract datatype dat vaak wordt gebruikt in toepassingen is een PRIORITEITSWACHTRIJ. Dit is een uitbreiding van de "gewone" FIFO wachtrij.

De elementen van een wachtrij bestaan typisch uit twee delen: een sleutel en een waarde. De sleutel geeft in dit geval de prioriteit aan. Meestal is het zo dat een kleinere sleutel wijst op een grotere prioriteit.

Bij een prioriteitswachtrij kan men

1. het element met de kleinste sleutel opzoeken;
2. het element met de kleinste sleutel verwijderen;
3. een nieuw element toevoegen aan de wachtrij.

Opmerking 4.30 In een prioriteitswachtrij kan *enkel* het element met de kleinste sleutel efficiënt bereikt worden. Dit betekent dat een prioriteitswachtrij



Figuur 4.16: Een binaire hoop.

flexibel is wat betreft het toevoegen van elementen maar niet wat betreft het verwijderen. ∎

4.6.2 Implementatie als binaire hoop

We bekijken nu hoe we een prioriteitswachtrij kunnen implementeren als een binaire hoop. In essentie is een binaire hoop een complete binaire boom die aan een extra ordeningseigenschap voldoet.

Definitie 4.31 Een COMPLETE BINAIRE BOOM is een binaire boom van diepte d waarbij het aantal toppen met diepte $k < d$ maximaal (dus 2^k, zie Eigenschap 4.12) is. De toppen met diepte d komen voor van "links naar rechts". ∎

Voorbeeld 4.32 In Figuur 4.16 ziet men een voorbeeld van een complete binaire boom. De niveaus met diepte 0, 1 en 2 zijn volledig gevuld. Het diepste niveau, met toppen van diepte 3, is gevuld van links naar rechts. ∎

Definitie 4.33 De ORDENINGSEIGENSCHAP VOOR BINAIRE HOPEN zegt dat de sleutel van *elke* top hoogstens gelijk is aan de kleinste sleutel van zijn kinderen. ∎

Voorbeeld 4.34 Men gaat gemakkelijk na dat de complete binaire boom in Figuur 4.16 aan de ordeningseigenschap voor binaire hopen voldoet. Dit zien we eenvoudig uit Tabel 4.4. ∎

Sleutel ouder	Sleutels kinderen
2	{5, 3}
5	{9, 6}
3	{11, 4}
9	{17, 10}
6	{8}
11	∅
4	∅
17	∅
10	∅
8	∅

Tabel 4.4: Voor elke top uit de boom in Figuur 4.16 wordt zijn sleutel en de sleutel van zijn kinderen gegeven. De sleutel van de ouder is steeds kleiner dan de sleutel van zijn kinderen.

4.6.3 Implementatie

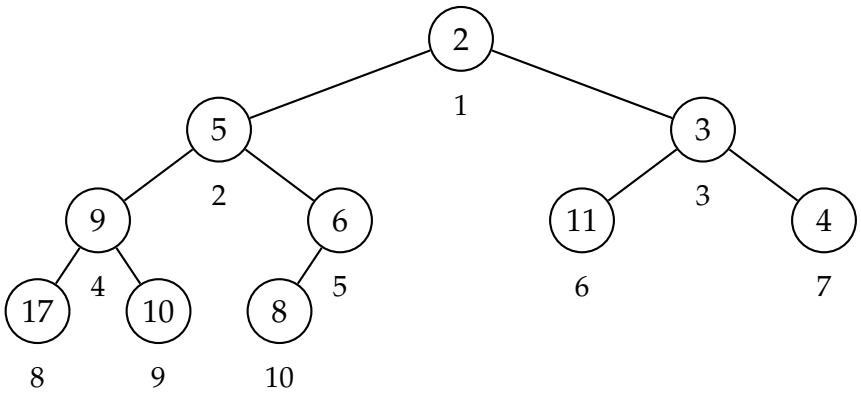
Het is mogelijk om een binaire hoop te implementeren als een binaire boom, waarbij men voor elke top bijhoudt wat zijn linker- en rechterkinderen zijn. Omdat de binaire boom echter steeds compleet is bestaat er een eenvoudigere (en snellere) manier die een gewone array gebruikt.

Veronderstel dat we de toppen van een complete binaire boom gaan nummeren op de volgende manier. De wortel van de boom heeft rangnummer 1. De toppen op diepte 1 worden consecutief genummerd van links naar rechts (en krijgen dus rangnummers 2 en 3). Daarna worden de toppen op diepte 2 genummerd (startend vanaf 4) van links naar rechts, enzovoort.

Voorbeeld 4.35 In Figuur 4.17 wordt de binaire hoop uit Figuur 4.16 herhaald, maar nu wordt voor elke top zijn rangnummer aangegeven. ∎

Er is een eenvoudig verband tussen het rangnummer van een top en het rangnummer van zijn kinderen. Dit verband wordt gegeven in de volgende eigenschap.

Eigenschap 4.36 Wanneer een top rangnummer i heeft, dan hebben zijn linker- en rechterkind (als die bestaan) respectievelijk rangnummer $2i$ en $2i + 1$.



Figuur 4.17: De binaire hoop uit Figuur 4.16 maar nu wordt voor elke top zijn rangnummer aangegeven.

Omgekeerd geldt: wanneer een top rangnummer i heeft (en deze top is niet de wortel van de boom), dan heeft zijn ouder rangummer $\text{floor}(i/2)$. We kunnen dus stellen dat:

$$\begin{aligned} \text{left}(i) &= 2i, \\ \text{right}(i) &= 2i+1, \\ \text{parent}(i) &= \text{floor}(i/2). \end{aligned}$$

Dit betekent dat we een binaire hoop kunnen opslaan in een array, waarbij de ouder-kind relaties afgeleid worden uit Eigenschap 4.36.

Voorbeeld 4.37 Hieronder ziet men hoe de voorgestelde nummering eruit ziet voor de binaire hoop in Figuur 4.16.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Sleutel		2	5	3	9	6	11	4	17	10	8				

Merk op dat positie 0 niet gebruikt wordt. De posities 11 t.e.m. 14 zijn nog beschikbaar voor nieuwe elementen.

4.6.4 Opzoeken van het element met de kleinste sleutel

Uit de ordeningseigenschap voor binaire hopen volgt dat het element met de kleinste sleutel steeds de wortel van de boom is (want elke andere top

heeft een ouder waarvoor de sleutelwaarde kleiner is dan de sleutelwaarde van die top). Het opzoeken van het element met de kleinste sleutel is dus een triviale bewerking, die de binaire hoop ongewijzigd laat. Deze bewerking kan uiteraard steeds uitgevoerd worden in constante tijd.

4.6.5 Toevoegen van een element

Om een nieuw element toe te voegen aan de binaire hoop gaan we als volgt tewerk:

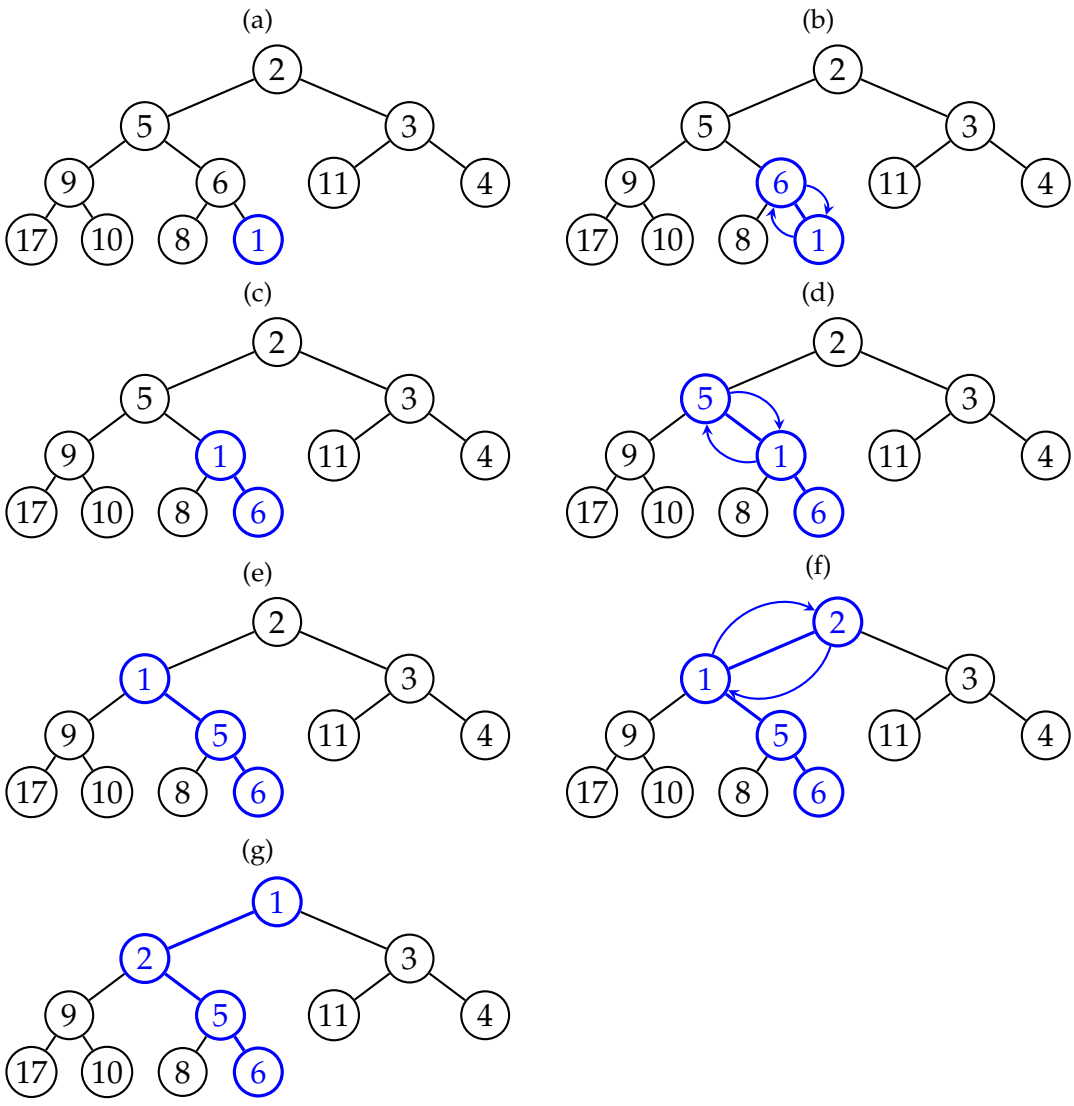
1. Creéer een nieuw element.
2. Voeg dit element toe op de eerste beschikbare plaats. Dit betekent dus als een nieuw blad, met rangnummer i, waarbij we er steeds voor zorgen dat het diepste niveau gevuld is van links naar rechts.
3. Op dit moment is het in het algemeen zo dat de ordeningseigenschap voor binaire hopen nu kan geschonden zijn tussen i en $\text{parent}(i)$. Indien dit zo is, verwissel dan i en $\text{parent}(i)$. Dit herstelt de ordeningseigenschap tussen i en zijn ouder. Eventueel is nu de ordeningseigenschap tussen $\text{parent}(i)$ en $\text{parent}(\text{parent}(i))$ geschonden. Indien dit zo is wissel dan beide elementen. Ga zo verder tot de binaire hoop is hersteld.

Opmerking 4.38 Het proces van het herstellen van de binaire hoop van beneden naar boven, noemt men soms ook OMHOOG BUBBELEN, omdat het nieuwe element als het ware omhoog bubbelt in de hoop tot het op zijn correcte plaats staat. ∎

Voorbeeld 4.39 In Figuur 4.18 zien we hoe we tewerk gaan om het element met sleutel 1 toe te voegen.

We voegen een nieuwe top toe met sleutel 1 op de plaats waar het volgende blad moet komen. Op dit moment voldoet de binaire boom niet meer aan de ordeningseigenschap voor binaire hopen: de sleutel 1 is kleiner dan de sleutel van zijn ouder (sleutelwaarde 6). Merk ook op dat dit de *enige* plaats is waar de ordeningseigenschap van binaire hopen geschonden is. We kunnen dit herstellen door de sleutelwaarden 1 en 6 van plaats te verwisselen.

Op dit moment hebben we nog steeds geen binaire hoop, aangezien 1 kleiner is dan 5. Merk opnieuw op dat dit de enige plaats is in de binaire hoop



Figuur 4.18: Toevoegen van sleutel 1 aan een binaire hoop.

waar de binaire ordeningseigenschap geschonden is. We wisselen 1 en 5 teneinde deze schending ongedaan te maken. Merk op dat in de nieuwe boom de top met sleutel 5 aan de ordeningseigenshap voldoet want 6 en 8 waren afstammelingen van 5 in de oorspronkelijke binaire hoop.

De binaire boom voldoet nog steeds niet aan de ordeningseigenschap, want 1 is kleiner dan de sleutel van zijn ouder die sleutelwaarde 2 heeft. Wisselen lost dit op, en nu hebben we terug een correcte binaire hoop. Merk op dat 2 kleiner is dan zijn kinderen: dit zal steeds zo zijn aangezien 9 en 5 afstammelingen waren van 2 in de oorspronkelijke binaire hoop.

In dit geval hebben we dus 3 verwisselingen moeten doorvoeren om de binaire hoop te herstellen. Dit is in deze binaire hoop ook het maximaal aantal verwisselingen dat nodig kan zijn. ∎

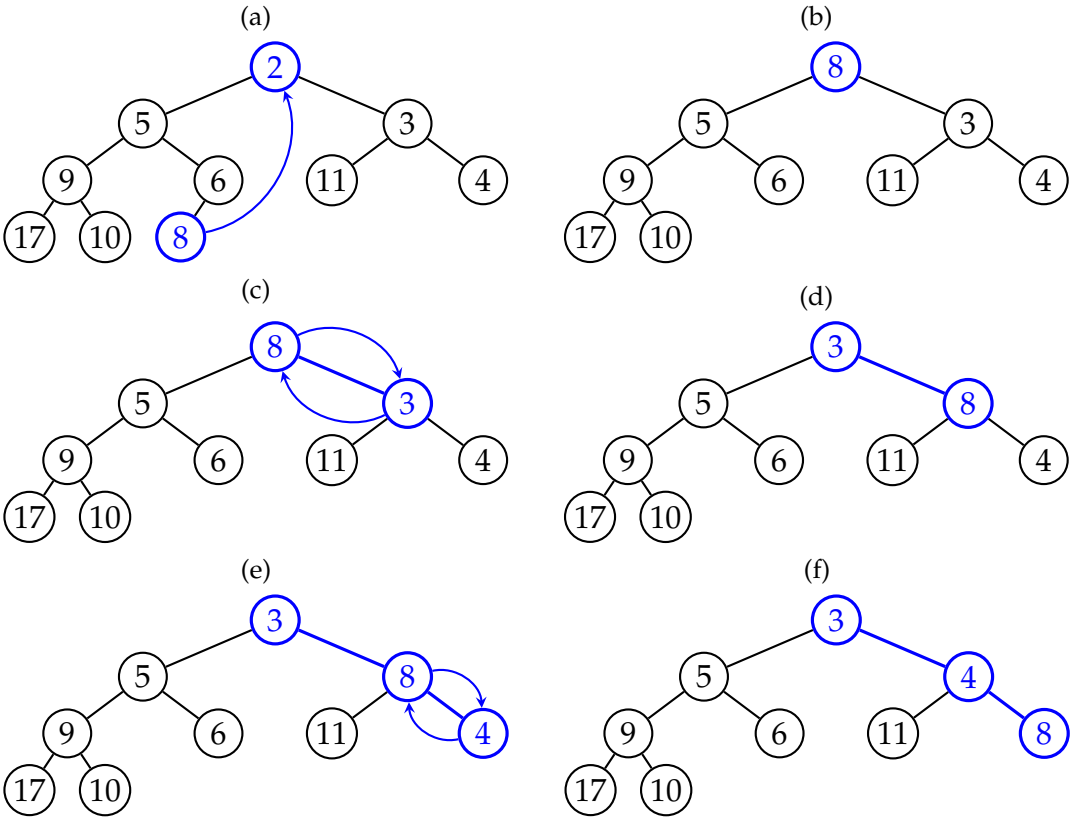
4.6.6 Verwijderen van het element met de kleinste sleutel

Wanneer we het element met de kleinste sleutel verwijderen, dan krijgen we een "gat" aan de wortel van de boom. We vullen dit gat op door het meest rechtse blad op het diepste niveau in dit "gat" te duwen (en het meest rechtse blad op het diepste niveau te verwijderen). Op dit moment is wellicht de ordeningseigenschap geschonden tussen de wortel en zijn kinderen. Herstel dit door de wortel om te wisselen met zijn *kleinste* kind. Ga zo verder tot de binaire hoop hersteld is.

Verwijderen van een element verloopt dus als volgt:

1. Verwissel de wortel met het meest rechtse blad met de grootste diepte.
2. Verwijder het meest rechtse blad; de binaire hoop heeft nu een element minder.
3. Indien de ordeningseigenschap geschonden is in de wortel, herstel deze dan door de wortel en zijn kleinste kind i van plaats te verwisselen. Indien de ordeningseigenschap nu geschonden is in i, herstel ze dan door i te verwisselen met de kleinste van zijn kinderen. Ga zo verder tot de binaire hoop hersteld is.

Opmerking 4.40 Het proces dat gebruikt wordt bij het verwijderen van een element noemt men soms ook OMLAAG BUBBELEN, omdat het laatste ele-



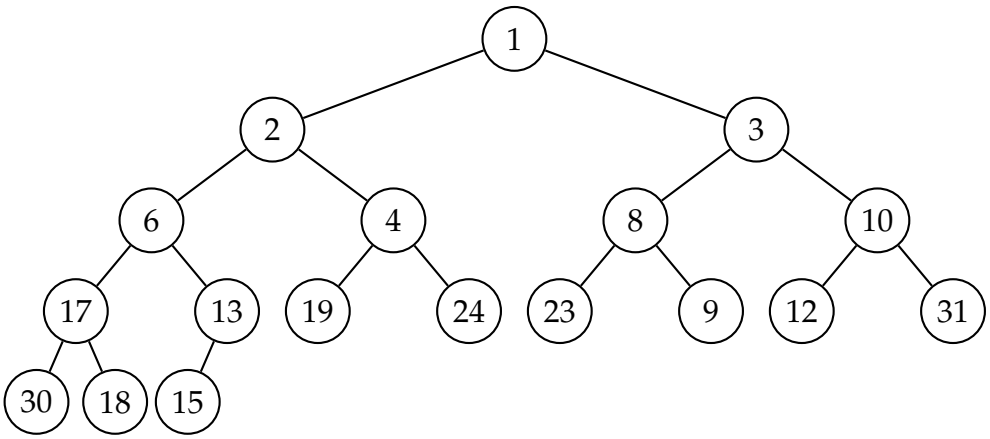
Figuur 4.19: Illustratie van de stappen die gezet worden bij het verwijderen van het kleinste element uit een binaire hoop.

ment van de hoop omlaag bubbelt doorheen de hoop tot het op zijn juiste plaats staat. ∎

Voorbeeld 4.41 In Figuur 4.19 zien we hoe het element met de kleinste sleutelwaarde (2 in dit geval) uit de binaire hoop wordt verwijderd.

In de eerste stap gaan we het "gat" dat achtergelaten wordt door 2, opvullen met de sleutelwaarde van het meest rechtste blad (8 in dit geval). Daarna wordt dit blad verwijderd. Op die manier hebben we nog steeds een complete binaire boom, maar de ordeningseigenschap is nu geschonden in de wortel want $8 > \min(5, 3)$. We wisselen dus 8 en 3 om deze schending van de ordeningseigenschap te herstellen.

Nu hebben we nog steeds geen binaire hoop, want de top met sleutelwaarde 8 voldoet nog steeds niet aan de ordeningseigenschap: $8 > \min(11, 4)$. De



Figuur 4.20: Een binaire hoop.

oplossing bestaat erin om de sleutelwaarden 8 en 4 van plaats te wisselen. Na deze verwisseling voldoet de binaire boom aan de ordeningseigenschap voor binaire hopen en is dus opnieuw een geldige binaire hoop. ∎

4.6.7 Tijdscomplexiteit van de bewerkingen

Een binaire hoop met n toppen heeft diepte $\mathcal{O}\big(\lg(n)\big)$. Dit betekent onmiddellijk dat de tijdscomplexiteit van de bewerkingen toevoegen van een element en verwijderen van het kleinste element in het slechtste geval $\mathcal{O}\big(\lg(n)\big)$ is.

Het ophalen van het kleinste element is uiteraard een bewerking die in constante tijd kan uitgevoerd worden: $T(n) = \mathcal{O}(1)$.

4.6.8 Oefeningen

1. Start met een lege binaire hoop. Voeg achtereenvolgens de volgende elementen toe aan de binaire hoop:

   $$11, 13, 1, 15, 6, 5, 9, 16, 3, 10, 7, 4, 12, 14, 2.$$

   Teken de resulterende hoop na elke toevoeging.

2. Beschouw de binaire hoop in Figuur 4.20. Verwijder de drie kleinste elementen uit deze binaire hoop. Teken de binaire hoop na elke verwijdering.

3. Wat worden de relaties in Eigenschap 4.36 wanneer een binaire hoop wordt opgeslaan in een array met als eerste index 0?

4. Waar kan het maximale element zich bevinden in een binaire hoop, aannemende dat de binaire hoop bestaat uit verschillende elementen.

HOOFDSTUK 5

Graafalgoritmes

We starten dit hoofdstuk met de definitie van en begrippen omtrent GRAFEN. Daarna volgen, net zoals in het hoofdstuk m.b.t. bomen, twee DATASTRUCTUREN om grafen voor te stellen in het computergeheugen. Vervolgens bekijken we verschillende manieren om te ZOEKEN in een graaf. Het DIEPTE-EERST zoekproces vormt de basis voor een efficiënt algoritme voor TOPOLOGISCH SORTEREN, i.e. het plaatsen van taken in een volgorde zodanig dat de afhankelijkheden voldaan zijn. Voor het vinden van het KORTSTE PAD tussen twee knopen zien we voor gewogen grafen het ALGORITME VAN DIJKSTRA. Vervolgens zien we twee algoritmen voor het vinden van een MINIMALE KOST OPSPANNENDE BOOM, nl. de algoritmes van PRIM en KRUSKAL. Om af te sluiten bespreken we kort het HANDELSREIZIGERSPROBLEEM waarvoor er waarschijnlijk geen efficiënt algoritme bestaat. We zien één benaderingsalgoritme gebaseerd op minimale kost opspannende bomen.

5.1 Terminologie m.b.t. grafen

In heel wat praktische situaties heeft men te maken met de situatie waarin 'objecten' verbonden zijn door een bepaalde relatie: steden zijn met elkaar verbonden m.b.v. wegen, computers zijn verbonden m.b.v. netwerkkabels, luchthavens zijn met elkaar verbonden door directe vluchten. In al de opgesomde situaties is het vaak zinvol om een bepaalde 'kost' te associëren met een verbinding: de afstand in kilometer, de communicatiesnelheid van de verbinding, of de duur van de rechtstreekse vlucht. Al deze situaties kunnen gemodelleerd worden aan de hand van een graaf. In dit hoofdstuk be-

studeren we grafen, samen met enkele fundamentele algoritmen voor grafen.

Definitie 5.1 Een GRAAF G bestaat uit een verzameling KNOPEN V, en een verzameling BOGEN E. Elke boog verbindt twee knopen, en we noteren $e = (v, w)$. De graaf G wordt genoteerd als het koppel (V, E), dus $G = (V, E)$. ∎

Wanneer v en w verbonden zijn door een boog, i.e. wanneer $(v, w) \in E$, dan zijn v en w ADJACENT. We zeggen dan ook dat de knopen v en w INCIDENT zijn met e, en omgekeerd ook dat e incident is met v en w. De BUREN van een knoop v zijn alle knopen w die adjacent zijn met v, en we schrijven buren(v) om deze verzameling aan te duiden. Het aantal buren van een knoop v noemt men de GRAAD van deze knoop.

Het aantal knopen van de graaf, i.e. $\#(V)$, wordt de ORDE van de graaf genoemd, en wordt vaak als n genoteerd. Het aantal bogen, i.e. $\#(E)$, noemt men de GROOTTE van de graaf en wordt traditioneel aangeduid met m.

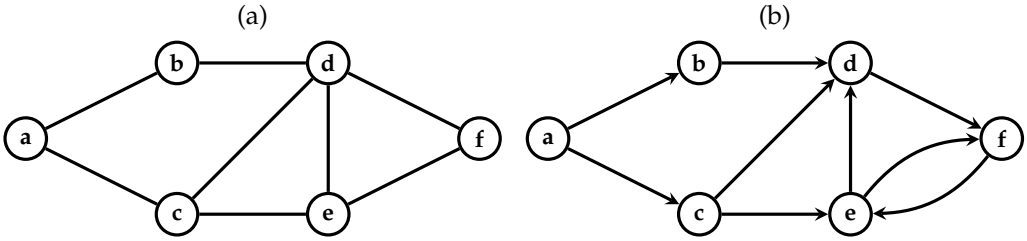
Opmerking 5.2 In plaats van "knoop" zullen we in deze tekst soms ook het synoniem "top" gebruiken om een element van V te benoemen. ∎

Opmerking 5.3 Wanneer de boogparen niet geordend zijn dan spreekt men van een ONGERICHTE graaf. Wanneer de boogparen wel geordend zijn dan heeft men een GERICHTE graaf. In een gerichte graaf heeft een boog (v, w) een STAART v en een KOP w. ∎

Voorbeeld 5.4 Een voorbeeld van een ongerichte graaf is de vriendschapsgraaf van Facebook. De knopen van deze graaf zijn de personen met een account op Facebook. Twee knopen zijn adjacent als de personen "vriend" zijn van elkaar op Facebook. Als v "vriend" is van w, dan is omgekeerd ook w "vriend" van v. De vriendschapgraaf van Facebook is dus een ongerichte graaf. ∎

Voorbeeld 5.5 De volgersgraaf van Twitter is een gerichte graaf. De knopen van deze graaf zijn de accounts op Twitter, en er is een boog van v naar w indien v het account w volgt. ∎

Grafen worden grafisch voorgesteld op een manier analoog aan bomen. De knopen van de graaf worden voorgesteld door een bolletje, eventueel voorzien van een label. Wanneer twee knopen adjacent zijn, dan worden de twee



Figuur 5.1: Een voorbeeld van een ongerichte en gerichte graaf.

knopen verbonden door een lijn: dit stelt de boog voor. Wanneer de boog geordend is (m.a.w. wanneer men te maken heeft met een gerichte graaf), dan wordt er een pijl toevoegd die de richting van de boog aangeeft. De pijl wijst van de staart naar de kop.

Voorbeeld 5.6 In Figuur 5.1(a) zien we een ongerichte graaf met als knopenverzameling

$$V = \{a, b, c, d, e, f\}.$$

Aangezien $\#(V) = 6$ is de orde van deze graaf 6. De bogenverzameling E is

$$E = \{(a, b), (a, c), (b, d), (c, d), (c, e), (d, e), (d, f), (e, f)\}.$$

Het aantal elementen van E is 8, en dus is de grootte van de graaf gelijk aan 8.

De buren van knoop c zijn de knopen a, d en e:

$$\text{buren}(c) = \{a, d, e\}.$$

De graad van de top c is dus 3.

In Figuur 5.1(b) zien we een gerichte graaf met dezelfde knopenverzameling

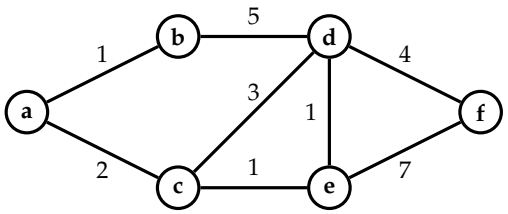
$$V = \{a, b, c, d, e, f\}.$$

Aangezien $\#(V) = 6$ is de orde van deze graaf eveneens 6. De bogenverzameling E is

$$E = \{(a, b), (a, c), (b, d), (c, d), (c, e), (d, f), (e, d), (e, f), (f, e)\}.$$

Het aantal elementen van E is 9, en dus is de grootte van de graaf 9. Merk op dat bogen voor een gerichte graaf steeds genoteerd worden van staart naar kop. De buren van knoop c in de gerichte graaf zijn de knopen d en e:

$$\text{buren}(c) = \{d, e\}.$$



Figuur 5.2: Een gewogen ongerichte graaf.

In de gerichte graaf is de graad van de top c dus 2. ∎

Soms kan het ook handig zijn om extra informatie te associëren met de bogen. Vaak neemt deze informatie de vorm aan van een getal. In dit geval spreekt men van een GEWOGEN graaf. Deze informatie kan bv. zijn: de afstand tussen twee steden in kilometer, de reistijd tussen twee luchthavens enzovoort.

Definitie 5.7 Een PAD in een graaf G is een opsomming van toppen $(v_1, v_2, \ldots, v_k)$ zodanig dat er een boog bestaat tussen v_i en v_{i+1} voor $i \in \{1, 2, \ldots, k-1\}$. De LENGTE van dit pad is $k-1$, zijnde het aantal bogen op dit pad. ∎

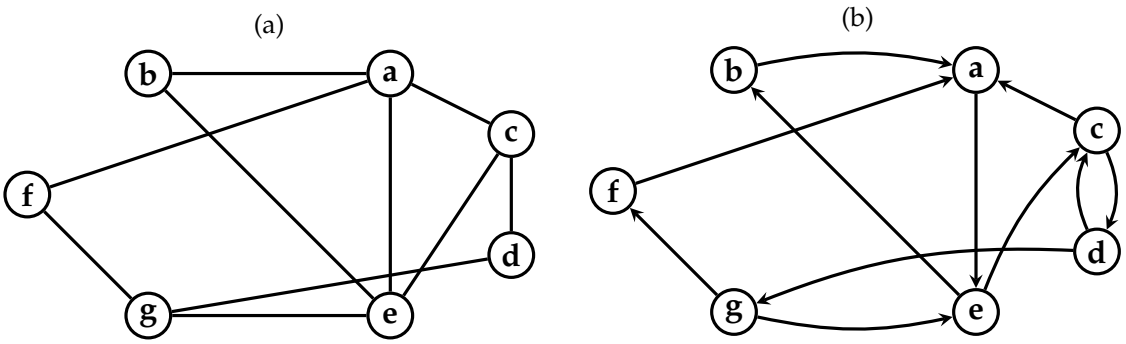
Opmerking 5.8 Het triviale geval $k = 1$ is toegestaan. Elke top v is een pad met lengte nul van v naar v. ∎

Definitie 5.9 Een ENKELVOUDIGE CYKEL in een graaf is een pad waarvan de lengte strikt positief is en dat begint en eindigt in dezelfde knoop, waarbij alle knopen (behalve de start- en eindknoop) verschillend zijn en waarbij bovendien geen boog méér dan een keer wordt doorlopen. ∎

Voorbeeld 5.10 In de ongerichte graaf van Figuur 5.1(a) is (a, b, d, c, e, f) een pad van lengte 5 van a naar f. Dit is *geen* pad in de gerichte graaf van Figuur 5.1(b) omdat er geen boog is van d naar c (enkel van c naar d).

In beide grafen is het pad (d, f, e, d) een enkelvoudige cykel van lengte 3. In de gerichte graaf is (e, f, e) een enkelvoudige cykel van lengte 2, maar dit is *niet* zo in de ongerichte graaf omdat we de boog (e, f) twee keer zouden gebruiken in dit pad.

In de ongerichte graaf is (a, b, d, f, e, d, c, a) een pad van lengte 7 van a naar a. Dit pad is *geen* enkelvoudige cykel want de knoop d wordt twee keer gebruikt! ∎



Figuur 5.3: Twee voorbeeldgrafen voor de oefeningen.

5.1.1 Oefeningen

1. Beschouw de gerichte en ongerichte graaf in Figuur 5.3.

   a) Geef de bogenverzameling van deze twee grafen.

   b) Geef voor beide grafen de verzameling buren(e). Wat is de graad van de knoop e in beide gevallen?

   c) Vind het kortste pad (i.e. het pad met de kleinste lengte) van b naar d in beide grafen.

   d) Vind in beide grafen de langste enkelvoudige cykel die d bevat.

5.2 Datastructuren voor grafen

5.2.1 De adjacentiematrix

We veronderstellen dat de knopen van de graaf $G = (V, E)$ genummerd zijn van 1 t.e.m. n. Wanneer we te maken hebben met een (ongewogen) graaf dan kunnen we deze voorstellen door een ADJACENTIEMATRIX A. Voor deze adjacentiematrix geldt:

$$A_{i,j} = \begin{cases} 1 & \text{als } (i,j) \in E \\ 0 & \text{anders.} \end{cases}$$

Opmerking 5.11 Wanneer de graaf G een ongerichte graaf is, dan is de adjacentiematrix A een symmetrische matrix: wanneer er een boog gaat van i naar j dan is er (bij definitie) ook een boog van j naar i. ∎

Voorbeeld 5.12 Hieronder zien we de adjacentiematrix van de graaf in Figuur 5.1(a):

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

Aangezien de graaf ongericht is, hebben we inderdaad dat A een symmetrische matrix is: $A^T = A$. In dit voorbeeld werd er stilzwijgend verondersteld dat de knoop met label a rangnummer 1 heeft, de knoop met label b rangnummer 2 enzovoort. ∎

Voorbeeld 5.13 Bekijken we de gerichte graaf in Figuur 5.1(b), dan is zijn adjacentiematrix:

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

Merk op dat deze matrix niet symmetrisch is. ∎

De adjacentiematrix kan uitgebreid worden om ook de gewichten van de bogen op te slaan. In dit geval bevat $A_{i,j}$ het gewicht van de boog (i, j). Men dient er dan wel op te letten dat men een "speciale" waarde voorziet om aan te geven dat i en j niet adjacent zijn. Wanneer men zeker weet dat de gewichten positief zijn (bv. omdat ze een afstand voorstellen), dan kan men de waarde -1 nemen voor deze speciale waarde.

De geheugenruimte die een adjacentiematrix in beslag neemt is steeds $\mathcal{O}(n^2)$, onafhankelijk van het aantal bogen in de graaf. Wanneer het aantal bogen klein is in vergelijking met het maximale aantal bogen (n^2 in een gerichte graaf), dan verspilt deze voorstellingswijze heel wat geheugenruimte. Een graaf waarvoor het aantal bogen "klein" is t.o.v. het aantal mogelijke bogen noemt men een IJLE graaf. Merk op dat er niet exact wordt gespecifieerd wat er bedoeld wordt met "klein".

Het bepalen of twee knopen adjacent zijn of niet gebeurt in constante tijd $\mathcal{O}(1)$ aangezien dit neerkomt op het ophalen van een element uit een tweedimensionale array.

Het overlopen van alle buren van een knoop is een bewerking die $\mathcal{O}(n)$ tijd in beslag neemt, onafhankelijk van de graad van de knoop. Men moet immers steeds de volledige rij corresponderend met de knoop overlopen.

Opmerking 5.14 Wanneer de toppen niet genummerd zijn van 1 t.e.m. n maar bv. gelabeld zijn met karakterstrings, dan kan men een "woordenboek" gebruiken om elk label om te zetten naar een geheel getal tussen 1 en n. Een mogelijke implementatie van zo'n woordenboek is m.b.v. een hashmap. ∎

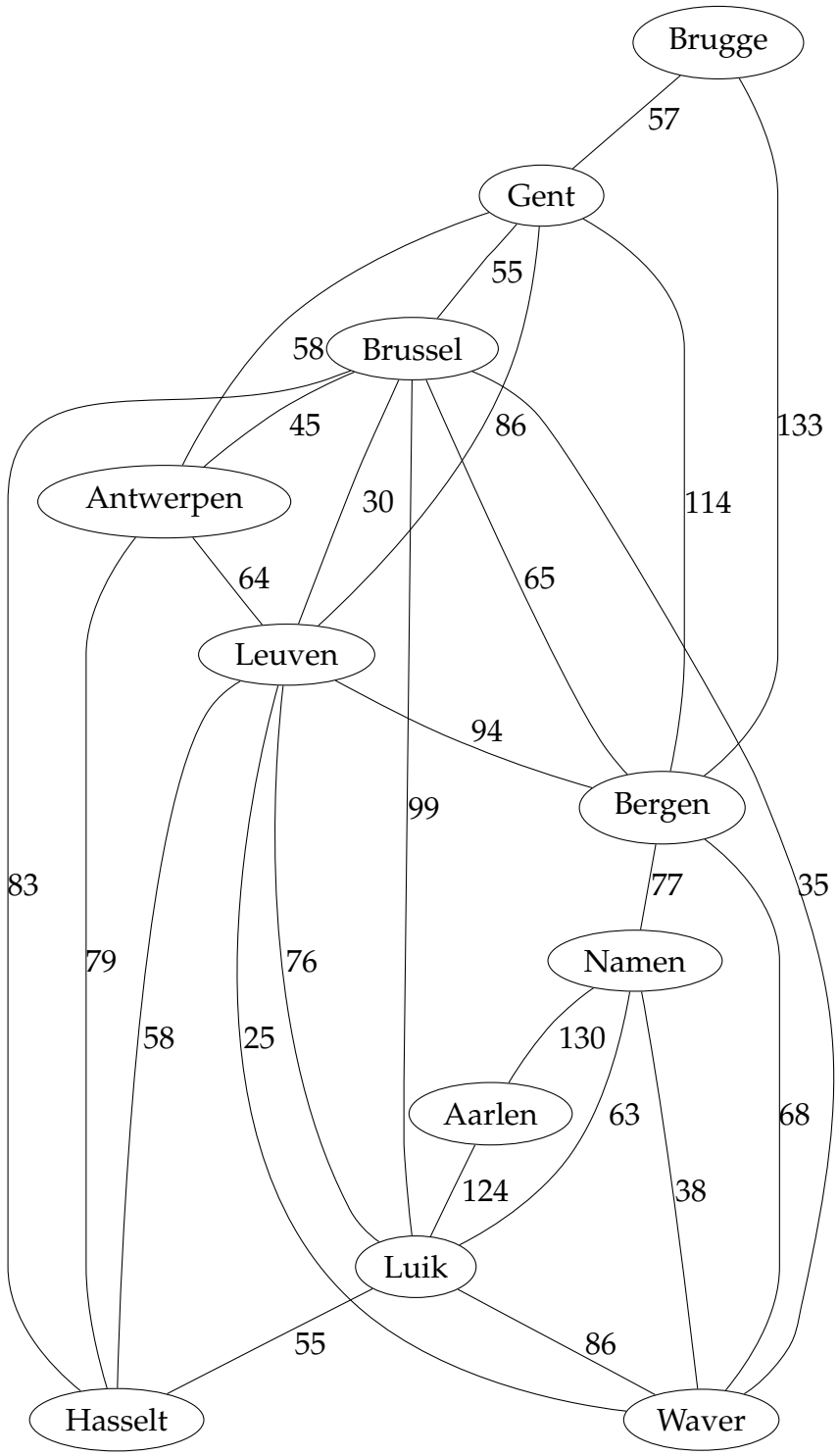
Voorbeeld 5.15 In Figuur 5.4 zien we een gewogen gelabelde graaf. Hieronder zien we het gebruikte woordenboek samen met de corresponderende adjacentiematrix.

Brugge	$\to 1$	∞	57	∞	∞	∞	133	∞	∞	∞	∞	∞
Gent	$\to 2$	57	∞	55	58	86	114	∞	∞	∞	∞	∞
Brussel	$\to 3$	∞	55	∞	45	30	65	∞	∞	99	35	83
Antwerpen	$\to 4$	∞	58	45	∞	64	∞	∞	∞	∞	∞	79
Leuven	$\to 5$	∞	86	30	64	∞	94	∞	∞	76	25	58
Bergen	$\to 6$	133	114	65	∞	94	∞	77	∞	∞	68	∞
Namen	$\to 7$	∞	∞	∞	∞	∞	77	∞	130	63	38	∞
Aarlen	$\to 8$	∞	∞	∞	∞	∞	∞	130	∞	124	∞	∞
Luik	$\to 9$	∞	∞	99	∞	76	∞	63	124	∞	86	55
Waver	$\to 10$	∞	∞	35	∞	25	68	38	∞	86	∞	∞
Hasselt	$\to 11$	∞	∞	83	79	58	∞	∞	∞	55	∞	∞

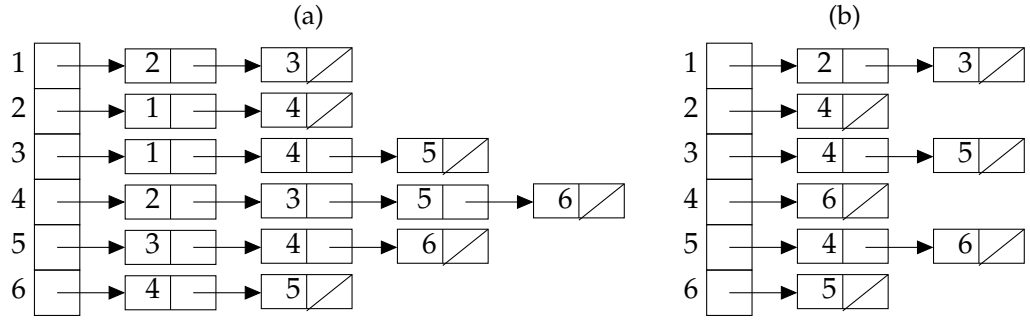
Merk op dat we als speciale waarde hier ∞ hebben gebruikt. We duiden aan dat de graaf geen lussen heeft door op de diagonaal eveneens ∞ te plaatsen. ∎

5.2.2 De adjacentielijst-voorstelling

De adjacentielijst-voorstelling lost het probleem van verspilling van geheugenruimte in de adjacentiematrix-voorstelling voor ijle grafen op.



Figuur 5.4: Graaf met Belgische provinciehoofsteden (en Brussel). Het gewicht van elke boog geeft de afstand tussen de twee betrokken steden weer zoals aangegeven door Google Maps. Gewichten staan steeds rechts van een boog.



Figuur 5.5: Adjacentielijsten van de grafen in Figuur 5.1.

