Hogeschool Gent
Departement Toegepaste Ingenieurswetenschappen
Vakgroep Informatica
Academiejaar 2018-2019

Algoritmen II

Rudy Stoop
aangevuld door Jan Cnops
14 september 2018

INHOUDSOPGAVE

INLEIDING

Deze nota's horen bij de hoorcolleges van het vak 'Algoritmen II', gegeven aan de masteropleiding industriële wetenschappen: informatica. Ze zijn niet bedoeld voor zelfstudie. Veel van dit materiaal is terug te vinden in goede (Engelstalige) leerboeken over algoritmen. (Bijvoorbeeld in [11, 35, 26, 41, 27, 18, 25, 2, 23, 34].) Voor bepaalde onderwerpen, en voor de recentste ontwikkelingen, maken we gebruik van wetenschappelijke literatuur. (En dan vooral van de overzichtsartikels uit de Computing Surveys van de ACM.[1])

Het vak 'Algoritmen I' gaf voornamelijk een overzicht van fundamentele algoritmen en gegevensstructuren. In dit vervolg komen meer gevorderde onderwerpen aan bod, waarbij vaak gebruik gemaakt wordt van dat basismateriaal:

- Gegevensstructuren II behandelt efficiënte zoekbomen, uitwendige gegevensstructuren, meerdimensionale gegevensstructuren, en samenvoegbare prioriteitswachtrijen.
- Grafen II gaat verder met het overzicht van fundamentele graafalgoritmen.
- Het deel over strings biedt een overzicht van specifieke gegevensstructuren voor strings, en zoekalgoritmen voor strings (met onder meer een inleiding tot abstracte automaten). Tenslotte worden tekstzoekmachines besproken.
- Het deel over hardnekkige problemen geeft een inleiding tot NP-complete problemen, een beperkt overzicht, en mogelijke oplossingsstrategieën.

Opnieuw worden er vrij uitgebreide oefeningensessies voorzien om een beter inzicht te verwerven via implementaties van verschillende van deze algoritmen, hun varianten, of toepassingen.

[1] De Association for Computing Machinery (New York) is de oudste (1947), en een van de grootste vakverenigingen van informatici (www.acm.org). Ze publiceert meer dan veertig wetenschappelijke toptijdschriften, richt talloze conferenties in, en maakt al die publicaties beschikbaar via haar uitstekende digitale bibliotheek.

DEEL 1

GEGEVENSSTRUCTUREN II

HOOFDSTUK 1
EFFICIËNTE ZOEKBOMEN

1.1 INLEIDING

De uitvoeringstijd van de meeste operaties op een binaire zoekboom met hoogte h is $O(h)$. Deze hoogte is sterk afhankelijk van de toevoegvolgorde van de gegevens. Als elke toevoegvolgorde even waarschijnlijk is, dan is de verwachtingswaarde voor de hoogte van een zoekboom met n gegevens $O(\lg n)$. Maar in het slechtste geval is die hoogte $O(n)$, zodat we eigenlijk met een soort gelinkte lijst te maken hebben, met alle gevolgen vandien. Elke toevoegvolgorde even waarschijnlijk onderstellen is niet zeer realistisch. Daarom werd er reeds vroeg gezocht naar manieren om de efficiëntie van zoekbomen te verbeteren. Er zijn verschillende mogelijkheden:

- *Elke operatie steeds efficiënt maken.* De vorm van *evenwichtige* zoekbomen ('balanced search trees') blijft steeds perfect of nagenoeg perfect. De operaties zijn dan ook altijd efficiënt:
  - De eerste evenwichtige bomen waren 'AVL-bomen'. (Genoemd naar de initialen van hun uitvinders, Adel'son-Vel'skiĭ en Landis, 1962.) Bij deze bomen mag het hoogteverschil tussen de twee deelbomen van elke knoop nooit groter zijn dan één. Daartoe slaat men in elke knoop zijn hoogte op. (Eigenlijk volstaat het hoogteverschil tussen zijn deelbomen, en daarvoor zijn in principe slechts twee bits nodig.) Toevoegen of verwijderen van een knoop kan het evenwicht in de boom verstoren, zodat er structuurwijzigingen moeten gebeuren om dat te herstellen. Daarbij mag echter niet aan de fundamentele eigenschap van een zoekboom geraakt worden. In de praktijk worden AVL-bomen niet vaak meer gebruikt, omdat hun implementatie ingewikkeld is, en de structuurwijzigingen soms veel werk vereisen. ($O(\lg n)$ rotaties, zie later.)
  - Later werden '2-3-bomen' uitgevonden (Hopcroft, 1970), waarvan elke (inwendige) knoop twee of drie kinderen heeft (vandaar de naam), en alle bladeren dezelfde diepte hebben. Bij toevoegen of verwijderen wordt dit ideale evenwicht behouden door het aantal kinderen van de knopen te manipuleren. Deze bomen zijn tevens de voorlopers van de 'B-trees' uit hoofdstuk 3. Analoog definieert men '2-3-4-bomen', waarop de operaties wat eenvoudiger uitvallen, omdat de drie soorten knopen meer flexibiliteit bieden.

Knopen van verschillende soorten gebruiken, die bovendien van soort kunnen veranderen, geeft praktische problemen bij implementatie. (Dat geldt niet voor B-trees, zie later.) Beide bomen kunnen echter door equivalente *binaire* bomen voorgesteld worden ('Binary B-trees', Bayer, 1971, en 'Symmetric binary B-trees', Bayer, 1972). Daarbij hebben de bladeren niet meer dezelfde diepte: het perfecte evenwicht wordt opgegeven, maar de hoogte blijft $O(\lg n)$. Deze ideeën leidden uiteindelijk tot rood-zwarte bomen, de meest gebruikte evenwichtige zoekbomen[1].