De ADJACENTIELIJST-VOORSTELLING van een graaf G bestaat uit een array van toppen, genummerd 1 t.e.m. n. Op de plaats i van deze array worden, in een lineair gelinkte lijst, de buren van top i bijgehouden.

De adjacentielijst-voorstelling gebruikt $\mathcal{O}(n+m)$ geheugenruimte. Er wordt een array van n toppen bijgehouden, en elk van de m bogen van de graaf geeft aanleiding tot één (of twee) knopen[1] in de geschakelde lijst.

Het bepalen of twee knopen adjacent zijn kan nu *niet* langer in constante tijd gebeuren. Om te weten of i en j adjacent zijn moeten we immers de gelinkte lijst horend bij i overlopen om na te gaan of j in deze lijst aanwezig is.

Als we alle buren van een knoop i willen overlopen dan gebeurt dit nu in een tijd die lineair is in het aantal buren van de knoop i. Dit is theoretisch de best mogelijke uitvoeringstijd.

Voorbeeld 5.16 In Figuur 5.5 ziet men de adjacentielijst-voorstelling voor de grafen in Figuur 5.1. Merk op dat voor de ongerichte graaf elke boog *twee* keer voorkomt in de adjacentielijst-voorstelling. ∎

Opmerking 5.17 (Graaf-implementatie in "moderne"programmeertalen) Een graaf geeft in essentie verbanden aan tussen verschillen objecten, i.e. tussen de knopen. Een Map-datastructuur, ook wel een dictionary genoemd, geeft precies zo'n verbanden aan. De sleutels zijn in dit geval de (labels van de) knopen van de graaf. De waarden zijn typisch één of andere collectie (bv. een set) bestaande uit de buren van refererende knoop. De elementen opgeslaan in deze collectie zijn ofwel (labels van) knopen (bij een ongewogen

[1] Dit zijn dus *niet* de knopen zoals in de definitie van een graaf, maar bv. `NodeList` componenten van een lineair geschakelde lijst.

graaf) ofwel bv. tupels (tweetallen) bestaande uit (het label van) de knoop en het gewicht van de boog. ∎

Voorbeeld 5.18 Stel dat g een Map of dictionary is die de graaf uit Figuur 5.4 voorstelt. Dan is

$$g[\text{"Brugge"}]$$

gelijk aan de verzameling

$$\{(\text{"Gent"}, 57), (\text{"Bergen"}, 133)\},$$

die bestaat uit twee tweetallen. Voor

$$g[\text{"Namen"}]$$

vinden we de volgende verzameling:

$$\{(\text{"Bergen"}, 77), (\text{"Aarlen"}, 130), (\text{"Luik"}, 63), (\text{"Waver"}, 38)\}.$$ ∎

5.2.3 Oefeningen

1. Beschouw de gerichte en ongerichte graaf in Figuur 5.3.
   a) Geef voor beide grafen de adjacentiematrix. Je mag veronderstellen dat a rangnummer 1 heeft, b rangnummer 2 enzovoort.
   b) Geef voor beide grafen de adjacentielijst-voorstelling. Je mag veronderstellen dat a rangnummer 1 heeft, b rangnummer 2 enzovoort.
2. Hoe berekent men de graad van een top i van een graaf G wanneer de adjacentiematrix A van de graaf gegeven is? Geef een algoritme. Wat is de tijdscomplexiteit van deze methode?
3. Hoe berekent men de graad van een top i van een graaf G wanneer de adjacentielijst-voorstelling van de graaf gegeven is? Geef een algoritme. Wat is de tijdscomplexiteit van deze methode?

5.3 Zoeken in Grafen

Veronderstel dat een graaf $G = (V, E)$ gegeven is. Dan kunnen we geïnteresseerd zijn om algoritmes te vinden die de volgende vragen kunnen beantwoorden:

1. Welke knopen kunnen we bereiken vanuit een knoop v?
2. Bestaat er een pad van v naar een specifieke knoop w?
3. Wat is het kortste pad van v naar w?

5.3.1 Generiek Zoeken

Het doel van een zoekalgoritme bestaat erin om in een graaf G vanaf een gegeven knoop v alle knopen te vinden die bereikbaar zijn vanaf v, i.e. alle knopen w waarvoor er een pad bestaat van v naar w. Bovendien zouden we graag hebben dat dit efficiënt gebeurt, i.e. dat de uitvoeringstijd $\mathcal{O}(n + m)$ is.

Een generiek zoekalgoritme voor een gegeven graaf G en startend vanuit een knoop s gaat als volgt: We markeren de knoop s als ontdekt, i.e. initieel bestaat het *ontdekte gebied* enkel uit de knoop s. In elke stap breiden we het ontdekte gebied uit door een boog (u, v) te kiezen waarbij u reeds ontdekt is en v nog niet ontdekt is, i.e. de boog (u, v) kruist als het ware de grens tussen het ontdekte en onontdekte gebied. Op deze manier breiden we het ontdekte gebied uit met de knoop v. We herhalen dit proces tot er geen bogen meer zijn die de grens tussen ontdekt en onontdekt gebied kruisen. Het algoritme wordt gegeven in Algoritme 6.1.

Opmerking 5.19 Dit algoritme is *ondergespecifiëerd* in die zin dat het niet duidelijk is *hoe* de keuze van de boog (u, v) gebeurt. We zullen dit in de volgende secties doen. ∎

Eigenschap 5.20 Wanneer het algoritme voor generiek zoeken eindigt dan geldt voor elke knoop v van G dat v gemarkeerd is als "ontdekt" (i.e. $D[v] =$ **true**) als en slechts als er een pad bestaat van s naar v in G. ∎

Bewijs **We bewijzen het $\Rightarrow$ deel eerst.**
Veronderstel dat het algoritme de top v markeert als ontdekt. We willen nu aantonen dat er een pad bestaat van s naar v. Het gestelde is duidelijk waar voor de eerste ontdekte knoop,

Algoritme 5.1 Generiek zoeken in een graaf.

Invoer Een gerichte of ongerichte graaf $G = (V, E)$ met orde $n > 0$. Een knoop s waarvan het zoeken vertrekt. De knopen zijn genummerd van 1 tot n, i.e. $V = \{1, 2, \ldots, n\}$.

Uitvoer Een array D met $D[v] =$ **true** als en slechts als er een pad bestaat van s naar v.

```
1: function ZoekGeneriek(G, s)
2:     D ← [false, false, ..., false]                ▷ n keer false
3:     D[s] ← true                                   ▷ markeer s
4:     while ∃(u, v): D[u] = true ∧ D[v] = false do
5:         kies een boog (u, v) met D[u] = true ∧ D[v] = false
6:         D[v] ← true                               ▷ markeer v
7:     end while
8:     return D
9: end function
```

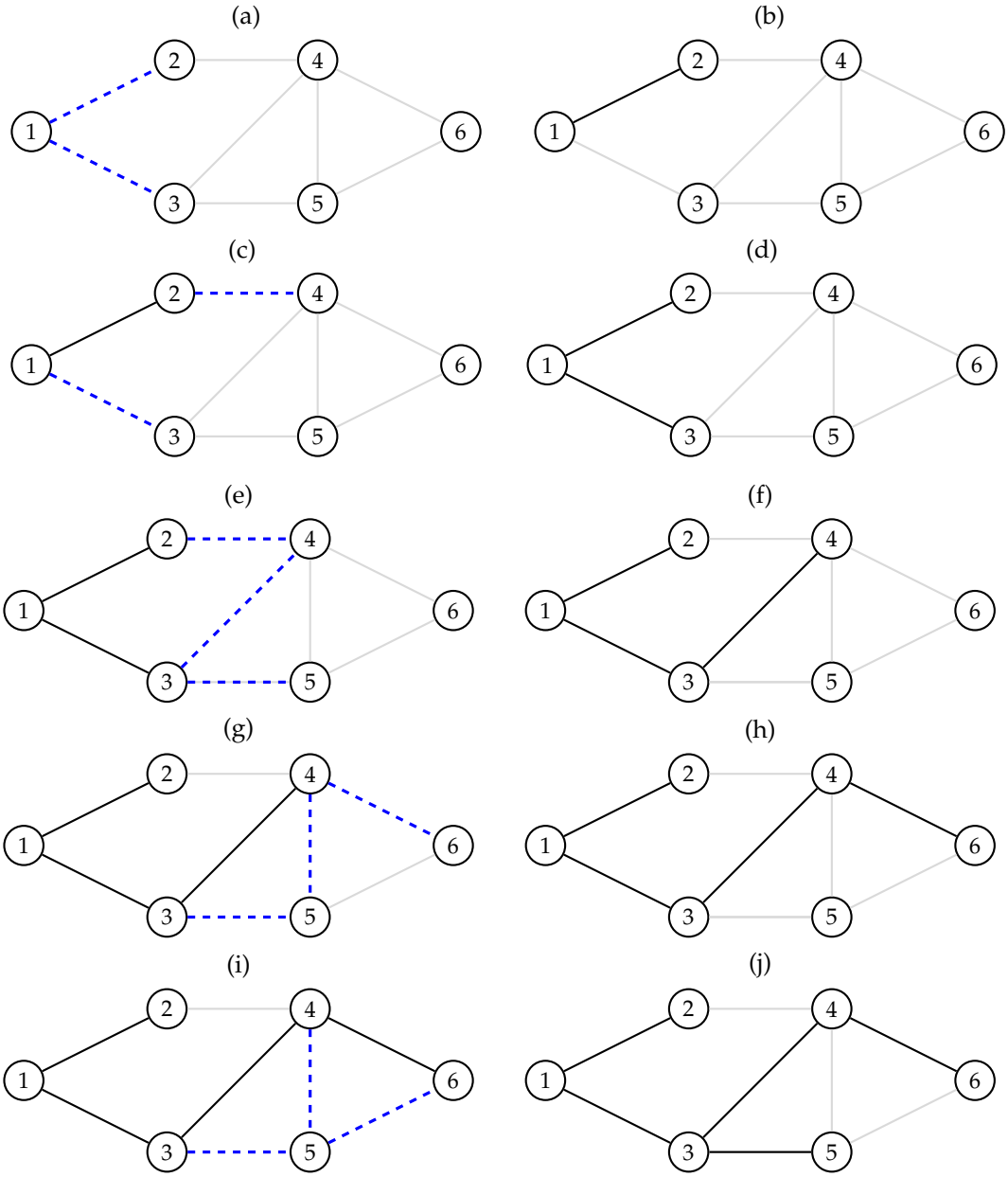
want dit is de knoop s. Veronderstel nu dat het gestelde waar is voor knopen die vóór de m-de knoop ontdekt zijn, en dat v als m-de knoop werd ontdekt, met (u, v) als gekozen boog. Uit de inductiehypothese volgt dat er een pad bestaat van s naar u, dat via de boog (u, v) uitgebreid wordt tot een pad van s naar v.

Nu tonen we het $\Leftarrow$ gedeelte aan.

Omgekeerd bewijzen we nu dat het algoritme alle knopen ontdekt waarvoor een pad bestaat vanaf s. Veronderstel dat dit niet zo is, dus veronderstel dat er een knoop v is waarvoor er een pad bestaat van s naar v maar waarvoor het algoritme eindigt met $D[v] =$ **false**. Beschouw een pad van s naar v en noem u de *eerste* knoop op dit pad waarvoor $D[u] =$ **false** (merk op dat $u \neq s$). Voor de voorganger van u op dit pad, noem deze w, geldt dus $D[w] =$ **true**. Maar dan zou het algoritme nog niet gestopt zijn want op regel 4 voldoet de boog (w, u) aan de voorwaarde, zodanig dat de lus nog minstens één keer zou uitgevoerd worden. M.a.w. wanneer het algoritme stopt zijn alle knopen op het pad van s naar v ontdekt. In het bijzonder is ook v ontdekt. ♢

Voorbeeld 5.21 In Figuur 5.6 ziet men een *mogelijk* verloop van het algoritme voor generiek zoeken. We gebruiken de ongerichte graaf uit Figuur 5.1(a) maar we hebben de knopen expliciet hernoemd als 1 t.e.m. 6.

De startknoop is de knoop 1. In blauwe stippellijn zijn telkens de bogen aangeduid die voldoen aan de voorwaarde op regel 4 van het algoritme. In de rechterkolom ziet men dan telkens de graaf nadat een (random) keuze werd gemaakt uit de blauwe bogen.



Figuur 5.6: Voorbeeld van generiek zoeken.

We sommen hieronder het verloop van het algoritme op:

gemarkeerde knopen	mogelijke bogen	gekozen boog
$\{1\}$	$(1,2),(1,3)$	$(1,2)$
$\{1,2\}$	$(1,3),(2,4)$	$(1,3)$
$\{1,2,3\}$	$(2,4),(3,4),(3,5)$	$(3,4)$
$\{1,2,3,4\}$	$(3,5),(4,5),(4,6)$	$(4,6)$
$\{1,2,3,4,6\}$	$(3,5),(4,5),(5,6)$	$(3,5)$
$\{1,2,3,4,5,6\}$	geen	geen

Merk op dat dit algoritme de bogen *niet* op een systematische manier afloopt. ∎

5.3.2 Breedte-Eerst Zoeken

Het algoritme voor breedte-eerst zoeken is een instantie van het algoritme voor generiek zoeken. Met dit algoritme gaan we de knopen van een graaf G bezoeken in "lagen", i.e. eerst bezoeken we de knoop s zelf, dan de knopen die juist één boog verwijderd zijn van s, dan de knopen die juist twee bogen verwijderd zijn van s, enzovoort.

De datastructuur die gebruikt wordt om breedte-eerst zoeken te implementeren is een FIFO-datastructuur, i.e. een wachtrij. Meer specifiek zal de wachtrij dié knopen v bevatten die reeds zijn ontdekt, maar waarvoor er mogelijks nog buren u zijn die nog niet ontdekt zijn.

De pseudocode voor breedte-eerst zoeken wordt gegeven in Algoritme 5.2.

Voorbeeld 5.22 We voeren breedte-eerst zoeken uit op de ongerichte voorbeeldgraaf in Figuur 5.1(a), maar we noemen de knopen nu expliciet 1 t.e.m. 6.

We duiden op de Figuur 5.7 aan welke *bogen* doorlopen werden door het algoritme, i.e. welke bogen ervoor zorgden dat een knoop werd toegevoegd aan de wachtrij. In de tabel hieronder zie je expliciet de inhoud van de wachtrij telkens regel 6 wordt uitgevoerd. We hebben er hier voor gekozen om de buren steeds te doorlopen in volgorde van hun label, i.e. alsof hun adjacentielijst-voorstelling die is in Figuur 5.5.

Algoritme 5.2 Breedte-eerst zoeken in een graaf.

Invoer Een gerichte of ongerichte graaf $G = (V, E)$ met orde $n > 0$. Een knoop s waarvan het zoeken vertrekt. De knopen zijn genummerd van 1 tot n, i.e. $V = \{1, 2, \ldots, n\}$.

Uitvoer Een array D met $D[v] =$ **true** als en slechts als er een pad bestaat van s naar v.

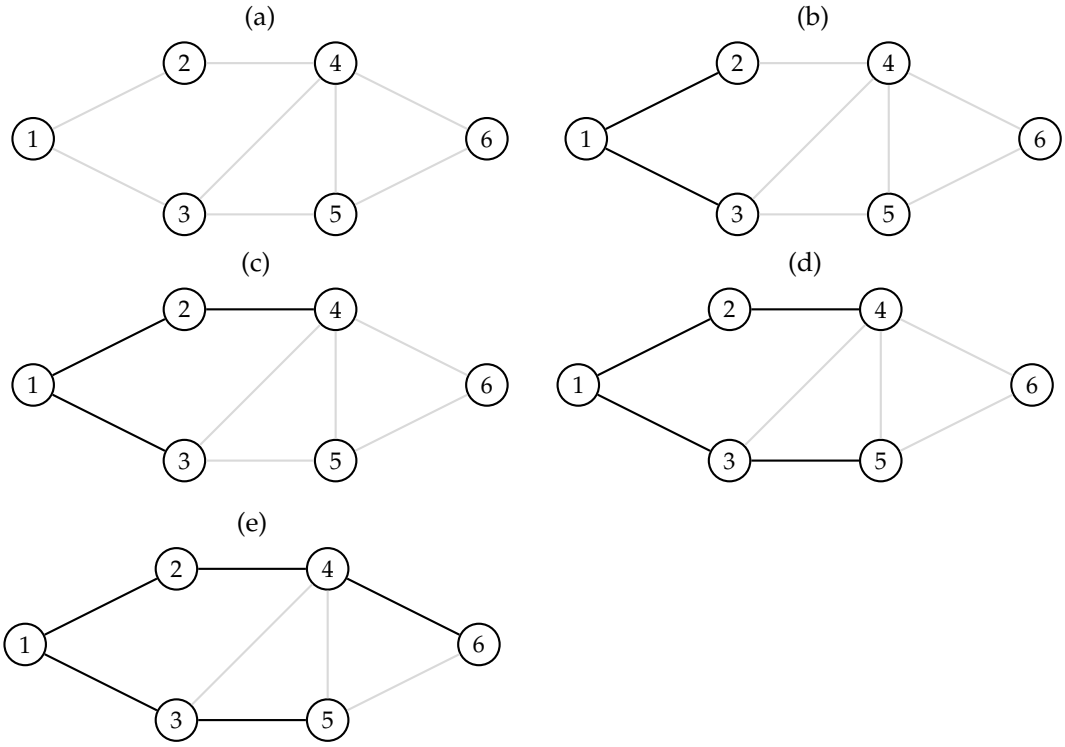
```
1:  function BREEDTEEERST(G, s)
2:      D ← [false, false, ..., false]          ▷ n keer false
3:      D[s] ← true                              ▷ markeer s
4:      Q.init()                                 ▷ wachtrij van knopen
5:      Q.enqueue(s)
6:      while Q ≠ ∅ do
7:          v ← Q.dequeue()
8:          for all w ∈ buren(v) do
9:              if D[w] = false then             ▷ w nog niet ontdekt
10:                 D[w] ← true
11:                 Q.enqueue(w)
12:             end if
13:         end for
14:     end while
15:     return D
16: end function
```

iteratie	Q	D
1	[1]	[T,F,F,F,F,F]
2	[2,3]	[T,T,T,F,F,F]
3	[3,4]	[T,T,T,T,F,F]
4	[4,5]	[T,T,T,T,T,F]
5	[5,6]	[T,T,T,T,T,T]
6	[6]	[T,T,T,T,T,T]
7	[]	[T,T,T,T,T,T]

Het is gemakkelijk om te controleren dat de wachtrij op elk moment knopen bevat van hoogstens twee verschillende "lagen". ∎

Eigenschap 5.23 Wanneer een adjacentielijst-voorstelling gebruikt wordt voor een graaf G, dan is de uitvoeringstijd $T(n, m)$ van Algoritme 5.2 van de grootte-orde $\mathcal{O}(n + m)$. ∎

Bewijs Om te zien dat de uitvoeringstijd van breedte-eerst zoeken inderdaad $\mathcal{O}(n + m)$ is wanneer een adjacentielijst-voorstelling gebruikt wordt redeneren we als volgt:



Figuur 5.7: Stap voor stap uitvoering van breedte-eerst zoeken. Elke figuur toont de bogen die gekozen zijn wanneer de test op regel 6 van Algoritme 5.2 wordt uitgevoerd.

1. Het initialiseren van de array D neemt tijd $\mathcal{O}(n)$.
2. Het is duidelijk dat elke knoop hoogstens één keer wordt toegevoegd aan de wachtrij, en dus wordt elke knoop er hoogstens één keer uit verwijderd. De totale tijd voor de lus op regel 8 is dus hoogstens

$$\sum_{v \in V} \#\big(\text{buren}(v)\big) = 2m \quad \text{in een ongerichte graaf}$$

of

$$\sum_{v \in V} \#\big(\text{buren}(v)\big) = m \quad \text{in een gerichte graaf.}$$

Dus we vinden inderdaad dat de uitvoeringstijd $\mathcal{O}(n + m)$ is aangezien alle bewerkingen in de lus een constante uitvoeringstijd hebben. ♢

5.3.3 Diepte-Eerst Zoeken

Diepte-eerst zoeken is eveneens een instantie van het generieke zoekalgoritme. Bij diepte-eerst zoeken gaan we zo snel mogelijk zo diep mogelijk in

de graaf, i.e. we bekijken steeds de buren van de 'diepste' ontdekte knoop. Dit wordt bereikt met een algoritme gelijkaardig aan breedte-eerst zoeken, maar we vervangen de wachtrij van breedte-eerst zoeken door een *stapel* (stack). In Algoritme 5.3 geven we een recursieve implementatie en maken dus gebruik van de (impliciete) stapel van methode-oproepen (call-stack).

Algoritme 5.3 Diepte-eerst zoeken in een graaf.

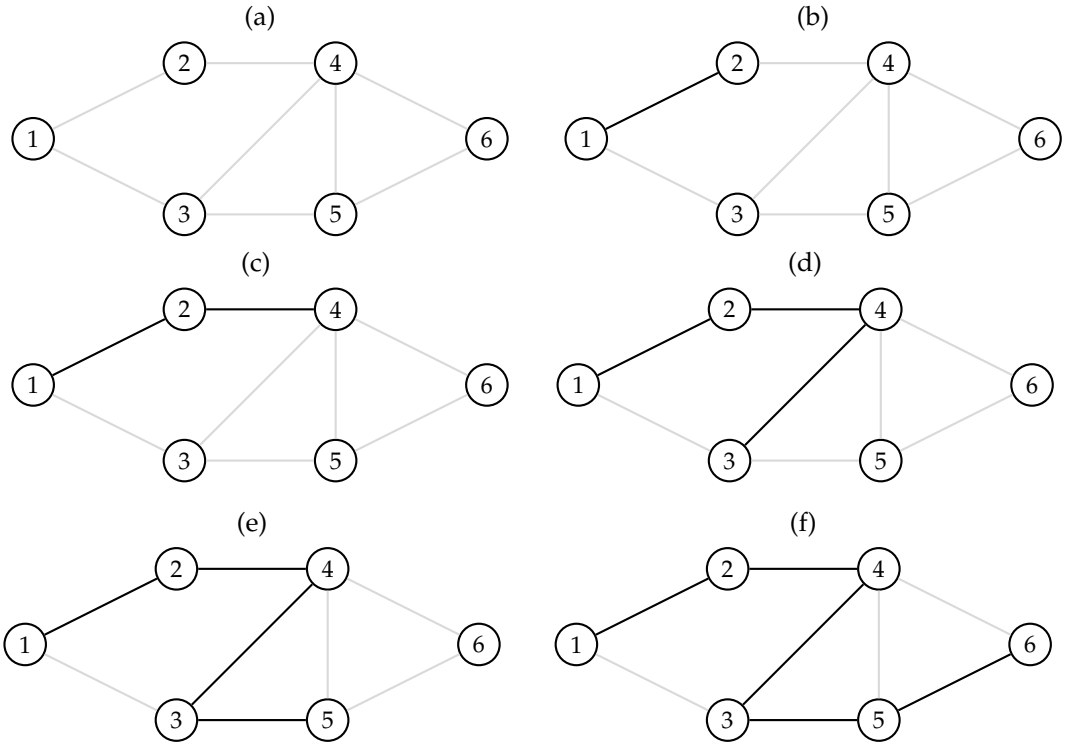
Invoer Een gerichte of ongerichte graaf $G = (V, E)$ met orde $n > 0$. Een knoop s waarvan het zoeken vertrekt. De knopen zijn genummerd van 1 tot n, i.e. $V = \{1, 2, \ldots, n\}$.

Uitvoer Een array D met $D[v] =$ **true** als en slechts als er een pad bestaat van s naar v.

```
 1: function DiepteEerst(G, s)
 2:     D ← [false, false, ..., false]                ▷ n keer false
 3:     DiepteEerstRecursief(G, s, D)
 4:     return D
 5: end function
 6: function DiepteEerstRecursief(G, v, D)
 7:     D[v] ← true                                   ▷ markeer v
 8:     for all w ∈ buren(v) do
 9:         if D[w] = false then                      ▷ w nog niet ontdekt
10:             DiepteEerstRecursief(G, w, D)
11:         end if
12:     end for
13: end function
```

Voorbeeld 5.24 In Figuur 5.8 zien we hoe diepte-eerst zoeken wordt uitgevoerd op de ongerichte graaf in Figuur 5.1(a), startend vanaf top 1. Telkens wanneer regel 10 uitgevoerd wordt voegen we de boog (v, w) toe op de figuur. Hieronder zien we de verschillende recursieve oproepen die gebeuren bij het uitvoeren van DiepteEerst.

```
DiepteEerstRecursief(G, 1, [false, false, false, false, false, false])
  DiepteEerstRecursief(G, 2, [true, false, false, false, false, false])
    DiepteEerstRecursief(G, 4, [true, true, false, false, false, false])
      DiepteEerstRecursief(G, 3, [true, true, false, true, false, false])
        DiepteEerstRecursief(G, 5, [true, true, true, true, false, false])
          DiepteEerstRecursief(G, 6, [true, true, true, true, true, false])
```



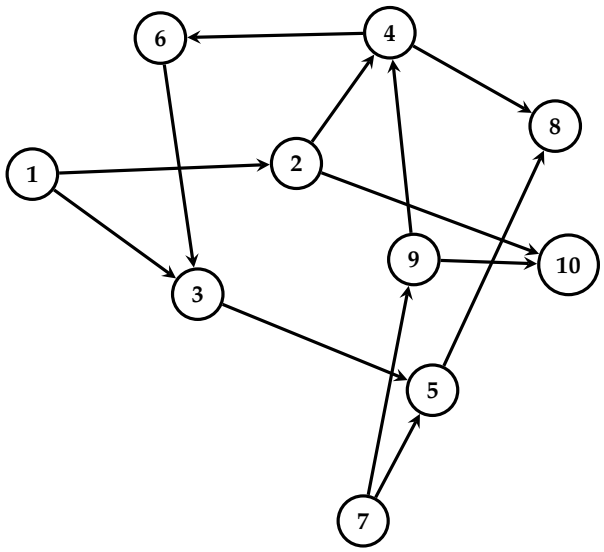
Figuur 5.8: Stap voor stap uitvoering van diepte-eerst zoeken.

Merk op dat het eerste statement van DIEPTEEERSTRECURSIEF ook aan $D[6]$ de waarde **true** zal toekennen. De uiteindelijke return-waarde zal dan ook een array zijn waarvan alle componenten de waarde **true** hebben. ∎

5.3.4 Toepassing: Topologisch Sorteren

We veronderstellen in deze sectie dat G een gerichte graaf is, zonder enkelvoudige cykels. Een voorbeeld van zo'n graaf is een PRECEDENTIEGRAAF, waarbij de verschillende knopen taken voorstellen. Er is een boog van knoop v naar w als taak v moet afgewerkt zijn alvorens taak w kan aangevat worden.

Voorbeeld 5.25 Veronderstel dat een softwareproject bestaat uit 10 modules. Sommige modules zijn afhankelijk van andere modules, en deze andere modules moeten dus eerst gecompileerd worden. We modelleren het project als een graaf, waarbij elke knoop een module voorstelt en er is een boog (v, w) als module w afhankelijk is van v, i.e. als v moet gecompileerd wor-



Figuur 5.9: Precedentiegraaf voor 10 software modules. Er is een boog van v naar w als v moet gecompileerd worden voor w.

den voor w. De graaf wordt gegeven in Figuur 5.9. We vragen ons af in welke volgorde we de modules kunnen compileren. ∎

Een topologische sortering is een volgorde van de taken zodanig dat voor elke taak w al zijn voorgaande taken zijn afgewerkt alvorens w wordt gestart. We maken dit meer formeel in de volgende definitie.

Definitie 5.26 Een TOPOLOGISCHE SORTERING van een gerichte graaf G kent aan elke knoop v een verschillend rangnummer $f(v)$ toe van 1 t.e.m. n zodanig dat de volgende eigenschap geldt:

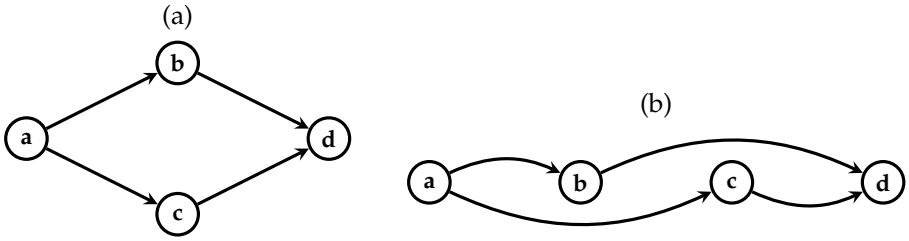
$$\forall (u, v) \in E \colon f(u) < f(v), \tag{5.1}$$

m.a.w. als (u, v) een boog is in de graaf dan is het rangnummer van de kop v groter dan het rangnummer van de staart u. ∎

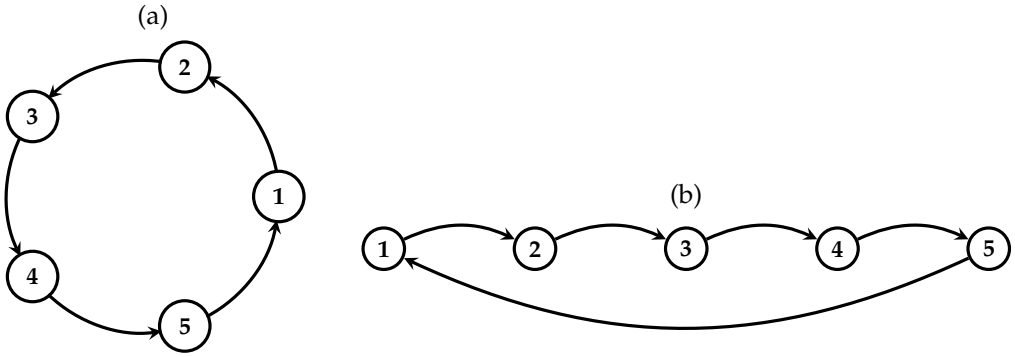
Wanneer we alle knopen op een rechte lijn tekenen, gesorteerd volgens het rangnummer van hun topologische sortering, dan zullen alle bogen vooruit wijzen.

Voorbeeld 5.27 In Figuur 5.10(a) zien we een gerichte graaf zonder cykels. Een mogelijke topologische sortering is de volgende:

$$a, b, c, d.$$



Figuur 5.10: Links zie je een gerichte graaf zonder cykels, rechts een topologische sortering van deze graaf (impliciet gegeven door de volgorde van de knopen op de lijn).



Figuur 5.11: Een cykel met 5 knopen en een *poging* tot topologische sortering van diezelfde cykel. Merk op dat de boog $(5, 1)$ achteruit wijst.

Inderdaad, kijken we naar Figuur 5.10(b) dan zien we een andere voorstelling van dezelfde graaf met alle knopen op een rechte lijn volgens de gegeven sortering. We zien onmiddellijk dat alle bogen vooruit wijzen, dus is dit een geldige topologische sortering. ∎

Opmerking 5.28 Het is duidelijk dat een gerichte graaf met een cykel *geen* topologische sortering heeft. Inderdaad, als we alle knopen van een cykel op een rechte lijn tekenen, dan zal de boog die de cykel "sluit" steeds achteruit wijzen. ∎

Voorbeeld 5.29 In Figuur 5.11 ziet men een cykel samen met een poging tot topologische sortering van die cykel. De boog $(5, 1)$ die de cykel sluit wijst achteruit; dit is niet toegelaten voor een topologische sortering. ∎

Eigenschap 5.30 Wanneer een gerichte graaf G geen enkelvoudige cykels heeft, dan bestaat er een topologische sortering van G. ∎

Bewijs We bewijzen eerst dat elke gerichte graaf G zonder enkelvoudige cykels steeds een knoop zonder buren heeft.

We geven een bewijs uit het ongerijmde: veronderstel dat G een gerichte graaf is zonder cykels maar dat elke knoop steeds minstens één buur heeft. We kunnen dan een pad

$$(v_1, v_2, v_3, \ldots, v_{n+1}) \tag{5.2}$$

vormen. (We lopen nooit "vast" want elke knoop heeft minstens één buur.) Het pad (5.2) bestaat uit $n+1$ knopen, maar de graaf G heeft slechts n knopen. Dit betekent dat minstens één knoop herhaald wordt in dit pad, en zo krijgen we een cykel in G. Dit is in strijd met de veronderstelling dat G geen cykel bevat. Er moet dus minstens één knoop zijn zonder buren.

Het is duidelijk dat wanneer G een topologische sortering heeft, dat dan de knoop met rangnummer n een knoop is zonder buren. Inderdaad, veronderstel dat v een knoop is die wel buren heeft en dat $f(v) = n$, dan betekent dit volgens eigenschap (5.1) dat $f(u) > n$ wanneer $(v, u) \in E$, maar dit is onmogelijk aangezien het grootste rangnummer dat we kunnen toekennen n is. M.a.w. in een topologische sortering zijn de enige knopen v waarvoor $f(v) = n$ kan zijn die knopen waarvoor buren$(v) = \emptyset$.

Een eenvoudig algoritme om een topologische sortering te vinden is dan het volgende.

1. Kies een knoop v zonder buren (zo is er minstens één), en stel $f(v) = n$.
2. Doe nu (recursief) hetzelfde voor de graaf $G - \{v\}$[2].

Dit algoritme werkt om de volgende reden: veronderstel dat de knoop v het rangnummer i krijgt, dan heeft deze knoop in de huidige graaf geen buren. Alle bogen die vertrekken vanuit v zijn dus reeds verwijderd uit de graaf G. Dat betekent dat voor een boog (v, u) er steeds geldt dat $f(u) > i$. ♢

Het bewijs van Eigenschap 5.30 geeft een algoritmische manier om een topologische sortering te vinden. We kunnen echter ook het algoritme voor diepte-eerst zoeken aanpassen zodanig dat het een topologische sortering produceert.

Het idee is het volgende: met diepte-eerst zoeken bekijken we vanuit *elke* knoop v steeds alle knopen w waarvoor er een pad bestaat van v naar w.

[2] $G - \{v\}$ is de graaf die men vindt door v en al de bogen incident met v te verwijderen uit de graaf G. Uiteraard heeft deze graaf ook geen (enkelvoudige) cykels.

Dit betekent dat in een topologische sortering v steeds vóór w zal moeten komen.

We houden een lijst van knopen S bij die initieel leeg is. Wanneer we *alle* knopen bereikbaar vanuit v hebben ontdekt dan voegen we v vooraan toe aan de lijst.

Het proces van diepte-eerst zoeken moet eventueel meerdere malen worden herhaald tot we alle knopen van de graaf hebben ontdekt. Het is immers mogelijk dat vanuit de gekozen startknoop niet alle knopen van de graaf bereikbaar zijn.

Om na te gaan of de graaf een cykel heeft of niet gaan we als volgt tewerk. Onontdekte knopen krijgen een statuscode 0; zolang een knoop nog niet is toegevoegd aan de lijst S (maar wel reeds ontdekt is) heeft die statuscode 1. Wanneer een knoop volledig is afgewerkt (en aan S wordt toegevoegd) dan krijgt deze statuscode 2. Stel nu dat we de buren van v bekijken en we zien een knoop w met statuscode 1. Dit betekent dat er een pad is van w naar v, maar de boog van v naar w sluit dit pad, en we hebben dus een cykel ontdekt in de graaf. De volledige pseudocode vind je in Algoritme 5.4.

Voorbeeld 5.31 We bepalen nu een mogelijke compilatievolgorde voor de modules. We voeren Algoritme 5.4 uit en we doorlopen de knopen steeds volgens stijgend label. We houden de array D bij, de stapel van methode-oproepen, alsook de lijst S.

Algoritme 5.4 Topologisch sorteren van een graaf.

Invoer Een gerichte graaf $G = (V, E)$ met orde $n > 0$. De knopen zijn genummerd van 1 tot n, i.e. $V = \{1, 2, \ldots, n\}$.

Uitvoer Een topologische sortering van G indien mogelijk, **false** anders.

```
1:  function SORTEERTOPOLOGISCH(G)
2:      global cycleDetected ← false                    ▷ globale variabele
3:      D ← [0, 0, ..., 0]                              ▷ n keer 0
4:      S ← ∅                                           ▷ S is lege lijst
5:      for all s ∈ V do
6:          if D[s] = 0 then                            ▷ s nog niet gezien
7:              DfsTopo(G, s, D, S)                     ▷ S en D referentieparameters
8:              if cycleDetected = true then            ▷ controleer op cykel
9:                  return false
10:             end if
11:         end if
12:     end for
13:     return S
14: end function
15: function DFSTOPO(G, v, D, S)
16:     D[v] ← 1                                        ▷ markeer v als 'bezig'
17:     for all w ∈ buren(v) do
18:         if D[w] = 0 ∧ cycleDetected = false then    ▷ w nog niet ontdekt
19:             DfsTopo(G, w, D, S)
20:         else if D[w] = 1 then                       ▷ cykel ontdekt w ⇝ v → w
21:             cycleDetected ← true
22:         end if
23:     end for
24:     D[v] ← 2                                        ▷ markeer v als 'voltooid'
25:     voeg v vooraan toe aan S                        ▷ ken rangnummer toe aan v
26: end function
```

	1	2	3	4	5	6	7	8	9	10	stack	S
(a)	0	0	0	0	0	0	0	0	0	0		$\emptyset$
(b)	1	0	0	0	0	0	0	0	0	0	1	$\emptyset$
(c)	1	1	0	0	0	0	0	0	0	0	1, 2	$\emptyset$
(d)	1	1	0	1	0	0	0	0	0	0	1, 2, 4	$\emptyset$
(e)	1	1	0	1	0	1	0	0	0	0	1, 2, 4, 6	$\emptyset$
(f)	1	1	1	1	0	1	0	0	0	0	1, 2, 4, 6, 3	$\emptyset$
(g)	1	1	1	1	1	1	0	0	0	0	1, 2, 4, 6, 3, 5	$\emptyset$
(h)	1	1	1	1	1	1	0	1	0	0	1, 2, 4, 6, 3, 5, 8	$\emptyset$
(i)	1	1	1	1	1	1	0	2	0	0	1, 2, 4, 6, 3, 5	8
(j)	1	1	1	1	2	1	0	2	0	0	1, 2, 4, 6, 3	5, 8
(k)	1	1	2	1	2	1	0	2	0	0	1, 2, 4, 6	3, 5, 8
(l)	1	1	2	1	2	2	0	2	0	0	1, 2, 4	6, 3, 5, 8
(m)	1	1	2	2	2	2	0	2	0	0	1, 2	4, 6, 3, 5, 8
(n)	1	1	2	2	2	2	0	2	0	1	1, 2, 10	4, 6, 3, 5, 8
(o)	1	1	2	2	2	2	0	2	0	2	1, 2	10, 4, 6, 3, 5, 8
(p)	1	2	2	2	2	2	0	2	0	2	1	2, 10, 4, 6, 3, 5, 8
(q)	2	2	2	2	2	2	0	2	0	2		1, 2, 10, 4, 6, 3, 5, 8
(r)	2	2	2	2	2	2	1	2	0	2	7	1, 2, 10, 4, 6, 3, 5, 8
(s)	2	2	2	2	2	2	1	2	1	2	7, 9	1, 2, 10, 4, 6, 3, 5, 8
(t)	2	2	2	2	2	2	1	2	2	2	7	9, 1, 2, 10, 4, 6, 3, 5, 8
(u)	2	2	2	2	2	2	2	2	2	2		7, 9, 1, 2, 10, 4, 6, 3, 5, 8

Uit de tabel leiden we af dat een mogelijke compilatievolgorde van de modules gegeven wordt door:

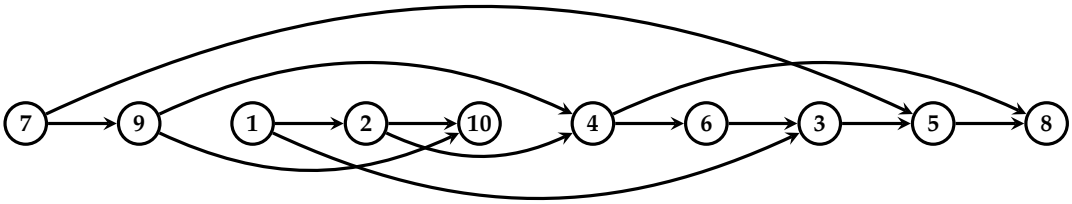
$$7, 9, 1, 2, 10, 4, 6, 3, 5, 8.$$

In Figuur 5.12 controleert men inderdaad dat alle bogen vooruit wijzen. ∎

5.3.5 Oefeningen

1. Een ongerichte graaf is GECONNECTEERD als en slechts als er een pad bestaat tussen elke twee knopen v en w.

   a) Ga na dat de bovenstaande definitie equivalent is met zeggen dat er een pad bestaat van een bepaalde knoop s naar alle andere knopen.



Figuur 5.12: Een topologische sortering van de graaf in Figuur 5.9.

b) Schrijf een methode IsGeconnecteerd die nagaat of een ongerichte graaf geconnecteerd is (return-waarde **true**) of niet (return-waarde **false**). Doe dit door de methode BreedteEerst aan te passen.

2. Vind alle mogelijke topologische sorteringen van de graaf in Figuur 5.10.

3. Vind de compilatievolgorde van de modules in Figuur 5.9 wanneer de labels in dalende volgorde worden doorlopen.

4. Veronderstel nu dat er in de graaf van Figuur 5.9 een extra boog $(8, 6)$ wordt toegevoegd. Pas nu het algoritme voor topologisch sorteren toe.

5.4 Kortste Pad Algoritmen

5.4.1 Kortste Pad in een Ongewogen Graaf

Wanneer een graaf G ongewogen is, dan is een kortste pad van knoop v naar een knoop u een pad van v naar u dat het kleinste aantal bogen bevat. Omdat bij breedte-eerst zoeken de graaf "laag per laag" overlopen wordt kunnen we het algoritme gemakkelijk aanpassen om voor elke knoop v de lengte van het kortste pad van s naar v terug te geven. Dit gaat als volgt: het kortste pad van de knoop s naar zichzelf heeft uiteraard lengte 0. Telkens wanneer we een knoop w ontdekken m.b.v. de boog (v, w) dan noteren we de afstand van s tot w als één meer dan de afstand van s tot v. De resulterende pseudocode vind je in Algoritme 5.5.

Voorbeeld 5.32 We bekijken opnieuw de uitvoering van het breedte-eerst algoritme 5.2 in Voorbeeld 5.22. De uitvoering van Algoritme 5.5 ontdekt de knopen in exact dezelfde volgorde. In de tabel hieronder geven we de

Algoritme 5.5 Kortste pad in een ongewogen graaf

Invoer Een gerichte of ongerichte ongewogen graaf $G = (V, E)$ met orde $n > 0$. Een knoop s waarvan het zoeken vertrekt. De knopen zijn genummerd van 1 tot n, i.e. $V = \{1, 2, \ldots, n\}$.

Uitvoer De array D met $D[v]$ de kortste afstand van s tot v; als $D[v] = \infty$ dan is er geen pad van s naar v.

```
1:  function KortstePadOngewogen(G, s)
2:      D ← [∞, ∞, ..., ∞]                                  ▷ n keer ∞
3:      D[s] ← 0                ▷ kortste pad van s naar zichzelf heeft lengte 0
4:      Q.init()                                             ▷ wachtrij van knopen
5:      Q.enqueue(s)
6:      while Q ≠ ∅ do
7:          v ← Q.dequeue()
8:          for all w ∈ buren(v) do
9:              if D[w] = ∞ then                             ▷ w nog niet ontdekt
10:                 D[w] ← D[v] + 1
11:                 Q.enqueue(w)
12:             end if
13:         end for
14:     end while
15:     return D
16: end function
```

inhoud van de afstandenarray D weer telkens wanneer regel 6 wordt uitgevoerd.

	1	2	3	4	5	6
(a)	0	∞	∞	∞	∞	∞
(b)	0	1	1	∞	∞	∞
(c)	0	1	1	2	∞	∞
(d)	0	1	1	2	2	∞
(e)	0	1	1	2	2	3

Men controleert gemakkelijk dat de laatste lijn in deze tabel inderdaad de correcte afstanden aanduidt van knoop 1 naar alle (andere) knopen. ∎

5.4.2 Dijkstra's Algoritme

Wanneer de graaf gewogen is, i.e. wanneer elke boog e een gewicht, genoteerd gewicht(e), heeft, dan is men meestal niet geïnteresseerd in het pad

met het minste aantal bogen, maar in het pad met het kleinste totale gewicht. In dit geval zal Algoritme 5.5 niet langer het correcte antwoord geven, zelfs niet als we regel 10 van het Algoritme 5.5 aanpassen om rekening te houden met de gewichten van de bogen.

Voorbeeld 5.33 Veronderstel dat we in Algoritme 5.5 regel 10 als volgt herschrijven:

$$D[w] \leftarrow D[v] + \text{gewicht}(v, w).$$

Wanneer we dit algoritme uitvoeren voor de gewogen graaf in Figuur 5.2 startend vanaf knoop 1, dan verandert de afstandenarray als volgt:

	1	2	3	4	5	6
(a)	0	∞	∞	∞	∞	∞
(b)	0	1	2	∞	∞	∞
(c)	0	1	2	6	∞	∞
(d)	0	1	2	6	3	∞
(e)	0	1	2	6	3	10

Men gaat echter eenvoudig na dat de lengte van het kortste pad van knoop 1 naar knoop 4, gewicht 4 heeft en niet 6 zoals dit algoritme aangeeft. Idem voor knoop 6: het kortste pad van knoop 1 naar knoop 6 heeft gewicht 8, i.p.v. 10 zoals aangegeven door dit foutieve algoritme. ∎

Een algoritme dat in het geval *alle gewichten positief zijn* het juiste antwoord geeft is Dijkstra's algoritme. De sleutelideeën van dit algoritme zijn de volgende:

1. Op elk moment in het algoritme zijn de knopen verdeeld in twee disjuncte verzamelingen: een verzameling S van knopen v waarvoor de kortste afstand van s tot v reeds gekend is, en een verzameling Q van knopen waarvoor de kortste afstand nog niet met zekerheid gekend is[3].

2. Voor elke knoop v die tot Q behoort houden we steeds de kortste afstand $D[v]$ bij van een pad van s naar v *dat enkel uit knopen van S bestaat, met uitzondering van de laatste knoop*.

[3]Omdat S en Q elkaars complement zijn moet in een implementatie maar één van beide verzamelingen bijgehouden worden.

3. We voegen telkens die knoop v van Q toe aan S waarvoor $D[v]$ minimaal is onder alle knopen van Q. Voor de buren w van v die tot Q behoren moeten we eventueel $D[w]$ aanpassen. Het pad van s naar v (dat nu enkel uit knopen van S bestaat) uitgebreid met de boog (v, w) zou eventueel korter kunnen zijn dan het tot dan toe gevonden kortste pad.

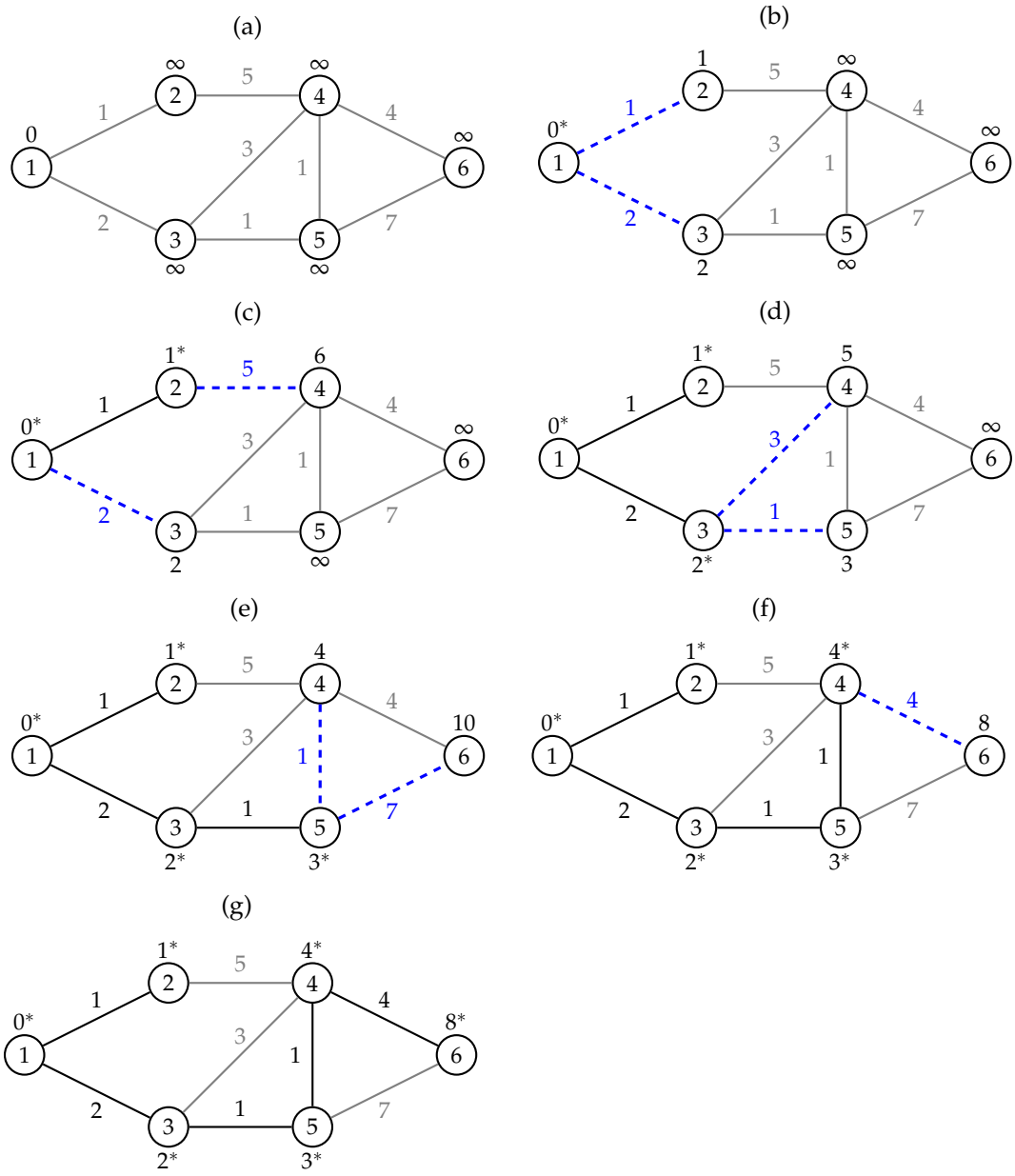
Algoritme 5.6 Het algoritme van Dijkstra

Invoer Een gerichte of ongerichte gewogen graaf $G = (V, E)$ met orde $n > 0$. Alle gewichten zijn positief. Een knoop s waarvan het zoeken vertrekt. De knopen zijn genummerd van 1 tot n, i.e. $V = \{1, 2, \ldots, n\}$.

Uitvoer De array D met $D[v]$ de kortste afstand van s tot v; als $D[v] = \infty$ dan is er geen pad van s naar v.

```
1:  function DIJKSTRA(G, s)
2:      D ← [∞, ∞, ..., ∞]                                  ▷ n keer ∞
3:      D[s] ← 0                ▷ kortste pad van s naar zichzelf heeft lengte 0
4:      Q ← V                   ▷ knopen waarvan kortste afstand nog niet is bepaald
5:      while Q ≠ ∅ do
6:          zoek v ∈ Q waarvoor D[v] minimaal is (voor knopen in Q)
7:          verwijder v uit Q
8:          for all w ∈ buren(v) ∩ Q do
9:              if D[w] > D[v] + gewicht(v, w) then
10:                 D[w] ← D[v] + gewicht(v, w)          ▷ korter pad naar w
11:             end if
12:         end for
13:     end while
14:     return D
15: end function
```

Voorbeeld 5.34 We passen Dijkstra's algoritme toe op de graaf in Figuur 5.2. In onderstaande tabel geven we telkens de inhoud van de array D wanneer regel 5 uitgevoerd wordt. Wanneer er een sterretje bij een afstand staat betekent dit dat de overeenkomstige knoop verwijderd is uit Q.



Figuur 5.13: Stap voor stap uitvoering van het algoritme van Dijkstra. In elke stap wordt de tot dan toe gevonden kortste afstand tot knoop 1 aangeduid. Wanneer er een sterretje bij staat dan is dit met 100% zekerheid de kortste afstand. De blauwe bogen in stippellijn geven voor elke knoop u (met afstand verschillend van ∞) aan wat de boog is van een knoop van S naar u die het gevonden kortste pad beëindigt.

	1	2	3	4	5	6
(a)	0	∞	∞	∞	∞	∞
(b)	0^*	1	2	∞	∞	∞
(c)	0^*	1^*	2	6	∞	∞
(d)	0^*	1^*	2^*	5	3	∞
(e)	0^*	1^*	2^*	4	3^*	10
(f)	0^*	1^*	2^*	4^*	3^*	8
(g)	0^*	1^*	2^*	4^*	3^*	8^*

De verschillende stappen worden ook geïllustreerd in Figuur 5.13. Het is interessant om te zien dat bv. voor knoop 4 de afstand tot drie keer toe wordt verlaagd omdat er steeds kortere paden worden ontdekt. ∎

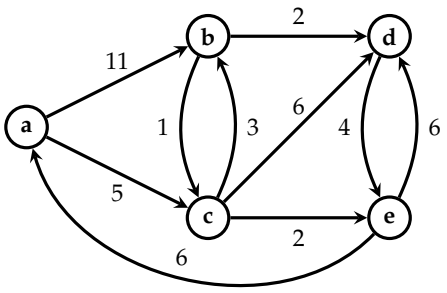
Implementatiedetails

We zien dat in Dijkstra's algoritme er herhaalde minimum berekeningen gebeuren: op regel 6 moet immers herhaaldelijk de knoop v gevonden worden met de minimale waarde voor $D[v]$. We hebben reeds gezien dat een *binaire hoop* hiervoor zeer geschikt is, omdat deze een prioriteitswachtrij implementeert. Een efficiënte implementatie van Dijkstra's algoritme zal dus gebruikmaken van een binaire hoop waarbij steeds aan de volgende twee invarianten voldaan is:

1. De elementen in de hoop zijn de elementen van Q.
2. De sleutel voor elk element v is de waarde van $D[v]$.

Op die manier wordt het ophalen en verwijderen van het minimum op regel 6 een efficiënte bewerking (logaritmisch in het aantal elementen van Q).

Op regel 10 moet echter de sleutelwaarde van w verkleind worden: deze operatie is standaard niet voorzien in een binaire hoop. Wanneer men echter *zou* weten wat de positie is van w in de binaire hoop, dan kan men gemakkelijk de sleutelwaarde aanpassen: dit komt neer op het laten omhoog bubbelen van w tot op zijn juiste plaats. Wat men dus moet doen is een array P bijhouden waarin voor elke knoop v zijn positie in de binaire hoop wordt bijgehouden. Operaties op de binaire hoop moeten dan natuurlijk ook deze array aanpassen (i.e. wanneer bv. twee elementen van plaats worden verwisseld in de binaire hoop dan moeten ook de overeenkomstige posities in de array P worden aangepast).



Figuur 5.14: Een gewogen, gerichte graaf.

5.4.3 Oefeningen

1. In Algoritme 5.5 wordt nu enkel de afstand van elke knoop v tot de startknoop s bijgehouden. In veel toepassingen heeft men echter ook een pad nodig dat deze minimale afstand realiseert.

   a) Pas de pseudo-code van Algoritme 5.5 aan zodanig dat er een tweede array P wordt teruggeven zodanig dat $P[v]$ de knoop geeft die de voorganger (predecessor) is van v op een kortste pad van s naar v.

   b) Pas je aangepaste algoritme toe op de gerichte graaf in Figuur 5.9 startend vanaf knoop 1. Ga ervan uit dat knopen steeds worden bezocht in stijgende volgorde. Hoe zit de array P er na afloop uit?

   c) Schrijf een algoritme dat als invoer de array P neemt en een knoop v. Het algoritme geeft een lijst terug die het kortste pad van s naar v bevat (in de juiste volgorde).

2. Beschrijf hoe je volgend probleem kan oplossen als een kortste pad probleem. Gegeven een lijst van Engelstalige 5-letterwoorden. Woorden worden *getransformeerd* door juist één letter van het woord te vervangen door een andere letter. Geef een algoritme dat nagaat of een woord w_1 omgezet kan worden in een woord w_2. Indien dit het geval is dan moet je algoritme ook de tussenliggende woorden tonen voor de kortste sequentie van transformaties die w_1 in w_2 omzet.

3. Vind voor de graaf in Figuur 5.4 de lengte van het kortste pad van Brugge naar alle andere steden. Voer hiertoe het algoritme van Dijkstra uit.

4. Vind voor de graaf in Figuur 5.14 (de lengte van) het kortste pad van de knoop *a* naar alle andere knopen. Voer hiertoe het algoritme van Dijkstra uit (en houd ook bij wat de kortste paden zijn).

5.5 Minimale Kost Opspannende Bomen

5.5.1 Minimale Kost Opspannende Bomen

Veronderstel dat een wegennetwerk gegeven is dat ervoor zorgt dat we van elke stad naar elke andere stad kunnen rijden, dan kunnen we ons afvragen welke wegen *essentieel* zijn om van elke stad naar elke andere stad te kunnen rijden. Wanneer het *aantal* wegen minimaal is, dan spreken we van een opspannende boom.

Definitie 5.35 Een OPSPANNENDE BOOM van een ongerichte graaf $G = (V, E)$ is een verzameling van bogen T, met $T \subseteq E$, zodanig dat $G' = (V, T)$ een pad heeft tussen elke twee knopen van V, en zodanig dat G' geen enkelvoudige cykels heeft. ∎

In het algemeen heeft een ongerichte graaf *veel* opspannende bomen. Wanneer de graaf gewogen is, dan zijn we vaak geïnteresseerd in de opspannende boom (of bomen) met het minimale gewicht, waarbij het gewicht (of de kost) van de boom gedefinieerd wordt als de som van de gewichten van zijn bogen, dus

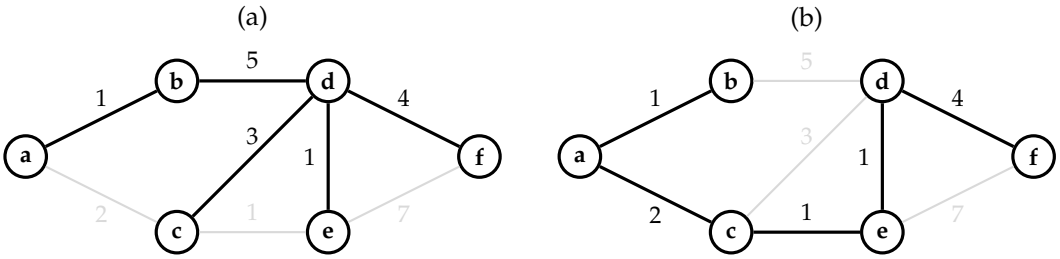
$$\text{gewicht}(T) = \sum_{t \in T} \text{gewicht}(t).$$

Definitie 5.36 Een MINIMALE KOST OPSPANNENDE BOOM T van de graaf G is een opspannende boom zodanig dat voor alle andere opspannende bomen T' van G geldt dat

$$\text{gewicht}(T) \leq \text{gewicht}(T').$$ ∎

Voorbeeld 5.37 In Figuur 5.15 ziet men twee opspannende bomen voor dezelfde graaf. Men controleert gemakkelijk dat dit inderdaad opspannende bomen zijn:

1. Het is duidelijk dat ze geen cykel bevatten.



Figuur 5.15: Twee opspannende bomen. De donkere bogen zijn deel van de opspannende boom.

2. Men ziet eenvoudig dat er een pad is tussen elke twee knopen van de graaf.

Voor de opspannende boom in Figuur 5.15(a) zien we dat

$$\begin{aligned}\text{gewicht}(T) &= \sum_{t \in T} \text{gewicht}(t) \\ &= 1 + 5 + 3 + 1 + 4 \\ &= 14.\end{aligned}$$

Het gewicht van de boom in Figuur 5.15(b) is

$$\begin{aligned}\text{gewicht}(T') &= \sum_{t \in T'} \text{gewicht}(t) \\ &= 1 + 2 + 1 + 1 + 4 \\ &= 9.\end{aligned}$$

De boom T is dus zeker *geen* minimale kost opspannende boom, want de boom T' heeft een lager gewicht. ∎

5.5.2 Prims Algoritme

Het algoritme voor generiek zoeken kan eenvoudig aangepast worden om een opspannende boom terug te geven. Inderdaad, in Figuur 5.6 ziet men in de laatste deelfiguur een opspannende boom. Het algoritme voor generiek zoeken genereert dus blijkbaar reeds een opspannende boom. We dienen enkel bij te houden welke bogen gekozen worden. Door de bogen op een *gulzige* manier te kiezen, i.e. door steeds de goedkoopste boog te kiezen die

het ontdekte gebied uitbreidt, bekomt men een minimale kost opspannende boom. Dit is de essentie van het algoritme van Prim.

Het is zo dat, in tegenstelling tot het algoritme voor generiek zoeken, er geen speciale startknoop is. Het blijkt dat dit niet uitmaakt: het algoritme werkt correct (zonder bewijs!) voor om het even welke startknoop.

Algoritme 5.7 Prims algoritme voor een minimale kost opspannende boom

Invoer Een ongerichte gewogen graaf $G = (V, E)$ met orde $n > 0$. De knopen zijn genummerd van 1 tot n, i.e. $V = \{1, 2, \ldots, n\}$.

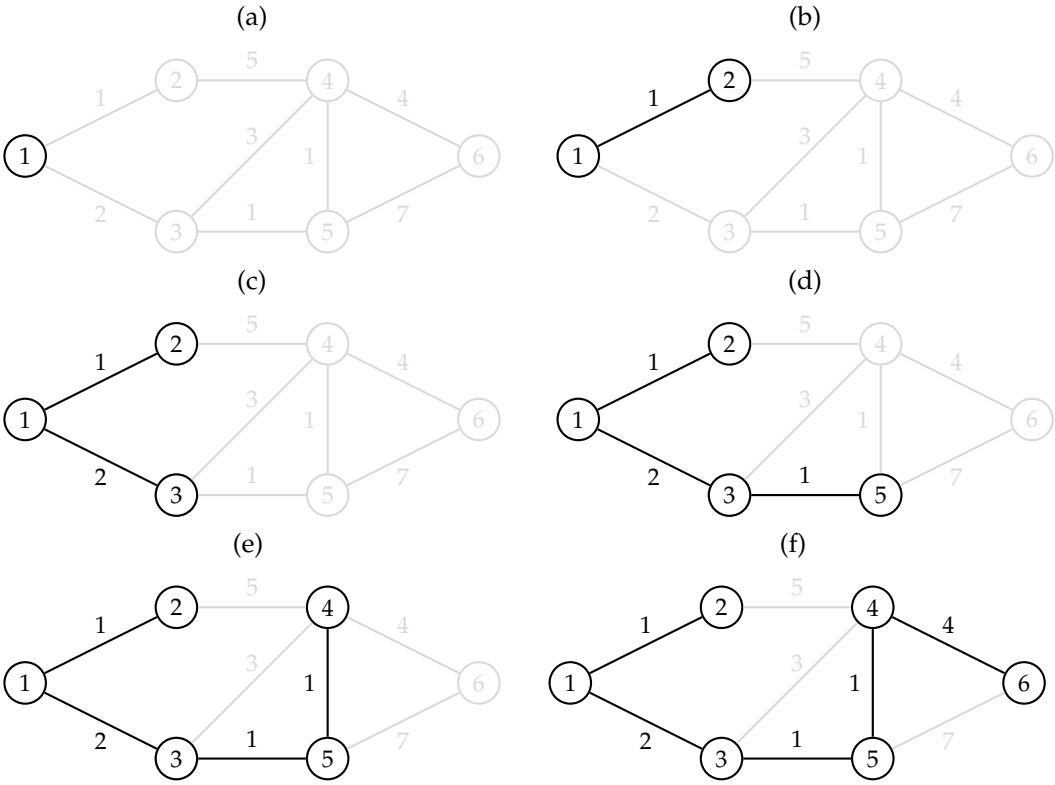
Uitvoer Een verzameling T van bogen die een minimale kost opspannende boom is.

```
1:  function PRIM(G)
2:      D ← [false, false, ..., false]                ▷ n keer false
3:      D[1] ← true                                   ▷ kies knoop 1 als startknoop
4:      T ← ∅                                         ▷ gekozen bogen
5:      while ∃(u, v): D[u] = true ∧ D[v] = false do
6:          kies (u, v) met D[u] = true ∧ D[v] = false met minimaal gewicht
7:          D[v] ← true
8:          T ← T ∪ {(u, v)}                          ▷ Voeg boog (u, v) toe aan boom
9:      end while
10:     return T
11: end function
```

Voorbeeld 5.38 We voeren Prims algoritme uit op de gewogen graaf van Figuur 5.2. In onderstaande tabel kunnen we zien in welke volgorde de bogen worden gekozen:

gemarkeerde knopen	mogelijke bogen	gekozen boog
$\{1\}$	$(1,2), (1,3)$	$(1,2)$
$\{1,2\}$	$(1,3), (2,4)$	$(1,3)$
$\{1,2,3\}$	$(2,4), (3,4), (3,5)$	$(3,5)$
$\{1,2,3,5\}$	$(2,4), (3,4), (5,4), (5,6)$	$(5,4)$
$\{1,2,3,4,5\}$	$(4,6), (5,6)$	$(4,6)$
$\{1,2,3,4,5,6\}$	geen	geen

Figuur 5.16 illustreert dit proces. ∎



Figuur 5.16: Stap voor stap uitvoering van Prims algoritme. De knopen waarvoor $D[v] =$ **true** alsook de inhoud van T worden getoond telkens regel 5 wordt uitgevoerd.

5.5.3 Kruskals Algoritme

Kruskals algoritme is een alternatief algoritme om het probleem van het vinden van een minimale kost opspannende boom op te lossen. Net als Prims algoritme is het een gulzig algoritme, maar daar waar in Prims algoritme de gekozen bogen steeds met elkaar verbonden zijn is dat in Kruskals algoritme niet het geval.

Samengevat komt het algoritme van Kruskal erop neer dat men eerst de bogen sorteert volgens stijgend gewicht. Vervolgens loopt men deze lijst van bogen af, waarbij men telkens een boog selecteert op voorwaarde dat deze geen cykel veroorzaakt onder de bogen die reeds gekozen zijn.

Voorbeeld 5.39 We voeren Kruskals algoritme uit op de graaf in Figuur 5.2. We sorteren de bogen volgens stijgend gewicht en we krijgen:

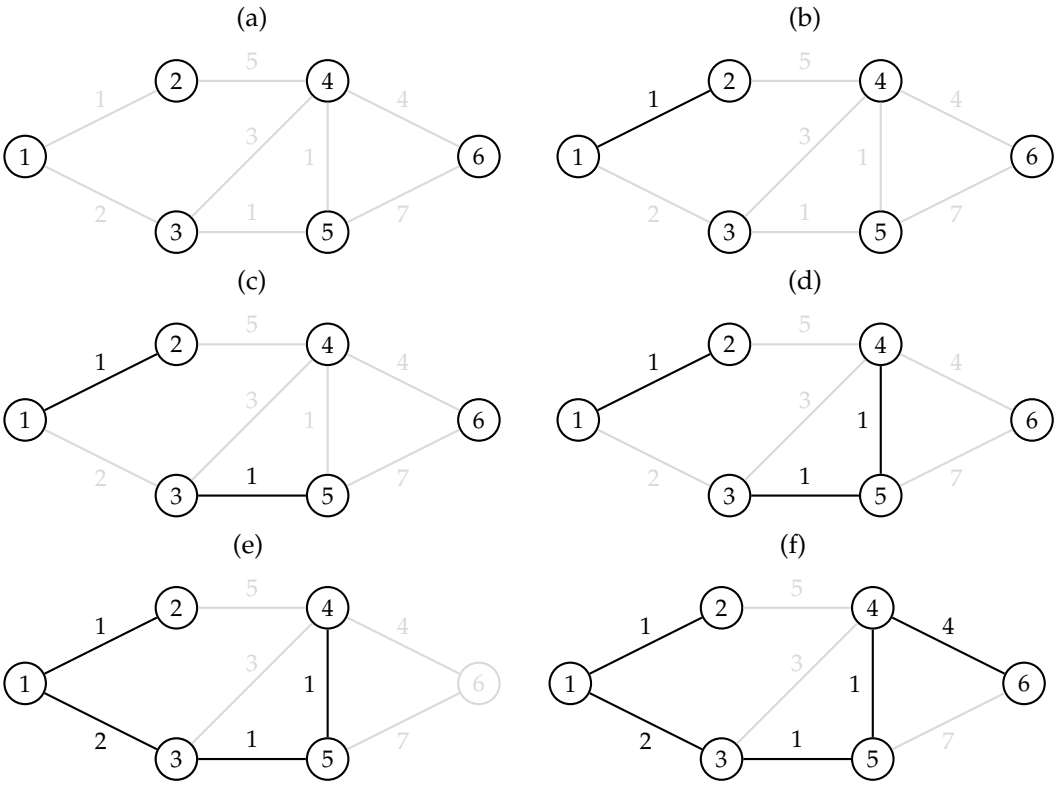
$$(1,2),(3,5),(4,5),(1,3),(3,4),(4,6),(2,4),(5,6).$$

Algoritme 5.8 Kruskals algoritme voor een minimale kost opspannende boom

Invoer Een ongerichte gewogen graaf $G = (V, E)$ met orde $n > 0$. De knopen zijn genummerd van 1 tot n, i.e. $V = \{1, 2, \ldots, n\}$.

Uitvoer Een verzameling T van bogen die een minimale kost opspannende boom is.

```
1:  function KRUSKAL(G)
2:      T ← ∅                                   ▷ Start met lege boom
3:      E' ← sorteer E volgens stijgend gewicht
4:      for all e' ∈ E' do
5:          if T ∪ e' heeft geen cykel then
6:              T ← T ∪ e'
7:          end if
8:      end for
9:      return T
10: end function
```



Figuur 5.17: Stap voor stap uitvoering van Kruskals algoritme.

Kruskals algoritme overloopt deze bogen één voor één en wanneer een boog geen cyckel veroorzaakt onder de reeds geselecteerde bogen wordt deze ook geselecteerd.

De eerste vier bogen worden zonder probleem geselecteerd. De volgende boog die in aanmerking komt is boog $(3, 4)$, maar dan zouden we een cykel krijgen, namelijk $(3, 5, 4, 3)$ (zie Figuur 5.17(e)), en dus wordt deze boog *niet* gekozen. De boog $(4, 6)$ wordt wel gekozen. De laatste twee bogen tenslotte worden ook niet gekozen want deze veroorzaken allebei een cykel onder de reeds gekozen bogen. ∎

Opmerking 5.40 Merk op dat in de beschrijving van het algoritme *niet* wordt gezegd hoe we een cykel gaan detecteren. Dit kan m.b.v. (een variant van) diepte-eerst zoeken, maar er bestaan efficiëntere manieren om dit te doen. Dit wordt hier verder niet behandeld. ∎

5.5.4 Oefeningen

1. Vind een minimale kost opspannende boom m.b.v. het algoritme van Prim voor de graaf in Figuur 5.4. Neem als startknoop "Brugge".
2. Vind een minimale kost opspannende boom m.b.v. het algoritme van Kruskal voor de graaf in Figuur 5.4.

5.6 Het Handelsreizigersprobleem

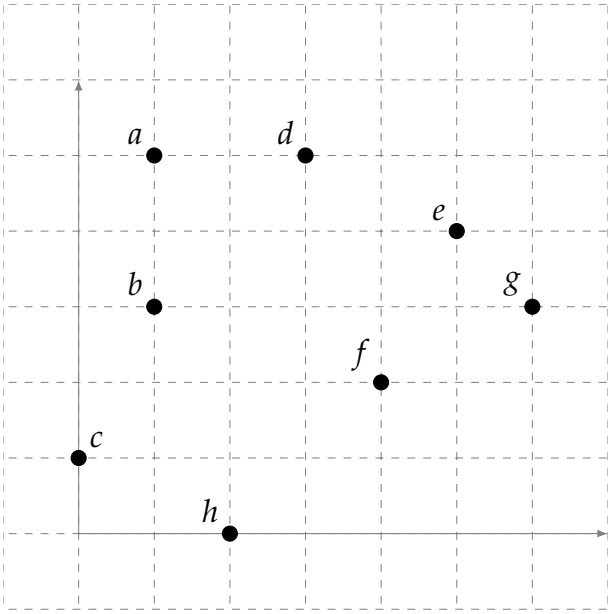
Definitie 5.41 Het HANDELSREIZIGERSPROBLEEM is het volgende: gegeven een complete[4] gewogen ongerichte graaf G met niet-negatieve gewichten, vind dan een ordening van de knopen zodanig dat elke knoop juist éénmaal wordt bezocht (behalve de start- en eindknoop die samenvallen) en zodanig dat de som van de gewichten van de gekozen bogen minimaal is. ∎

Het handelsreizigersprobleem is van een veel grotere moeilijkheidsgraad dan de andere problemen die we reeds hebben besproken in de grafentheorie. Het is een zogenaamd NP-COMPLEET probleem[5], wat hoogstwaar-

[4] Een graaf is compleet als er een boog is tussen elke twee (verschillende) knopen.

[5] Juister gezegd: het overeenkomstig *beslissingsprobleem* ("bestaat er een rondreis met kost hoogstens k?") is NP-compleet. Het algemene probleem is echter minstens zo moeilijk als het beslissingsprobleem.

schijnlijk betekent dat er *geen* polynomiaal algoritme bestaat dat alle gevallen correct kan oplossen. Men beperkt zich dan ook vaak tot snellere maar *benaderende* algoritmen, waarvan men hoopt (of kan bewijzen) dat de oplossing die ze produceren niet te ver afwijkt van de optimale oplossing.



Figuur 5.18: Acht steden in het vlak. Iedere stad is met iedere andere stad verbonden via een rechte lijn (lijnen niet getekend).

Voorbeeld 5.42 We beschouwen een graaf met 8 knopen die allen geheeltallige coördinaten hebben, zoals aangegeven in Figuur 5.18. De afstand tussen twee knopen (i.e. het gewicht van de overeenkomstige boog) wordt gegeven door de afstand in rechte lijn tussen de twee knopen. De adjacentiematrix (met gewichten) van de graaf is dus

$$A = \begin{pmatrix} 0 & 2 & \sqrt{17} & 2 & \sqrt{17} & \sqrt{18} & \sqrt{29} & \sqrt{26} \\ 2 & 0 & \sqrt{5} & \sqrt{8} & \sqrt{17} & \sqrt{10} & 5 & \sqrt{10} \\ \sqrt{17} & \sqrt{5} & 0 & 5 & \sqrt{34} & \sqrt{17} & \sqrt{40} & \sqrt{5} \\ 2 & \sqrt{8} & 5 & 0 & \sqrt{5} & \sqrt{10} & \sqrt{13} & \sqrt{26} \\ \sqrt{17} & \sqrt{17} & \sqrt{34} & \sqrt{5} & 0 & \sqrt{5} & \sqrt{2} & 5 \\ \sqrt{18} & \sqrt{10} & \sqrt{17} & \sqrt{10} & \sqrt{5} & 0 & \sqrt{5} & \sqrt{8} \\ \sqrt{29} & 5 & \sqrt{40} & \sqrt{13} & \sqrt{2} & \sqrt{5} & 0 & 5 \\ \sqrt{26} & \sqrt{10} & \sqrt{5} & \sqrt{26} & 5 & \sqrt{8} & 5 & 0 \end{pmatrix}$$

Aangezien de gewichten gebaseerd zijn op de Euclidische afstand zal het steeds korter (of juister gezegd: niet langer) zijn om rechtstreeks van een

knoop v naar w te gaan dan om een omweg te maken via een knoop u. Wiskundig uitgedrukt voldoet deze graaf aan de driehoeksongelijkheid. ∎

Definitie 5.43 Een graaf G voldoet aan de DRIEHOEKSONGELIJKHEID wanneer voor alle knopen u, v en w geldt dat

$$\text{gewicht}(v, w) \leq \text{gewicht}(v, u) + \text{gewicht}(u, w).$$ ∎

Een graaf voldoet m.a.w. aan de driehoeksongelijkheid wanneer rechtstreeks van v naar w gaan nooit langer is dan een omweg nemen via u. Wanneer de gewichten afgeleid zijn van de Euclidische afstand dan is steeds aan de driehoeksongelijkheid voldaan.

Wanneer de graaf aan de driehoeksongelijkheid voldoet dan geeft volgend *benaderingsalgoritme* een oplossing die hoogstens tweemaal zolang is als de optimale oplossing (zonder bewijs). Deze garantie is echter niet geldig wanneer de graaf niet aan de driehoeksongelijkheid voldoet.

1. Bereken een minimale kost opspannende boom T voor de graaf.
2. Kies willekeurig een wortel r van deze boom.
3. Geef de cykel terug die correspondeert met het in *preorde* doorlopen van deze boom.

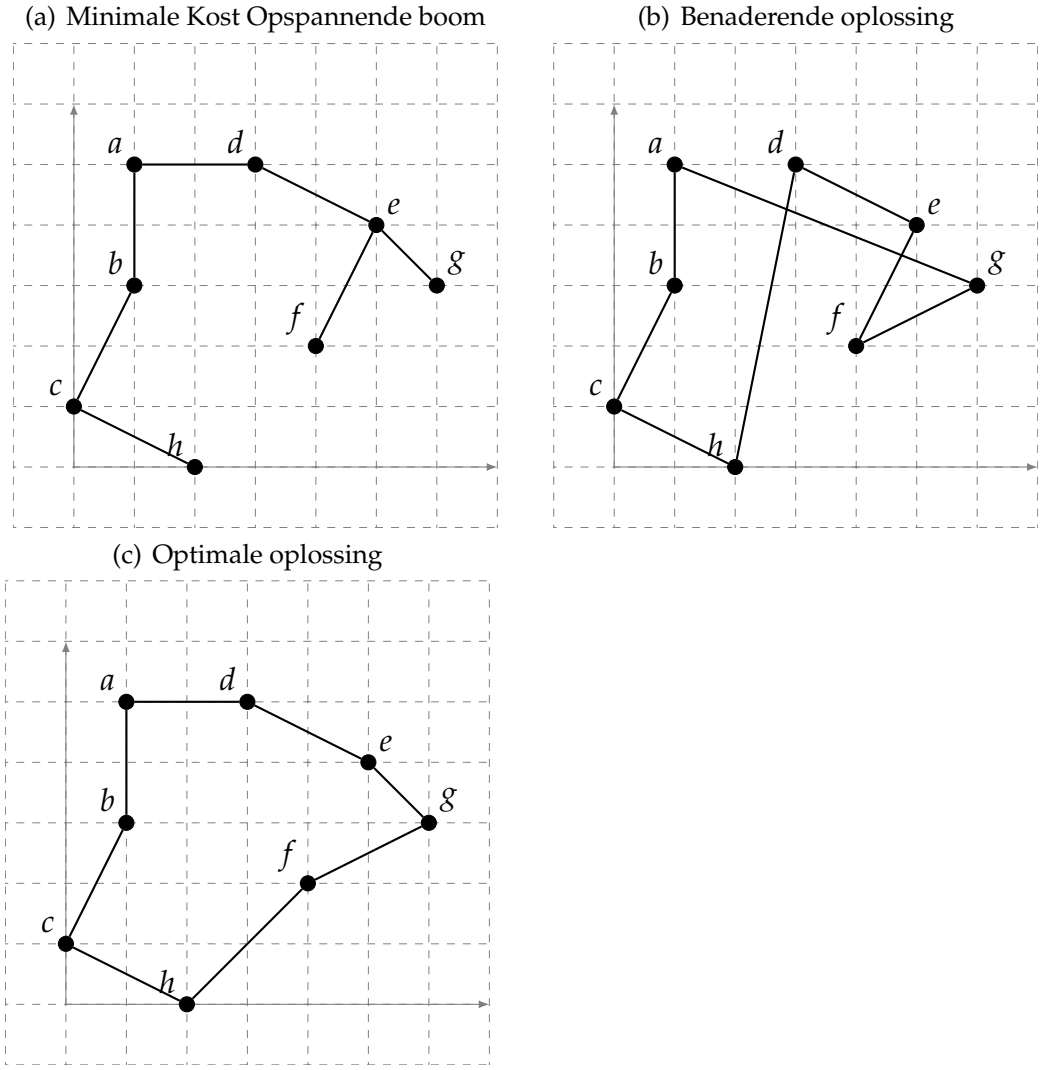
Voorbeeld 5.44 In Figuur 5.19(a) is een minimale kost opspannende boom gegeven. Deze boom werd bekomen m.b.v. Prims algoritme waarbij bij gelijke kost steeds de lexicografische kleinste boog werd gekozen. De startknoop was knoop a.

Als we dan als wortel van deze boom a nemen, en de knopen van de boom in preorde doorlopen (met de kinderen lexicografisch geordend), dan vinden we:

$$a, b, c, h, d, e, f, g.$$

Dit geeft dan aanleiding tot de rondreis in Figuur 5.19(b). De kost van deze rondreis is:

$$2 + \sqrt{5} + \sqrt{5} + \sqrt{26} + \sqrt{5} + \sqrt{5} + \sqrt{5} + \sqrt{29} \approx 23.66.$$



Figuur 5.19: Illustratie van het benaderende algoritme voor het handelsreizigersprobleem.

De optimale oplossing wordt gegeven in Figuur 5.19(c), en deze rondreis heeft de volgende kost:

$$2 + \sqrt{5} + \sqrt{2} + \sqrt{5} + \sqrt{8} + \sqrt{5} + \sqrt{5} + 2 \approx 17.19.$$

De benaderende rondreis is dus ongeveer 38% langer dan de optimale rondreis. ∎

5.6.1 Oefeningen

1. Beschouw opnieuw de acht steden in Figuur 5.18, maar veronderstel nu dat het gewicht van een boog gegeven wordt door de zogenaamde Manhattan-distance tussen de twee knopen, dus

$$d\big((x_1, y_1), (x_2, y_2)\big) = |x_2 - x_1| + |y_2 - y_1|$$

   a) Ga na dat de Manhattan-distance aan de driehoeksongelijkheid voldoet. **Hint**: voor de absolute waarde geldt dat

$$|x + y| \leq |x| + |y|.$$

   b) Pas het benaderende algoritme voor het oplossen van het handelsreizigersprobleem toe op dit probleem. Gebruik Kruskals algoritme om de minimale opspannende boom te construeren. Wanneer meerdere bogen kunnen gekozen worden, kies dan steeds de lexicografisch kleinste boog. Neem de knoop a als wortel van de opspannende boom.

HOOFDSTUK 6

Zoekalgoritmes

In hoofdstuk 5 zagen we reeds enkele algoritmes om een pad te zoeken doorheen een graaf. We bekijken nu een gelijkaardig probleem dat in principe ook kan gemodelleerd worden als een graafprobleem maar waarbij de graaf *veel te groot* is om effectief op te bouwen. We starten met de definitie van een ZOEKPROBLEEM en onderstrepen het belang van een efficiënte representatie. Vervolgens bekijken we twee algoritmische templates, nl. BOOMGEBASEERD en GRAAFGEBASEERD ZOEKEN. Die worden dan verder gespecialiseerd in een heel aantal verschillende algoritmes, waarbij A* ZOEKEN een zeer belangrijk algoritme is. We bestuderen eveneens het gebruik en ontwerp van HEURISTIEKEN in de context van het A* zoekalgoritme.

6.1 Inleiding

In dit hoofdstuk bespreken we hoe een agent een zoekprobleem kan oplossen. Voor een zoekprobleem nemen we aan dat de agent zich bevindt in een eenpersoons omgeving die compleet observeerbaar, deterministisch, statisch en discreet is. De agent bevindt zich steeds in een bepaalde beginpositie en de bedoeling is dat de agent acties onderneemt die hem in een toestand brengen waar één of andere voorwaarde voldaan is.