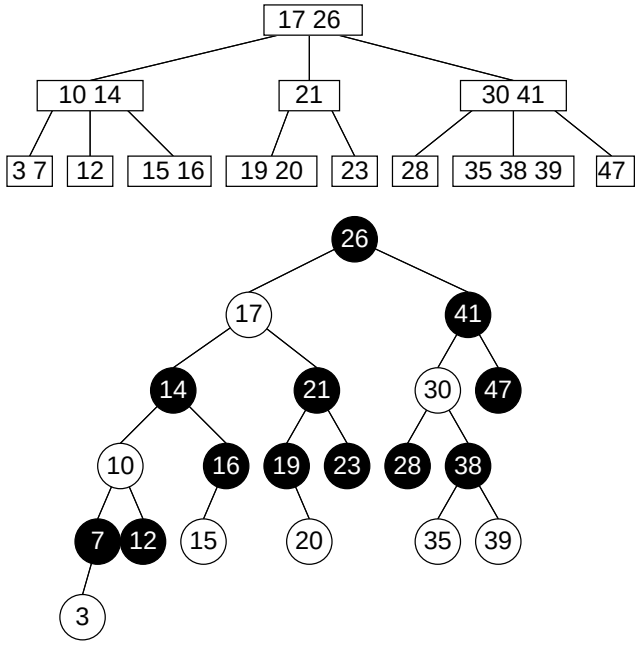
- *Elke reeks operaties steeds efficiënt maken.* Bij 'splay trees' wordt de vorm van de boom meermaals aangepast, zodat die nooit slecht blijft. Met als gevolg dat elke *reeks* opeenvolgende operaties gegarandeerd efficiënt is, ook al kan een individuele operatie zeer traag uitvallen. Met andere woorden, *uitgemiddeld over de reeks* ('geamortiseerd') is de performantie per operatie goed, ook in het slechtste geval.
- *De gemiddelde efficiëntie onafhankelijk maken van de operatievolgorde.* Door gebruik te maken van een random generator zorgen 'randomized search trees' ervoor dat de boom random is en ook blijft, onafhankelijk van de volgorde van toevoegen en verwijderen. Met als gevolg dat de verwachtingswaarde van de performantie van hun operaties steeds $O(\lg n)$ is. De eenvoudigste zoekboom uit deze familie is een 'treap'.

1.2 ROOD-ZWARTE BOMEN

1.2.1 Definitie en eigenschappen

Definitie. Een rood-zwarte boom ('red-black tree', Guibas en Sedgewick, 1978) is een binaire zoekboom die een 2-3-4-boom simuleert. Daartoe vervangt men een 3-knoop door twee, en een 4-knoop door drie binaire knopen. Daarbij definieert men twee soorten ouder-kindverbindingen. De originele verbindingen van de 2-3-4-boom noemt men zwart, de nieuwe verbindingen rood. Die kleur moet in principe opgeslagen worden bij de verbindingen (pointers), maar het is gemakkelijker om ze op te slaan bij de kinderen. De hoogte van de boom wordt laag gehouden door beperkingen op te leggen aan de manier waarop knopen op elke (dalende) weg vanuit de wortel gekleurd kunnen worden. Figuur 1.1 toont een 2-3-4-boom en een equivalente rood-zwarte boom.

[1] Associatieve containers uit zowel de Standard Template Library van C++ (`map`, `set`) als uit de Collections API van Java (`TreeMap`, `TreeSet`) worden geïmplementeerd met rood-zwarte bomen.



Figuur 1.1. 2-3-4-boom en equivalente rood-zwarte boom (wit stelt hier rood voor).

We gebruiken dezelfde knoopstructuur als bij een gewone binaire zoekboom, met een nieuw veld voor de kleur. (Waarvoor in principe één bit volstaat.) Ontbrekende kinderen worden voorgesteld door nullwijzers, die we echter conceptueel beschouwen als wijzers naar virtuele knopen van de boom, die geen gegevens bevatten, maar wel een kleur hebben. Alle gewone knopen worden aldus inwendig. In de literatuur noemt men deze virtuele knopen bladeren, wat voor verwarring zorgt bij vergelijking met gewone binaire bomen, waarbij bladeren echte knopen zonder echte kinderen zijn.

Een rood-zwarte boom wordt dan gedefinieerd als een binaire zoekboom, waarbij bovendien:

(1) Elke knoop rood of zwart gekleurd is.
(2) Elke virtuele knoop zwart is.
(3) Een rode knoop steeds twee zwarte kinderen heeft. (Zijn er steeds twee?)
(4) Elke mogelijke (rechtstreekse) weg vanuit een knoop naar een virtuele knoop evenveel zwarte knopen heeft. Dat aantal noemt men de *zwarte hoogte* van de knoop. Daarbij wordt de knoop zelf niet meegerekend (zoals bij de gewone hoogte), de (zwarte) virtuele knoop echter wel. Elke echte knoop heeft dus zwarte diepte minstens 1.
(5) De wortel zwart is. Deze vereiste is niet strikt noodzakelijk, maar wel handig. Een rode wortel kan trouwens zonder problemen zwart gemaakt worden.

Eigenschappen. Uit deze definitie kunnen we afleiden dat een deelboom met wortel w en zwarte hoogte z tenminste $2^z - 1$ *inwendige* knopen (en dus gegevens) bevat. Dat is eenvoudig aan te tonen door te kijken naar de 2-3-4-bomen. Elke knoop heeft daar twee, drie of vier kinderen. Deze zijn virtueel als en slechts als de knoop op niveau $z - 1$ zit. Op niveau d, met $d < z$ zijn er dus minstens 2^d en hoogstens 4^d knopen. Het totale aantal is dus minimaal

$$1 + 2 + \ldots + 2^{z-1} = 2^z - 1.$$

Dit minimum wordt gehaald als alle knopen in de boom 2-knopen zijn. De equivalente rood-zwarte boom heeft dus allemaal zwarte knopen.