Het feit dat de omgeving aan alle bovenstaande voorwaarden voldoet betekent dat de agent *op voorhand* de acties kan berekenen die hem naar het doel zullen brengen zonder deze effectief uit te moeten voeren. Op het moment dat de agent dan effectief de acties gaat uitvoeren in de "echte" wereld kan

hij de waarnemingen negeren: hij weet immers toch al wat er gaat komen.

Definitie 6.1 Een ZOEKPROBLEEM bestaat uit de volgende elementen:

- Een TOESTANDSRUIMTE S die alle mogelijke toestanden bevat.
- Een verzameling van mogelijke acties A.
- Een TRANSITIEMODEL dat zegt wat het effect is van het uitvoeren van een actie op een bepaalde toestand:

  $$T\colon (S, A) \to S\colon (s, a) \mapsto s'.$$

  Wanneer s' bereikt wordt door het uitvoeren van een actie a op een toestand s dan wordt s' een *opvolger* van s genoemd[1].

  Het uitvoeren van een actie op een bepaalde toestand heeft meestal een bepaalde KOST:

  $$c\colon (S, A, S) \to \mathbb{R}\colon (s, a, s') \mapsto c(s, a, s').$$

  De kost kan dus afhangen van zowel s, de gekozen actie a, als van de opvolger s'. In deterministische omgevingen ligt de opvolger s' vast wanneer men s en a weet, maar deze definitie kan ook gebruikt worden in een stochastische omgeving waar s' onzeker is.
- Een initiële toestand $s_0 \in S$, dit is de toestand van waaruit het zoeken zal vertrekken.
- een DOELTEST. Dit is een functie die voor elke toestand s aangeeft of het doel bereikt is of niet. Een toestand waarvoor de doeltest voldaan is noemen we een DOELTOESTAND. ∎

Opmerking 6.2 De toestandsruimte S kan samen met de transitie- en kostfunctie gebruikt worden om een TOESTANDSRUIMTEGRAAF (Eng. *state space graph*) G op te bouwen. In deze graaf stellen de knopen de toestanden voor en twee knopen zijn verbonden door een (gerichte) boog wanneer de ene knoop de opvolger is van de andere door het uitvoeren van een bepaalde actie. De kost (of het gewicht) van een boog is dan uiteraard de kost van de bijhorende actie.

[1] Hier zie je dat we te maken hebben met een deterministische omgeving: wanneer s en a gekend zijn is er juist één opvolger s'.

In een toestandsruimtegraaf komt elke toestand juist één keer voor.

De toestandsruimtegraaf is een abstract concept: het aantal knopen (en bogen) van deze graaf is vaak veel te groot om deze volledig bij te houden in het (hoofd)geheugen van een computer. ∎

Voorbeeld 6.3 (De 8-puzzel) De 8-puzzel is een typisch zoekprobleem. Hierbij is een vierkant rooster gegeven van 9 vakjes: 8 vakjes zijn genummerd van 1 t.e.m. 8 en één vakje is leeg. Men kan een genummerd vakje verschuiven naar het lege vakje, waardoor de originele plaats van het genummerd vakje leeg wordt. De bedoeling is om een vooraf bepaalde "mooie" configuratie te bereiken. In Figuur 6.1 ziet men twee voorbeelden van 8-puzzels.

We beschrijven de verschillende onderdelen van het zoekprobleem.

- De verzameling S bestaat uit alle mogelijke configuraties van de puzzel.
- Er zijn vier acties, nl. `Boven`, `Onder`, `Links` en `Rechts`. Deze worden beschreven in termen van het lege vakje.
- Het transitiemodel is een eenvoudige vertaling van wat er gebeurt in de fysieke puzzel. Wanneer men bv. de actie `Rechts` uitvoert op de linkerpuzzel in Figuur 6.1 dan bekomt men een nieuwe puzzel waarbij het lege vakje en vakje 6 omgewisseld zijn. De kost van elke actie is één.
- De initiële toestand is een willekeurige configuratie van de puzzel[2].
- De doeltest bestaat uit verifiëren of de vooraf vastgelegde configuratie werd bereikt.

Het aantal toestanden voor de 8-puzzel is $9! = 362880$ wat voor een moderne computer nog beheersbaar is. Men kan de puzzel echter ook uitbreiden naar grotere borden, bv. 4 op 4 (de 15-puzzel) met $16! \approx 2.1 \times 10^{13}$ of 5 op 5 (de 24-puzzel) met $25! \approx 1.6 \times 10^{25}$ toestanden. ∎

[2]Echter, slechts de helft van de puzzels kan omgezet worden in een gegeven doelconfiguratie.

7	2	4
5		6
8	3	1

	1	2
3	4	5
6	7	8

Figuur 6.1: Een voorbeeld van een 8-puzzel. Links ziet men een typische beginconfiguratie (initiële toestand) en rechts ziet men de doeltoestand.

Voorbeeld 6.4 (Het 8-koninginnenprobleem) Het doel van het 8-koninginnenprobleem is om 8 koninginnen op een standaard 8 op 8 schaakbord te plaatsen zodanig dat geen enkel paar koninginnen elkaar aanvalt volgens de standaard schaakregels waarin een koningin horizontaal, verticaal en diagonaal kan bewegen. In Figuur 6.2 ziet men een oplossing van het 8-koninginnenprobleem.

We bekijken nu een eerste manier om dit probleem voor te stellen als een zoekprobleem, waarbij we met een leeg bord beginnen en telkens een koningin toevoegen aan het bord.

- De toestandsruimte S bevat alle mogelijke configuraties waarbij er tussen de 0 en 8 koninginnen op het bord staan.
- Wanneer er minder dan 8 koninginnen op het bord staan dan kunnen we op een willekeurig leeg vakje een koningin toevoegen; dit zijn de acties. Het transitiemodel wordt gegeven door de fysica van het model en de kost van de acties is voor dit probleem irrelevant en kan dus gelijk aan nul worden genomen.
- De initiële toestand is het lege bord.
- De doeltest bestaat uit nagaan of er inderdaad 8 koninginnen op het bord staan én uit controleren dat er geen paar koninginnen is dat elkaar aanvalt. In dit geval is de doeltest effectief een functie en niet zomaar een eenvoudige opsomming van de doeltoestanden.

Met deze representatie is het aantal toestanden in de toestandsruimte

$$\binom{64}{0} + \binom{64}{1} + \cdots + \binom{64}{8} \approx 5.13 \times 10^9,$$

wat met 8 bytes per toestand reeds 40 GB aan hoofdgeheugen vraagt! ∎

Voorbeeld 6.5 (Het 8-koninginnenprobleem (bis)) Een efficiëntere representatie bestaat erin om toestanden waarin twee koninginnen elkaar aanvallen te verbieden. In de vorige representatie worden eerst alle acht de koninginnen geplaatst en pas dan wordt gecontroleerd of deze configuratie voldoet aan de doeltest. Om het aantal toestanden verder te beperken gaan we koninginnen kolom per kolom toevoegen, startend vanaf links. We bekomen de volgende formulering als een zoekprobleem.

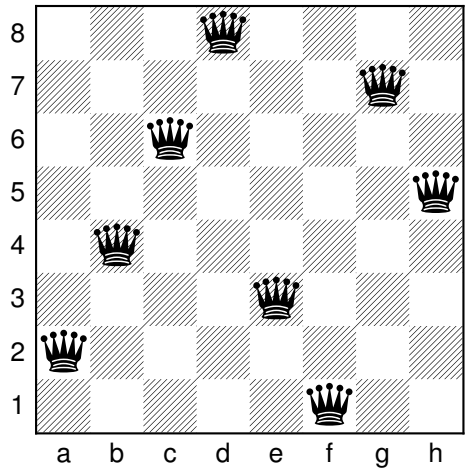
- De toestandsruimte bevat toestanden waarbij de eerste n (met $0 \leq n \leq 8$) kolommen van het bord juist één koningin bevatten. Bovendien is er geen paar koninginnen dat elkaar aanvalt.
- De mogelijke acties op een gegeven bord bestaan erin om een koningin toe te voegen in de eerste vrije kolom op een vakje dat niet wordt aangevallen door één van de reeds geplaatste koninginnen. Het transitiemodel is zoals men verwacht en de kost van elke actie is nog steeds gelijk aan nul.
- De initiële toestand is het lege bord.
- De doeltest is zoals voorheen.

In dit geval bevat de toestandsruimte 2057 toestanden en wordt het triviaal om een oplossing te vinden.

Men ziet dus dat *de formulering van de toestanden en acties een grote invloed kan hebben op de mogelijkheid om al dan niet een oplossing te vinden.* ∎

Voorbeeld 6.6 (Route- en rondreisproblemen) Veel "real-life" problemen hebben te maken met het vinden van een route tussen twee locaties. Veronderstel dat er n steden zijn: in het *routeprobleem* wil men een weg vinden tussen twee van deze steden.

- Er zijn n toestanden, de toestand i duidt aan dat "we in stad i zijn".
- Vanuit een bepaalde stad kan men acties ondernemen om een andere stad te bereiken, bv. de `RijdNaar` actie.



Figuur 6.2: Eén van de 92 oplossingen van het 8-koninginnenprobleem.

- Wanneer men in toestand i de `RijdNaar`(j) actie onderneemt, dan is de volgende toestand die waarin men zich in locatie j bevindt. Dit is het transitiemodel. De kost van de actie kan bv. de afstand in km zijn tussen i en j, of de gemiddelde reistijd of het gemiddeld benzineverbruik, ...
- De initiële toestand is een willekeurige stad.
- De doeltest bestaat uit verifiëren dat men in de gewenste stad is aangekomen.

Voor het routeprobleem is het aantal toestanden dus gelijk aan het aantal steden.

Bij een *rondreisprobleem*, waarbij men elke stad minstens éénmaal moet bezoeken en terug moet keren naar de eerste stad, zijn de toestanden fundamenteel anders. Het is immers niet langer voldoende om te weten in welke locatie men *nu* is, men moet ook onthouden *waar men reeds geweest is*.

- Een toestand bestaat uit de huidige stad, en de verzameling van de reeds bezochte steden. Het aantal toestanden is dus $n \times 2^n$. Nu is het aantal toestanden exponentieel in het aantal steden.
- De acties zijn dezelfde als voorheen.

- In het transitiemodel moet men niet enkel de huidige stad aanpassen maar ook de verzameling van reeds bezochte steden uitbreiden met de stad waar men net vandaan komt. De kost van de acties is zoals voorheen.
- Initieel zijn we in een willekeurige stad en hebben we geen enkele andere stad bezocht.
- De doeltest bestaat uit verifiëren dat we opnieuw in de startstad zijn en dat alle steden minstens éénmaal bezocht zijn.

Bij het *handelsreizigersprobleem* moet elke tussenliggende stad juist éénmaal worden bezocht. ∎

We hebben nu enkele voorbeelden gezien van de definitie van een zoekprobleem. We bekijken nu wat het betekent om een zoekprobleem op te lossen.

Definitie 6.7 Een OPLOSSING VAN ZOEKPROBLEEM bestaat uit een sequentie van acties zodanig dat startend vanuit de initiële toestand een doeltoestand wordt bereikt.

De KOST van een oplossing is de som van de kosten van de individuele acties.

Een OPTIMALE OPLOSSING is een oplossing waarvoor de kost minimaal is onder alle mogelijke oplossingen. ∎

Opmerking 6.8 Het oplossen van een $(n^2 - 1)$-puzzel en het handelsreizigersprobleem zijn NP-complete problemen die kunnen geformuleerd worden als zoekproblemen. We kunnen m.a.w. *niet* verwachten dat de oplossingsmethodes binnen de artificiële intelligentie een algoritme zullen opleveren dat alle instanties efficiënt kan oplossen. ∎

Voorbeeld 6.9 Voor de 8-puzzel in Figuur 6.1 bestaat een optimale uit 26 acties. Een mogelijke oplossing, met `L` als afkorting voor actie `Links` en analoog voor de andere acties, is de volgende:

`L, B, R, O, R, O, L, L, B, R, R, O, L, L, B, R, R, B, L, L, O, R, R, B, L, L` ∎

6.2 Algemene Zoekalgoritmen

6.2.1 Boomgebaseerd Zoeken

Het algemene algoritme gekend als BOOMGEBASEERD ZOEKEN houdt een lijst bij van mogelijke partiële oplossingen (plannen) die nog verder uitgewerkt (geëxpandeerd) moeten worden. Deze lijst wordt de OPEN LIJST genoemd. Bij de start van de uitvoering bestaat deze open lijst enkel uit het plan corresponderend met de initiële toestand van het zoekprobleem. Bij elke iteratie van het algoritme wordt een plan gekozen uit deze lijst (volgens één of andere strategie). Wanneer de (eind)toestand van het gekozen plan voldoet aan de doeltest dan stopt het algoritme. Wanneer dit niet het geval is dan worden de plannen voor alle opvolgers van de (eind)toestand van het gekozen plan toegevoegd aan de open lijst (waardoor deze ook beschikbaar worden voor expansie). Wanneer de open lijst op een bepaald moment leeg is dan geeft het algoritme aan dat er geen oplossing werd gevonden.

Conceptueel bouwen we dus een ZOEKBOOM op. Elke top van deze zoekboom stelt een sequentie van acties voor. Het is uiteraard de bedoeling om de zoekproblemen op te lossen aan de hand van een zo klein mogelijke zoekboom. In een zoekboom stelt elke top een plan voor dat o.a. de huidige toestand bijhoudt. Dezelfde toestand kan (en zal) in het algemeen *meerdere malen* voorkomen in een zoekboom.

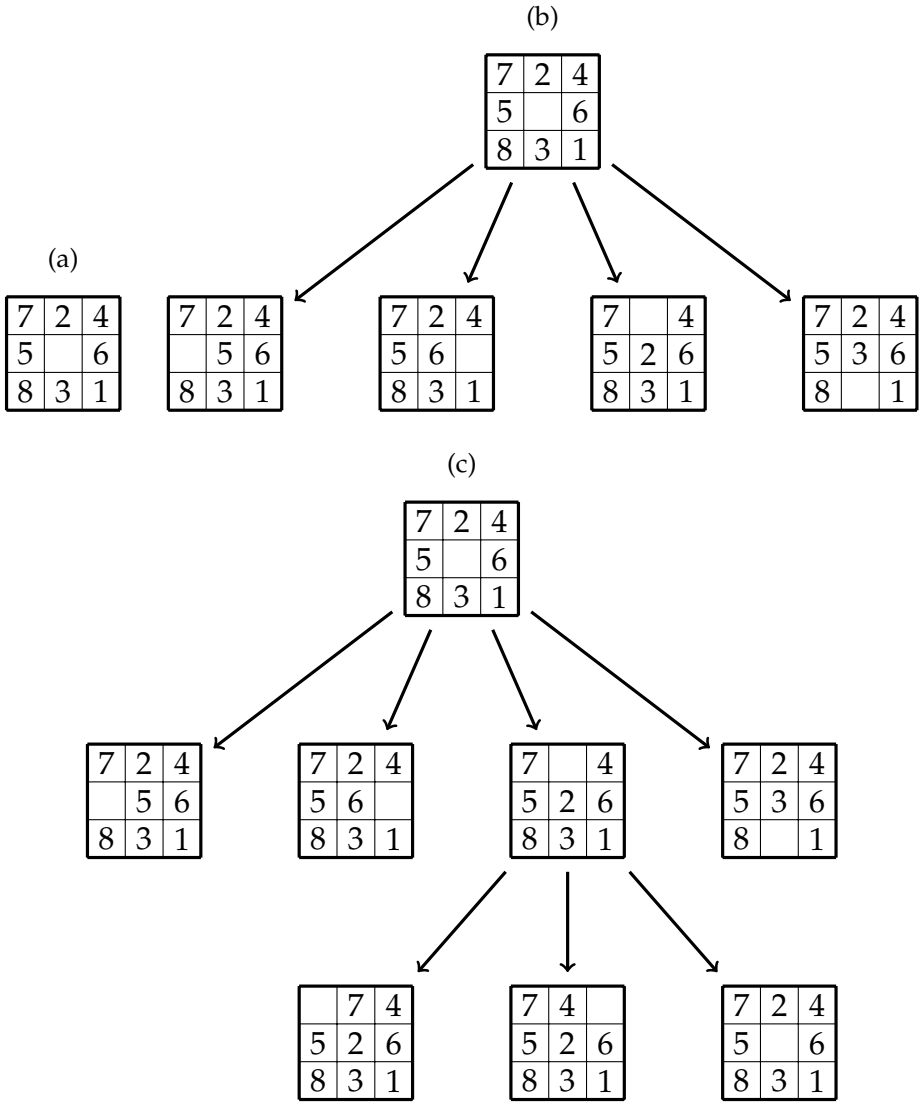
Voorbeeld 6.10 In Figuur 6.3 ziet men de start van een zoekboom voor de 8-puzzel. Hier ziet men reeds dat dezelfde toestand meerdere malen kan voorkomen op de open lijst. ∎

Implementatie van een plan

Conceptueel stelt elke top van de zoekboom een sequentie van acties voor om een bepaalde toestand te bereiken startend vanaf de initiële toestand van het zoekprobleem.

Het is echter *niet nodig* om in elke top het volledige pad op te slaan. Zolang we weten wat het vorige plan is kunnen we, in combinatie met de laatst gekozen actie, het volledige plan opstellen.

We gebruiken een klasse `Plan` om een plan voor te stellen. Zo'n `Plan` bestaat uit vier velden:



Figuur 6.3: Deel van de zoekboom opgebouwd door boomgebaseerd zoeken. We kiezen "willekeurig" welk plan op de open lijst (i.e. welk blad) wordt geëxpandeerd. Initieel (zie (a)) bevat de open lijst slechts één plan. Dit plan wordt verwijderd van de open lijst en wordt geëxpandeerd. De vier opvolgers van de initiële toestand worden toegevoegd aan de open lijst (zie (b)). Veronderstel nu dat (om één of andere reden) de toestand met de blanco bovenaan wordt gekozen voor expansie. De drie opvolgers worden toegevoegd aan de open lijst die nu 6 elementen bevat (zie (c)). Merk op dat één van de gegenereerde opvolgers opnieuw de initiële toestand is. Bij boomgebaseerd zoeken wordt hier geen rekening mee gehouden en wordt deze toestand opnieuw toegevoegd aan de open lijst.

Algoritme 6.1 Boomgebaseerd zoeken.

Invoer Een zoekprobleem P.

Uitvoer Een sequentie van acties die een oplossing is van het zoekprobleem of error wanneer er geen oplossing werd gevonden.

```
1:  function TREESEARCH(P)
2:      f ← nieuwe lege lijst                          ▷ De open lijst
3:      f.ADD(nieuw plan gebaseerd op initiële toestand P)
4:      while f ≠ ∅ do
5:          c ← f.CHOOSEANDREMOVEPLAN                  ▷ Kies het volgende plan
6:          if P.GOALTEST(c.GETSTATE) = true then
7:              return GETACTIONSEQ(c)
8:          else
9:              for (s, a) ∈ c.GETSTATE.GETSUCCESSORS do
10:                 f.ADD(nieuw plan gebaseerd op (s, a) en c)
11:             end for
12:         end if
13:     end while
14:     return error: geen oplossing gevonden          ▷ Open lijst is leeg.
15: end function
```

- De huidige toestand.
- De laatst gekozen actie a; deze is enkel leeg voor het plan geassocieerd met de initiële toestand.
- De *voorganger* of *ouder* van dit plan. Een referentie naar het plan waarvan dit plan is afgeleid door het toepassen van de huidige actie a.
- De totale kost van dit plan. Traditioneel wordt deze kost met g genoteerd. Strikt genomen kunnen we deze kost ook berekenen door het volgen van de voorganger-referenties. Deze berekening heeft echter een uitvoeringstijd die lineair is in het aantal acties van het plan.

6.2.2 Criteria voor Zoekalgoritmen

Zoekalgoritmen kunnen op verschillende manieren worden geëvalueerd. De volgende vier criteria worden vaak gebruikt.

1. Een zoekalgoritme is COMPLEET wanneer het algoritme, voor elk zoekprobleem met een oplossing, effectief een oplossing vindt.

2. Een zoekalgoritme is OPTIMAAL wanneer het niet enkel *een* oplossing vindt maar steeds een optimale oplossing teruggeeft voor elk zoekprobleem met een oplossing.
3. De TIJDSCOMPLEXITEIT van een zoekalgoritme bepaalt de uitvoeringstijd van het algoritme. We nemen aan dat de uitvoeringstijd evenredig is met het aantal gegenereerde toppen.
4. De RUIMTECOMPLEXITEIT van een zoekalgoritme bepaalt de hoeveelheid geheugen die het algoritme nodig heeft tijdens de uitvoering. Dit wordt meestal uitgedrukt als het maximaal aantal toestanden dat gelijktijdig moet worden bijgehouden.

De volgende maten worden vaak gebruikt om de tijds- en ruimtecomplexiteit van zoekalgoritmen uit te drukken. Vooreerst is er de VERTAKKINGSFACTOR b. Deze geeft het maximaal aantal opvolgers van een top in de zoekboom. De diepte van de meest ondiepe top waarvan de toestand een doeltoestand is (kortweg een *doeltop* genoemd) wordt genoteerd met d. Met m, tenslotte, duidt men de maximale lengte (gemeten als het aantal genomen acties) van een pad in de toestandsruimte aan.

In Figuur 6.4 ziet men een illustratie van deze concepten.

Eigenschap 6.11 Het aantal toppen in een zoekboom met vertakkingsfactor b en maximale diepte m wordt gegeven door

$$\frac{b^{m+1}-1}{b-1} = \mathcal{O}(b^m). \tag{6.1}$$

∎

Bewijs We nemen aan dat elke top exact b opvolgers heeft.

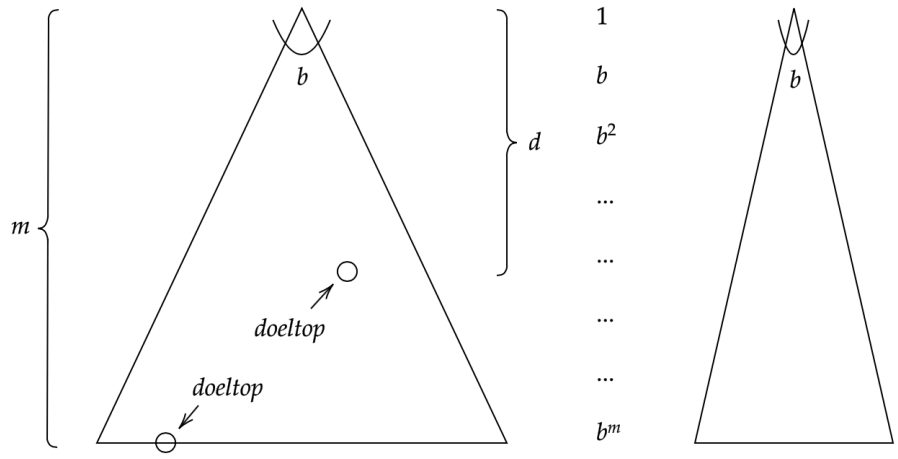
Het aantal toppen op diepte nul is 1, het aantal toppen op diepte 1 is b, het aantal toppen op diepte 2 is b^2, enzovoort. Het totaal aantal toppen is dus

$$1 + b + b^2 + \cdots + b^m.$$

Een gesloten formule voor deze som wordt inderdaad gegeven door (6.1).♢

Opmerking 6.12 De laag met diepte m in een zoekboom met vertakkingsfactor b bevat b^m toppen. Dit is méér dan alle voorgaande lagen samen; deze bevatten in totaal slechts

$$1 + b + b^2 + \cdots + b^{m-1} = \frac{b^m - 1}{b-1} \approx b^{m-1}$$



Figuur 6.4: Illustratie van de grootheden b, d en m voor een zoekboom. Links een voorbeeld van een zoekboom met een "grote" vertakkingsfactor, rechts een zoekboom met een "kleine" vertakkingsfactor. *(Elektronische versie van deze figuur aangeleverd door oud-student Cedric De Vylder. Waarvoor dank.)*

toppen. ∎

6.2.3 Graafgebaseerd Zoeken

Het grootste probleem van boomgebaseerd zoeken is dat dit algoritme *niet onthoudt waar het reeds geweest is*. Dit zorgt ervoor dat we in sommige gevallen (bv. diepte eerst zoeken, cfr. infra) te maken krijgen met *oneindige lussen*, en dat we in veel andere gevallen een grote hoeveelheid werk herhaaldelijk uitvoeren.

Voorbeeld 6.13 (Herhaalde toestanden in een grid) Een vaak voorkomende situatie is die van een agent die leeft in een grid met vier acties: `Boven`, `Onder`, `Links` en `Rechts`. De vertakkingsfactor b is in dit geval gelijk aan 4. Vergelijken we nu eventjes het aantal toestanden op afstand d van een willekeurig toestand met het aantal toppen van de zoekboom op diepte d bij boomgebaseerd zoeken. Figuur 6.5 toont een grid. We kunnen nu gemakkelijk vergelijken:

			3			
		3	2	3		
	3	2	1	2	3	
3	2	1	0	1	2	3
	3	2	1	2	3	
		3	2	3		
			3			

Figuur 6.5: Toestanden in een grid. Centraal staat de starttoestand. Voor toestanden die op afstand drie of minder liggen staat de afstand tot de initiële toestand aangeduid in de desbetreffende cel.

d	aantal toestanden op afstand d	aantal toppen in zoekboom op diepte d
0	1	1
1	4	4
2	8	16
3	12	64
⋮	⋮	⋮
d	$4d$	4^d

Uit deze tabel zien we dat het aantal verschillende toestanden *lineair* toeneemt met de diepte, terwijl het aantal toppen in de zoekboom *exponentieel* toeneemt met de diepte. ∎

De oplossing voor het probleem van de herhaalde toestanden bestaat erin om eenvoudigweg te onthouden welke toestanden reeds geëxpandeerd zijn in, wat men noemt, een GESLOTEN LIJST[3]. Merk op dat de gesloten lijst *toestanden* bevat terwijl de open lijst *plannen* bevat.

Bij GRAAFGEBASEERD ZOEKEN wordt elke toestand hoogstens éénmaal geëxpandeerd. Wanneer een plan van de open lijst wordt gehaald dat een toestand bevat die reeds geëxpandeerd is, dan wordt deze niet opnieuw geëxpandeerd. De pseudocode voor graafgebaseerd zoeken vind je in Algoritme 6.2.

[3]De gesloten lijst wordt best geïmplementeerd als een verzameling.

Algoritme 6.2 Graafgebaseerd zoeken.

Invoer Een zoekprobleem P.
Uitvoer Een sequentie van acties of error wanneer er geen oplossing werd gevonden.

```
1:  function GRAPHSEARCH(P)
2:      f ← nieuwe lege lijst                          ▷ De open lijst
3:      closed ← ∅                  ▷ Verzameling geëxpandeerde toestanden
4:      f.ADD(nieuw plan gebaseerd op initiële toestand P)
5:      while f ≠ ∅ do
6:          c ← f.CHOOSEANDREMOVEPLAN                  ▷ Kies het volgende plan
7:          if P.GOALTEST(c.GETSTATE) = true then
8:              return GETACTIONSEQ(c)
9:          else
10:             if c.GETSTATE ∉ closed then
11:                 closed ← closed ∪ c.GETSTATE
12:                 for (s, a) ∈ c.GETSTATE.GETSUCCESSORS do
13:                     f.ADD(nieuw plan gebaseerd op (s, a) en c)
14:                 end for
15:             end if
16:         end if
17:     end while
18:     return error: geen oplossing gevonden
19: end function
```

6.3 Blinde Zoekmethoden

Blinde zoekmethoden kunnen enkel gebruikmaken van de informatie die verschaft wordt door de definitie van het zoekprobleem. Ze beschikken niet over extra informatie die hen kan helpen bij het zoekproces. We bespreken nu vier blinde zoekmethoden en hun eigenschappen.

6.3.1 Breedte Eerst Zoeken

Bij breedte eerst zoeken wordt voor de open lijst een *wachtrij* gebruikt. Dit is een FIFO datastructuur. Bij breedte eerst wordt de zoekboom systematisch laag per laag opgebouwd. In Figuur 6.6 ziet men een illustratie van het breedte eerst algoritme op een binaire boom.

Omdat breedte eerst systematisch de lagen in de zoekboom onderzoekt zal het algoritme steeds een oplossing vinden voor elk zoekprobleem dat effectief een oplossing heeft. Dit betekent dat breedte eerst een compleet zoe-

kalgoritme is. Het algoritme vindt steeds de meest ondiepe doeltop, i.e. het retourneert een oplossing met een minimaal *aantal* acties. Wanneer acties een verschillende kost hebben is dit niet noodzakelijk een oplossing met de kleinste kost. Het kan immers beter zijn om meerdere goedkope acties te doen i.p.v. een kleiner aantal duurdere acties. Breedte eerst is m.a.w. niet optimaal. Breedte eerst is wel optimaal in het bijzonder geval dat alle acties dezelfde kost hebben.

We onderzoeken nu de tijdscomplexiteit van breedte eerst zoeken. Veronderstel dat het zoekprobleem een oplossing heeft en dat de meest ondiepe doelknoop diepte d heeft. Als deze doeltop de "meest rechtse" top is dan worden alle toppen op de dieptes 0 t.e.m. d geëxpandeerd. Het aantal *gegenereerde* toppen is m.a.w. (op een term b na) gelijk aan

$$1 + b + b^2 + \cdots + b^d + b^{d+1} = \mathcal{O}(b^{d+1}).$$

De tijdscomplexiteit van breedte eerst is m.a.w. exponentieel in de diepte van de meest ondiepe doeltop.

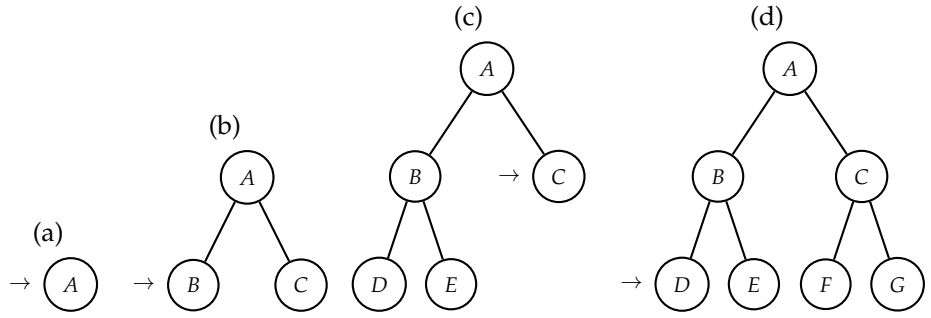
Het maximaal aantal toppen dat moet worden bijgehouden in de open lijst wordt bereikt wanneer men de doeltop expandeert. Op dit moment wordt zo goed als de volledige laag op diepte $d + 1$ bijgehouden in de open lijst. Deze laag bevat b^{d+1} toppen[4]. De ruimtecomplexiteit is m.a.w. eveneens $\mathcal{O}(b^{d+1})$.

Bij graafgebaseerd breedte eerst zoeken kan men veel tijd winnen (t.o.v. boomgebaseerd zoeken) wanneer veel toestanden meerdere malen voorkomen in de zoekboom. Het extra geheugen dat men moet spenderen aan het bijhouden van de gesloten lijst weegt niet op tegen de tijdswinst die men kan maken. Om deze reden wordt breedte eerst zoeken meestal uitgevoerd in zijn graafgebaseerde versie.

6.3.2 Diepte Eerst Zoeken

Diepte eerst zoeken is in zekere zin duaal aan breedte eerst zoeken: hier gebruikt men een LIFO structuur voor het bijhouden van de open lijst. Deze stapel zorgt ervoor dat men zo snel mogelijk zo diep mogelijk in de boom afdaalt. In Figuur 6.7 ziet men een illustratie van diepte eerst zoeken op een ternaire boom.

[4]De verwijzingen naar de voorgangers zorgen ervoor dat in principe de volledige zoekboom in het geheugen aanwezig blijft. Zie echter opmerking 6.12.



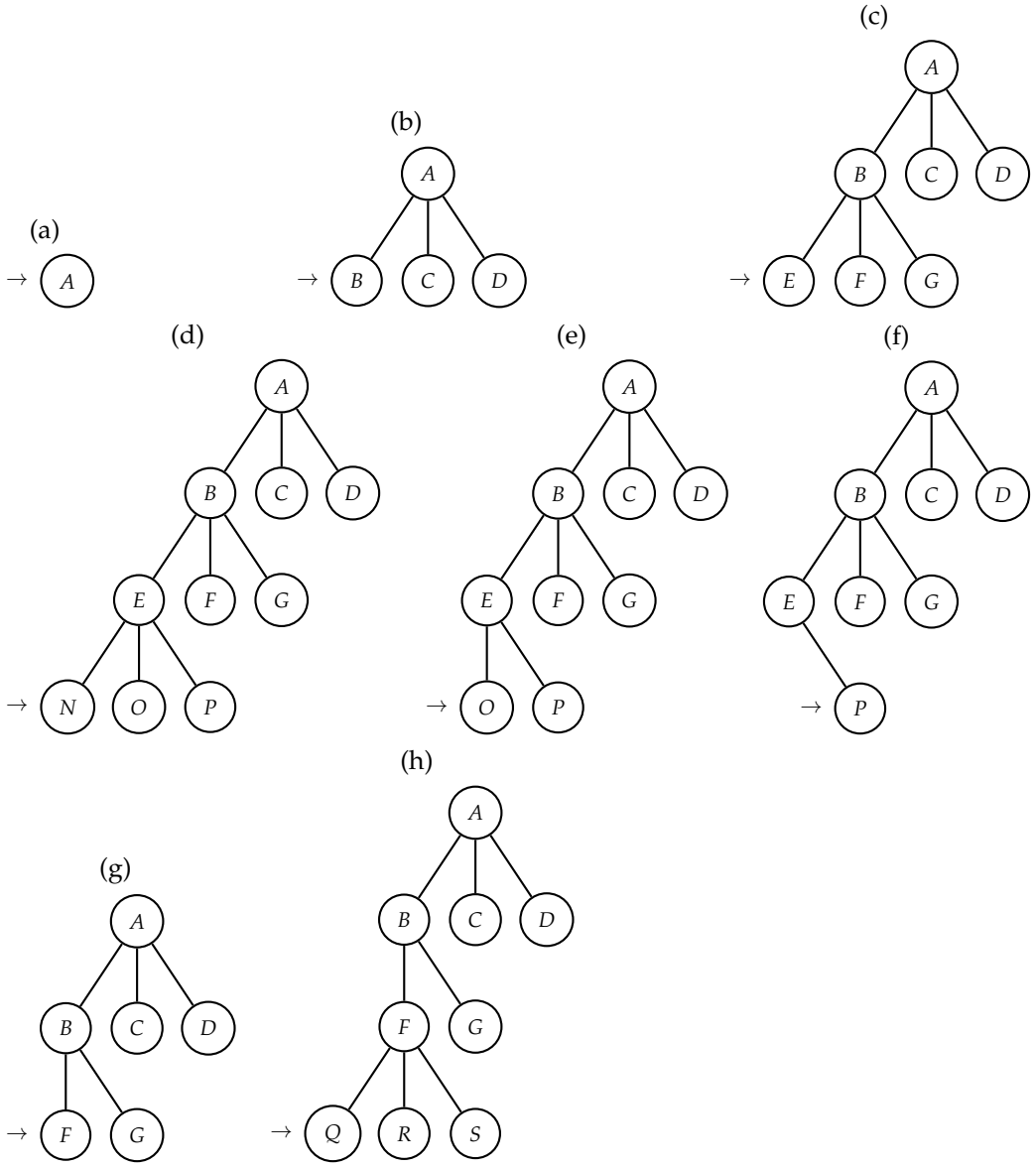
Figuur 6.6: Illustratie van breedte eerst zoeken op een binaire boom. De knoop die zal geëxpandeerd worden is steeds aangeduid met een pijltje. De doelknoop is D. Net vóór het algoritme eindigt bestaat de open lijst uit de volgende plannen: $\{D \to B \to A, E \to B \to A, F \to C \to A, G \to C \to A\}$ (in die volgorde).

Diepte eerst zoeken genereert steeds een linkerdeel van de boom. Wanneer m eindig is en de enige doeltop helemaal rechts onderaan in de boom zit dan worden alle toppen van de boom gegenereerd. In het slechtste geval is de tijdscomplexiteit m.a.w. $\mathcal{O}(b^m)$. Dit is dezelfde exponentiële (en dus slechte) tijdscomplexiteit als bij breedte eerst.

Diepte eerst kan, zelfs wanneer er een oplossing is, in bepaalde gevallen toch in een oneindige lus geraken. Diepte eerst is m.a.w. niet compleet en bijgevolg ook niet optimaal. Zelfs wanneer diepte eerst een oplossing vindt is deze niet gegarandeerd een optimale oplossing. De oplossing ontdekt door diepte eerst is immers steeds de "meest linkse" doeltop.

Waar diepte eerst wel goed op scoort is op het vlak van benodigd geheugen. Wanneer een bepaalde top wordt geëxpandeerd dan behoren enkel de broers van zijn voorouders tot de open lijst. Aangezien er maximaal m niveaus zijn en er hoogstens b broers zijn is de ruimtecomplexiteit van diepte eerst van de orde $\mathcal{O}(b \cdot m)$. Dit is een *lineaire* functie van b. Dit is dus een heel stuk beter dan de exponentiële ruimtecomplexiteit in het geval van breedte eerst.

Normaalgesproken is het niet zeer zinvol om diepte eerst uit te voeren in de graafgebaseerde vorm. Men verliest immers de goede eigenschappen betreffende de ruimtecomplexiteit. Het kan wel zinvol zijn om diepte eerst aan te passen zodanig dat er geen actiesequenties worden geprobeerd die terugkeren naar een toestand die reeds op het huidige pad ligt. Op die ma-



Figuur 6.7: Illustratie van diepte eerst zoeken op een ternaire boom. De toppen op diepte drie (i.e. *N*, *O* en *P*) hebben geen opvolgers. De doeltoestand is *Q*. De top die als volgende gekozen wordt voor expansie wordt aangeduid met een pijltje. De enige blaadjes van de boom (elementen van de open lijst) zijn de top gekozen voor expansie, zijn broers en voorouders en hun broers. Dit aantal is *lineair* in b en m.

nier kan eenvoudig vermeden worden dat diepte eerst terechtkomt in een oneindige lus. Men kan er echter niet mee vermijden dat diepte eerst oneindig lange actiesequenties probeert in toestandsruimten met een oneindig aantal toestanden.

6.3.3 Iteratief Verdiepen

Iteratief verdiepen is geen directe toepassing van het boomgebaseerd zoekalgoritme dat gegeven werd in Algoritme 6.1. Het is in essentie een lus rond diepte eerst zoeken waarbij het zoekproces wordt afgebroken wanneer een bepaalde diepte wordt bereikt; dit zoekproces wordt DIEPTE-GELIMITEERD ZOEKEN genoemd. Algoritme 6.3 geeft een implementatie voor diepte-gelimiteerd zoeken. Het algoritme wordt recursief geïmplementeerd en bij elke recursieve oproep wordt de maximale toegelaten diepte met één verminderd. Wanneer de toegelaten diepte de waarde nul bereikt dan wordt het meegegeven plan niet verder geëxpandeerd (en gebeuren er dus ook geen recursieve oproepen meer).

Het algoritme heeft een bijzondere returnwaarde nl. “hit boundary” om aan te geven dat er geen oplossing werd gevonden binnen de opgegeven dieptelimiet maar dat tijdens het zoekproces de dieptelimiet minstens éénmaal werd bereikt. Deze returnwaarde geeft m.a.w. aan dat een oplossing *eventueel* kan gevonden worden wanneer de maximale toegelaten diepte wordt verhoogd.

Bij elke iteratie van ITERATIEF VERDIEPEN, zie Algoritme 6.4, wordt de maximaal toegelaten diepte met één verhoogd. Het algoritme stopt de eerste maal dat een oplossing wordt gevonden of wanneer het duidelijk is dat er geen oplossing is. Op die manier vermijden we het probleem van diepte eerst dat we terechtkomen in een oneindige lus. We vinden op deze manier immers steeds de meest ondiepe doeltop. Tegelijkertijd behouden we de goede eigenschappen m.b.t. de de ruimtecomplexiteit van diepte eerst. Figuur 6.8 illustreert iteratief verdiepen op een binaire boom.

Op het eerste zicht zou men denken dat dit proces een gigantische hoeveelheid werk te veel doet. De eerste lagen van de zoekboom worden immers meerdere malen opgebouwd. De eerste lagen van de zoekboom bevatten echter relatief weinig toppen tegenover de diepere lagen zodat de hoeveelheid werk die “te veel” wordt verricht relatief beperkt blijft. Een berekening maakt dit duidelijk. Veronderstel dat de meest ondiepe oplossing zich be-

Algoritme 6.3 Diepte-gelimiteerd zoeken

Invoer Een zoekprobleem P, een maximale diepte d.

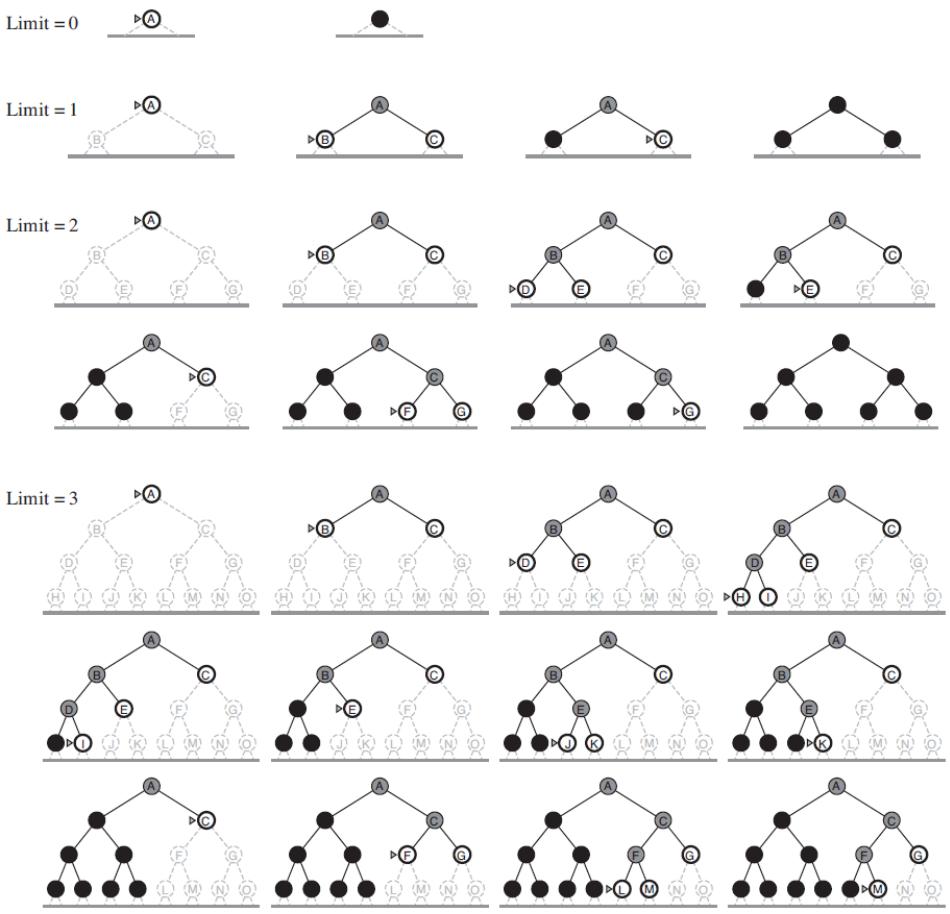
Uitvoer Een sequentie van acties wanneer een oplossing werd gevonden met diepte d of minder; een "hit boundary" conditie wanneer tijdens het zoekproces de maximale diepte werd bereikt of "error" wanneer er geen oplossing werd gevonden.

```
1:  function DEPTHLIMITEDSEARCH(P, d)
2:      c ← nieuw plan gebaseerd op initiële toestand P
3:      return DLSRECURSIVE(c, P, d)
4:  end function
```

Invoer Een huidig plan c, een zoekprobleem P en een maximale diepte d.

Uitvoer Een sequentie van acties wanneer een oplossing werd gevonden met diepte d of minder startend vanaf het huidig plan; een "hit boundary" conditie wanneer tijdens het zoekproces de maximale diepte werd bereikt of "error" wanneer er geen oplossing werd gevonden.

```
5:  function DLSRECURSIVE(c, P, d)
6:      if P.GOALTEST(c.GETSTATE) = true then
7:          return GETACTIONSEQ(c)                    ▷ Oplossing gevonden
8:      end if
9:      if d = 0 then
10:         return "hit boundary"                     ▷ Grens bereikt
11:     end if
12:     boundaryHit ← false  ▷ Grens bereikt in één van de rec. oproepen?
13:     for (s, a) ∈ c.GETSTATE.GETSUCCESSORS do
14:         child ← nieuw plan gebaseerd op (s, a) en c
15:         sol ← DLSRECURSIVE(child, P, d − 1)       ▷ Recursieve oproep
16:         if sol = "hit boundary" then
17:             boundaryHit ← true
18:         else
19:             if sol ≠ "error: geen oplossing gevonden" then
20:                 return sol                        ▷ Effectieve oplossing gevonden
21:             end if
22:         end if
23:     end for
24:     if boundaryHit = true then
25:         return "hit boundary"
26:     else
27:         return "error: geen oplossing gevonden"
28:     end if
29: end function
```



Figuur 6.8: Illustratie van 4 iteraties van iteratief verdiepen op een binaire boom. De doeltoestand is M. Het aantal gegenereerde toppen in de eerste drie iteraties samen is $1 + 3 + 7 = 11$. Het aantal gegenereerde toppen in de vierde iteratie is 13. Bron: (Russell and Norvig, 2014).

Algoritme 6.4 Iteratief verdiepen

Invoer Een zoekprobleem P

Uitvoer Een sequentie van acties wanneer een oplossing werd gevonden of "error" wanneer er geen oplossing werd gevonden.

```
1: function ITERATIVEDEEPENING(P)
2:     d ← 0
3:     sol ← DEPTHLIMITEDSEARCH(P, d)
4:     while sol = "hit boundary" do
5:         d ← d + 1
6:         sol ← DEPTHLIMITEDSEARCH(P, d)
7:     end while
8:     return sol                          ▷ Oplossing of error
9: end function
```

vindt op diepte d. We bekijken nu hoeveel toppen er gegenereerd worden bij elke oproep van diepte-gelimiteerd zoeken, waarna we al deze gegenereerde toppen optellen.

diepte	aantal gegenereerde knopen
0	1
1	$1 + b$
2	$1 + b + b^2$
$\vdots$	$\vdots$
d	$1 + b + b^2 + \cdots + b^d$

Als we nu de som uitvoeren dan zien we dat de term 1 in totaal $d + 1$ keer voorkomt, de term b komt d keer voor, de term b^2 komt $d - 1$ keer voor enzovoort. Het totaal aantal gegenereerde toppen is dus

$$(d + 1) + db + (d - 1)b^2 + (d - 2)b^3 + \cdots + b^d = \mathcal{O}(b^d).$$

Om onszelf te overtuigen dat de hoeveelheid "extra" werk beperkt blijft is het interessant om het aantal toppen gegenereerd in de laatste iteratie te vergelijken met het totaal aantal toppen gegenereerd in alle iteraties ervoor.

Het aantal toppen gegenereerd in de laatste iteratie (met maximale toegelaten diepte d) is

$$N(d) = 1 + b + b^2 + \cdots + b^d = \frac{b^{d+1} - 1}{b - 1}.$$

In het algemeen is het aantal toppen gegenereerd in de iteratie met maximale toegelaten diepte i gelijk aan

$$N(i) = \frac{b^{i+1} - 1}{b - 1}.$$

Het totaal aantal genereerde toppen in alle iteraties *behalve de laatste* is dus:

$$\begin{aligned}
\sum_{i=0}^{d-1} \frac{b^{i+1} - 1}{b - 1} &= \frac{1}{b-1} \left((\sum_{i=0}^{d-1} b^{i+1}) - d \right) \\
&= \frac{1}{b-1} \left((1 + b + b^2 + \cdots + b^d - 1) - d \right) \\
&= \frac{1}{b-1} \left(\frac{b^{d+1} - 1}{b - 1} - 1 - d \right) \\
&= \frac{1}{b-1} (N(d) - 1 - d) \\
&\approx \frac{1}{b-1} N(d).
\end{aligned}$$

Wanneer de vertakkingsfactor b bv. gelijk is aan 4, dan hebben we dus ongeveer 33% van het werk "te veel" gedaan.

Voorbeeld 6.14 Veronderstel dat de vertakkingsfactor b gelijk is aan 4 en dat de meest ondiepe oplossing zich bevindt op diepte 10. Het aantal toppen gegenereerd bij de laatste iteratie (met dieptelimiet 10) is gelijk aan:

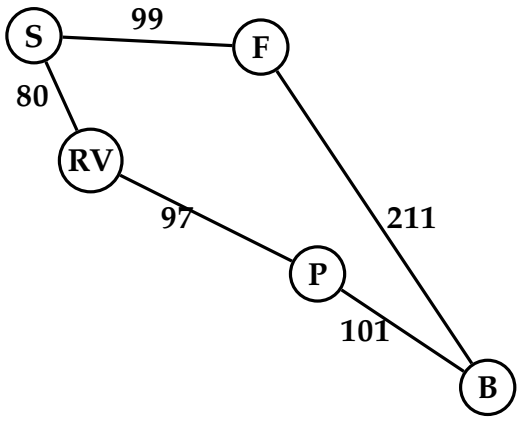
$$\frac{4^{11} - 1}{4 - 1} = 1\,398\,101.$$

Het aantal toppen gegenereerd in alle voorgaande iteraties is

$$\frac{1}{3}(1\,398\,101 - 1 - 10) = 466\,030.$$

∎

Iteratief verdiepen is een compleet zoekalgoritme en zal steeds de oplossing met het minste aantal acties vinden. Het algoritme is in het algemeen niet optimaal maar wel in het bijzondere geval dat alle acties dezelfde kost hebben. Het algoritme heeft een exponentiële tijdscomplexiteit maar slechts een lineaire ruimtecomplexiteit.



Figuur 6.9: Deel van de kaart van Roemenië. De kost om van de ene stad naar de andere te gaan staat bij de bogen. Figuur gebaseerd op: (Russell and Norvig, 2014).

6.3.4 Uniforme Kost Zoeken

Uniforme kost zoeken (Eng. *uniform cost search*) tracht het probleem dat breedte eerst niet noodzakelijk optimaal is (wanneer acties een verschillende kost hebben) op te lossen door steeds het plan te expanderen waarvoor de totale kost van dit plan minimaal is. De open lijst wordt hier m.a.w. geïmplementeerd aan de hand van een prioriteitswachtrij. Een kleinere kost betekent een grotere prioriteit.

Het idee achter uniforme kost zoeken is dus in essentie gelijk aan het algoritme van Dijkstra.

Uniforme kost zoeken is, wanneer alle acties een kost hebben die groter of gelijk is aan één of andere positieve ϵ, een compleet en optimaal algoritme. De tijd- en ruimtecomplexiteit is $\mathcal{O}(b^{1+\lfloor C^*/\epsilon \rfloor})$, waarbij C^* de kost van de optimale oplossing voorstelt.

Voorbeeld 6.15 (Belang van plaats doeltest) Veronderstel dat men op het deel van de kaart van Roemenië zoals getoond in Figuur 6.9 van S(ibiu) naar B(ucharest) wenst te reizen. We voeren het graafgebaseerde uniforme kost zoeken algoritme uit. Om plaats te winnen korten we de toestanden af tot hun eerste letters. Het verloop van het algoritme zie je in Tabel 6.1.

Bij dit voorbeeld zie je dat de *plaats* van de doeltest van cruciaal belang is. Na het expanderen van F(agaras) verschijnt het doel, nl. B(ucharest),

open lijst $\{(\text{pad}, g)\}$	gekozen plan	geëxpandeerde toestanden
$\{(S, 0)\}$	$(S, 0)$	$\emptyset$
$\{(F \to S, 99), (RV \to S, 80)\}$	$(RV \to S, 80)$	$\{S\}$
$\{(F \to S, 99),$ $(P \to RV \to S, 177),$ $(S \to RV \to S, 160)\}$	$(F \to S, 99)$	$\{S, RV\}$
$\{(P \to RV \to S, 177),$ $(S \to RV \to S, 160),$ $(B \to F \to S, 310),$ $(S \to F \to S, 198)\}$	$(S \to RV \to S, 160)$ niet geëxpandeerd	$\{S, RV, F\}$
$\{(P \to RV \to S, 177),$ $(B \to F \to S, 310),$ $(S \to F \to S, 198)\}$	$(P \to RV \to S, 177)$	$\{S, RV, F\}$
$\{(RV \to P \to RV \to S, 274),$ $(B \to P \to RV \to S, 278)$ $(B \to F \to S, 310),$ $(S \to F \to S, 198)\}$	$(S \to F \to S, 198)$ niet geëxpandeerd	$\{S, RV, F, P\}$
$\{(RV \to P \to RV \to S, 274),$ $(B \to P \to RV \to S, 278)$ $(B \to F \to S, 310)\}$	$(RV \to P \to RV \to S, 274)$ niet geëxpandeerd	$\{S, RV, F, P\}$
$\{(B \to P \to RV \to S, 278),$ $(B \to F \to S, 310)\}$	$(B \to P \to RV \to S, 278)$ doel bereikt	$\{S, RV, F, P\}$

Tabel 6.1: Verloop van uniforme kost zoeken om van Sibiu naar Bucharest te gaan volgens de kaart in Figuur 6.9.

reeds op de open lijst (met een actiesequentie die *niet* de optimale is). Het algoritme stopt echter niet op dit moment en verklaart slechts succes eens de doeltest slaagt voor het plan dat gekozen werd voor expansie. Enkel dan is het gegarandeerd dat het algoritme steeds een optimale oplossing vindt.∎

6.4 Geïnformeerde Zoekmethoden

De blinde of niet geïnformeerde zoekmethoden hebben geen toegang tot domeinkennis om het zoeken meer efficiënt te laten verlopen. Daardoor gaan ze vaak toestanden expanderen die wij (als mens) "stom" vinden. Probeer bv. maar eens het uniforme kost algoritme toe te passen wanneer je van Sibiu naar Bucharest wenst te gaan, gebruikmakend van de volledige kaart van Roemenië in Figuur 6.11!

Wanneer wij bv. een weg moeten gaan plannen tussen twee steden in een gebied waar we de weg niet kennen dan gaan wij toch altijd te neiging hebben om eerst te gaan kijken naar steden die al "in de juiste richting" liggen, i.e. steden waarvoor de afstand in vogelvlucht tot de doelstad klein is. We beschouwen de afstand in vogelvlucht als een goede indicator of schatting voor de werkelijke afstand[5]. We gebruiken een *heuristiek* om het zoekproces efficiënter te laten verlopen. We bekijken nu hoe dit idee geïmplementeerd kan worden a.d.h.v. twee zoekalgoritmes. Eerst bespreken we echter nog enkele eigenschappen van heuristieken.

6.4.1 Heuristieken

Definitie 6.16 Een HEURISTIEK h is een afbeelding van de verzameling toestanden S naar de verzameling van niet-negatieve reële getallen $\mathbb{R}^+$, i.e.

$$h\colon S \to \mathbb{R}^+ \colon s \mapsto h(s).$$ ∎

Opmerking 6.17 Het is uiteraard de bedoeling dat de heuristiek een goede schatting is voor de werkelijke kost naar het doel, i.e. wanneer $h(s)$ klein is dan is dit, hopelijk, een goede indicator dat s ook effectief niet ver van een doeltoestand verwijderd is, en omgekeerd wanneer $h(s)$ groot is dan moet dit erop wijzen dat s ver van het doel verwijderd is.

[5]Dit hoeft niet noodzakelijk zo te zijn, bv. wanneer er meren, rivieren of bergketens aanwezig zijn.

Aangezien de heuristiek h zal gebruikt worden in de zoekalgoritmes is het ook noodzakelijk dat *h snel te berekenen* is. ∎

Voorbeeld 6.18 (Heuristieken voor 8-puzzel) Voor de 8-puzzel kan bv. gekeken worden naar het aantal vakjes (genummerd van 1 t.e.m. 8) dat niet op zijn juiste plaats staat in vergelijking met doel

$$h_1 = \text{aantal niet-lege vakjes (tegeltjes) dat niet op zijn juiste plaats staat} \tag{6.2}$$

Deze heuristiek volgt de redenering dat hoe meer vakjes verkeerd zijn, hoe meer zetten nog nodig zijn om de puzzel volledig correct te maken.

We kunnen echter niet enkel rekening houden met het aantal verkeerde vakjes maar ook met de afstand (uitgedrukt m.b.v. de Manhattan afstand) tot hun doel:

$$h_2 = \sum_{i=1}^{8} (\text{Manhattan afstand van vakje } i \text{ tot zijn correcte plaats}). \tag{6.3}$$

Voor de linkerpuzzel in Figuur 6.1 neemt h_1 de waarde 8 aan, want alle niet-lege vakjes staan op de verkeerde plaats. De heuristiek h_2 neemt hier de waarde

$$3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$$

aan. Inderdaad, vakje 1 moet minstens 3 maal verschoven worden, vakje 2 minstens 1 maal, enzovoort. ∎

De heuristieken h_1 en h_2 zijn bijzonder in die zin dat ze de werkelijke afstand *nooit overschatten*. Zo'n heuristieken noemen we toelaatbaar.

Definitie 6.19 Een heuristiek $h\colon S \to \mathbb{R}^+$ is TOELAATBAAR als voor elke toestand s geldt dat $h(s) \leq C^*(s)$ waarbij C^* de kost van een optimale oplossing voorstelt van s naar een doeltoestand. ∎

Stelling 6.20 Wanneer de heuristiek h toelaatbaar is, dan is $h(g) = 0$ voor elke doeltoestand g. ∎

Bewijs Voor een doeltoestand g geldt dat de kost van de optimale oplossing gelijk is aan nul, zodat

$$0 \leq h(g) \leq C^*(g) = 0,$$

waaruit onmiddellijk volgt dat $h(g) = 0$. ♢

Definitie 6.21 Een heuristiek $h\colon S \to \mathbb{R}^+$ is CONSISTENT als voor elke doeltoestand g geldt dat $h(g) = 0$ en als bovendien voor elke toestand s en elke actie a op s met $s' = T(s, a)$ geldt dat

$$h(s) \leq c(s, a, s') + h(s').$$ ∎

Opmerking 6.22 Wanneer we de heuristiek beschouwen als "in vogelvlucht op doel afgaan" dan is een heuristiek consistent wanneer het steeds korter is om in vogelvlucht op doel af te gaan ($h(s)$) dan om eerst een actie te nemen a.d.h.v. het transitiemodel ($c(s, a, s')$) en dan in vogelvlucht op doel af te gaan ($h(s')$). ∎

Stelling 6.23 Als een heuristiek consistent is, dan is ze ook onmiddellijk toelaatbaar. ∎

Bewijs We kunnen voor elke actie a die een toestand s omzet in s' de *heuristische kost* definiëren als $h(s) - h(s')$. Voor een consistente heuristiek geldt dat

$$h(s) - h(s') \leq c(s, a, s'). \tag{6.4}$$

Voor een consistente heuristiek is de heuristische kost m.a.w. steeds kleiner dan of gelijk aan de werkelijke kost.

Beschouw nu een willekeurige toestand s waarvoor de kost van een optimale oplossing gegeven wordt door $C^*(s)$. We moeten aantonen dat $h(s) \leq C^*(s)$. Neem aan dat een optimale oplossing bestaat uit de volgende opeenvolging van acties (en bijhorende toestanden):

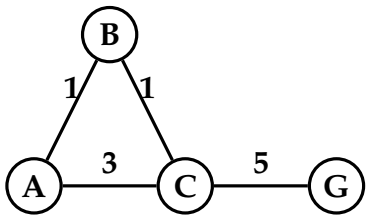
$$s = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \longrightarrow \cdots \longrightarrow s_{n-2} \xrightarrow{a_{n-1}} s_{n-1} \xrightarrow{a_n} s_n = g.$$

Nu geldt uiteraard dat

$$\begin{aligned}
C^*(s) &= c(s_0, a_1, s_1) + c(s_1, a_2, s_2) + \cdots + c(s_{n-2}, a_{n-1}, s_{n-1}) \\
&\quad + c(s_{n-1}, a_n, s_n) \\
&\geq \big(h(s_0) - h(s_1)\big) + \big(h(s_1) - h(s_2)\big) + \cdots + \big(h(s_{n-2}) - h(s_{n-1})\big) \\
&\quad + \big(h(s_{n-1}) - h(s_n)\big) \\
&= h(s_0) - h(s_n) \\
&= h(s) - h(g) \\
&= h(s).
\end{aligned}$$

Hierbij hebben we in de eerste stap gebruikgemaakt van het feit dat, voor een consistente heuristiek, de heuristische kost van een actie hoogstens gelijk is aan de werkelijke kost, zie vgl. (6.4). In de laatste stap maakten we gebruik van het feit dat, bij definitie, $h(g) = 0$ voor elke consistente heuristiek.

Dit toont aan dat elke consistente heuristiek ook toelaatbaar is. ♢



s	G	A	B	C
$h(s)$	0	4	6	1

Figuur 6.10: Voorbeeld van een eenvoudige toestandsruimte met een toelaatbare maar niet-consistente heuristiek.

Opmerking 6.24 (Toelaatbare maar inconsistente heuristiek) Het is *niet* zo dat elke toelaatbare heuristiek ook consistent is. Beschouw de toestandsruimte in Figuur 6.10, waarbij G de doeltoestand voorstelt. We zien op het zicht dat de kost van de optimale oplossing voor de verschillende toestanden gegeven wordt door

s	G	A	B	C
$C^*(s)$	0	7	6	5

Wanneer we dit vergelijken met de heuristiek h uit Figuur 6.10, dan zien we dat voor elke toestand s geldt dat

$$h(s) \leq C^*(s).$$

De heuristiek h is m.a.w. toelaatbaar.

Opdat de heuristiek h consistent zou zijn moet voor elke toestand s en voor elke opvolger s' gelden dat

$$h(s) \leq c(s, a, s') + h(s'). \tag{6.5}$$

We controleren dit nu systematisch voor de acht verschillende mogelijkheden

s	s'	$h(s)$	$c(s,a,s') + h(s')$	vgl. (6.5) voldaan?
A	B	4	$1+6=7$	ja
A	C	4	$3+1=4$	ja
B	A	6	$1+4=5$	nee
B	C	6	$1+1=2$	nee
C	A	1	$3+4=7$	ja
C	B	1	$1+6=7$	ja
C	G	1	$5+0=5$	ja
G	C	0	$5+1=6$	ja

We zien dat in twee gevallen de ongelijkheid (6.5) niet voldaan is, zo is bv. voor $s = B$ en $s' = A$ de heuristische kost $6 - 4 = 2$, en dit is groter dan de werkelijke kost. Een analoge redenering geldt voor $s = B$ en $s' = C$.∎

6.4.2 Gulzig Beste Eerst

De gulzig beste eerst zoekmethode maakt gebruik van een heuristiek h. De methode kiest steeds de top met de kleinste waarde van h als de volgende top die wordt geëxpandeerd. De open lijst wordt hier dus, net als bij uniforme kost zoeken, geïmplementeerd als een prioriteitswachtrij en een kleinere waarde voor h betekent een grotere prioriteit.

Voorbeeld 6.25 (Niet-optimaliteit gulzig beste eerst) In dit voorbeeld passen we het gulzig beste eerst algoritme toe op de toestandsruimtegraaf in Figuur 6.10 waarbij we de volgende consistente heuristiek gebruiken:

s	G	A	B	C
$h(s)$	0	7	6	5

De gebruikte heuristiek is in zekere zin "perfect" omdat ze voor elke toestand s gelijk is aan $C^*(s)$. De initiële toestand is A en de doeltoestand is G.

open lijst $\{(\text{pad}, f = h)\}$	gekozen top
$\{(A,7)\}$	$(A,7)$
$\{(B \to A,6),(C \to A,5)\}$	$(C \to A,5)$
$\{(B \to A,6),(A \to C \to A,7)$ $(G \to C \to A,0),(B \to C \to A,6)\}$	$(G \to C \to A,0)$

Het algoritme gaat recht op doel af. In dit geval behoort elke geëxpandeerde toestand ook effectief tot de geretourneerde oplossing. Helaas is de gevonden oplossing niet de optimale oplossing. In dit geval komt dit omdat het algoritme geen rekening houdt met de (hoge) kost om vanuit A de toestand C te bereiken. Het algoritme houdt enkel rekening met het feit dat C er "beter" uitziet dan B op basis van hun heuristische waarden. ∎

Voorbeeld 6.26 (Incompleetheid gulzig beste eerst) In dit voorbeeld passen we het gulzig beste eerst algoritme toe op de toestandsruimtegraaf in Figuur 6.10 waarbij we de volgende consistente heuristiek gebruiken:

s	G	A	B	C
$h(s)$	0	3	4	5

De initiële toestand is A en de doeltoestand is G. Bij de start bestaat de open lijst enkel uit de toestand A met bijhorende heuristische waarde 3. We expanderen A en voegen B en C toe aan de open lijst. Aangezien B een lagere heuristische waarde heeft dan C wordt B als eerste geëxpandeerd. De opvolgers A en C worden aan de open lijst toegevoegd. De open lijst bevat op dit moment dus drie plannen: $C \to A$, $C \to B \to A$ en $A \to B \to A$. Het laatste plan (voor toestand A) heeft de laagste heuristische waarde ($h = 3$) en bijgevolg wordt dit als volgende plan geëxpandeerd: het algoritme zit vast in een oneindige lus.

Zelfs met een consistente heuristiek is het boomgebaseerde gulzig beste eerst algoritme niet compleet. ∎

Opmerking 6.27 Omdat we in Voorbeeld 6.26 werken in een eindige toestandsruimte zou de graafgebaseerde versie van gulzig beste eerst *wel* een oplossing vinden. Of het gulzig beste eerst algoritme hier eindigt met de optimale oplossing hangt af van de manier waarop gekozen wordt tussen de plannen

$$(C \to A, h = 5) \quad \text{en} \quad (C \to B \to A, h = 5).$$ ∎

We hebben reeds gezien dat het gulzig beste eerst algoritme niet compleet en niet optimaal is. Ook de tijd- en ruimte complexiteit zijn in het slechtste geval van de orde $\mathcal{O}(b^m)$. Deze exponentiële tijdscomplexiteit kan door het gebruik van een goede heuristiek echter sterk teruggedrongen worden.

Stad	Afstand	Stad	Afstand
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Tabel 6.2: Afstanden in vogelvlucht tot Bucharest. Gebaseerd op (Russell and Norvig, 2014).

6.4.3 A* Zoekalgoritme

Het probleem van de gulzig beste eerst zoekmethode is dat er *enkel* rekening wordt gehouden met de waarde van de heuristiek en niet met de kost van de reeds afgelegde weg. Er wordt m.a.w. waardevolle informatie genegeerd.

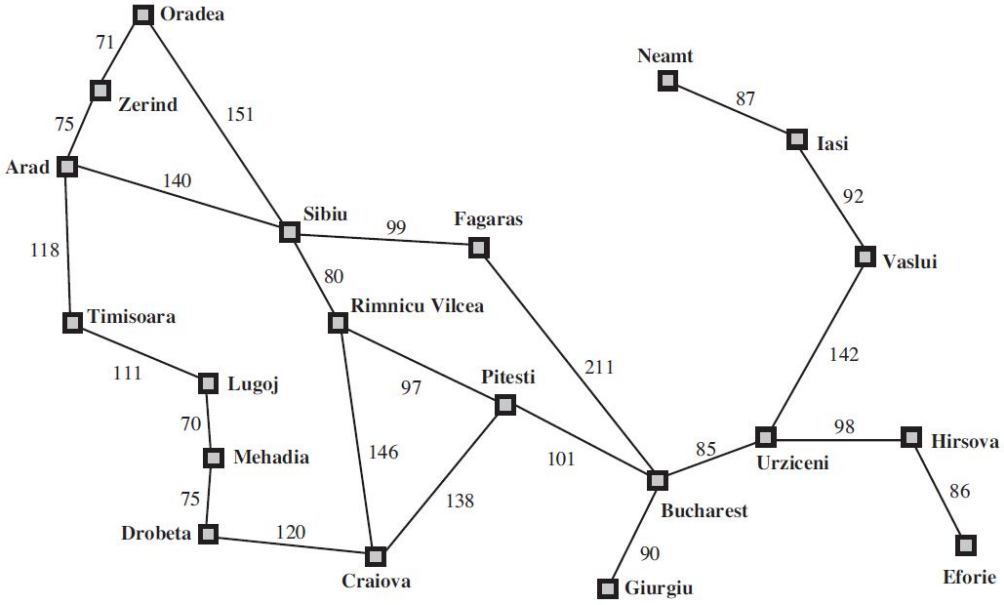
Bij de A* zoekmethode wordt de open lijst nog steeds geïmplementeerd als een prioriteitswachtrij maar de volgende top die wordt geëxpandeerd is de top (plan) n waarvoor

$$f(n) = g(n) + h(n)$$

minimaal is. Hierbij is $g(n)$ de totale kost van het plan n, i.e. de kost van de reeds gekozen acties, en is $h(n)$ (met misbruik van notatie) de waarde van de heuristiek voor de toestand die hoort bij deze top.

Voorbeeld 6.28 (A* met een toelaatbare heuristiek) We proberen op de kaart van Roemenië, zie Figuur 6.11, een weg te vinden van Arad naar Bucharest. We gebruiken het A* algoritme met als heuristiek de afstand in vogelvlucht (tot Bucharest), zoals gegeven in Tabel 6.2. De opbouw van de zoekboom kan je volgen in Figuur 6.12.

We starten in Arad en initieel bestaat de open lijst enkel uit het plan A. De f-waarde die bij dit plan hoort is $f = 0 + 366$, want de afgelegde afstand



Figuur 6.11: Een vereenvoudige kaart van Roemenië. Bron: (Russell and Norvig, 2014).

om van Arad in Arad te geraken is 0 en de geschatte afstand naar Bucharest (in vogelvlucht) is 366.

Dit enige plan wordt van de open lijst gehaald. Aangezien Arad niet voldoet aan de doeltest wordt dit plan geëxpandeerd. De plannen $S \rightarrow A$, $T \rightarrow A$ en $Z \rightarrow A$ worden aan de open lijst toegevoegd:

$$(S \rightarrow A, 140 + 253 = 393), (T \rightarrow A, 118 + 329 = 447),$$
$$(Z \rightarrow A, 75 + 374 = 449).$$

Aangezien het plan $(S \rightarrow A, 140 + 253 = 393)$ de kleinste f-waarde heeft wordt dit als volgende van de open lijst gehaald. Het doel is nog niet bereikt en dus wordt dit plan geëxpandeerd. De volgende plannen worden toegevoegd aan de open lijst:

$$(A \rightarrow S \rightarrow A, 280 + 366 = 646), (F \rightarrow S \rightarrow A, 239 + 176 = 415),$$
$$(O \rightarrow S \rightarrow A, 291 + 380 = 671), (RV \rightarrow S \rightarrow A, 220 + 193 = 413).$$

De open lijst bestaat nu uit 6 plannen waarvan het plan $RV \rightarrow S \rightarrow A$ de kleinste f-waarde heeft. Dit plan wordt van de open lijst gehaald, en aangezien het doel nog niet werd bereikt wordt het geëxpandeerd en worden de

volgende plannen toegevoegd aan de open lijst:

$$(C \to RV \to S \to A, 366 + 160 = 526), (P \to RV \to S \to A, 317 + 100 = 417)$$
$$(S \to RV \to S \to A, 300 + 253 = 553).$$

Van de 8 plannen op de open lijst heeft het plan $F \to S \to A$ de laagste f-waarde, nl. 415. Dit plan wordt dus als volgende geëxpandeerd aangezien het doel nog niet werd bereikt. De plannen

$$(S \to F \to S \to A, 338 + 253 = 591), (B \to F \to S \to A, 450 + 0)$$

worden toegevoegd aan de open lijst. Merk op dat de open lijst op dit moment reeds een plan bevat dat aan de doeltest voldoet. Het algoritme stopt echter nog niet. Aangezien het plan $P \to RV \to S \to A$ een f-waarde heeft van 417 is het eventueel nog mogelijk om hierlangs een betere oplossing te vinden. We expanderen dit plan en voegen de volgende plannen toe aan de open lijst:

$$(B \to P \to RV \to S \to A, 418 + 0 = 418),$$
$$(C \to P \to RV \to S \to A, 455 + 160 = 615),$$
$$(RV \to P \to RV \to S \to A, 414 + 193 = 607).$$

Het plan $B \to P \to RV \to S \to A$ is het plan met de laagste f-waarde. Dit plan wordt van de open lijst gehaald. De doeltest slaagt voor dit plan (we hebben Bucharest bereikt) en het algoritme eindigt en geeft de volgende oplossing terug:

$$A \to S \to RV \to P \to B.$$

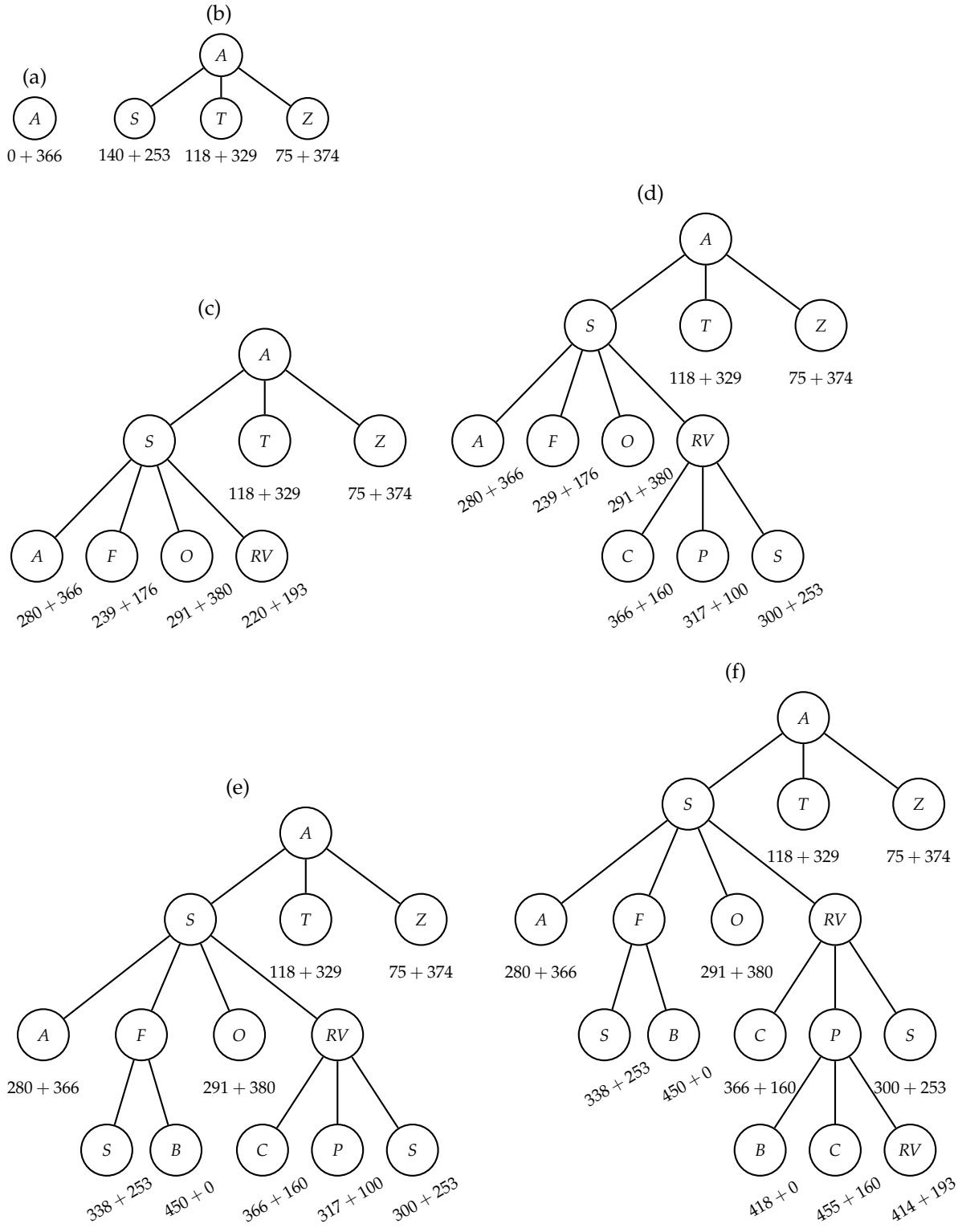
Dit is de optimale oplossing. ∎

Voorbeeld 6.29 (A* met een niet toelaatbare heuristiek) Beschouw de toestandsruimtegraaf in Figuur 6.10 maar gebruik de volgende heuristiek

s	G	A	B	C
$h(s)$	0	4	8	1

Deze heuristiek is niet toelaatbaar omdat de waarde voor $h(B)$ strikt groter is dan de kost van de optimale oplossing van B naar de doeltoestand G.

We voeren nu A* uit startend in de toestand A.



Figuur 6.12: Progressie van A* bij het zoeken van Arad naar Bucharest in Roemenië.

open lijst $\{(\text{pad}, f)\}$	gekozen knoop
$\{(A, 0+4)\}$	$(A, 0+4)$
$\{(B \to A, 1+8), (C \to A, 3+1)\}$	$(C \to A, 3+1)$
$\{(B \to A, 1+8), (A \to C \to A, 6+4)$ $(B \to C \to A, 4+8), (G \to C \to A, 8+0)\}$	$(G \to C \to A, 8+0)$

Het algoritme eindigt dus met de oplossing $A \to C \to G$. Deze oplossing is niet optimaal want de oplossing $A \to B \to C \to G$ heeft een kleinere kost.

Dit voorbeeld toont aan dat A* niet noodzakelijk optimaal is wanneer een niet-toelaatbare heuristiek wordt gebruikt. ∎

De volgende stelling toont aan dat het gebruik van een toelaatbare heuristiek *voldoende* is om optimaliteit te garanderen wanneer men het A* algoritme gebruikt in zijn boomgebaseerde versie (op voorwaarde dat alle acties een strikt positieve kost hebben).

Stelling 6.30 Wanneer boomgebaseerde A* gebruikmaakt van een toelaatbare heuristiek h en wanneer alle acties een kost hebben groter of gelijk aan een zekere strikt positieve ϵ, dan is A* compleet en optimaal, i.e. dan vindt het algoritme steeds een optimale oplossing wanneer die bestaat. ∎

Bewijs We tonen eerst aan dat de open lijst steeds een plan bevat dat uitgebreid kan worden tot een optimale oplossing; zo'n plan noemen we een "uitbreidbaar" plan. Dit is uiteraard zo bij de start van het algoritme aangezien de open lijst een plan bevat met een leeg pad. Bij elke iteratie van het algoritme zijn er twee mogelijkheden: ofwel werd een niet-uitbreidbaar plan gekozen voor expansie en staan de uitbreidbare plannen nog steeds op de open lijst. Ofwel werd een uitbreidbaar plan gekozen: in dit geval is minstens één van zijn opvolgers ook een "uitbreidbaar" plan.

Noem $C^*(s)$ de kost van de optimale oplossing vanaf de toestand s tot een doeltoestand. Voor zo'n uitbreidbaar plan p geldt wegens de toelaatbaarheid van de heuristiek h dat

$$\begin{aligned} f(p) &= g(p) + h(p) \\ &\leq g(p) + C^*(p) && \text{toelaatbaarheid } h \\ &= C^*(s_0). && s_0 \text{ is de starttoestand} \end{aligned}$$

Veronderstel dat op een bepaald moment in het algoritme een "gevaarlijke" situatie optreedt in die zin dat er een plan n op de open lijst verschijnt waarvoor de doeltest voldaan is maar waarvoor de kost van het pad suboptimaal is. We tonen nu aan dat dit plan *n nooit*

zal gekozen worden voor expansie. Noem p een uitbreidbaar plan op de open lijst. Er geldt

$$\begin{aligned} f(n) &= g(n) + h(n) \\ &= g(n) && n \text{ is doeltoestand en } h \text{ is toelaatbaar} \\ &> C^*(s_0) && \text{want } n \text{ is suboptimaal} \\ &\geq f(p) && \text{zie voorgaande berekening} \end{aligned}$$

Dit toont inderdaad aan dat het plan n nooit zal gekozen worden voor expansie, want p moet zeker eerder worden gekozen. Het algoritme kan bijgevolg nooit een suboptimale oplossing teruggeven.

We tonen tenslotte aan dat A* steeds eindigt wanneer er een oplossing is. Inderdaad, omdat elke actie een kost heeft van minstens ϵ, kunnen er slechts een eindig aantal plannen zijn met kost kleiner of gelijk aan $f(p)$ met p het uitbreidbare plan met minimale $f(p)$ dat nu op de open lijst staat. Dit plan wordt dus binnen een eindig aantal iteraties geëxpandeerd, waardoor er nu een uitbreidbaar plan op de open lijst staat dat één actie minder ver verwijderd is van de doeltoestand dan p. Uiteindelijk zal er dus een plan worden gekozen dat aan de doeltest voldoet. Zoals reeds aangetoond kan dit plan geen suboptimale oplossing naar het doel bevatten. ♢

Uit volgend voorbeeld blijkt dat het voor het A* algoritme niet voldoende is om te werken met een toelaatbare heuristiek wanneer men graafgebaseerd zoeken uitvoert.

Voorbeeld 6.31 (Toelaatbaarheid en graafgebaseerd zoeken) Bekijk de toestandsruimte en de heuristiek in Figuur 6.10. Zoals reeds gezien in Opmerking 6.24 is deze heuristiek toelaatbaar maar inconsistent.

We voeren A* uit in de graafgebaseerde versie. Het verloop van het algoritme kan je volgen in Tabel 6.3.

Zoals je ziet eindigt A* hier met een foutief antwoord: de sequentie $A \rightarrow C \rightarrow G$ is *geen* optimale oplossing. Het probleem is dat de sequentie van acties naar de toestand C die het eerst werd geëxpandeerd *niet* de optimale sequentie is. De betere sequentie die later werd ontdekt ($A \rightarrow B \rightarrow C$) wordt niet meer in beschouwing genomen want C is reeds geëxpandeerd en is dus reeds een element van de gesloten lijst.

Dit kan niet gebeuren wanneer er gewerkt wordt met een consistente heuristiek. ∎

Stelling 6.32 Wanneer graafgebaseerde A* gebruikmaakt van een consistente heuristiek h en wanneer alle acties een kost hebben groter of gelijk aan

open lijst $\{(\text{pad}, f)\}$	gekozen top	geëxpandeerde toestanden
$\{(A, 0+4)\}$	$(A, 0+4)$	$\emptyset$
$\{(B \to A, 1+6), (C \to A, 3+1)\}$	$(C \to A, 3+1)$	$\{A\}$
$\{(B \to A, 1+6), (A \to C \to A, 6+4)$ $(B \to C \to A, 4+6), (G \to C \to A, 8+0)\}$	$(B \to A, 1+6)$	$\{A, C\}$
$\{(A \to C \to A, 6+4), (B \to C \to A, 4+6)$ $(G \to C \to A, 8+0), (A \to B \to A, 2+4)$ $(C \to B \to A, 2+1)$	$(C \to B \to A, 2+1)$ niet geëxpandeerd	$\{A, C, B\}$
$\{(A \to C \to A, 6+4), (B \to C \to A, 4+6)$ $(G \to C \to A, 8+0), (A \to B \to A, 2+4)\}$	$(A \to B \to A, 2+4)$ niet geëxpandeerd	$\{A, C, B\}$
$\{(A \to C \to A, 6+4), (B \to C \to A, 4+6)$ $(G \to C \to A, 8+0)\}$	$(G \to C \to A, 8+0)$ doeltest geslaagd	$\{A, C, B\}$

Tabel 6.3: Uitvoering van het graafgebaseerde A*-algoritme met een toelaatbare maar inconsistente heuristiek.

een zekere strikt positieve ϵ, dan is A* compleet en optimaal, i.e. dan vindt het algoritme steeds een optimale oplossing wanneer die bestaat. ∎

Bewijs Zonder bewijs. ♢

A* is duidelijk een interessant algoritme: onder "milde" beperkingen van de gebruikte heuristiek is A* compleet en optimaal. De toppen die door A* worden geëxpandeerd zijn alle toppen n waarvoor

$$f(n) = g(n) + h(n) < C^*(s_0).$$

Er wordt geen enkele top geëxpandeerd waarvoor $f(n)$ strikt groter is dan $C^*(s_0)$. Het hangt van de implementatie van het algoritme of toppen met $f(n) = C^*(s_0)$ worden geëxpandeerd of niet. M.a.w. hoe beter (groter) de heuristiek hoe minder toppen worden geëxpandeerd[6].