Stel nu dat een rood-zwarte boom met n inwendige knopen een hoogte h heeft (virtuele knopen meegerekend). Daar er geen twee opeenvolgende rode knopen op elke weg vanuit een knoop naar een virtuele knoop mogen voorkomen, geldt dat $z \geq h/2$. Uit de vorige eigenschap volgt dat

$$n \geq 2^z - 1 \geq 2^{h/2} - 1$$

wat uiteindelijk leert dat

$$h \leq 2 \lg(n + 1).$$

De hoogte van een rood-zwarte boom met n knopen is dus steeds $O(\lg n)$.

De boom is nu slechts bij benadering evenwichtig. Omdat de eisen hier minder streng zijn dan bij een AVL-boom, zijn de operaties op deze bomen eenvoudiger en efficiënter te implementeren.

1.2.2 Zoeken

Bij zoekoperaties speelt de kleur van de knopen geen rol, en wordt de rood-zwarte boom een gewone binaire zoekboom. Zijn hoogte is echter steeds $O(\lg n)$, zodat zoeken naar een willekeurige sleutel, zoeken naar de kleinste en de grootste sleutel, en zoeken naar de opvolger en de voorloper van een sleutel, die allemaal $O(h)$ waren, nu gegarandeerd $O(\lg n)$ worden. Zoeken is dus altijd efficiënt.

1.2.3 Toevoegen en verwijderen

Een element toevoegen aan een rood-zwarte boom, of er een uit verwijderen, zonder rekening te houden met de kleur, is dus ook $O(\lg n)$. Het is echter niet zeker dat deze gewijzigde boom nog rood-zwart zal zijn, zodat de efficiëntie van erop volgende operaties niet meer verzekerd kan worden. Gelukkig zal blijken dat toevoegen en verwijderen, met als resultaat een nieuwe rood-zwarte boom, toch efficiënt kunnen gebeuren, als er bepaalde kleur- en structuurwijzigingen aan de boom gebeuren.

Bij toevoegen kan men op twee manieren te werk gaan:

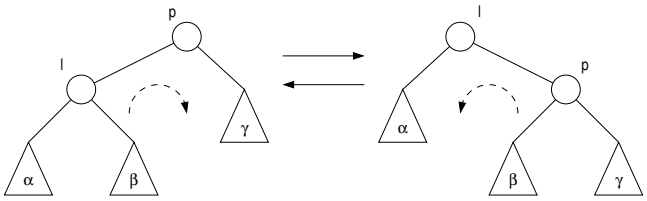
- Ofwel voegt men eerst toe zonder op de kleur te letten, zoals bij een gewone binaire zoekboom (een virtuele knoop wordt vervangen door de nieuwe knoop). Daarna herstelt men de rood-zwarte boom, te beginnen bij de nieuwe knoop, en desnoods tot bij de wortel. We dalen dus af, om nadien langs dezelfde weg weer te stijgen. Ouderwijzers of een stapel zijn hiervoor vereist. Men spreekt van een 'bottom-up' rood-zwarte boom.
- Ofwel wordt de boom reeds aangepast langs de dalende zoekweg. Deze variant heet natuurlijk 'top-down', en blijkt in de praktijk efficiënter zowel in tijd als in plaats, omdat ouderwijzers noch stapel nodig zijn.

Ook verwijderen kan 'bottom-up' of 'top-down' gebeuren.

In beide soorten rood-zwarte bomen gebeuren de structuurwijzigingen met *rotaties*, die reeds bij AVL-bomen gebruikt werden.

1.2.3.1 Rotaties

Rotaties wijzigen de vorm van de boom, maar behouden de inorder volgorde van de sleutels. (Waarom is dat nodig?) De kleur van de knopen blijft onveranderd. Een rotatie is een lokale operatie, waarbij twee *inwendige* knopen betrokken zijn. (Een ouder en een van zijn kinderen.) De rotatie 'draait' rond de ouder-kindverbinding. Er zijn twee soorten, een rotatie naar links voor een rechterkind en een naar rechts voor een linkerkind, en ze zijn elkaars inverse (figuur 1.2). Een rotatie moet drie ouder-



Figuur 1.2. Rotaties

kindverbindingen aanpassen. (In beide richtingen, als er ouderwijzers zijn.) Bij een rotatie naar rechts van een ouder p en zijn linkerkind l bijvoorbeeld wordt het rechterkind van l (als het bestaat) het linkerkind van p, de ouder van p (als die bestaat) wordt de ouder van l, en p wordt het rechterkind van l. Een rotatie is dus $O(1)$.

1.2.3.2 Bottom-up rood-zwarte bomen

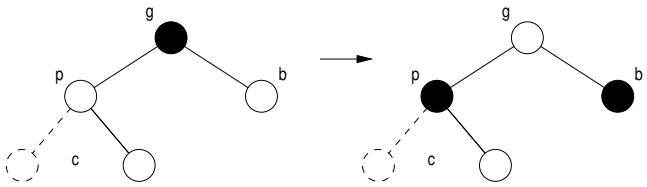
Toevoegen. De nieuwe knoop wordt dus eerst op de gewone manier toegevoegd. Maar welke kleur geven we hem? Een zwarte knoop kan de zwarte hoogte van veel

knopen ontregelen, en een rode knoop mag enkel als de ouder zwart is. We kiezen toch maar voor rood, omdat de zwarte hoogte moeilijker te herstellen valt. Er is dus een probleem als de ouder ook rood is: er zijn dan twee opeenvolgende rode knopen. We zullen deze storing trachten te verwijderen door enkele rotaties en kleurwijzigingen, nadat we ze eventueel eerst naar boven in de boom opgeschoven hebben, desnoods tot bij de wortel.