A* is echter geen wonderalgoritme: de tijds- en ruimtecomplexiteit zijn in het slechtste geval nog steeds exponentieel, waarbij in de meeste gevallen A* sneller een tekort heeft aan geheugen dan aan tijd.

[6]Opletten: indien h "te groot" wordt is deze misschien niet meer toelaatbaar!

6.5 Ontwerpen van Heuristieken

Het is duidelijk dat een goede heuristiek een grote positieve invloed kan hebben op de tijds- en ruimtecomplexiteit van een algoritme zoals A*. Hoe kunnen we nu zo'n heuristieken construeren?

We bekijken twee manieren om dit aan te pakken.

6.5.1 Gebruik van Vereenvoudigde Problemen

Voorbeeld 6.33 (Agent in Doolhof) Veronderstel dat de agent zich in een grid beweegt en dat de agent zich door middel van de acties `Boven`, `Onder`, `Links` en `Rechts` naar een bepaalde locatie in het grid moet bewegen. Echter, sommige locaties zijn muren zodat het grid een doolhof voorstelt. Als we nu de *muren wegdenken* dan is de oplossing van dit vereenvoudigde probleem (Eng. *relaxed problem*) onmiddellijk te berekenen. Het aantal benodigde acties is niets anders dan de Manhattan-afstand tussen de huidige locatie en de doellocatie; dit is meteen ook een aanvaardbare (en consistente) heuristiek voor het oorspronkelijke probleem. ∎

Voorbeeld 6.34 (De 8-puzzel) Wanneer we de regels voor de 8-puzzel wat formeler opschrijven dan kunnen we zeggen dat vakje A naar vakje B kan verplaatst worden als

1. de vakjes A en B aangrenzend zijn; **en**
2. het vakje B is het lege vakje.

Door nu één of meerdere van deze condities (restricties) weg te laten bekomen we de volgende drie vereenvoudigde problemen.

1. Vakje A kan naar vakje B worden verplaatst als A en B aangrenzend zijn.
2. Vakje A kan naar vakje B worden verplaatst als B het lege vakje is.
3. Vakje A kan naar vakje B worden verplaatst (zonder voorwaarden).

Wanneer we het derde vereenvoudigde probleem bekijken dan zien we dat de optimale oplossing van dit probleem erin bestaat om elk vakje dat verkeerd staat te verplaatsen naar zijn juiste positie. Het aantal acties dat hiervoor nodig is precies het aantal vakjes dat niet op zijn juiste plaats staat; dit is niets anders de heuristiek h_1 uit vergelijking (6.2).

De optimale oplossing van het eerste vereenvoudigde probleem bestaat erin om elk vakje naar zijn correcte positie te schuiven (zonder er rekening mee te houden dat er andere vakjes "in de weg" kunnen staan). Het aantal acties nodig per vakje is de Manhattan-afstand van zijn huidige positie tot zijn doelpositie. Uit dit vereenvoudigde probleem leiden we m.a.w. de heuristiek h_2 uit vergelijking (6.3) af.

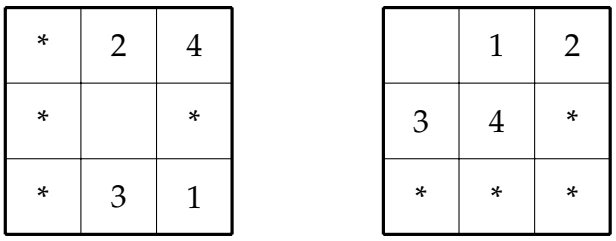
Het tweede vereenvoudigde probleem geeft aanleiding tot nog een andere heuristiek. ∎

Het is uiterst belangrijk dat de vereenvoudigde problemen efficiënt kunnen opgelost worden, i.e. *zonder het uitvoeren van een zoekalgoritme*! Immers, zoals reeds vermeld is het belangrijk dat een heuristiek efficiënt berekend kan worden.

Opmerking 6.35 Het is interessant om op te merken dat het proces van het vinden van heuristieken kan geautomatiseerd worden. Het programma ABSolver heeft zo een nieuwe heuristiek voor de 8-puzzel ontwikkeld die beter was dan de reeds bestaande (Prieditis, 1993). Het programma vond ook de eerste nuttige heuristiek voor het oplossen van Rubik's kubus. ∎

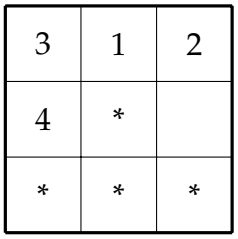
6.5.2 Patroon Databanken

Een aanvaardbare heuristiek kan ook gevonden worden als de kost van een optimale oplossing voor een *deelprobleem*. Veronderstel dat we in de 8-puzzel de vakjes 5 t.e.m. 8 vervangen door een identiek symbool, bv. een sterretje. In Figuur 6.13 zie je de instanties van dit soort puzzel wanneer dit proces wordt toegepast op de 8-puzzels uit Figuur 6.1. Het is duidelijk dat het minimum aantal stappen nodig om de puzzel aan de linkerkant van Figuur 6.13 om te zetten naar de puzzel aan de rechterkant van Figuur 6.13 een ondergrens is voor het aantal stappen nodig om de originele puzzel op te lossen.

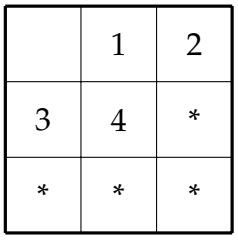


Figuur 6.13: Een voorbeeld van een deelprobleem voor de 8-puzzel.

Wat men nu kan doen is een databank aanleggen met voor elk patroon de optimale oplossingskost van het deelprobleem. In dit geval is het aantal deelproblemen $9 \times 8 \times 7 \times 6 \times 5 = 15120$. Het aanleggen van de databank kan (omdat de acties omkeerbaar zijn) vereenvoudigd worden door (graaf-gebaseerde) breedte-eerst uit te voeren startend vanaf het doelpatroon, en bij te houden wat de afstand is vanaf het doelpatroon (dat gebruikt werd als initiële toestand). Figuur 6.14 geeft een illustratie van dit proces. Uit de boom in Figuur 6.14 lezen we af dat de optimale oplossingkost van

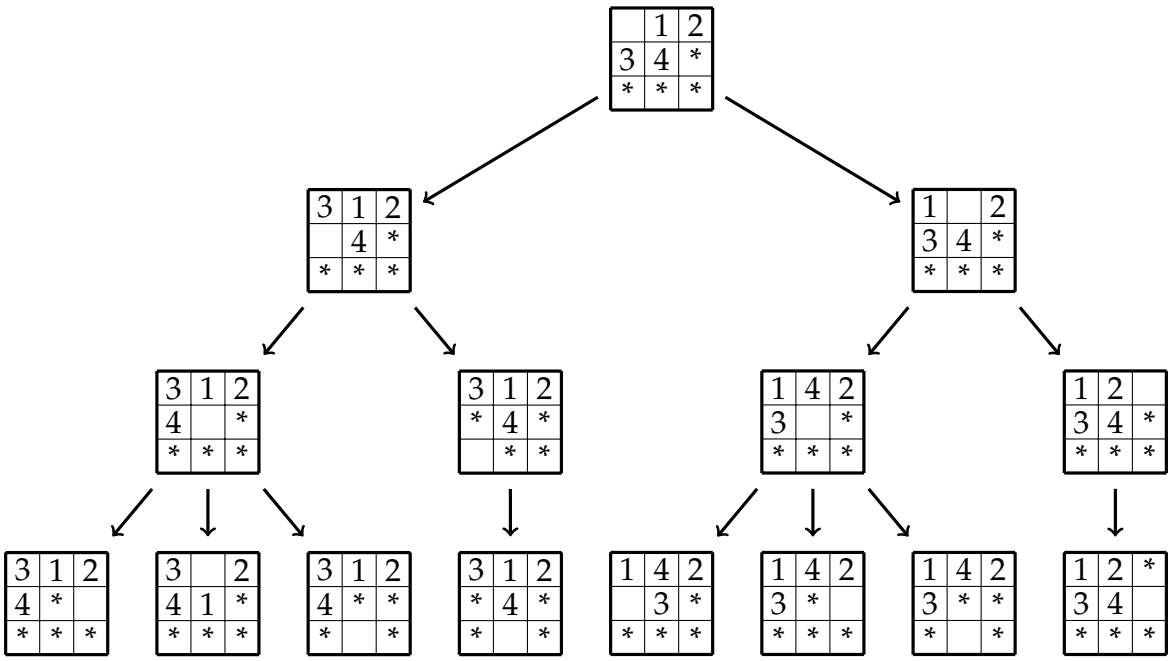


naar



gelijk is aan drie.

Uiteraard is het tamelijk arbitrair om de vakjes 5 t.e.m. 8 te vervangen door sterretjes. Men kan evengoed de vakjes 1 t.e.m. 4 vervangen. Op die manier bekomt men een tweede heuristiek. Voor sommige probleeminstanties zal



Figuur 6.14: Illustratie van het achterwaarts zoeken voor het opbouwen van een patroon databank. De drie eerste lagen van het breedte-eerst zoekproces worden getoond. Er wordt graafgebaseerd zoeken gebruikt dus elk patroon komt hoogstens éénmaal voor in de zoekboom. (Om de tekening te vereenvoudigen tonen we toestanden die niet opnieuw zullen geëxpandeerd worden niet.) Bovendien stopt hier het zoeken pas nadat de volledige boom is opgebouwd; er is m.a.w. geen expliciete doeltoestand. In de figuur worden slechts de eerste drie lagen van de zoekboom getoond.

de eerste heuristiek een betere (grotere) waarde geven dan de tweede en voor andere probleeminstanties zal dit net omgekeerd zijn. Men kan echter de twee heuristieken *combineren* door het nemen van het maximum:

$$h(s) = \max\big(h_1(s), h_2(s)\big).$$

Het resultaat is een betere heuristiek die nog steeds aanvaardbaar is.

6.6 Oefeningen

1. (Russell and Norvig, 2014, Oefening 3.4) Beschouw twee vrienden die in verschillende steden wonen, bv. in Roemenië. Bij elke actie kunnen

we elke vriend simultaan naar een naburige stad op de kaart verplaatsen. De hoeveelheid tijd nodig om zich van stad i naar de aanpalende stad j te verplaatsen is gelijk aan de afstand $d(i, j)$. Bij elke actie moet de vriend die eerst aankomt wachten tot de andere ook aankomt. De twee vrienden willen zo vlug als mogelijk samenkomen.

a) Geef een gedetailleerde beschrijving van het zoekprobleem.

b) Beschouw $D(i, j)$ als de afstand in vogelvlucht tussen de twee steden i en j. Welke van volgende heuristieken zijn toelaatbaar wanneer de eerste vriend zich in stad i en de tweede zich in stad j bevindt.

   i. $D(i, j)$
   ii. $2 \cdot D(i, j)$
   iii. $D(i, j)/2$

c) Zijn er toestanden (in de wetenschap dat er een pad is tussen alle steden op de kaart) waarvoor geen oplossing bestaat? Leg uit.

2. Een aantal robots (bv. k) leven in een rooster waarin sommige locaties muren zijn. Twee robots kunnen zich nooit op dezelfde locatie bevinden. Elke robot heeft zijn eigen bestemming. Bij elke tijdseenheid verplaatsen de robots zich simultaan naar een aanpalend (vrij) vierkant of blijven ze staan. Twee robots die zich naast elkaar bevinden kunnen niet van plaats wisselen in één tijdseenheid. Elke tijdseenheid kost één punt. Beantwoord de volgende vragen:

   a) Geef een minimale correcte voorstelling van een toestand in de toestandsruimte.

   b) Schat de grootte van de toestandsruimte in voor een rooster met als afmetingen $M \times N$.

   c) Welke van volgende heuristieken zijn toelaatbaar? Beargumenteer je antwoord, meer bepaald: geef een situatie waarvan je aangeeft dat de gegeven heuristiek niet toelaatbaar is wanneer je dit beweert.

      i. Som van de Manhattan afstanden voor elke robot tot zijn doellocatie.
      ii. Som van de kosten van de optimale paden indien de robots zich alleen in de omgeving voortbewegen, m.a.w. zonder obstructie door andere robots.

iii. Maximum van de Manhattan afstanden vanuit elke robotpositie tot zijn doelpositie.

iv. Maximum van de kosten van de optimale paden indien de robots zich alleen in de omgeving voortbewegen, m.a.w. zonder obstructie door andere robots.

v. Aantal robots die zich niet op hun doellocatie bevinden.

3. Beschouw de toestandsruimtegraaf in Figuur 6.15. De starttoestand is steeds S en de doeltoestand is G. Voer een aantal zoekalgoritmes uit op deze toestandsruimtegraaf. Indien er ergens "random" een plan uit de open lijst moet gekozen worden, neem je het plan dat eindigt in de lexicografisch kleinste toestand. Hierdoor wordt de oplossing steeds uniek.

   De heuristiek gebruikt door de geïnformeerde zoekmethoden staat gegeven in onderstaande tabel:

toestand	S	A	B	C	D	E	G
h	6	0	6	4	1	10	0

   Geef voor onderstaande blinde en geïnformeerde zoekmethodes het pad naar de doeltoestand. Geef ook aan welke toestanden geëxpandeerd werden en dit in de juiste volgorde van hun expansie. De geïnformeerde zoekmethoden gebruiken de heuristiek h.

   a) Diepte-eerst (boomgebaseerd)

   b) Gulzig beste-eerst (boomgebaseerd)

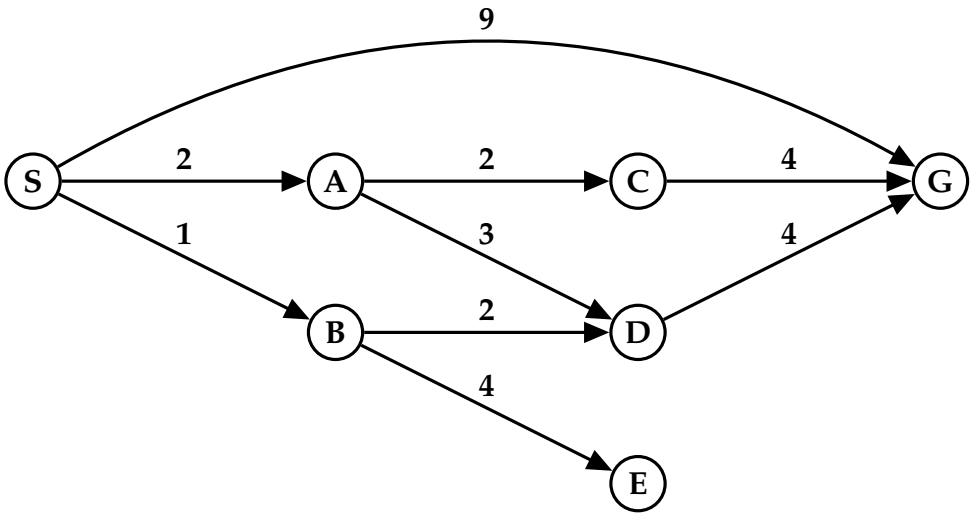
   c) Uniforme kost zoeken (boomgebaseerd)

   d) A^* (boomgebaseerd)

   e) A^* (graafgebaseerd)

4. Pas het gulzig beste eerst algoritme toe om van Arad naar Bucharest te gaan op de kaart van Roemenië in Figuur 6.11. Gebruik de afstand in vogelvlucht uit Tabel 6.2 als heuristiek.

5. Gebruik A* om van Lugoj naar Bucharest te gaan op de kaart van Roemenië in Figuur 6.11 met de afstand in vogelvlucht als heuristiek. Teken de opgebouwde zoekboom en geef aan in welke volgorde de plannen verwijderd worden van de open lijst. Los deze oefening eerst op



Figuur 6.15: Toestandsruimtegraaf voor vraag 3.

voor boomgebaseerd zoeken en vervolgens voor graafgebaseerd zoeken.

6. Rush Hour wordt gespeeld op een $n \times n$ spelbord. Op het spelbord staan een aantal auto's, die elk twee aaneengrenzende vakjes beslaan, en een aantal vrachtwagens, die elk drie aaneengrenzende vakjes beslaan. Het spelbord bevat aan één van de zijden een uitgang. De spelregels zijn heel eenvoudig: je kan bij elke beurt een willekeurige auto of vrachtwagen een aantal vakjes voorwaarts of achterwaarts verplaatsen in de richting waarin deze geplaatst is op het spelbord. Uiteraard kunnen voertuigen elkaar niet overlappen en kunnen ze niet over elkaar springen. Het probleem is dat een specifieke auto (op Figuur 6.16 gemarkeerd met X) naar de uitgang van het spelbord moet geleid worden. Hierbij moet de totale afgelegde afstand (som van alle afstanden door verplaatsen van auto's en/of vrachtwagens) zo laag mogelijk gehouden worden.

   Geef minstens twee aanvaardbare heuristieken voor dit probleem verschillend van de triviale aanvaardbare heuristiek $h = 0$.

Figuur 6.16: Illustratie van het spel Rush Hour.

HOOFDSTUK 7

Zoeken met een Tegenstander

In hoofdstuk 6 kon het zoekprobleem opgelost worden door eenvoudigweg "vooruit te denken" omdat men in dit geval te maken heeft met toestanden die enkel wijzigen wanneer men een actie onderneemt. In dit hoofdstuk voegen we een tegenstander toe, zodat het algoritme niet enkel rekening moet houden met zijn eigen acties maar ook met de acties die de tegenstander zal ondernemen. We starten met de definitie van een TWEE PERSOONS NULSOMSPEL en bestuderen het MINIMAX ALGORITME dat aangeeft hoe men optimaal moet spelen (onder bepaalde aannames). Het minimax algoritme kan versneld worden door bepaalde delen van de spelboom te snoeien; dit gebeurt m.b.v. α-β-SNOEIEN. In de laatste sectie bekijken we nog kort de praktische uitwerking van dergelijke algoritmen.

7.1 Inleiding

In dit hoofdstuk bekijken we hoe een agent moet handelen om zijn performantiemaat te maximaliseren wanneer er in de omgeving nog een andere (competitieve) agent aanwezig is. We gaan er verder van uit dat de omgeving (voor beide agenten) compleet observeerbaar is, en dat er voor elke toestand slechts een eindig aantal mogelijke deterministische acties zijn. De agenten spelen om de beurt. Zo'n omgeving wordt vaak een "spel" genoemd.

Voorbeeld 7.1 Voorbeelden van spellen die voldoen aan de bovenstaande

beschrijving zijn: vier-op-een-rij, dammen, schaken[1] en Go. Backgammon valt bijvoorbeeld niet onder deze noemer: het spel is weliswaar volledig observeerbaar maar is niet deterministisch want het rollen van een dobbelsteen bepaalt de volgende toestand. Veel kaartspellen, zoals poker, zijn weliswaar deterministisch maar zijn partieel observeerbaar omdat men de kaarten van de tegenspeler(s) niet kent. ∎

We starten met de definities die we in het vervolg van dit hoofdstuk zullen gebruiken.

Definitie 7.2 Een TWEE PERSOONS NULSOMSPEL (Eng. *two person zero-sum game*) wordt gespeeld door twee spelers (genaamd Max en Min) en bestaat verder uit de volgende componenten:

- Een verzameling van toestanden S.
- Een verzameling van mogelijke acties A.
- Een TRANSITIEMODEL dat zegt wat het effect is van het uitvoeren van een actie op een bepaalde toestand:

  $$T\colon (S, A) \to S\colon (s, a) \to s'.$$

  Wanneer s' bereikt wordt door het uitvoeren van een actie a op een toestand s dan wordt s' een *opvolger* van s genoemd.
- De initiële toestand s_o: deze bepaalt (uiteraard) de toestand voor er een zet werd gedaan.
- Een EINDTEST (Eng. *terminal test*) die voor een toestand aangeeft of het spel in deze toestand gedaan is of niet. De toestanden waarvoor de eindtest **true** teruggeeft worden EINDTOESTANDEN genoemd.
- Een OPBRENGSTFUNCTIE U (Eng. *utility function, payoff function*). De functie U zegt voor elke eindtoestand en voor elke speler wat de opbrengst is in deze toestand voor de gegeven speler. Omdat het spel een nulsomspel is geldt voor alle eindtoestanden s dat

  $$U(s, \text{Max}) + U(s, \text{Min}) = K,$$

  waarbij K een constante is die hoort bij het spel. ∎

[1] Schaken heeft een aantal zeer kleine niet-observeerbare componenten, bv. de rokade.

Opmerking 7.3 Merk op dat de constante K niet noodzakelijk gelijk is aan nul. "Constante-somspel" was dus wellicht een betere naam geweest dan de historisch gegroeide naam nulsomspel. ∎

De speler Max tracht te eindigen in een toestand waarvoor $U(\cdot, \text{Max})$ zo groot mogelijk is. Hij tracht m.a.w. zijn opbrengstfunctie te maximaliseren. De speler Min tracht uiteraard $U(\cdot, \text{Max})$ te minimaliseren, want dan is, gezien het feit dat het een nulsomspel is, zijn eigen opbrengst maximaal. In het vervolg bedoelen we met de *opbrengst U steeds de opbrengst van de speler Max*. De speler Max is steeds eerst aan zet.

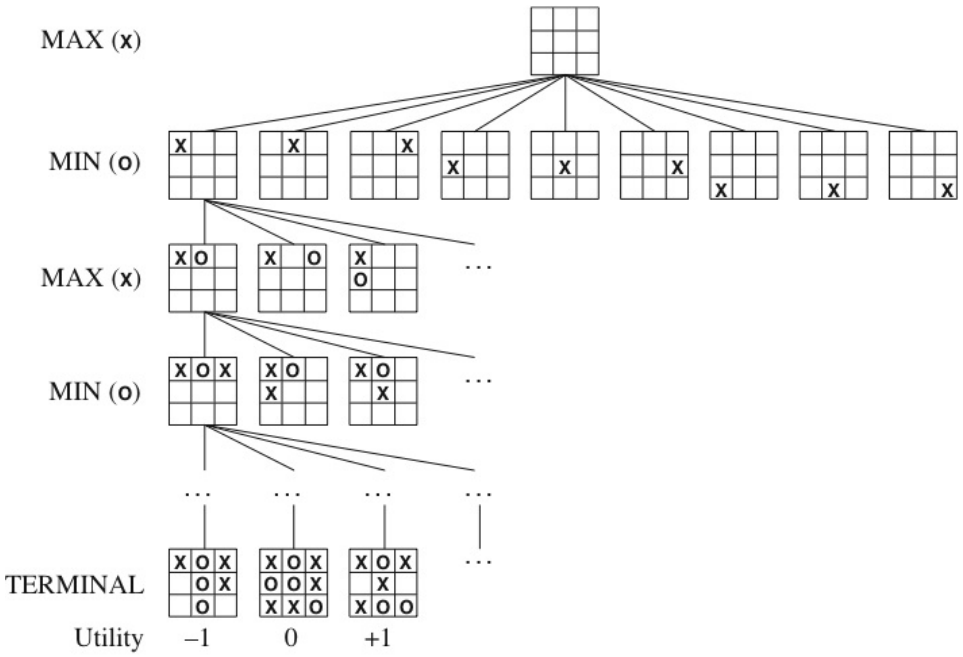
7.2 Spelbomen en het Minimax Algoritme

De initiële toestand s_o bepaalt, samen met het transitiemodel T en de eindtest een SPELBOOM voor het gegeven spel. De wortel van de spelboom is de initiële toestand. De kinderen van een top zijn de opvolgers van de toestand horend bij de top onder het gegeven transitiemodel. De blaadjes van de boom corresponderen met de eindtoestanden van het spel. De waarden die worden geschreven bij deze blaadjes corresponderen met de opbrengstfunctie vanuit het standpunt van de speler Max.

Voorbeeld 7.4 In Figuur 7.1 zie je een deel van de spelboom voor het spel tic tac toe (OXO). De speler Max speelt met X, en Min speelt met O. Om de beurt plaatsen de spelers hun symbool op een vrije plaats in het 3×3 rooster. De speler die als eerste drie identieke symbolen op een rij, kolom of diagonaal heeft, wint. Wanneer het bord vol is en geen van beide spelers is erin geslaagd om drie symbolen gealigneerd te krijgen dan is er gelijkspel.

In Figuur 7.1 zie je onderaan drie eindtoestanden. In het eerste geval wint de speler Min (die met O speelt). De opbrengstfunctie, vanuit het standpunt van Max, is dus negatief. In de tweede eindtoestand in Figuur 7.1 is het bord vol en zijn er geen drie symbolen gealigneerd. Er is gelijkspel en de opbrengst voor Max is nul. In de derde eindtoestand heeft Max drie gelijke symbolen op de tweede diagonaal en wint het spel. Dit wordt gereflecteerd in de positieve opbrengst van deze eindtoestand.

Voor het toch wel eenvoudige spel dat tic tac toe is, bevat de spelboom reeds 255 168 blaadjes. Dit is meteen ook het aantal mogelijke manieren om tic tac toe te spelen. Merk op dat niet al deze blaadjes een verschillende toestand



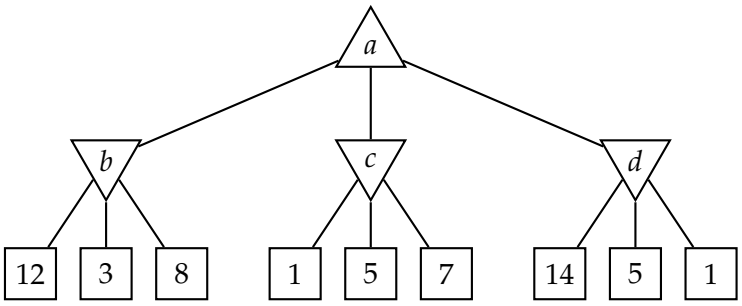
Figuur 7.1: Deel van de spelboom voor het spel tic tac toe. Bron: (Russell and Norvig, 2014)

voorstellen. Inderdaad, een eenvoudige bovengrens voor het aantal mogelijke toestanden van een tic tac toe bord is $3^9 = 19\,683$. ∎

In Figuur 7.2 zie je een voorbeeld van een spelboom voor een hypothetisch spel. Een toestand waarin speler Max aan zet is wordt steeds aangeduid met een naar boven gerichte driehoek, wanneer Min aan zet is dan wordt de driehoek naar onder gericht. Een eindtoestand wordt steeds aangeduid met een vierkant.

Opmerking 7.5 De spelboom is vooral een theoretisch concept, want voor realistische spellen is deze veel te groot om volledig op te bouwen, of zelfs maar alle toppen te evalueren zonder ze allemaal tegelijkertijd in het geheugen op te slaan. Het aantal mogelijke toestanden van een schaakbord bijvoorbeeld wordt geschat op 10^{43}. Vergelijk dit met de schatting voor het aantal atomen in het zichtbare universum, nl. 10^{80}. ∎

Speler Max is als eerste aan de beurt en wil uiteraard die actie selecteren die zijn opbrengst zo groot mogelijk zal maken. Wanneer het spel gedaan zou



Figuur 7.2: Een spelboom voor een hypothetisch spel.

zijn na één zet dan zou Max eenvoudigweg de waarde van de opbrengstfunctie in de eindtoestanden kunnen gebruiken. Echter, Min is ook nog in het spel en Max weet dat Min er alles aan gaat doen om de opbrengst voor Max te minimaliseren, terwijl Max ook weet dat Min weet dat Max de opbrengst voor zichzelf wil maximaliseren. We zien hier dus een *recursief* proces. De MINIMAX WAARDE van een toestand wordt m.a.w. als volgt bepaald:

$$\text{minimax}(s) = \begin{cases} U(s, \text{Max}) & \text{als } s \text{ een eindtoestand is} \\ \max\limits_{a \in A} \text{minimax}(T(s,a)) & \text{als Max aan zet is in toestand } s \\ \min\limits_{a \in A} \text{minimax}(T(s,a)) & \text{als Min aan zet is in toestand } s. \end{cases}$$

Opmerking 7.6 Uiteraard beschouwen we in de toestand s enkel die acties a die zinvol kunnen uitgevoerd worden op de toestand s. ∎

Eens de minimax waarde van een toestand is bepaald is het eenvoudig om de bijhorende actie voor Max te selecteren. Men kiest eenvoudigweg die actie (of één van de acties) waarvoor het maximum wordt bereikt. Die noemt men dan de MINIMAX BESLISSING.

Opmerking 7.7 (Optimaliteit minimax beslissing) De minimax beslissing is de beste beslissing wanneer er wordt gespeeld tegen een tegenstander die ook optimaal speelt. Een andere beslissing dan de minimax beslissing kan (en zal) door een optimaal spelende tegenstander uitgebuit worden om ervoor te zorgen dat zijn eigen opbrengst zal stijgen (of gelijk blijven).

In het geval de tegenstander niet optimaal speelt, dan is het wel mogelijk dat een andere beslissing dan de minimax beslissing een nog hogere opbrengst kan opleveren. ∎

In Algoritme 7.1 wordt code getoond die de minimax beslissing berekent. Dit gebeurt door de spelboom te doorlopen op een diepte-eerst manier. De recursie gaat eerst volledig door tot aan een blad van de boom. Voor zo'n blad (eindtoestand) is de minimax waarde onmiddellijk gekend. Het zijn deze waarden die via het gebruik van de max en min operatoren naar de wortel van de boom worden teruggebracht. Het minimax algoritme genereert alle toppen van de spelboom. Als we er opnieuw van uitgaan dat de maximale diepte van de boom gelijk is aan m en dat er b geldige zetten zijn in iedere toestand dan gebruikt het algoritme $\mathcal{O}(bm)$ aan geheugen (wanneer alle opvolgers tegelijkertijd worden berekend). Dit is een lineaire complexiteit en betekent dat het algoritme normaliter nooit in de problemen geraakt omwille van een tekort aan hoofdgeheugen. Echter, *alle toppen* van de spelboom worden gegenereerd. Het aantal toppen is van de orde $\mathcal{O}(b^m)$. We hebben dus te maken met een exponentiële tijdscomplexiteit. Dit betekent dat het voor de meeste spellen onmogelijk is om het minimax algoritme volledig uit te voeren.

Voorbeeld 7.8 (Uitwerking minimax algoritme) We bekijken opnieuw de spelboom in Figuur 7.2 en we berekenen de minimax beslissing aan de hand van het minimax algoritme zoals gegeven in Algoritme 7.1.

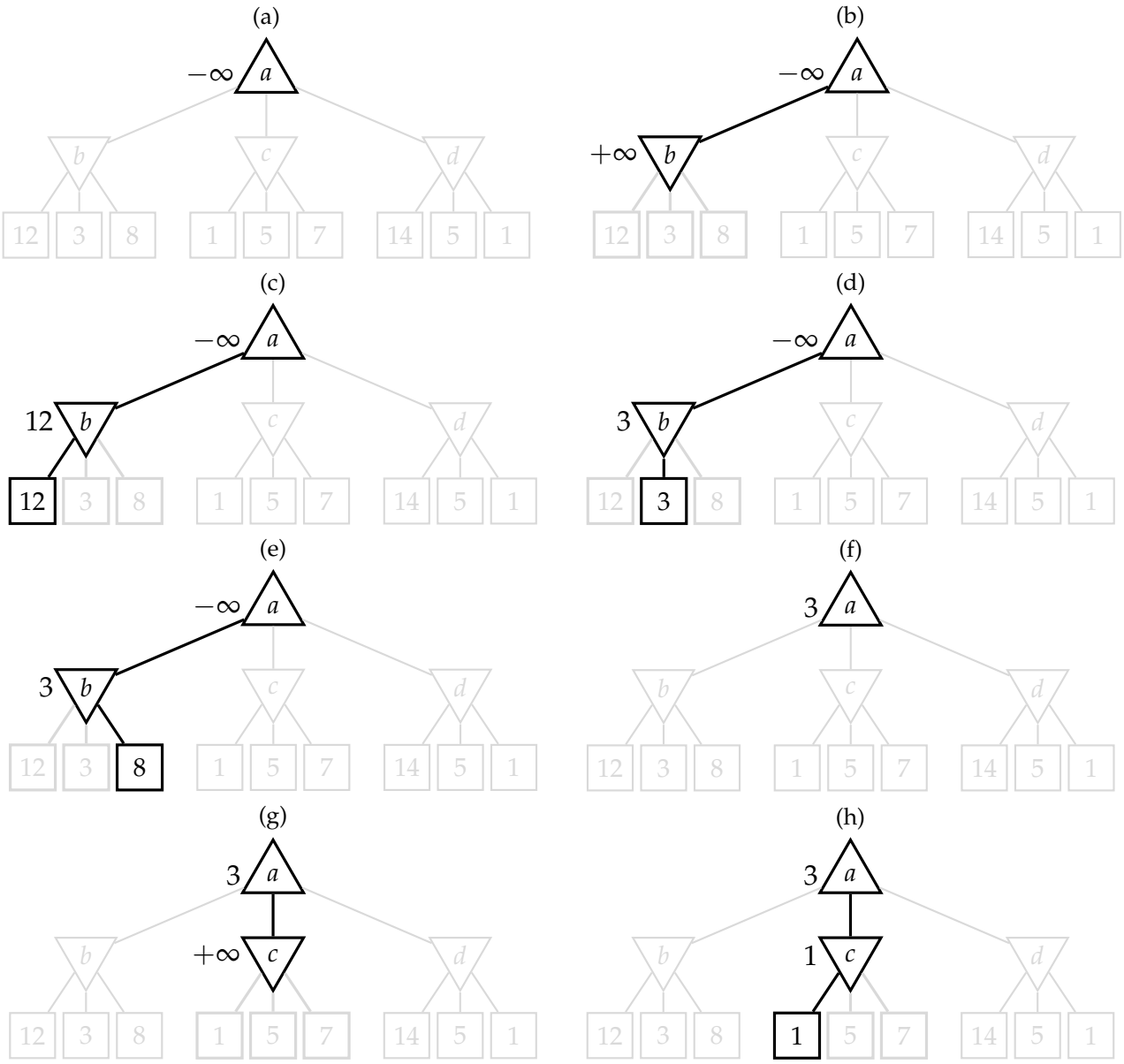
We starten in de initiële toestand a. Aangezien nog geen enkele mogelijke actie werd beschouwd, is de waarde van deze toestand (waarin speler Max aan zet is) voorlopig $-\infty$. Er wordt een eerste mogelijke actie beschouwd. In de resulterende toestand b is speler Min aan zet. De waarde van deze toestand is op dit moment $+\infty$, de slechtst mogelijke waarde voor speler Min. Speler Min probeert de eerste van zijn drie mogelijke acties en belandt in een eindtoestand met opbrengst 12. Aangezien 12 (vanuit het standpunt van Min) beter is dan $+\infty$ wordt de waarde in de toestand b geüpdatet naar 12. Vervolgens probeert Min de volgende actie; die leidt naar een eindtoestand met waarde 3. Opnieuw wordt de waarde in b geüpdatet, dit keer naar de waarde 3, (want 3 is beter dan 12 vanuit het standpunt van Min). De laatste actie die Min probeert leidt ook tot een eindtoestand maar aangezien de opbrengst van de eindtoestand, nl. 8, niet beter is voor Min dan de 3 die hij reeds kan bereiken wordt de waarde van de toestand b niet aan-

Algoritme 7.1 Berekenen van de minimax beslissing.

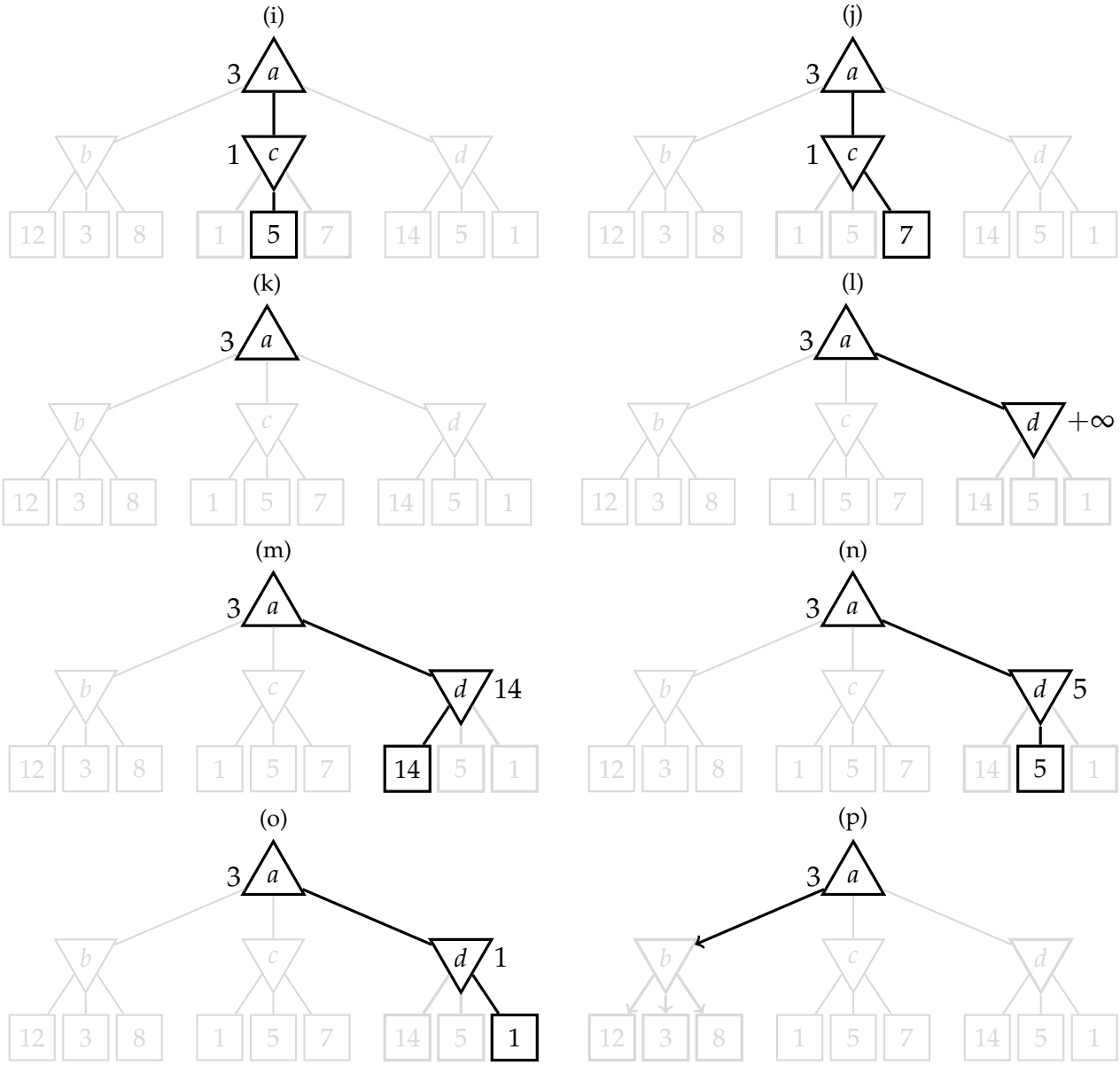
Invoer Een initiële toestand s_o

Uitvoer De actie overeenkomend met de minimax beslissing voor Max.

```
1:  function MINIMAXDECISION(s_o)
2:      chosenAction ← ∅
3:      maxVal ← −∞
4:      for (s, a) ∈ s_o.GETSUCCESSORS do
5:          v ← MINVALUE(s)
6:          if v > maxVal then                    ▷ betere actie gevonden
7:              chosenAction ← a
8:              maxVal ← v
9:          end if
10:     end for
11:     return chosenAction
12: end function
13: function MINVALUE(t)
14:     if t.TERMINALTEST = true then
15:         return t.UTILITY                      ▷ eindgeval van de recursie
16:     end if
17:     v ← +∞
18:     for (s, a) ∈ t.GETSUCCESSORS do
19:         v ← MIN(v, MAXVALUE(s))               ▷ Min zoekt de kleinste waarde
20:     end for
21:     return v
22: end function
23: function MAXVALUE(t)
24:     if t.TERMINALTEST = true then
25:         return t.UTILITY                      ▷ eindgeval van de recursie
26:     end if
27:     v ← −∞
28:     for (s, a) ∈ t.GETSUCCESSORS do
29:         v ← MAX(v, MINVALUE(s))               ▷ Max zoekt de grootste waarde
30:     end for
31:     return v
32: end function
```



Figuur 7.3: Stap voor stap uitwerking van het minimax algoritme. Het actieve pad is in het donker aangeduid. De delen van de spelboom die *niet* in het hoofdgeheugen zitten zijn grijs. De getallen naast de toppen zijn de (voorlopige) minimaxwaarden zoals berekend door het minimax algoritme.



Figuur 7.4: Vervolg uitwerking minimax algoritme.

gepast. Alle mogelijke acties in b zijn nu afgehandeld, de minimax waarde van de toestand b is gelijk aan 3.

De (voorlopige) minimax waarde voor de toestand a wordt nu aangepast naar 3, want 3 is (vanuit het standpunt van Max) beter dan de huidige waarde $-\infty$. Nu wordt in toestand a de volgende actie uitgevoerd. Deze leidt naar de toestand c. De minimax waarde van c wordt geïnitialiseerd op $+\infty$, want c is een toestand waarin speler Min aan zet is. De eerste actie in c wordt geprobeerd en deze leidt naar een eindtoestand met opbrengst 1 en de (voorlopige) minimax waarde van c wordt geüpdatet naar 1. De volgende actie wordt gesimuleerd. Deze leidt naar een eindtoestand met opbrengst 5. Dit is niet beter (vanuit het standpunt van Min) dan de huidige waarde 1, dus wordt onmiddellijk overgegaan naar de volgende actie. Deze leidt naar een eindtoestand met waarde 7. Er is opnieuw geen aanpassing van de (voorlopige) minimax waarde in toestand c. Op dit moment is de minimax waarde van de toestand c bepaald als 1, en deze wordt geretourneerd.

In toestand a gebeurt er nu *geen* aanpassing van de (voorlopige) minimax waarde, want 1 is (vanuit het standpunt van Max) niet beter dan de 3 die reeds bereikt kan worden door de eerste actie te ondernemen. Speler Max probeert tenslotte de laatste actie die kan uitgevoerd worden in de initiële toestand. Deze leidt naar de toestand d. Door de verschillende acties uit te proberen wordt de (voorlopige) minimax waarde van d achtereenvolgens aangepast van $+\infty$, naar 14, 5 en tenslotte 1. Deze minimax waarde wordt geretourneerd (naar de toestand a) maar de minimax waarde van a wordt niet aangepast aangezien Max reeds de waarde 3 kan bereiken worden door de eerste actie uit te voeren.

De minimaxwaarde van a is m.a.w. gelijk aan 3. De eerste actie wordt geretourneerd als de minimax beslissing. ∎

7.3 Snoeien van Spelbomen

Als we kijken naar de spelboom in Figuur 7.2 dan zien we dat *de waarde van sommige toppen irrelevant is* voor het eindresultaat.

We berekenen de minimax waarde van toestand a. We vinden:

$$\begin{aligned}\text{minimax}(a) &= \max\big(\text{minimax}(b), \text{minimax}(c), \text{minimax}(d)\big)\\ &= \max\big(\min(12,3,8), \min(1,5,7), \min(14,5,1)\big)\\ &= \max(3,1,1)\\ &= 3.\end{aligned}$$

Veronderstel nu dat de laatste twee opvolgers van toestand c onbepaalde waarden x en y hebben. In dit geval vinden we voor de minimax waarde van a:

$$\begin{aligned}\text{minimax}(a) &= \max\big(\text{minimax}(b), \text{minimax}(c), \text{minimax}(d)\big)\\ &= \max\big(\min(12,3,8), \min(1,x,y), \min(14,5,1)\big)\\ &= \max(3,z,1) && \text{met } z \leq 1\\ &= 3.\end{aligned}$$

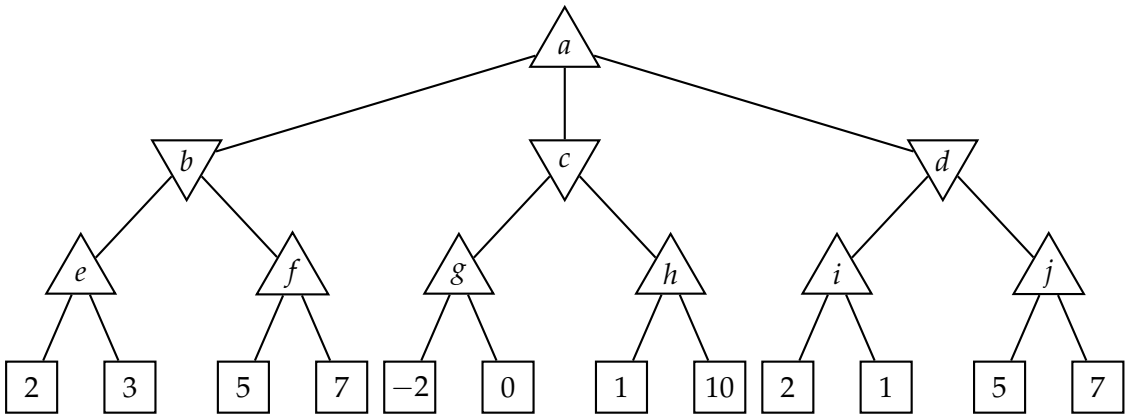
Wat x en y ook zijn, de minimaxwaarde van a blijft 3!

Merk op dat het niet uitmaakt dat de laatste twee opvolgers van c eindtoestanden zijn. De redenering gaat ook op als de waarden x en y het gevolg zijn van een spel met duizenden mogelijke zetten! In dit geval kan veel tijd bespaard worden door deze takken van de spelboom niet te evalueren.

Het niet evalueren van een tak in een spelboom (omdat die het eindresultaat toch niet kan beïnvloeden) wordt het SNOEIEN (Eng. *pruning*) van die tak genoemd.

Hoe kunnen we nu gemakkelijk zien welke takken kunnen gesnoeid worden? Veronderstel dat de toestand c in Figuur 7.2 geëvalueerd wordt (m.b.v. de methode MINVALUE). Veronderstel verder dat aan deze methode werd doorgegeven dat Max reeds de mogelijkheid heeft om een opbrengst van 3 te realiseren. Nadat het eerste kind van c werd geëvalueerd weet Min reeds dat in deze toestand de opbrengst voor Max *hoogstens 1* is. Min weet dus dat *een rationale (optimale) Max-speler het spel nooit in de toestand c zal laten belanden*. Het is dan ook zinloos om de resterende opvolgers van c te evalueren en deze kunnen gesnoeid worden.

We wensen nu het minimax algoritme aan te passen om te kunnen snoeien waar mogelijk. Het minimax algoritme is een diepte eerst algoritme dus op elk moment bekijken we één enkel pad in de spelboom. Het resulterende



Figuur 7.5: Een voorbeeldboom voor het toepassen van α-β-snoeien.

algoritme word α-β-snoeien (Eng. *α-β-pruning*) genoemd naar de twee extra parameters die worden bijgehouden t.o.v. het minimax algoritme:

Definitie 7.9 De parameter α houdt de waarde bij van de beste keuze (i.e. de hoogste waarde) *op het huidig pad* voor Max. De parameter β houdt de waarde bij van de beste keuze (i.e. de laagste waarde) *op het huidig pad* voor Min. ∎

Max wijzigt de α-waarden en Min de β-waarden. Op elk moment heeft de top in de spelboom een huidige waarde v; deze waarde stijgt voor Max en daalt voor Min. Er kan gesnoeid worden als één van volgende voorwaarden voldaan is:

- Min merkt dat de huidige waarde v kleiner (of gelijk) is aan α. Een rationele Max zal immers het spel nooit hier laten komen aangezien hij op het huidig pad reeds een betere keuze heeft.
- Max merkt dat de huidige waarde v groter (of gelijk) is aan β. Een rationele Min zal immers het spel nooit hier laten komen aangezien hij op het huidig pad reeds een betere keuze heeft.

Voorbeeld 7.10 (Uitwerking α-β-snoeien) We passen α-β-snoeien toe op de spelboom in Figuur 7.5.

Het algoritme start en stuurt $\alpha = -\infty$ en $\beta = +\infty$ mee langs de tak van a naar b. Dezelfde waarden worden ook meegestuurd langs de tak van b naar

Algoritme 7.2 Berekenen van de minimax beslissing m.b.v. α-β-snoeien.

Invoer Een initiële toestand s_o
Uitvoer De actie overeenkomend met de minimax beslissing voor Max.

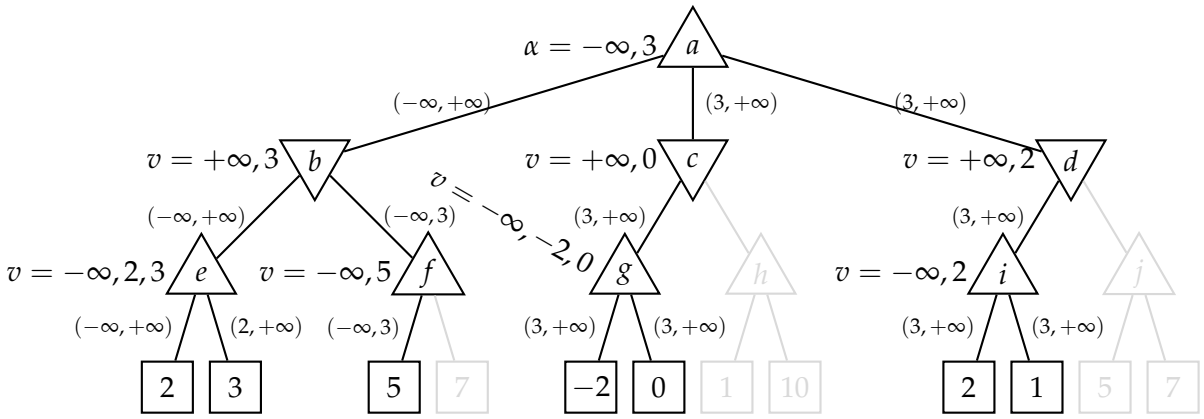
```
 1: function ALPHABETADECISION(s_o)
 2:     chosenAction ← ∅
 3:     α ← −∞
 4:     for (s, a) ∈ s_o.GETSUCCESSORS do
 5:         v ← MINVALUE(s, α, +∞)                          ▷ β = +∞
 6:         if v > α then                                   ▷ betere actie gevonden
 7:             chosenAction ← a
 8:             α ← v
 9:         end if
10:     end for
11:     return chosenAction
12: end function
13: function MINVALUE(t, α, β)
14:     if t.TERMINALTEST = true then
15:         return t.UTILITY                                ▷ eindgeval van de recursie
16:     end if
17:     v ← +∞
18:     for (s, a) ∈ t.GETSUCCESSORS do
19:         v ← MIN(v, MAXVALUE(s, α, β))
20:         if v ≤ α then
21:             return v                                    ▷ snoeien
22:         end if
23:         β ← MIN(v, β)                                   ▷ aanpassen β
24:     end for
25:     return v
26: end function
27: function MAXVALUE(t, α, β)
28:     if t.TERMINALTEST = true then
29:         return t.UTILITY                                ▷ eindgeval van de recursie
30:     end if
31:     v ← −∞
32:     for (s, a) ∈ t.GETSUCCESSORS do
33:         v ← MAX(v, MINVALUE(s, α, β))
34:         if v ≥ β then
35:             return v                                    ▷ snoeien
36:         end if
37:         α ← MAX(v, α)                                   ▷ aanpassen α
38:     end for
39:     return v
40: end function
```

e. Wanneer de opbrengstfunctie van de eerste eindtoestand wordt teruggegeven wordt in *e* de waarde van v en α aangepast naar 2. De volgende actie wordt getest: bij deze oproep heeft α de waarde 2, en β is nog steeds $+\infty$. In *e* worden v en α aangepast naar de waarde 3 (de waarde geretourneerd door de tweede eindtoestand). Dit is meteen ook de waarde die wordt geretourneerd aan *b*.

In *b* (waar speler Min aan zet is) worden v en β aangepast naar 3. Bij de oproep van *b* naar *f* nemen α en β respectievelijk de waarden $-\infty$ en 3 aan. In toestand *f* wordt v geïnitaliseerd op $-\infty$, en de eerste actie wordt geprobeerd (met nog steeds de waarden $-\infty$ en 3 voor α en β bij deze oproep). De eindtoestand rapporteert 5 als waarde en v wordt aangepast naar 5. Op dit moment wordt v (met waarde 5) vergeleken met β (met waarde 3). Omdat $v \geq \beta$ weet Max dat een rationele speler Min het spel *nooit* in deze toestand zal laten komen: Min heeft immers op het huidige pad een keuze die voor hem de waarde drie oplevert i.p.v. de vijf (of meer!) die hier zal bekomen worden. De tweede actie in *f* wordt niet geprobeerd: er wordt gesnoeid en de waarde 5 wordt geretourneerd naar de toestand *b*.

In *b* blijft de waarde v ongewijzigd op 3. Aangezien alle acties in *b* geprobeerd zijn wordt 3 teruggegeven naar de toestand *a*. Hier wordt α aangepast naar 3. Bij de oproep op de tak van *a* naar *c* hebben α en β dus respectievelijk de waarden 3 en $+\infty$. Dezelfde waarden worden doorgegeven langs de tak van *c* naar *g* en van *g* naar de vijfde eindtoestand met waarde -2. Deze rapporteert -2 als waarde en in *g* wordt v aangepast naar -2 (i.p.v. de initiële $-\infty$). Aangezien -2 niet groter is dan β kan er niet gesnoeid worden. Ook de waarde van α wordt *niet* aangepast. Bij de oproep van *g* naar de zesde eindtoestand worden dezelfde waarden voor α en β meegegeven; daarna wordt v aangepast naar 0 en wordt de waarde 0 teruggegeven naar toestand *c*. De waarde v in toestand *c* wordt aangepast naar 0. Op dit moment weet de speler Min dat hij in deze toestand een waarde van 0 *of minder* kan bekomen. Maar, aangezien α gelijk is aan 3 weet hij ook dat de speler Max ergens op het pad naar deze toestand de mogelijkheid heeft om de waarde 3 te bekomen. De volgende acties in toestand *c* moeten niet bekeken worden. In het algoritme zien we dat de conditie $v \leq \alpha$ evalueert naar **true**. De waarde 0 wordt geretourneerd naar de toestand *a*.

In toestand *a* wordt α niet gewijzigd. De volgende actie wordt geprobeerd. Op de tak van *a* naar *d* nemen α en β respectievelijk de waarden 3 en $+\infty$



Figuur 7.6: Geannoteerde boom voor α-β-snoeien. De takken in het grijs zijn gesnoeid. Bij elke niet-gesnoeide tak staan de waarden (α, β) zoals ze werden doorgegeven bij die methodeoproep. De sequenties $v = \cdots$ geven aan hoe de waarde van v wijzigt in elke top.

aan. Dezelfde waarden worden doorgegeven van d naar i en van i naar de negende eindtoestand. Daarna krijgt v in toestand i de waarde 2 (zonder aanpassing van α) en ook de tiende eindtoestand wijzigt deze waarde niet. De waarde 2 wordt dus geretourneerd van i naar d. In d krijgt v nu de waarde 2; deze waarde wordt vergeleken met de huidige waarde van α. Aangezien er inderdaad geldt dat $v \leq \alpha$ kan de volgende tak gesnoeid worden.

De waarde 2 wordt teruggegeven naar a waar beslist wordt dat α ongewijzigd blijft en dat de beste actie de eerste actie is. ∎

Het is duidelijk dat de volgorde waarin de opvolgers bekeken en uitgewerkt worden een zeer grote invloed kan hebben op de effectiviteit van het snoeialgoritme. Zo is er bv. in toestand d van de spelboom in Figuur 7.2 geen snoeien mogelijk omdat de beste opvolger (vanuit het standpunt voor Min) als laatste bekeken wordt. Indien deze als eerste bekeken werd, dan zouden opnieuw de andere twee opvolgers kunnen gesnoeid worden.

Het komt er dus op aan om op een snelle manier een goede ordening van de verschillende zetten te vinden. Voor een spel als schaken bekijkt men bv. eerst "slagen", dan bedreigingen, dan voorwaartse zetten en tenslotte achterwaartse zetten. Op die manier komt men ongeveer binnen een factor twee van het best mogelijke theoretische resultaat.

7.4 Praktische Uitwerking

In veel spelbomen komt dezelfde toestand meerdere malen voor. Dit leidt natuurlijk tot (veel) extra werk. Een oplossing bestaat erin om de toestanden (en hun minimax waarde) bij te houden in een hashtabel, analoog aan de gesloten lijst bij graafgebaseerd zoeken. In de context van spelbomen wordt deze hashtabel de TRANSPOSITIETABEL genoemd. Het bijhouden van een transpositietabel kan een grote tijdswinst betekenen. Als het aantal toestanden te groot is, dan is het natuurlijk onmogelijk om *alle* toestanden bij te houden. Er bestaan allerhande strategieën om te beslissen welke toestanden bijgehouden worden.

Voor praktische spellen, zoals schaken, dammen of Go, is het natuurlijk onrealistisch om het minimax algoritme of zelfs α-β-snoeien te gebruiken. Het antwoord van de computer kan immers voor zulke spellen niet binnen een redelijke termijn berekend worden. Ook α-β-snoeien moet immers de paden die niet kunnen gesnoeid worden uitwerken totdat een eindtoestand wordt bereikt. Voor schaken bv. doet elke speler gemiddeld 40 zetten alvorens het spel eindigt. Zelfs met een vertakkingsfactor van 6 i.p.v. 35 is het totaal onpraktisch om de boom volledig uit te werken.

In de praktijk stelt men een limiet in op de diepte (het aantal halve zetten) van de boom. Wanneer die dieptelimiet wordt bereikt voordat een eindtoestand wordt bereikt dan gebruikt men een HEURISTISCHE EVALUATIEFUNCTIE om de waarde van die toestand te benaderen.

We krijgen dan de volgende (recursieve) formule om een heuristische minimax waarde te berekenen voor maximale diepte d:

$$\text{h-minimax}(s,d) = \begin{cases} \text{EVAL}(s) & \text{als } d = 0 \text{ of } s \text{ een eindtoestand is} \\ \max\limits_{a \in A} \text{h-minimax}(T(s,a), d-1) & \text{als Max aan zet is in toestand } s \\ \min\limits_{a \in A} \text{h-minimax}(T(s,a), d-1) & \text{als Min aan zet is in toestand } s. \end{cases}$$

Het is duidelijk dat de kwaliteit van de heuristische evaluatiefunctie een grote invloed heeft op de performantie van het algoritme. Wanneer een heuristische evaluatiefunctie verkeerdelijk een hoge waarde toekent aan slechte posities, dan zal het algoritme in de richting van de slechte posities gestuurd worden. Dit is uiteraard niet de bedoeling.

Aan welke eigenschappen moet een goede heuristische evaluatiefunctie voldoen? Ten eerste moet ze de eindtoestanden op dezelfde manier ordenen als de opbrengstfunctie U: toestanden met winst moeten als beter geëvalueerd worden dan toestanden met gelijkspel en die moeten op hun beurt beter geëvalueerd worden dan toestanden waarin er verloren wordt. Ten tweede moet de evaluatiefunctie snel kunnen berekend worden. De bedoeling is immers om tijd te winnen. Tenslotte moet de waarde van de heuristische evaluatiefunctie voor niet-eindtoestanden sterk gecorreleerd zijn met de kans om effectief te winnen vanuit die toestand.

Voorbeeld 7.11 (Voorbeeld heuristische evaluatiefunctie tic tac toe) Voor het spel tic tac toe definiëren we de volgende features:

$$\begin{aligned} X_n &= \text{het aantal rijen, kolommen of diagonalen met} \\ &\qquad n\ X\text{-en en geen enkele } O \\ O_n &= \text{het aantal rijen, kolommen of diagonalen met} \\ &\qquad n\ O\text{'s en geen enkele } X. \end{aligned}$$

Het is intuïtief duidelijk dat een hoge waarde van X_2 goed is voor de speler Max (die met X speelt), en dat rijen met 2 X-en meer waard zijn dan rijen met slechts één X. Evengoed is duidelijk dat een grote waarde van O_2 (niet-geblokkeerde rijen met twee keer O erop) nadelig is voor speler Max, méér nadelig dan een grote waarde voor O_1.

We kunnen m.a.w. de volgende formule gebruiken als evaluatiefunctie voor een toestand s:

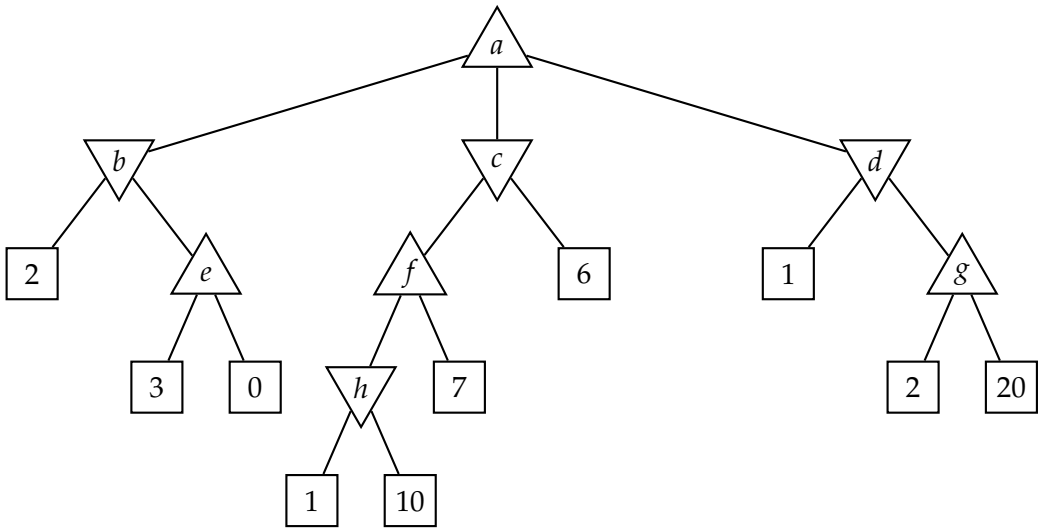
$$\text{EVAL}(s) = 3X_2 + X_1 - 3O_2 - O_1. \tag{7.1}$$

Gunstige features krijgen een positieve coëfficiënt, hoe gunstiger hoe groter de coëfficiënt. Nadelige features krijgen een negatieve coëfficiënt, hoe nadeliger de feature hoe groter de absolute waarde van de bijhorende coëfficiënt. ∎

We zien uit het voorbeeld dat een toestand geëvalueerd wordt als een gewogen lineaire combinatie van features:

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s).$$

Wanneer het niet duidelijk is wat "juiste" waarden voor de gewichten w_i zijn, dan kunnen technieken voor machinaal leren gebruikt worden om deze te bepalen. Daar gaan we hier verder niet op in.



Figuur 7.7: Spelboom voor de oefeningen.

7.5 Oefeningen

1. Gegeven de spelboom in Figuur 7.7 voor een twee-speler nulsomspel waarbij Max als eerste aan de beurt is. Beide spelers zijn rationaal. In elk blad van de boom (i.e. voor elke eindtoestand) staat de waarde van de opbrengstfunctie (vanuit het oogpunt van Max) genoteerd.

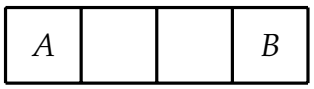
   a) Geef (door aanduiding op de figuur) voor elke toestand de minimax-waarde van die toestand.

   b) Welke actie zal Max ondernemen volgens het minimax algoritme?

   c) In welke toestand zal het spel eindigen? Omcirkel deze op de figuur.

   d) Wanneer α-β-snoeien wordt toegepast, dan kunnen er eventueel bepaalde delen van de boom gesnoeid worden. Duid deze aan op de figuur door een kruis te plaatsen op de takken die kunnen gesnoeid worden.

2. (Russell and Norvig, 2014, Oef 5.8) Beschouw volgend eenvoudig spel. De begintoestand is gegeven in Figuur 7.8. Speler A, die start in vakje één, speelt de rol van Max, B, die start in vakje vier, van Min. Bij elke zet moet een speler zijn pion verplaatsen naar aan aangrenzend vakje



Figuur 7.8: Begintoestand van het spel beschreven in Vraag 2.

(in beide richtingen). Indien het aangrenzend vakje ingenomen wordt door de pion van de tegenstander dan mag er over deze pion worden gesprongen. De speler die eerst aan de overkant van het bord geraakt wint. Dus als A als eerste vakje vier bereikt dan is de waarde van het spel voor A gelijk aan $+1$. Indien B als eerste vakje één bereikt dan is de waarde van het spel voor A gelijk aan -1.

a) Teken de volledige spelboom gebruikmakend van de volgende conventies:
   - Schrijf elke toestand als (s_A, s_B) waarbij s_A en s_B de locatie van de pion van respectievelijk A en B aanduiden. De begintoestand wordt bv. genoteerd als $(1, 4)$.
   - Trek rond elke eindtoestand een vierkant en schrijf de waarde (vanuit het standpunt van Max) naast dit vierkant.
   - Trek een dubbel vierkant rond *lusstaten*, i.e. rond staten die reeds voorkomen op het pad naar de wortel van de spelboom. Aangezien hun waarde voorlopig onbepaald is noteer je dit met een "?".

b) Schrijf bij elke knoop zijn berekende minimax waarde. Leg uit hoe je bent omgegaan met de "?".

c) Leg uit waarom het standaard minimax algoritme zou falen voor deze spelboom. Schets hoe je dit probleem zou aanpakken. Baseer je hiervoor op je antwoord in puntje (b) van deze vraag. Is je oplossing algemeen geldig?

3. Werk de spelboom voor tic tac toe uit tot op diepte twee, maar houd rekening met de symmetrieën van het bord om de spelboom beheersbaar te houden. Voor de eerste zet zijn er dan geen 9 maar slechts drie mogelijke zetten, nl. in een hoek, aan een rand en centraal. Beantwoord vervolgens de volgende vragen.

   a) Geef voor elke toestand op diepte twee de waarde van de evaluatiefunctie EVAL zoals gedefinieerd in (7.1).

b) Geef op basis van deze evaluatiefunctie de h-minimax waarde en de h-minimax beslissing.

c) Pas α-β snoeien toe. Welke takken kunnen er gesnoeid worden? Neem voor deze vraag aan dat het evalueren van de toppen in de *optimale* volgorde gebeurt zodanig dat er maximaal kan gesnoeid worden.

4. Veronderstel dat, voor de spelboom in Figuur 7.2, de speler Max *weet* dat Min "lui" is en steeds de eerste actie kiest. Wat is dan de beste actie voor Max? Is deze gelijk aan de minimax beslissing?

5. Schrijf, in pseudocode, het hoofdprogramma voor een agent die een bepaald spel gaat spelen tegen een menselijke speler. De computer speelt eerst. Je mag gebruikmaken van de functie MINIMAXDECISION. Je mag verder veronderstellen dat *s*.SHOWSTATE de toestand van het spel op het scherm toont (zodat de menselijke speler kan zien wat de toestand is). Met GETANDPARSEACTION lees je de actie die de menselijke speler wil doen in van het toetsenbord en zet je deze om naar een "actie" a. Met s.DOACTION(a) voer je actie a uit op de toestand s.

HOOFDSTUK 8

Machinaal Leren

In dit hoofdstuk starten we met een definitie van MACHINAAL LEREN om dan direct over te gaan op de drie types van machinaal leren, nl. GESUPERVISEERD LEREN, ONGESUPERVISEERD LEREN en REINFORCEMENT LEARNING. Voor de eerste twee types worden nog een aantal deeltypes besproken, zoals REGRESSIE, CLASSIFICATIE en CLUSTERING. In de laatste sectie bestuderen we de implementatie van het *K*-GEMIDDELDEN ALGORITME, een algoritme voor clustering. We bekijken eveneens hoe dit algoritme in de praktijk kan worden gebruikt.

8.1 Inleiding

Machineel leren is een belangrijk deelveld van artificiële intelligentie en wordt gebruikt in zeer veel toepassingen die door miljoenen mensen dagelijks gebruikt worden, zoals bv. spam filters voor email, gezichts*detectie* door de camera in je smartphone en gezichts*herkenning* in fotoapplicaties of op sociale netwerksites. Wanneer men een brief verzendt[1] dan zijn er machines in postsorteercentra die automatisch het (handgeschreven) adres lezen a.d.h.v. karakterherkenningssoftware.

Machinaal leren wordt ook gebruikt bij het verwerken en herkennen van natuurlijke taal, zoals dit bv. gebruikt wordt door de "assistenten" in moderne smartphones. Ook wanneer men Amazon of Netflix gebruikt krijgt men te maken met machinaal leren: deze systemen zijn, gebaseerd op je gedrag en

[1] Ja, dat gebeurt soms nog, of misschien is het wel dat pakketje dat je hebt besteld op het Internet dat wordt verstuurd.

gegeven beoordelingen, zeer goed in het voorstellen van andere producten of films waarin je waarschijnlijk geïnteresseerd bent.

Definitie 8.1 MACHINAAL LEREN is het deelveld van artificiële intelligentie dat computers de mogelijkheid geeft om te leren zonder hiervoor *expliciet* geprogrammeerd te zijn. ∎

8.2 Drie Types van Machinaal Leren

Algemeen worden er drie grote types van machinaal leren beschouwd. We definiëren deze nu kort één voor één. Later bekijken we enkele algoritmes die kunnen gebruikt worden om deze types effectief te implementeren.

8.2.1 Gesuperviseerd Leren

We starten met enkele voorbeelden van gesuperviseerd leren.

Voorbeeld 8.2 (Huisprijs voorspellen) Veronderstel dat we trachten te voorspellen wat de verkoopswaarde van een huis is op basis van de bewoonbare oppervlakte van dit huis. Om dit te realiseren beschikken we over een (groot) aantal voorbeelden van koppels

$$(\text{bewoonbare oppervlakte}, \text{verkoopswaarde}).$$

De taak bestaat er in dit geval in om een reëel getal, de verkoopswaarde, te voorspellen wanneer een bepaalde bewoonbare oppervlakte werd gegeven.

In een meer realistische opgave zullen we van elk huis meerdere eigenschappen (attributen) krijgen en gebruiken. Voorbeelden van attributen zijn: het aantal slaapkamers, het aantal badkamers, het huistype, het bouwjaar, enzovoort. De taak bestaat er dan nog steeds in om, op basis van de beschikbare voorbeelden, voor een nieuw huis waarvan de eigenschappen gegeven zijn te voorspellen wat zijn verkoopswaarde is. ∎

Voorbeeld 8.3 (Spamdetectie) Bij spamdetectie bestaat de opdracht erin om, op basis van een (groot) aantal emails waarvan geweten is of ze spam zijn of niet, te voorspellen of een nieuwe email al dan niet spam is.

Typisch gaat men eerst de tekst van de email gaan bewerken door woordnormalisatie (alles in kleine letters, woorden in enkelvoud) uit te voeren. Daarna gaat men alle stopwoorden (zoals 'and', 'the', etc.) die praktisch in

alle emails voorkomen en dus geen enkele bijdrage leveren tot de inhoud van de email gaan verwijderen. Tenslotte wordt de tekst van de email voorgesteld als een *bag of words*, waarbij men voor elk woord gaat bijhouden hoe vaak het voorkomt in de email. Een email wordt m.a.w. voorgesteld aan de hand van een attribuutvector. Het zijn deze attribuutvectoren die gebruikt worden door het leeralgoritme om een hypothese op te stellen. ∎

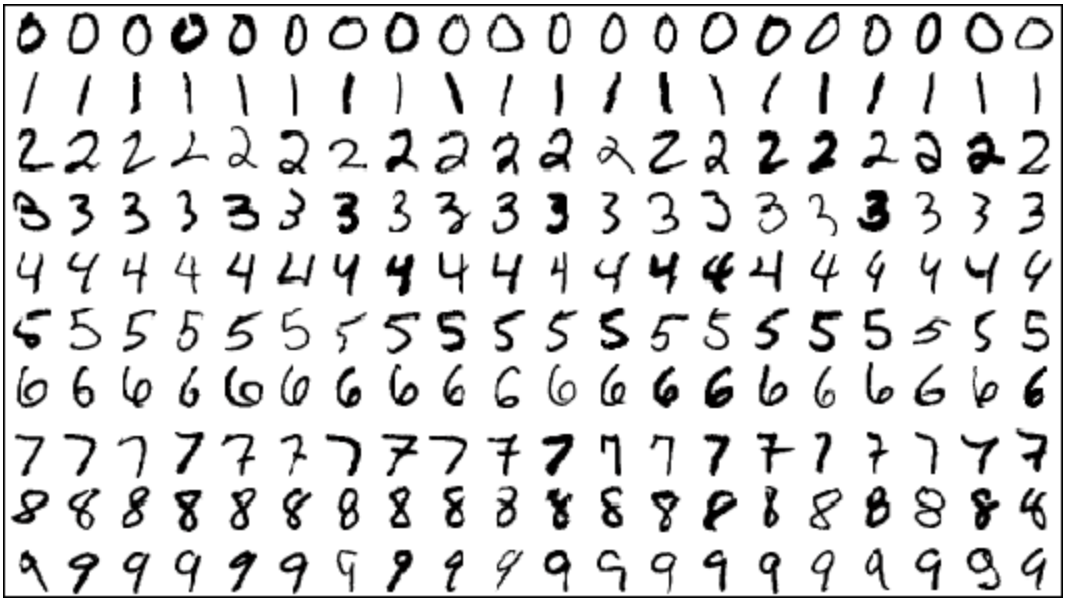
Voorbeeld 8.4 (Cijferherkenning) Veronderstel dat men een (groot) aantal afbeeldingen (bitmaps) heeft waarvan gegeven is of ze respectievelijk een 0, 1, 2, ... of een 9 voorstellen. De taak van een cijferherkenningsalgoritme bestaat er dan in om voor een nieuwe afbeelding (van een cijfer) te beslissen welk cijfer deze afbeelding voorstelt.

In Figuur 8.1 zie je in totaal 200 voorbeelden van cijfers. Elk voorbeeld is een 28×28 bitmap van grijswaarden en bestaat dus uit 784 getallen tussen 0 en 255. In de trainingsdataset is elk van de voorbeelden gelabeld met het cijfer dat dit voorbeeld voorstelt. ∎

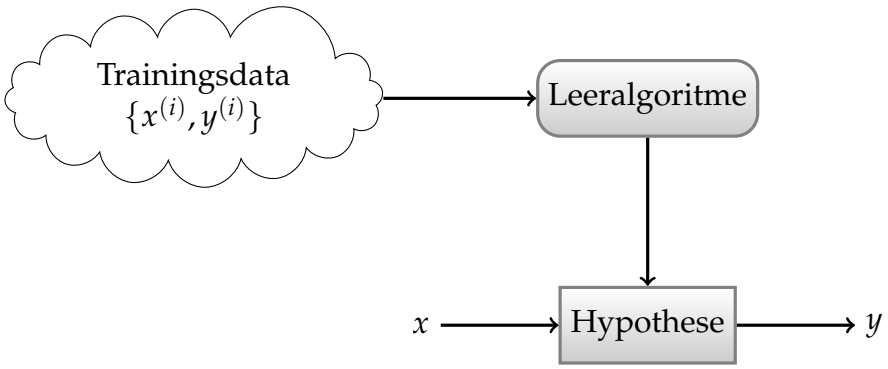
Definitie 8.5 De taak van GESUPERVISEERD LEREN bestaat erin om op basis van een *gelabelde trainingsdataset* een *hypothese* op te bouwen waarmee, voor een (nieuwe) invoer het label kan voorspeld worden. Wanneer het label een (reëel) getal is, dan spreekt men van een REGRESSIEPROBLEEM; wanneer het label één van een (klein) aantal voorgedefinieerde klassen is dan spreekt men van een CLASSIFICATIEPROBLEEM. ∎

Voorbeeld 8.6 Het voorspellen van de verkoopswaarde is een regressieprobleem; spamdetectie en cijferherkenning zijn voorbeelden van classificatieproblemen. Omdat er bij spamdetectie slechts twee klassen zijn spreekt men ook van een BINAIR classificatieprobleem. ∎

In Figuur 8.2 ziet men een overzicht van de werking van gesuperviseerd leren. We geven een groot aantal voorbeelden $(x^{(i)}, y^{(i)})$ met $i \in \{1, 2, \ldots, m\}$ aan het leeralgoritme, waarbij $y^{(i)}$ de "juiste" uitvoer is bij de attribuutvector $x^{(i)}$. Het leeralgoritme heeft als uitvoer een *functie* h die de HYPOTHESE genoemd wordt. Wanneer men aan deze functie h een vector x als invoer geeft dan antwoordt deze met het voorspelde label y dat bij de gegeven x hoort.



Figuur 8.1: Enkele voorbeelden van cijfers. In elke rij staan er telkens 20 voorbeelden voor een 0, 1,2, enzovoort. Elk voorbeeld kan voorgesteld worden door een attribuutvector van lengte 28×28. Elk attribuut is een grijswaarde, i.e. een getal tussen 0 en 255. Bron: http://myselph.de/neuralNet.html.



Figuur 8.2: Overzicht van de werking van een algoritme voor gesuperviseerd leren. In de eerste fase gebruikt men het leeralgoritme om een hypothese op te bouwen aan de hand van de trainingsdata, i.e. aan de hand van de gelabelde voorbeelden. Vervolgens kan men deze hypothese gaan gebruiken om voor een (ongeziene) attribuutvector x te gaan voorspellen wat het bijhorende label y is.

8.2.2 Ongesuperviseerd Leren

We starten met enkele voorbeelden van ongesuperviseerd leren.

Voorbeeld 8.7 (Marktsegmentatie) Gegeven een ongelabelde dataset met attributen van klanten. Je wil uitvissen of er verschillende *soorten* of *groepen* van klanten zijn, zodat je deze groepen op verschillende manieren kan gaan benaderen bv. door het ontwikkelen van een specifiek aanbod voor de verschillende groepen.

De taak van het algoritme bestaat erin om de verschillende coherente groepen aan te duiden. De *interpretatie* van de betekenis van deze groepen ligt wel nog steeds bij de mens. Zo zou een telecomoperator kunnen interpreteren dat de drie gevonden groepen betekenen: "honkvaste" klanten, die hoogstwaarschijnlijk gewoon hun contract verlengen, "ontevreden" klanten die zo goed als zeker hun contract opzeggen en "twijfelaars" die mits de juiste aanbieding hun contract zullen verlengen. ∎

Voorbeeld 8.8 (Nieuwsaggregatie) Nieuwsaggregatie websites tonen artikels van verschillende bronnen *gegroepeerd volgens onderwerp*. Een van de onderliggende algoritmen bestaat er dus in om, gegeven de verschillende nieuwsartikels op het internet, te beslissen welke artikels over hetzelfde onderwerp handelen. Gelijkaardige artikels worden m.a.w. gegroepeerd. ∎

Voorbeeld 8.9 (Fraudedetectie) Hier hebben we een dataset waarbij $x^{(i)}$ bestaat uit attributen die het gedrag van gebruiker i beschrijven (bv. op een website). In fraudedetectie wenst men uit te vissen welke gebruikers er *ongewoon gedrag* vertonen. ∎

Voorbeeld 8.10 (Dimensiereductie) Veronderstel dat men een dataset heeft waarbij elk (ongelabeld) voorbeeld voorgesteld wordt door honderden of zelfs duizenden attributen. Vaak is het zo dat sommige van deze attributen een hoge mate van correlatie vertonen, bv. wanneer twee attributen een meting voorstellen van dezelfde grootheid maar in een andere schaal. Het kan dan interessant zijn om over te gaan om een nieuwe, veel kleinere, verzameling van attributen zodanig dat de voorbeelden zo goed mogelijk "gelijken" op de originele voorbeelden. ∎

Definitie 8.11 De taak van een algoritme voor ONGESUPERVISEERD LEREN bestaat erin om *structuur te ontdekken* in een *ongelabelde* dataset. De meest

voorkomende taak in ongesuperviseerd leren is CLUSTERING, het ontdekken van coherente groepen. Andere taken zijn ANOMALIEDETECTIE en PRIMAIRE COMPONENTEN ANALYSE. ∎

Voorbeeld 8.12 Marktsegmentatie en nieuwsaggregatie zijn voorbeelden van clustering. Fraudedectectie is een voorbeeld van anomaliedetectie terwijl dimensiereductie kan gerealiseerd worden m.b.v. primaire componenten analyse. ∎

8.2.3 Reinforcement Learning

Reinforcement learning is verschillend van gesuperviseerd en ongesuperviseerd leren in die zin dat er hier niet wordt gewerkt met datasets. In de plaats hiervan leert de agent van een reeks van "beloningssignalen", die negatief zijn wanneer de agent een "slechte" handeling stelt en positief wanneer de agent een goede handeling stelt. Eenvoudig gezegd bestaat de taak van reinforcement learning erin om te leren welke acties, i.e. welk *beleid*, leiden tot de hoogste totale beloning.

8.3 Clustering en het K-Gemiddelden Algoritme

Bij *ongesuperviseerd leren* is de invoer een *ongelabelde* dataset

$$\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\},$$

met elke $x^{(i)} \in \mathbb{R}^n$. We zien dat de dataset ongelabeld is door het ontbreken van de labels y. De taak van het algoritme bestaat erin om *structuur* te ontdekken in deze ongelabelde dataset.

De meest voorkomende vorm van het ontdekken van structuur is *clustering*, het vinden van *coherente groepen*.

Het K-GEMIDDELDEN ALGORITME (Eng. *K-means algorithm*) is een heel vaak gebruikt algoritme voor het vinden van clusters. Uiteraard zijn er nog veel andere algoritmen om clustering uit te voeren maar deze worden niet besproken in deze tekst.

De invoer voor het algoritme bestaat uit de ongelabelde dataset en het aantal clusters K dat men wenst te ontdekken in de dataset. Elke cluster wordt voorgesteld a.d.h.v. zijn ZWAARTEPUNT (Eng. *centroid*). Een element $x^{(i)}$ van

de dataset wordt toegewezen aan de cluster waarvan het zwaartepunt het dichtst bij $x^{(i)}$ ligt. De afstand tussen twee punten x en μ in $\mathbb{R}^n$ wordt voor de eenvoud berekend a.d.h.v. de Euclidische afstand:

$$\begin{aligned}\text{afstand tussen } x \text{ en } \mu &= \|x - \mu\| \\ &= \sqrt{(x_1 - \mu_1)^2 + (x_2 - \mu_2)^2 + \cdots + (x_n - \mu_n)^2}.\end{aligned}$$

Opmerking 8.13 (Berekening zwaartepunt) De berekening van het zwaartepunt is een eenvoudige veralgemening van het "midden" van twee punten. Wanneer men twee punten $x^{(1)}$ en $x^{(2)}$ heeft dan berekent men het midden μ als

$$\mu = \frac{1}{2}(x^{(1)} + x^{(2)}),$$

of explicieter

$$\begin{pmatrix}\mu_1 \\ \mu_2 \\ \vdots \\ \mu_n\end{pmatrix} = \frac{1}{2}\left(\begin{pmatrix}x_1^{(1)} \\ x_2^{(1)} \\ \vdots \\ x_n^{(1)}\end{pmatrix} + \begin{pmatrix}x_1^{(2)} \\ x_2^{(2)} \\ \vdots \\ x_n^{(2)}\end{pmatrix}\right) = \begin{pmatrix}(x_1^{(1)} + x_1^{(2)})/2 \\ (x_2^{(1)} + x_2^{(2)})/2 \\ \vdots \\ (x_n^{(1)} + x_n^{(2)})/2\end{pmatrix}$$

Het zwaartepunt van m punten $x^{(i)}$ wordt dan gegeven door:

$$\mu = \frac{1}{m}(x^{(1)} + x^{(2)} + \cdots + x^{(m)})$$

∎

Het K-gemiddelden algoritme is een iteratief algoritme. De K initiële zwaartepunten worden op een random manier bepaald. Daarna start de hoofdlus van het algoritme. In elke iteratie worden de volgende twee stappen na elkaar uitgevoerd:

- De *toewijzingsstap*, waarin voor elk punt $x^{(i)}$ van de dataset wordt berekend tot welke cluster het behoort, i.e. er wordt bepaald welk zwaartepunt er het dichtst bij $x^{(i)}$ ligt.
- De *update zwaartepuntenstap*, waarin, op basis van de toekenningen in de vorige stap, het nieuwe zwaartepunt voor elk van de K clusters wordt berekend.