Aangezien de ouder p van de nieuwe knoop c rood is, kan hij geen wortel zijn (want die is steeds zwart). Er is dus een grootouder g, die zwart is (want de boom waaraan we c toevoegen was rood-zwart, en die heeft geen opeenvolgende rode knopen). We krijgen dan *zes* mogelijke gevallen, die uiteenvallen in twee groepen van drie, naar gelang dat p een linker- of rechterkind is van g. De operaties voor beide groepen zijn elkaars spiegelbeeld, zodat we slechts één groep moeten bespreken.

Onderstel dat p een linkerkind is van g. We kunnen de rode ouder p elimineren ofwel door hem zwart te maken (dat vereist een zwarte knoop die rood moet worden, ter compensatie), ofwel door hem weg te nemen via een rotatie naar rechts (als dit niet opnieuw twee opeenvolgende rode knopen oplevert in de rechterdeelboom). Er zijn dan ook twee hoofdgevallen, naar gelang van de kleur van de broer b van p:

(1) *De broer b van p is rood.* De ligging van c ten opzichte van zijn ouder p is hier



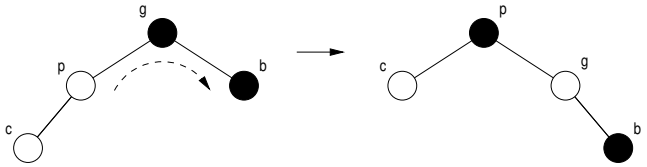
Figuur 1.3. Rode broer.

onbelangrijk. De rode ouder p kan zwart worden, als er aan dezelfde kant een zwarte knoop rood wordt, ter compensatie. Wijzigingen moeten zo lokaal mogelijk blijven, zodat we de zwarte grootouder g rood maken. Maar daardoor nemen we ook een zwarte kleur weg aan de andere kant, wat gelukkig kan opgevangen worden door de rode broer b zwart te maken.
Als de rood geworden g een zwarte ouder heeft, of geen ouder, dan is het probleem opgelost. Is deze overgrootouder echter rood, dan zijn er opnieuw twee opeenvolgende rode knopen. Het probleem werd dus opgeschoven, in de richting van de wortel. Ofwel wordt het dan opgelost door een van de andere gevallen, ofwel wordt het verder omhoog geschoven. In het slechtste geval gaat dat opschuiven door tot bij de wortel, die dan nog enkel (opnieuw) zwart moet gemaakt worden.

(2) *De broer b van p is zwart.* Initieel is b een virtuele knoop, maar als dit geval later hogerop voorkomt zal b een gewone knoop zijn. Nu nemen we de rode ouder p weg via een rotatie naar rechts, die de rechterdeelboom een extra knoop bezorgt.

Omdat p nu bovenaan komt, en dus invloed heeft op twee wegen, neemt die best de zwarte kleur van g over. De overgebrachte knoop g moet dan rood worden om de zwarte hoogte aan die kant niet te ontregelen.
Dit werkt echter enkel als c, p en g op één lijn liggen. (Ga eens na.) Er zijn dus nog twee mogelijkheden, naar gelang dat c een linker- of rechterkind is van p:

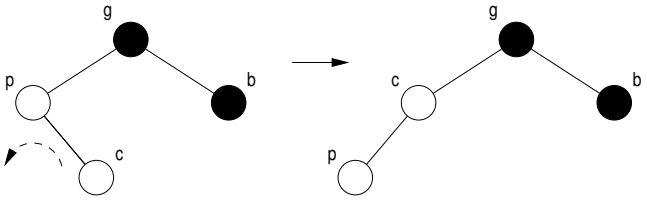
a. *Knoop c is een linkerkind van p.* Hier liggen de drie knopen op één lijn,



Figuur 1.4. Rood kind langs buiten.

zoals gewenst. We roteren ouder p en grootouder g naar rechts, maken p zwart, en g rood. Er liggen nu twee rode knopen aan weerszijden van een zwarte ouder, zodat het probleem opgelost is.

b. *Knoop c is een rechterkind van p.* Om de drie knopen op één lijn te krijgen,



Figuur 1.5. Rood kind langs binnen.

roteren we p en c naar links. Dan krijgen we het vorige geval (mutatis mutandis).

In totaal zijn er dus hoogstens twee rotaties nodig om de boom te herstellen, eventueel voorafgegaan door $O(\lg n)$ keer opschuiven (beperkt door de hoogte van de boom). Zowel roteren als opschuiven zijn $O(1)$, en het initiële afdalen is $O(\lg n)$, zodat toevoegen steeds $O(\lg n)$ is.

Verwijderen. Verwijderen is zoals gewoonlijk wat ingewikkelder. Bovendien wordt de implementatie vlug onoverzichtelijk omdat men steeds moet nagaan of bepaalde knopen wel bestaan. (Virtuele knopen zijn conceptueel zwart, maar er is geen knoop om hun kleur op te slaan!) Daarom gebruikt men in dergelijke gevallen soms een speciale 'null'knoop, die een virtuele knoop voorstelt. De structuur van deze knoop is dezelfde als die van de gewone knopen. Zijn kleur is steeds zwart, zijn andere velden

kan men naar believen gebruiken. Bemerk wel dat slechts één nullknoop álle virtuele knopen voorstelt: alle nullwijzers worden door een wijzer naar deze knoop vervangen. Een kind heeft nu altijd een ouderpointer, ook als het een virtuele knoop is. De waarde van deze pointer is zinloos, maar men kan hem wel invullen zonder eerst te testen of de knoop bestaat. Verwijderen maakt eveneens gebruik van rotaties, waarvan de implementatie ook wat vereenvoudigt met de nullknoop.

Eerst verwijdert men de knoop (of zijn voorloper of opvolger), zoals bij een gewone zoekboom. Als de fysisch te verwijderen knoop rood is, dan heeft hij geen 'echte' kinderen (enkel virtuele). Verwijderen is dan eenvoudig, en zonder gevolgen voor de zwarte hoogten. Als de fysisch te verwijderen knoop zwart is, dan heeft hij ofwel geen 'echte' kinderen, ofwel één rood kind. Dat rood kind kan de zwarte kleur van zijn verdwenen ouder overnemen, zodat de zwarte hoogten intact blijven. (Waarom kan dat niet met een eventueel rode ouder?) Als er geen 'echte' kinderen zijn, mag de zwarte kleur toch niet zomaar verdwijnen, of de zwarte hoogten komen in het gedrang. Daarom geeft men die kleur aan een van de 'kinderen' (een zwarte virtuele knoop), die dan 'dubbelzwart' wordt. Dat is niet in overeenstemming met de definitie van een rood-zwarte boom, zodat we deze anomalie moeten verwijderen.

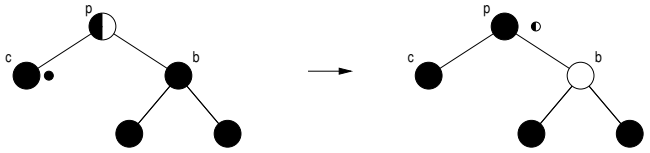
In bepaalde gevallen zal men het probleem naar boven moeten opschuiven, zodat niet alleen een blad, maar ook een inwendige knoop dubbelzwart kan worden. Het elimineren van een dubbele zwarte kleur moet dus voor een willekeurige knoop c opgelost worden. (Door een zwarte nullknoop te gebruiken, vervalt het onderscheid tussen een virtuele en een inwendige knoop. Als ouder krijgt dat blad dan zijn vroegere grootouder, de ouder van de verwijderde knoop.)

Als de dubbelzwarte knoop c wortel is, dan kan de overtollige zwarte kleur gewoon verdwijnen, omdat de wortel in geen enkele zwarte hoogte meetelt. Stel dus dat c geen wortel is, met ouder p. Dan zijn er *acht* mogelijke gevallen, die uiteenvallen in twee groepen van vier, naar gelang dat c een linker- of rechterkind van p is. De operaties voor beide groepen zijn elkaars spiegelbeeld, zodat we slechts één groep moeten bespreken.

Onderstel dat c een linkerkind is van p. We kunnen het overtollige zwart elimineren ofwel door het samen met een vrijgekomen zwarte kleur uit de rechterdeelboom naar boven op te schuiven (dat vereist dat er rechts een knoop rood kan worden), ofwel door de linkerdeelboom een extra knoop te bezorgen, via een rotatie naar links, en die zwart te maken. Beide gevallen vereisen dat c een zwarte broer heeft. Er zijn dus twee hoofdgevallen, naar gelang van de kleur van deze broer b:

(1) *Broer b van c is zwart.* De kleur van hun ouder p is willekeurig. Of b zijn kleur afstaat, of roteert, hangt af van de kleur van zijn (eventueel virtuele kinderen). Om rood te kunnen worden moet b immers twee zwarte kinderen hebben, en wanneer b roteert moet zijn verdwenen zwarte kleur gecompenseerd worden door een van zijn rode kinderen zwart te maken. Er zijn dus nog drie gevallen, naar gelang van de kleur van de kinderen van b:

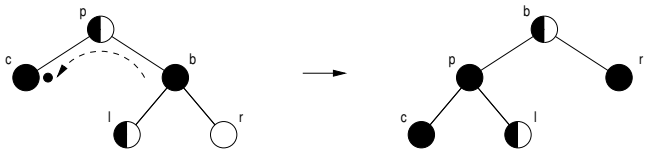
   a. *Broer b heeft twee zwarte kinderen.* Dan kan b rood worden, en de vrijgeko-



Figuur 1.6.

men zwarte kleur schuiven we samen met het overtollige zwart van c door naar hun ouder p. De zwarte hoogte op beide takken blijft aldus ongewijzigd. Een rode ouder p kan zwart worden, zodat alles in orde is.
Een zwarte ouder p wordt echter dubbelzwart, zodat het probleem naar boven opgeschoven werd. Ofwel wordt het daar opgelost door een van de andere gevallen, ofwel wordt het verder opgeschoven, desnoods tot bij de wortel, waar dat overtollige zwart gewoon verdwijnt.

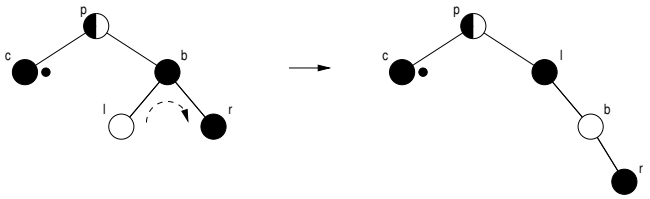
b. *Broer b heeft een rood rechterkind.* De kleur van het linkerkind l van b is



Figuur 1.7.

willekeurig. Nu bezorgen we de linkerdeelboom een extra knoop (via rotatie naar links), zodat die de overtollige zwarte kleur van c kan overnemen. Daarbij verliest de rechterdeelboom echter een zwarte knoop. Gelukkig is er het rode rechterkind r van b, dat zwart kan worden. (Dit werkt niet met een rood linkerkind.) We roteren dus p en b naar links, en b krijgt de (willekeurige) kleur van p, die zelf zwart wordt, net als r. Daarmee is het probleem van de baan.