De bovenstaande stappen worden herhaald totdat er in de toewijzingsstap geen wijzigingen meer gebeuren. De volledige pseudocode wordt getoond in Algoritme 8.1.

Algoritme 8.1 Pseudocode voor het K-gemiddelde algoritme.

Invoer het aantal clusters K, een ongelabelde dataset $T = \{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$, waarbij elke $x^{(i)}$ een element is van $\mathbb{R}^n$.

Uitvoer Een array $(\mu^{(1)}, \mu^{(2)}, \ldots, \mu^{(K)})$ bestaande uit de zwaartepunten van de K clusters. Elke $\mu^{(k)}$ behoort tot $\mathbb{R}^n$.

function K-MEANS(K, T)
  for $k = 1 \ldots K$ **do** ▷ Random initialisatie zwaartepunten
    $\mu^{(k)} \leftarrow$ random trainingsvoorbeeld
  end for
  while zwaartepunten zijn veranderd **do**
    for $i = 1 \ldots m$ **do** ▷ Cluster assignatie
      $c^{(i)} \leftarrow \arg\min_{k \in \{1,\ldots,K\}} \|x^{(i)} - \mu^{(k)}\|$
    end for
    for $k = 1 \ldots K$ **do** ▷ Cluster zwaartepunt update
      $$\mu^{(k)} \leftarrow \frac{1}{\#\{i \mid c^{(i)} = k\}} \sum_{\{i \mid c^{(i)} = k\}} x^{(i)}$$
    end for
  end while
  return $(\mu^{(1)}, \mu^{(2)}, \ldots, \mu^{(K)})$
end function

Voorbeeld 8.14 (Voorbeeld *K*-gemiddelden algoritme) Veronderstel dat we beschikken over de volgende (zeer kleine) ongelabelde dataset in $\mathbb{R}^2$:

i	$x_1^{(i)}$	$x_2^{(i)}$
1	1	1
2	1	0
3	0	2
4	2	4
5	3	5

We werken met $K = 2$ en we kiezen random de punten

$$\mu^{(1)} = x^{(1)} = (1,1) \quad \text{en} \quad \mu^{(2)} = x^{(3)} = (0,2)$$

als initiële zwaartepunten. Als we bv. $\|x^{(1)} - \mu^{(1)}\|$ berekenen dan vinden we (uiteraard) dat

$$\|x^{(1)} - \mu^{(1)}\| = 0,$$

terwijl de afstand tot het tweede zwaartepunt gegeven wordt door

$$\|x^{(1)} - \mu^{(2)}\| = \sqrt{(1-0)^2 + (1-2)^2} = \sqrt{2} \approx 1.414\,.$$

Dit betekent dat (uiteraard!) het eerste punt aan de eerste cluster wordt toegewezen, of $c^{(1)} = 1$. We doen dit nu voor elk van de vijf punten in de dataset. Het resultaat wordt samengevat in onderstaande tabel

i	$\|x^{(i)} - \mu^{(1)}\|$	$\|x^{(i)} - \mu^{(2)}\|$	$c^{(i)}$
1	0	1.414	1
2	1	2.236	1
3	1.414	0	2
4	3.162	2.828	2
5	4.472	4.243	2

Vervolgens worden de nieuwe zwaartepunten $\mu^{(1)}$ en $\mu^{(2)}$ berekend. We vinden

$$\begin{aligned}\mu^{(1)} &= \frac{1}{2}\big(x^{(1)} + x^{(2)}\big) \\ &= \frac{1}{2}\big((1,1) + (1,0)\big) \\ &= (1, 0.5)\end{aligned}$$

en

$$\begin{aligned}\mu^{(2)} &= \frac{1}{3}\big(x^{(3)} + x^{(4)} + x^{(5)}\big) \\ &= \frac{1}{3}\big((0,2) + (2,4) + (3,5)\big) \\ &\approx (1.667, 3.667)\end{aligned}$$

Nu wordt de afstand van de vijf punten in de dataset tot deze *nieuwe* zwaartepunten berekend. Elk punt wordt toegewezen aan de cluster bepaald door het dichtstbijzijnde zwaartepunt. De volgende tabel vat de berekeningen samen:

i	$\|x^{(i)} - \mu^{(1)}\|$	$\|x^{(i)} - \mu^{(2)}\|$	$c^{(i)}$
1	0.5	2.749	1
2	0.5	3.727	1
3	1.803	2.357	1
4	3.640	0.471	2
5	4.924	1.886	2

Het derde punt wisselde van cluster. We berekenen de nieuwe zwaartepunten:

$$\begin{aligned}\mu^{(1)} &= \frac{1}{3}\left(x^{(1)} + x^{(2)} + x^{(3)}\right) \\ &\approx (0.667, 1.000)\end{aligned}$$

en

$$\begin{aligned}\mu^{(2)} &= \frac{1}{2}\left(x^{(4)} + x^{(5)}\right) \\ &= (2.5, 4.5)\end{aligned}$$

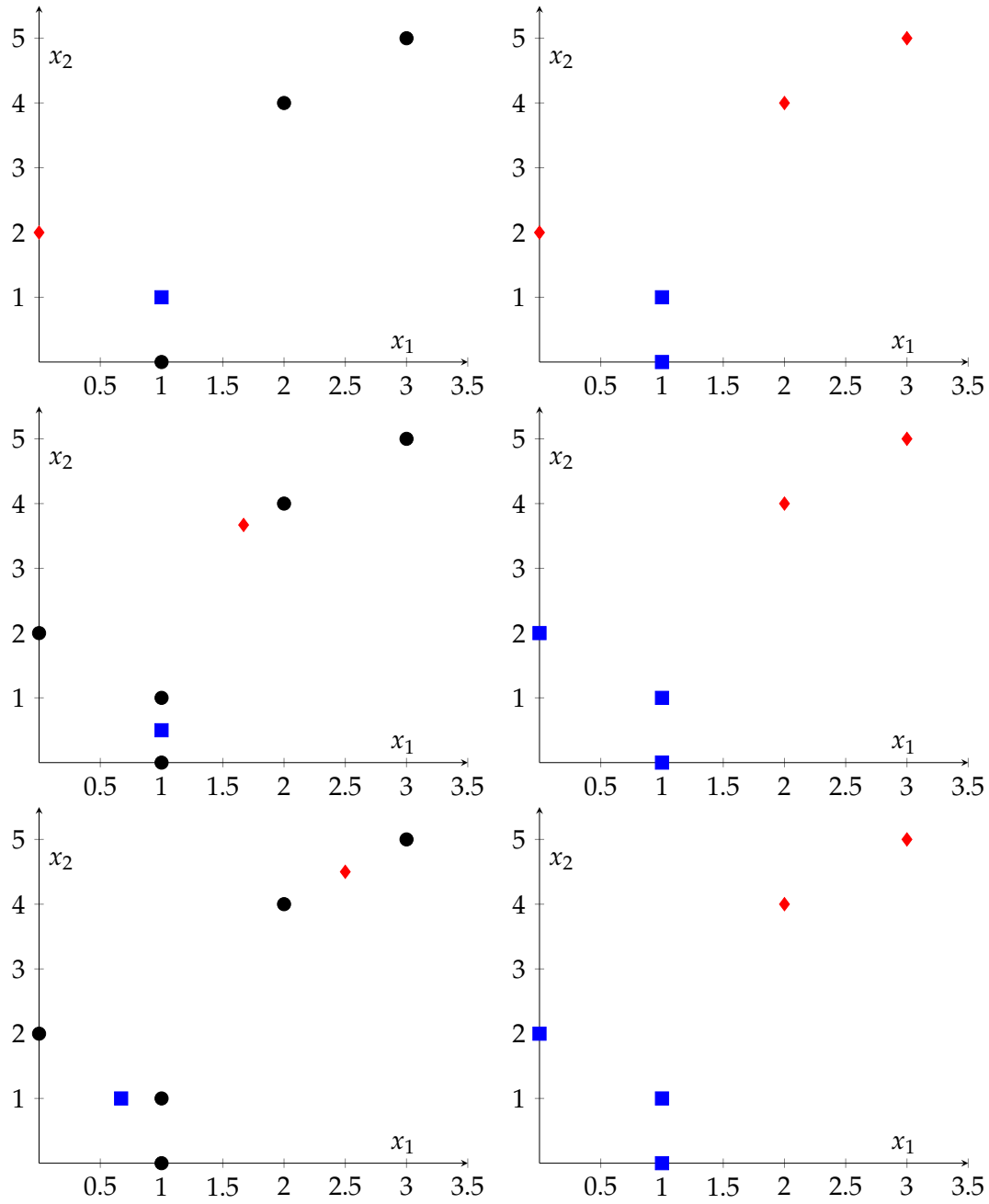
Als men nu nogmaals de afstanden van de vijf punten tot de twee nieuwe zwaartepunten berekent dan ziet men dat er geen verschuivingen meer zijn: alle punten blijven toegewezen aan hun huidige cluster. Het algoritme is geconvergeerd en stopt. ∎

8.3.1 Het K-Gemiddelden Algoritme als Optimalisatie

In zekere zin is het K-gemiddelden algoritme ook een *optimalisatieprobleem*. De kennis van dit optimalisatieprobleem kan ten eerste helpen om eventuele fouten in de implementatie van het K-gemiddelden algoritme op te sporen, maar ten tweede (en belangrijker) het kan helpen bij het bepalen van het juiste aantal clusters K.

Men kan aantonen dat het K-gemiddelden algoritme de *gemiddelde kwadratische afwijking tot de zwaartepunten* minimaliseert. Onthoud dat $c^{(i)}$ de index van de huidige cluster van het punt $x^{(i)}$ voorstelt en dat de zwaartepunten worden aangeduid met $\mu^{(k)}$. Dit betekent dat

$$\mu^{(c^{(i)})}$$



Figuur 8.3: Visuele voorstelling van het *K*-gemiddelden algoritme. Men start met twee random datapunten als zwaartepunten (boven links). Elk punt wordt toegewezen aan het dichtsbijzijnde zwaartepunt (boven rechts). De zwaartepunten worden herberekend (midden links), en opnieuw worden de vijf punten toegewezen aan het dichtstbijzijnde zwaartepunt. De zwaartepunten worden opnieuw herberekend (onder links). Men ziet dat er geen punten meer wisselen van cluster (onder rechts). Dit is het punt waarop het algoritme eindigt.

het zwaartepunt van de cluster waartoe $x^{(i)}$ behoort voorstelt.

We noemen de kostfunctie J en stellen

$$J\big(c^{(1)}, c^{(2)}, \ldots, c^{(m)}, \mu^{(1)}, \mu^{(2)}, \ldots, \mu^{(K)}\big) = \frac{1}{m}\sum_{i=1}^{m} \|x^{(i)} - \mu^{(c^{(i)})}\|^2.$$

Het K-gemiddelden algoritme zoekt zwaartepunten $\mu^{(k)}$ en toewijzingen $c^{(i)}$ zodanig dat de kostfunctie J minimaal wordt. Het is nu echter wel zo dat J *meerdere lokale minima* kan hebben en dat het K-gemiddelden algoritme dus kan vastraken in één van die lokale minima. Random herstart kan helpen om in een "goed" lokaal minimum uit te komen.

Voorbeeld 8.15 (Lokale minima voor clustering) Als een (klein) voorbeeld dat lokale minima voor de kostfunctie mogelijk zijn beschouwen we volgende dataset:

i	$x_1^{(i)}$	$x_2^{(i)}$
1	0	0
2	0	1
3	4	0
4	4	1

Een grafische voorstelling vindt men in Figuur 8.4.

Veronderstel dat men als initiële zwaartepunten $\mu^{(1)} = x^{(1)}$ en $\mu^{(2)} = x^{(2)}$ neemt. Het is dan niet moeilijk om te verifiëren dat het algoritme de volgende assignaties maakt:

i	$\|x^{(i)} - \mu^{(1)}\|$	$\|x^{(i)} - \mu^{(2)}\|$	$c^{(i)}$
1	0	1	1
2	1	0	2
3	4	$\sqrt{17}$	1
4	$\sqrt{17}$	4	2

De kostfunctie voor deze initiële assignatie is

$$J = \frac{1}{4}\big(0^2 + 0^2 + 4^2 + 4^2\big) = 8.$$

De zwaartepunten worden aangepast en we vinden:

$$\mu^{(1)} = \frac{1}{2}(x^{(1)} + x^{(3)}) = (2, 0)$$

en

$$\mu^{(2)} = \frac{1}{2}(x^{(2)} + x^{(4)}) = (2, 1).$$

Men ziet dat de assignaties niet veranderen. De waarde van de kostfunctie in dit geval is:

$$\begin{aligned} J &= \frac{1}{4}\left(\|x^{(1)} - \mu^{(1)}\|^2 + \|x^{(2)} - \mu^{(2)}\|^2 + \|x^{(3)} - \mu^{(1)}\|^2 + \|x^{(4)} - \mu^{(2)}\|^2\right) \\ &= \frac{1}{4}\left(2^2 + 2^2 + 2^2 + 2^2\right) \\ &= 4. \end{aligned}$$

Men ziet dat de waarde van de kostfunctie in de loop van het algoritme is gedaald en eindigt met een waarde gelijk aan 4.

Wanneer men echter start men de initiële keuze $\mu^{(1)} = x^{(1)}$ en $\mu^{(2)} = x^{(3)}$ dan controleert men dat na één iteratie de nieuwe zwaartepunten

$$\mu^{(1)} = (0, 0.5) \quad \text{en} \quad \mu^{(2)} = (4, 0.5)$$

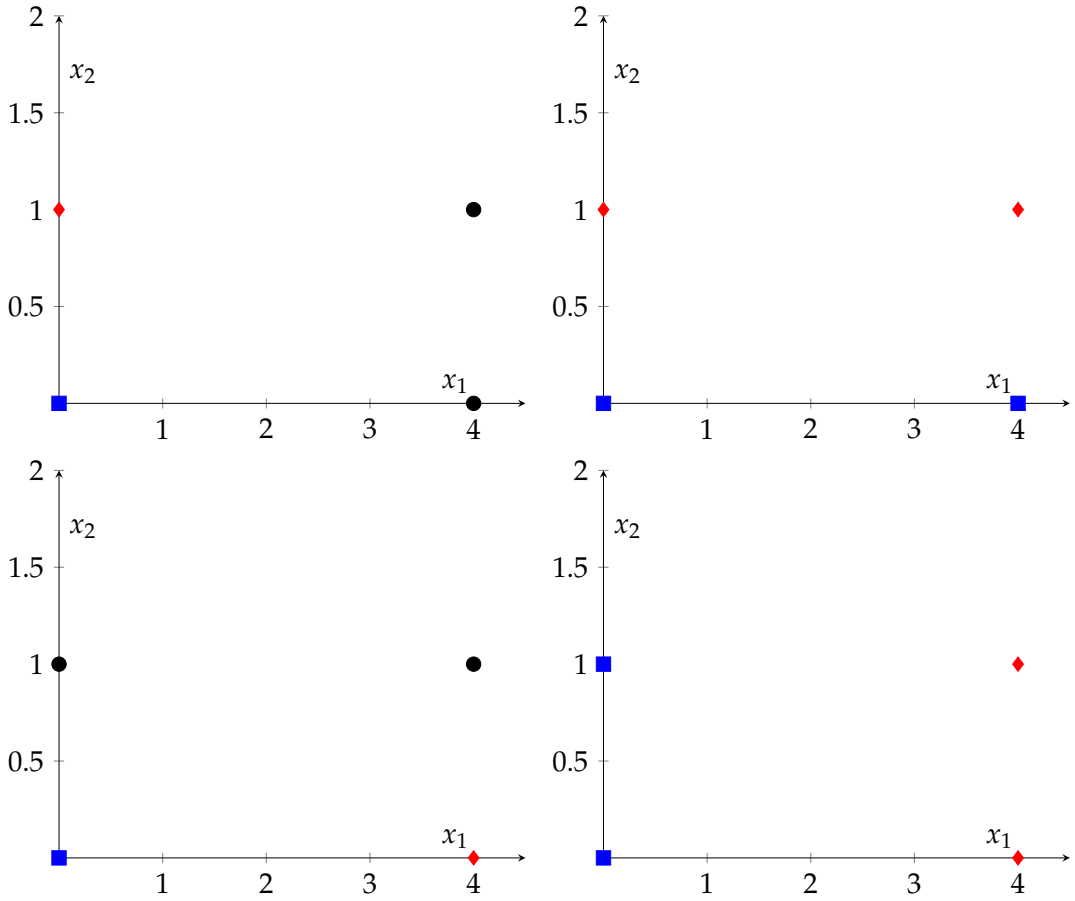
worden. Dit zijn meteen ook de zwaartepunten waarmee het algoritme zal eindigen. De waarde van de kostfunctie is nu

$$\begin{aligned} J &= \frac{1}{4}\left(\frac{1}{2^2} + \frac{1}{2^2} + \frac{1}{2^2} + \frac{1}{2^2}\right) \\ &= \frac{1}{4}. \end{aligned}$$

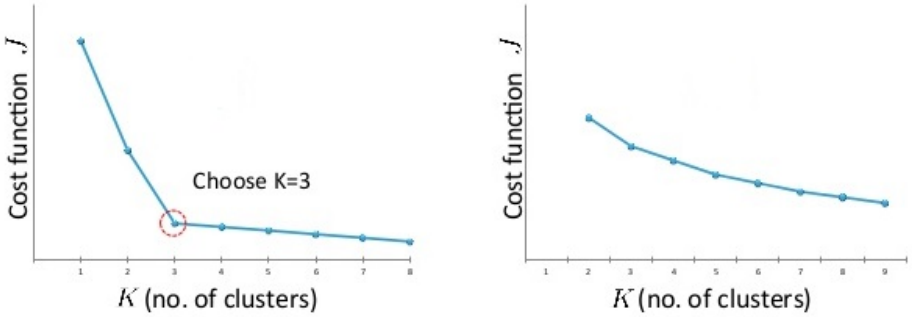
De kostfunctie bevestigt wat intuïtief duidelijk is, nl. dat de tweede clustering "beter" is. ∎

8.3.2 Bepalen van het Aantal Clusters

In sommige gevallen dicteert het probleem het aantal clusters dat men moet "ontdekken", bv. wanneer men kleding wil ontwerpen zoekt men clusters die de maten S(mall), M(edium) en L(arge) voorstellen. In andere gevallen moet men vaak zelf ontdekken wat de "juiste" waarde voor K is. Hier kan de kostfunctie J soms nuttig zijn.



Figuur 8.4: Voorbeeld waarbij het K-gemiddelden algoritme twee verschillende oplossingen vindt afhankelijk van de initiële keuze voor de zwaartepunten. Op de bovenste rij ziet men een voorbeeld waarbij men eindigt met een "horizontale" clustering waarbij de punten in dezelfde cluster relatief ver van elkaar liggen. Op de onderste rij ziet men een betere "verticale" clustering. Deze tweede clustering heeft een lagere waarde voor de kostfunctie J.



Figuur 8.5: Illustratie van een clusteringsprobleem waarbij de elleboog methode goed werkt (links), en één waarbij het aantal clusters niet kan afgeleid worden met deze methode. Bron: http://www.slideshare.net/gopass2002/kmeans-iwth

Wanneer het aantal clusters K toeneemt, dan zal de minimumwaarde van de kostfunctie J dalen (omdat er meer vrijheidsgraden zijn). Het idee is dat, zolang er echte, nuttige clusters worden ontdekt de kostfunctie J snel zal dalen. Eens er geen echte nieuwe clusters meer te ontdekken zijn zal J waarschijnlijk veel minder snel dalen.

Een methode, die zeker niet altijd werkt in de praktijk, bestaat erin om voor de waarden $K = 1$, $K = 2$, $K = 3$ enzovoort steeds de waarde van de kostfunctie te plotten. In het begin verwacht men een sterk dalende functie; de waarde voor K waar de daling opeens sterk afvlakt is dan waarschijnlijk een goede waarde voor het aantal clusters K. Wanneer de methode goed werkt lijkt de grafiek op een menselijke arm die gebogen is aan de elleboog, vandaar ook de naam ELLEBOOG METHODE (Eng. *elbow method*). Figuur 8.5 geeft een illustratie.

8.4 Oefeningen

1. a) De hoeveelheid regenval wordt gewoonlijk uitgedrukt in een aantal mm per dag. Veronderstel dat je wil voorspellen hoeveel regen er morgen zal vallen op een bepaalde plaats. Je beschikt hiervoor over de historische weerobservaties.

   Wat is er van toepassing?

- classificatie
- regressie
- clustering
- gesuperviseerd leren
- ongesuperviseerd leren

b) Sommige problemen worden best aangepakt met algoritmes voor gesuperviseerd leren en andere met algoritmes voor ongesuperviseerd leren. Welke van onderstaande problemen worden het best aangepakt met *gesuperviseerd leren*?

- Het onderzoeken van een grote collectie emails waarvan geweten is dat ze spam zijn teneinde uit te vissen of er verschillende categorieën van spam zijn.
- Gegeven historische data m.b.t. de leeftijd en lengte van kinderen, tracht de lengte van kinderen te voorspellen op basis van hun leeftijd.
- Trachten de artikels van verschillende nieuwsbronnen te groeperen in "gerelateerde" artikels.
- Gegeven 1000 artikels van mannelijke auteurs en 1000 artikels van vrouwelijke auteurs, tracht uit te vissen of een nieuw artikel (waarvan de auteur niet gekend is) geschreven is door een mannelijke of vrouwelijke auteur.

2. Veronderstel dat het K-gemiddelden algoritme uitgevoerd wordt met $K = 3$ en dat de drie zwaartepunten momenteel gegeven worden door:

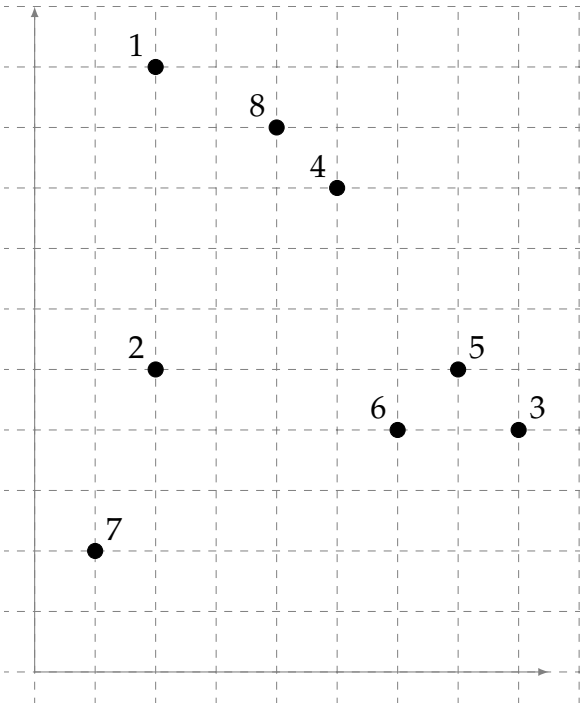
$$\mu^{(1)} = (1,2), \quad \mu^{(2)} = (-3,0) \quad \text{en} \quad \mu^{(3)} = (4,2).$$

Gegeven het voorbeeld

$$x^{(i)} = (-1,2).$$

Wat wordt de waarde van $c^{(i)}$, of anders gezegd, aan welke cluster wordt dit voorbeeld toegewezen?

3. Gegeven de volgende ongelabelde dataset bestaande uit 8 voorbeelden:



Figuur 8.6: Grafische voorstelling van de dataset in Oefening 3.

i	$x_1^{(i)}$	$x_2^{(i)}$	i	$x_1^{(i)}$	$x_2^{(i)}$
1	2	10	5	7	5
2	2	5	6	6	4
3	8	4	7	1	2
4	5	8	8	4	9

Deze dataset wordt ook voorgesteld in Figuur 8.6.

a) Veronderstel dat de drie initiële zwaartepunten gekozen worden bij $\mu^{(1)} = x^{(1)}$, $\mu^{(2)} = x^{(4)}$ en $\mu^{(3)} = x^{(7)}$. Aan welke cluster wordt elk van de datapunten toegewezen. Wat worden de nieuwe zwaartepunten? Duid deze aan op een figuur.

b) Voer nu bijkomende iteraties van het algoritme uit tot het algoritme convergeert. Wat zijn de resulterende clusters? Komen deze overeen met hetgeen je verwacht?

BIJLAGE **A**

Complexiteitstheorie

In deze appendix geven we een korte en informele inleiding tot de complexiteitstheorie. Dat is de theorie die problemen classificeert volgens hun moeilijkheidsgraad. Deze eerste klasse van problemen die besproken wordt is de klasse **P**, de klasse van de "eenvoudige" problemen. We zien hierbij ook dat niet alle problemen tot de klasse **P** behoren, méér zelfs, we zien dat er problemen bestaan, zoals Turing's STOPPROBLEEM, die ONBESLISBAAR zijn. REDUCTIES geven een manier om aan te tonen dat een bepaald probleem minstens zo moeilijk is als een ander probleem. Dit leidt dan onmiddellijk tot het begrip COMPLEETHEID voor een bepaalde klasse. De klasse **NP** wordt geïntroduceerd als de klasse waarvan oplossingen eenvoudig te verifiëren zijn. De klasse **NP** is zeer groot en bevat ook een heel aantal **NP**-complete problemen. Het is een open vraag of de klasse **NP** strikt groter is dan de klasse **P**, en dit is één van de belangrijkste open vragen binnen de informatica!

A.1 De complexiteitsklasse P

In de complexiteitstheorie is men voornamelijk geïnteresseerd om "gemakkelijke" van "moeilijke" problemen te onderscheiden en dit op basis van hun tijdscomplexiteit.

We starten met de definitie van de klasse **P** wat de klasse van de "gemakkelijke" problemen is.

Definitie A.1 De KLASSE **P** is de verzameling van alle problemen die in polynomiale tijd kunnen opgelost worden door een (deterministisch) algo-

ritme. ∎

Opmerking A.2 Met polynomiale tijd bedoelen we dat de uitvoeringstijd van het algoritme $O(n^k)$ is waarbij n de grootte van de invoer voorstelt en k een constante is (onafhankelijk van n). ∎

De meeste van de problemen die we in deze cursus hebben gezien behoren tot de klasse **P**, zoals bv. het vinden van een kortste pad tussen twee knopen in een gewogen graaf met positieve gewichten (Dijkstra), het vinden van een minimale kost opspannende boom (algoritmes van Prim en Kruskal).

Men kan zich nu afvragen of *alle* problemen tot de klasse **P** behoren. Het negatief antwoord op deze vraag werd reeds in 1936 gegeven door Turing (Turing, 1936). Turing's STOPPROBLEEM (halting problem) is het volgende: "schrijf een algoritme A (programma) dat bepaalt of een willekeurig programma P met als input I stopt of niet". Het blijkt dat het stopprobleem een ONBESLISBAAR probleem is: het algoritme A bestaat niet!

Eigenschap A.3 Turing's stopprobleem is onbeslisbaar. ∎

Bewijs We schetsen het bewijs. We moeten aantonen dat er geen algoritme A bestaat dat voor alle programma's P en elke mogelijke invoer I kan bepalen of het programma P stopt voor de invoer I of niet.

We geven een bewijs uit het ongerijmde en veronderstellen m.a.w. dat het programma A wel bestaat:

$$A(P,I) = \begin{cases} 1 & \text{als programma } P \text{ stopt voor invoer } I \\ 0 & \text{als programma } P \text{ niet stopt voor invoer } I. \end{cases}$$

Met dit programma A kunnen we een programma Q construeren met als invoer een willekeurig programma P. Dit programma stopt als P niet stopt wanneer het wordt uitgevoerd met als invoer zichzelf en omgekeerd:

$$Q(P) = \begin{cases} \text{stopt} & \text{als } A(P,P) = 0 \\ \text{stopt niet} & \text{als } A(P,P) = 1. \end{cases}$$

Het programma Q kan gemakkelijk geschreven worden door programma A aan te roepen.

We voeren nu het programma Q uit met zichzelf als invoer en we krijgen:

$$Q(Q) = \begin{cases} \text{stopt} & \text{als } A(Q,Q) = 0, \\ & \text{i.e. programma } Q \text{ stopt niet met zichzelf als invoer} \\ \text{stopt niet} & \text{als } A(Q,Q) = 1, \\ & \text{i.e. programma } Q \text{ stopt wel met zichzelf als invoer.} \end{cases}$$

Beide gevallen leiden tot een contradictie die een logisch gevolg is van het bestaan van het programma A. Dit toont aan dat het programma A niet kan bestaan en dat het stopprobleem inderdaad onbeslisbaar is. ♢

Turing's stopprobleem toont aan dat er wel degelijk grenzen zijn aan wat er kan *berekend* worden m.b.v. een computer. Dit is een interessante observatie voor een informaticus.

A.2 Reducties

Reducties zijn een fundamenteel begrip in de informatica en worden in de praktijk ook vaak gebruikt.

Voorbeeld A.4 (Voorbeeld reductie) Om de mediaan van een rij getallen te vinden kunnen we eenvoudigweg deze rij sorteren en dan het middelste getal of het gemiddelde van de twee middelste getallen teruggeven. Het vinden van de mediaan *reduceert* tot sorteren, of anders gezegd sorteren kan gebruikt worden om het probleem van het vinden van de mediaan op te lossen[1]. ∎

Voorbeeld A.5 (Voorbeeld reductie) Als we bv. de kortste afstanden willen vinden tussen *alle* paren van knopen in een graaf met positieve gewichten dan kunnen we het algoritme van Dijkstra n keer aanroepen, één keer voor elk van de n startknopen. Het probleem van het vinden van de afstanden tussen *alle* paren van knopen is m.a.w. *gereduceerd* tot het n keer aanroepen van het algoritme van Dijkstra. Het algoritme van Dijkstra kan gebruikt worden om het oorspronkelijke probleem op te lossen. ∎

We maken dit nu iets formeler met de volgende definitie.

Definitie A.6 Een probleem π_1 reduceert tot een probleem π_2 wanneer een polynomiaal algoritme voor π_2 kan gebruikt worden om probleem π_1 op te lossen in polynomiale tijd. ∎

Als probleem π_1 reduceert tot π_2 dan betekent dit dat π_1 "gemakkelijk" is wanneer π_2 "gemakkelijk" is:

$$\pi_2 \text{ gemakkelijk} \implies \pi_1 \text{ gemakkelijk}.$$

[1] Er zijn snellere manieren om de mediaan te vinden.

Als we hiervan de contrapositie nemen dan betekent dit

$$\pi_1 \text{ moeilijk} \implies \pi_2 \text{ moeilijk}$$

of

$$\pi_1 \notin \mathbf{P} \implies \pi_2 \notin \mathbf{P}.$$

M.a.w. **als π_1 reduceert tot π_2 dan is π_2 minstens zo moeilijk als π_1**, want je kan π_2 niet alleen gebruiken om π_1 op te lossen maar eventueel ook om nog andere zaken te doen.

A.3 Compleetheid en de klasse NP

Veronderstel dat C één of andere klasse van problemen is. We wensen nu een definitie die zegt dat het probleem π het moeilijkste probleem is van deze klasse of juister gezegd dat het minstens zo moeilijk is als alle problemen uit deze klasse. Dit is de definitie van C-compleetheid.

Definitie A.7 Als C een klasse van problemen is dan is een probleem π C-COMPLEET als en slechts als π tot de klasse C behoort en alle problemen uit de klasse C reduceren naar π. ∎

Als we van een bepaald probleem willen aantonen dat het een moeilijk probleem is, dan kunnen we proberen aan te tonen dat het een C-compleet probleem is voor een zeer grote klasse van problemen C.

Als we wensen aan te tonen dat het handelsreizigersprobleem "moeilijk" is dan zouden we kunnen proberen om als klasse C *alle* computationele problemen te nemen. Helaas werkt dit niet want Turing's stopprobleem is strikt moeilijker dan het handelsreizigersprobleem want dit laatste probleem kan opgelost worden m.b.v. een algoritme met exponentiële uitvoeringstijd terwijl voor het stopprobleem helemaal geen algoritme bestaat.

Zoals net gezegd kan het handelsreizigersprobleem opgelost worden door een brutekracht algoritme. We wensen nu aan te tonen dat het handelsreizigersprobleem minstens zo moeilijk is als alle problemen die kunnen opgelost met zo'n brutekracht algoritme.

Definitie A.8 De KLASSE **NP** bestaat uit de problemen waarvoor oplossingen een lengte hebben die hoogstens polynomiaal is in de lengte van de in-

voer en waarvoor de correctheid van een oplossing kan *geverifieerd* worden in polynomiale tijd. ∎

Opmerking A.9 De klasse **NP** bestaat dus uit die problemen waarvoor het eenvoudig is om te *herkennen* dat je een oplossing hebt gevonden. Dit is anders dan de klasse *P* waar het eenvoudig is om een oplossing te *vinden*. ∎

Voorbeeld A.10 Veronderstel dat men zich in het handelsreizigersprobleem afvraagt of er een rondreis bestaat met kost hoogstens k.

Wanneer iemand je een *voorstel* van een rondreis geeft dan is het eenvoudig om te verifiëren of de totale kost van deze rondreis hoogstens k is. Inderdaad, je moet enkel maar controleren of de som van kosten van de bogen hoogstens k is. Dit kan gebeuren in lineaire tijd. ∎

Alles wat nodig is om tot de klasse **NP** te behoren is dat men op een efficiënte manier oplossingen kan herkennen en dit betekent dat de klasse **NP** enorm veel problemen omvat.

Wanneer een probleem **NP**-COMPLEET is dan betekent dit (door definitie van compleetheid) dat dit probleem minstens zo moeilijk is als alle problemen in **NP**. Dit suggereert dat het waarschijnlijk zeer moeilijk, zo niet onmogelijk is om voor een **NP**-compleet probleem een algoritme te bedenken dat kan uitgevoerd worden in polynomiale tijd. Bovendien: het vinden van zo'n algoritme voor een **NP**-compleet probleem zou betekenen dat *alle* problemen in **NP** kunnen opgelost worden in polynomiale tijd!

De hamvraag blijft natuurlijk of er wel zo'n **NP**-compleet probleem bestaat. Deze vraag werd, onafhankelijk van elkaar, door Cook in 1971 (Cook, 1971) en Levin in 1973 (Levin, 1973) positief beantwoord. In 1972 werd door Karp (Karp, 1972) een lijst van 21 **NP**-complete problemen beschreven. Sindsdien is van honderden problemen bewezen dat ze **NP**-compleet zijn, waaronder ook het handelsreizigersprobleem en het knapzakprobleem.

Om aan te tonen dat een probleem **NP**-compleet is gaat men als volgt tewerk. Eerst toont men aan dat het probleem tot de klasse **NP** behoort. Dit is meestal gemakkelijk. In een tweede stap reduceert men een gekend **NP**-compleet probleem tot het nieuwe probleem.

Voorbeeld A.11 (TSP is NP-compleet) We beschouwen de *beslissingsversie* van het handelsreizigersprobleem: "gegeven een gewogen complete onge-

richte graaf $G = (V, E)$ en een reëel getal d, bestaat er een rondreis die alle knopen juist éénmaal bezoekt zodanig dat de kost van de rondreis hoogstens d is." We noemen dit probleem TSP.

Het is duidelijk dat dit beslissingsprobleem tot de klasse **NP** behoort: als iemand je een voorstel van rondreis doet dan is het eenvoudig om te controleren of het een geldige rondreis is en of de kost inderdaad hoogstens d is. Dit kan gebeuren in een tijd $\Theta(n)$.

We *veronderstellen* nu dat we reeds weten dat het Hamiltoniaanse cykel probleem een **NP**-compleet probleem is. Het Hamiltoniaanse cykel probleem is het volgende: "gegeven een ongerichte (ongewogen) graaf $G = (V, E)$, bestaat er een Hamiltoniaanse cykel, i.e. bestaat er een cykel die alle knopen van de graaf bevat?" Dit probleem noemen we HC.

We tonen nu aan dat HC reduceert tot TSP. Gegeven een graaf $G = (V, E)$ waarop we HC willen toepassen. Construeer nu een nieuwe (gewogen) en complete graaf $G' = (V, E')$ (met dezelfde knopenverzameling) als volgt:

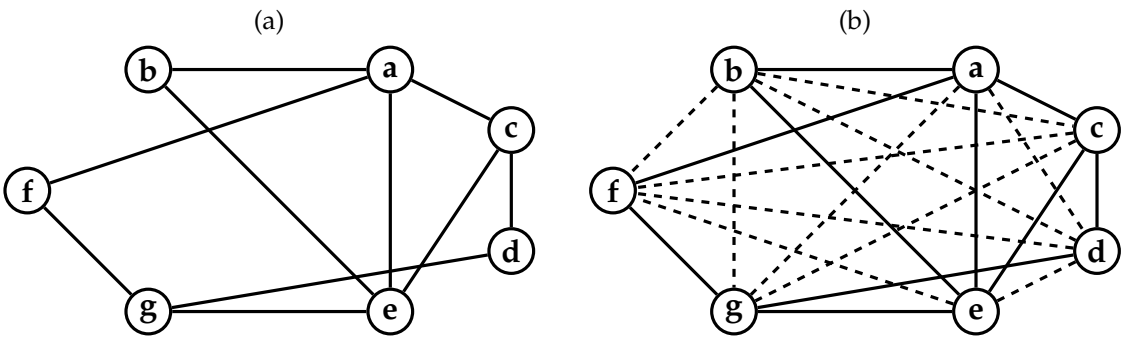
$$\text{gewicht}(e'_{i,j}) = \begin{cases} 1 & \text{als knopen } i \text{ en } j \text{ adjacent zijn in } G \\ 2 & \text{als knopen } i \text{ en } j \text{ niet adjacent zijn in } G. \end{cases}$$

We tonen nu aan dat het antwoord op HC voor $G = (V, E)$ "ja" is als en slechts als het antwoord op TSP voor $G' = (V, E')$ met $d = n$ "ja" is.

Veronderstel dat $G = (V, E)$ een Hamiltoniaanse cykel bezit, dan kunnen in $G' = (V, E')$ al de bogen met gewicht 1 gevolgd worden om een rondreis te bekomen met gewicht gelijk aan n. En dus als HC "ja" heeft als antwoord heeft ook TSP "ja" als antwoord.

Omgekeerd: veronderstel dat er een rondreis is met kost hoogstens n in $G' = (V, E')$. Aangezien deze rondreis n bogen moet bevatten kan deze enkel bogen met gewicht gelijk aan 1 bevatten. Deze bogen vormen een Hamiltoniaanse cykel in de oorspronkelijke graaf. M.a.w. als TSP "ja" heeft als antwoord heeft ook HC "ja" als antwoord.

Figuur A.1 geeft een illustratie van dit proces. ∎



Figuur A.1: Links ziet men de originele ongewogen graaf $G = (V, E)$ voor probleem HC. Rechts ziet men de gewogen graaf $G' = (V, E')$. Om de figuur niet te zwaar te maken zijn de bogen met gewicht 1 met een volle lijn getekend, terwijl de bogen met gewicht 2 met een streepjeslijn zijn aangeduid. Het is duidelijk dat de rechtse graaf een rondreis heeft met kost hoogstens 7 als en slechts de linkse graaf een Hamiltoniaanse cykel heeft. We kunnen TSP dus gebruiken om HC op te lossen, of anders gezegd HC reduceert naar TSP.

A.4 De klasse P versus NP

Het is duidelijk dat de klasse **P** tot de klasse **NP** behoort. Inderdaad, wanneer men efficiënt een oplossing kan berekenen (de klasse **P**) kan men ook efficiënt controleren of een gegeven oplossing correct is (de klasse **NP**). Er geldt dus zeker

$$\mathbf{P} \subseteq \mathbf{NP}.$$

Dé cruciale vraag is echter of de twee klassen samenvallen of niet. Tot op heden is het antwoord hierop nog onbekend al gaan de meeste computerwetenschappers er wel van uit dat de klasse **NP** strikt groter is dan de klasse **P**.

Het **P** versus **NP** probleem is één van de zeven problemen op de lijst van "Millennium Prizes" van het Clay Institute of Mathematics. Een definitief antwoord op deze vraag is dan ook 1 miljoen dollar waard!

Wanneer men in de praktijk te maken heeft met een **NP**-compleet probleem dan kan men dit probleem meestal slechts voor "kleine" instanties exact oplossen binnen een redelijke tijd. Voor grotere instanties neemt men vaak zijn toevlucht tot benaderingsalgoritmen.

Bibliografie

Cook, S. A. (1971). The Complexity of Theorem-proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA. ACM.

Karp, R. M. (1972). *Reducibility Among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA.

Levin, L. A. (1973). Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116.

Prieditis, A. E. (1993). Machine discovery of effective admissible heuristics. *Machine learning*, 12(1-3):117–141.

Russell, S. J. and Norvig, P. (2014). *Artificial Intelligence: A Modern Approach*. Pearson Education, Limited, Harlow, third edition.

Slaats, N. (2019). Probleemoplossend denken 1. Lesnota's.

Turing, A. (1936). On computable numbers. *Proceedings of the London Mathematical Society*, 42:230–265.