c. *Broer b heeft een zwart rechterkind en een rood linkerkind.* We herleiden

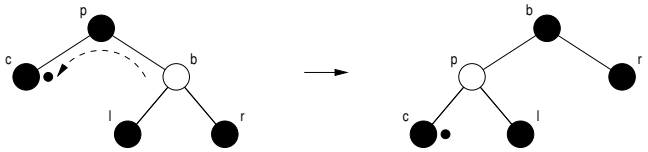


Figuur 1.8.

dit tot het vorige geval door de (nieuwe) broer van c een rood rechterkind te bezorgen. Daartoe roteren we b en zijn rood linkerkind l naar rechts, maken

b rood, en l zwart. De nieuwe broer l van c is zwart, en heeft nu een rood rechterkind b. Dat is precies het vorige geval (mutatis mutandis).

(2) *Broer b van c is rood.* Hun ouder p moet dan zwart zijn. We herleiden dit tot



Figuur 1.9.

het eerste hoofdgeval, door c een nieuwe zwarte broer te bezorgen. Dat moet een *inwendige* knoop zijn, want we hebben zijn kinderen nodig. Daarvoor gebruiken we het zwarte linkerkind l van broer b, die immers twee inwendige zwarte kinderen heeft. (De zwarte hoogte van b is minstens twee, omdat c dubbelzwart is.) We roteren ouder p en broer b naar links, maken b zwart, en p rood. De dubbelzwarte knoop c heeft nu een nieuwe zwarte broer l, en een rood geworden ouder p. Dat is dus het eerste hoofdgeval.

Het eerste hoofdgeval lost het probleem op met één of twee rotaties, of schuift het naar boven. Het tweede hoofdgeval verricht één rotatie, om bij het eerste hoofdgeval terecht te komen. Daar wordt het ofwel opgelost met één of twee bijkomende rotaties, ofwel met twee kleurwijzigingen (zonder op te schuiven, omdat de ouder van c rood is, en dus probleemloos zwart kan worden). Een naar boven opgeschoven probleem wordt ofwel daar opgelost met maximaal drie rotaties, of nog verder opgeschoven.

Verwijderen heeft dus hoogstens drie rotaties nodig om de boom te herstellen, eventueel voorafgegaan door $O(\lg n)$ keer opschuiven (beperkt door de hoogte van de boom). Zowel roteren als opschuiven zijn $O(1)$, en het initiële afdalen is $O(\lg n)$, zodat ook verwijderen steeds $O(\lg n)$ is.

1.2.3.3 Top-down rood-zwarte bomen

Bij deze bomen moeten we nooit naar boven terugkeren, zodat ouderwijzers noch stapel vereist zijn.

Toevoegen. Net zoals bij de bottom-up versie vervangt de nieuwe knoop een virtuele knoop, en maken we hem rood. Dat geeft enkel problemen als zijn ouder ook rood is. We hebben gezien dat dit met rotaties en kleurwijzigingen kan opgelost worden als de oom van de nieuwe knoop zwart is. Bij een rode oom echter moesten we soms terugkeren in de boom, en dat is hier onmogelijk. Daarom zullen we er tijdens het afdalen voor zorgen dat deze oom steeds zwart zal zijn.

Op de weg naar beneden mogen we daarom geen rode broers toelaten, want de nieuwe (rode) knoop zou een kind van een van beide kunnen worden. Wanneer we dus voorbij een *zwarte knoop met twee rode kinderen* komen, dan maken we die knoop rood en zijn kinderen zwart. Als die nieuwe rode knoop zelf een rode ouder heeft is er een probleem, dat echter weer met rotaties en kleurwijzigingen kan opgelost worden, omdat zijn oom gegarandeerd zwart is. Rode broers hoger op de zoekweg werden immers op dezelfde manier geëlimineerd. Daarbij kunnen nieuwe rode broers gecreëerd worden, maar die liggen dan hoger in de boom, zodat ze hier geen problemen meer veroorzaken. (Bemerk trouwens dat door het zwart maken van de rode broers, de zoektocht naar beneden tenminste twee generaties zwarte knopen zal ontmoeten.)

Als er onderweg een rotatie moet gebeuren, zijn daar drie generaties knopen bij betrokken, want om twee opeenvolgende rode knopen (ouder en kind) te elimineren moeten ouder en grootouder roteren, waarbij ook de boompointer naar de grootouder moet aangepast worden. Tijdens de afdaling moeten er dus steeds (wijzers naar) drie opeenvolgende boompointers bijgehouden worden.

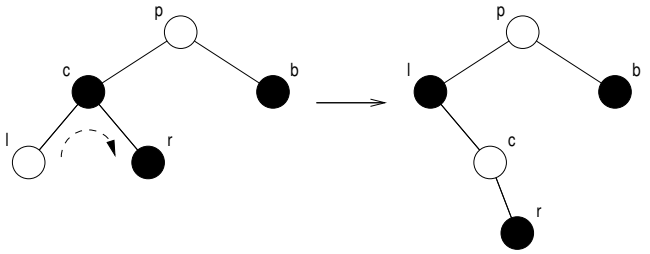
Toevoegen daalt enkel in de boom, waarbij rotaties en kleurwijzigingen kunnen gebeuren. De performantie is dus steeds $O(\lg n)$.

Verwijderen. Zoals bij een gewone binaire zoekboom moet een te verwijderen knoop met twee echte kinderen vervangen worden door zijn voorloper of opvolger. De zwarte hoogte van de fysisch te verwijderen knoop is dus één, omdat minstens een van zijn kinderen virtueel is. Om geen problemen te krijgen met de zwarte hoogte, moeten we ervoor zorgen dat deze knoop rood is. Maar dan is ook zijn tweede kind steeds virtueel, want een rode knoop mag geen rood kind hebben.

Omdat we niet weten wanneer we deze knoop op de dalende zoekweg zullen vinden, maken we elke volgende knoop op die weg rood. Stel dus dat we tijdens het afdalen in een rode of rood gemaakte knoop p beland zijn, en dat we nog verder moeten afdalen. We zullen dan bij een (uiteraard) zwart kind c terechtkomen, dat rood moet worden, waarna we opnieuw in de beginsituatie zijn. (We zullen ons later bezighouden met het begin, bij de wortel). Naar gelang dat c een linker- of rechterkind is van p zijn er twee groepen van mogelijke gevallen te onderscheiden. Opnieuw zijn de operaties elkaars spiegelbeeld, zodat we slechts één groep moeten bespreken.

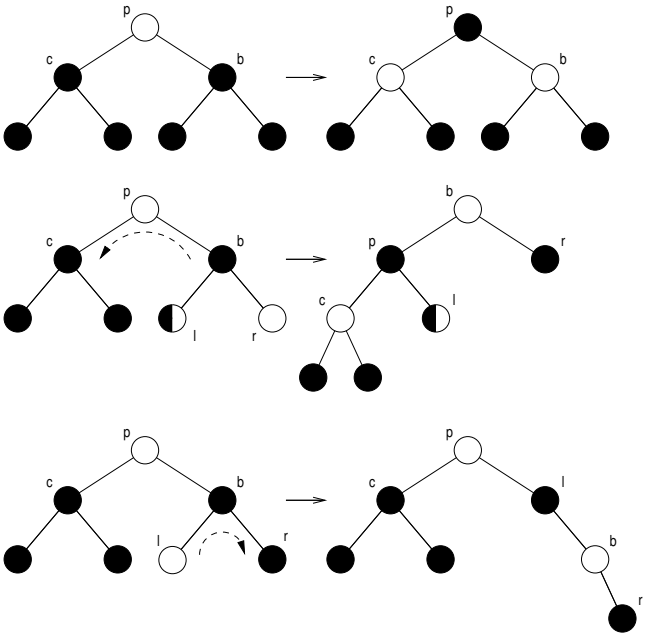
Stel dat c een linkerkind is van p. Ofwel kan het probleem opgelost worden via een van de rode kinderen van c, ofwel moet er een beroep gedaan worden op broer b. Er zijn dus twee hoofdgevallen:

(1) *Knoop c heeft minstens één rood kind.* Als we geluk hebben moeten we verder naar een rood kind, zodat we weer in de beginsituatie zijn.
Als we echter naar een zwart kind moeten afdalen, of als c de (fysisch) te verwijderen knoop is, dan maken we c rood door hem samen met zijn rood kind te roteren, en hun beide kleuren om te keren. We zijn dan opnieuw in de beginsituatie, of we kunnen c zonder meer verwijderen.



Figuur 1.10.

(2) *Knoop c heeft twee zwarte kinderen.* (Dat geldt ook als c geen inwendige kinderen



Figuur 1.11.

heeft.) Om c rood te maken zullen we nu een beroep moeten doen op zijn (zwarte) broer b. Ofwel geeft c zijn zwarte kleur door aan ouder p, maar dan moet b ook rood kunnen worden. Ofwel brengen we een knoop over (via rotatie) om de zwarte kleur van c over te nemen, en zullen we het verlies van een zwarte knoop aan de andere kant moeten compenseren. (Bemerk de analogie met het elimineren van een dubbelzwarte knoop bij bottom-up verwijderen.) Er zijn dus nog drie mogelijkheden, afhankelijk van de kleur van de kinderen van b (b is nooit virtueel en heeft dus kinderen):

a. *Broer b heeft twee zwarte kinderen.* Dan maken we b en c rood, en p zwart.
b. *Broer b heeft een rood rechterkind.* De kleur van het linkerkind is willekeurig. Omdat c rood moet worden, is er een bijkomende zwarte knoop in de linkse deelboom nodig, die we bekomen via een rotatie naar links. Daarbij verliest de rechterdeelboom een zwarte knoop, waarvan de kleur bewaard wordt door het rood rechterkind r van b zwart te maken. We roteren dus p en b naar links, maken c en b rood, en zowel p als r zwart.
c. *Broer b heeft een zwart rechterkind en een rood linkerkind.* In principe zorgen we ervoor dat de (nieuwe) broer van c een rood rechterkind krijgt, zodat we in het vorige geval terechtkomen. We roteren dus het rood linkerkind l en b naar rechts en keren hun kleuren om. We zijn dan in het vorige geval, dat p en l naar links roteert, en de kleuren van p, c, b en l omkeert.
Het kan echter eenvoudiger, aangezien de kleuren van b en l tweemaal omkeren, zodat we ze beter onveranderd laten. Het volstaat dus om l en b naar rechts te roteren, dan p en l naar links te roteren, en tenslotte c rood te maken en p zwart.

Ofwel kan de rood geworden knoop c nu (fysisch) verwijderd worden, ofwel moeten we verder afdalen, zodat we weer in de beginsituatie zijn.

Blijft de vraag hoe we beginnen bij de (zwarte) wortel. Die kunnen we beschouwen als een zwarte knoop c waarheen we moeten vanuit een fictieve rode ouder. Afhankelijk van de kinderen van de wortel krijgen we dan dezelfde hoofdgevallen als hierboven. Het tweede hoofdgeval (beide kinderen zwart) is echter eenvoudiger, omdat de wortel steeds (tijdelijk) rood mag worden, want hij heeft geen broer waarmee hij rekening moet houden.

Net als toevoegen gebeurt verwijderen door af te dalen in de boom, waarbij rotaties en kleurwijzigingen kunnen gebeuren. De performantie is dus ook steeds $O(\lg n)$.

1.2.4 Vereenvoudigde rood-zwarte bomen

De implementatie van rood-zwarte bomen is omslachtig door de talrijke speciale gevallen. Daarom gebruikt men soms eenvoudiger varianten. Zo mag bij een AA-boom (Andersson, 1993) enkel een *rechterkind* rood zijn. Ook ‘Binary B-trees’ hadden reeds dezelfde beperking ten opzichte van ‘Symmetric binary B-trees’, zonder echter de rood-zwarte terminologie te gebruiken. Deze beperkingen reduceren het aantal gevallen, maar behouden toch de (asymptotische) efficiëntie.

Een andere variant is de *left-leaning red-black tree* (Sedgewick, 2008). Hierbij mag een zwarte knoop alleen een rood rechterkind hebben als hij ook een rood linkerkind heeft. Ook deze beperking vermindert het aantal te implementeren speciale gevallen drastisch.

1.3 SPLAY TREES

In tegenstelling tot evenwichtige zoekbomen garanderen splay trees (Sleator en Tarjan, 1985) niet dat elke afzonderlijke operatie efficiënt is. In plaats daarvan zorgen ze ervoor dat elke *opeenvolgende reeks* operaties steeds efficiënt is, zodat uitgemiddeld over de reeks de performantie per operatie goed uitvalt. (Dat gemiddelde is dus *geen* verwachtingswaarde, zoals bij gewone zoekbomen. Men noemt dit 'amorticed efficiency', want uitgesmeerd over de reeks.) Aangezien men efficiënte zoekbomen niet gebruikt om er slechts enkele operaties op uit te voeren, vormen ze een interessant alternatief voor de meer ingewikkelde rood-zwarte bomen.

Concreter, wanneer men een reeks van m operaties verricht op een initieel ledige splay tree, waaronder n keer toevoegen (zodat $m \geq n$), dan is de performantie van deze reeks $O(m \lg n)$, ook in het slechtste geval. Uitgemiddeld over de reeks is dat dus $O(\lg n)$ per operatie. (Dat is slechter dan de $O(\lg n)$ voor een individuele operatie in het slechtste geval bij rood-zwarte bomen, aangezien hier een individuele operatie wel slecht kan zijn.)

Aangezien individuele operaties nu inefficiënt mogen zijn, zal men bij een dergelijke operatie de vorm van de boom moeten aanpassen, zoniet zou het herhalen van diezelfde operatie een slechte reeks opleveren. De basisidee van een splay tree bestaat erin om elke knoop die gezocht, toegevoegd of verwijderd wordt, tot *wortel* van de boom te maken, zodat volgende operaties op die knoop (of zijn omgeving) zeer efficiënt worden. Dat is zeker nuttig als men vaak dezelfde knopen nodig heeft: splay trees presteren dan ook bijzonder goed als men in dat geval is. Dit is vergelijkbaar met gelinkte lijsten die zelf hun gemiddelde zoektijd verbeteren, door elk gevonden element vooraan de lijst te plaatsen (zie hiervoor het vak Gegevensstructuren.) Een knoop wortel maken is wel iets ingewikkelder, omdat de boom een binaire zoekboom moet blijven. Deze zogenaamde *splay-operatie* gebeurt dus opnieuw met rotaties.

De weg naar een diep liggende knoop bevat veel knopen die ook relatief diep liggen. Terwijl we die knoop wortel maken, zullen we tegelijkertijd de structuur van de boom zo moeten aanpassen, dat ook de toegangstijd van deze knopen verbetert, zoniet blijft de kans bestaan dat een reeks operaties inefficiënt is.

Bij splay trees moet er geen hoogte van knopen of andere informatie over het evenwicht van de boom (zoals kleuren) bijgehouden worden. Dit spaart geheugen uit, en zorgt meestal voor meer eenvoudige implementaties. Bemerk echter dat ook zoeken de boom kan herstructureren. Net zoals bij rood-zwarte bomen bestaan er 'top-down' en 'bottom-up' versies van splay trees.

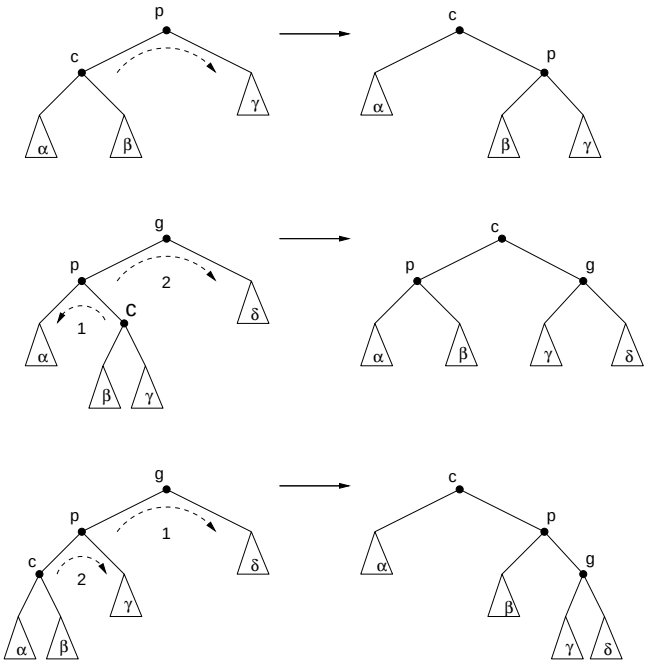
1.3.1 Bottom-up splay trees

Eerst zoeken we de knoop zoals bij een gewone zoekboom, daarna gebeurt de splay-operatie van onder naar boven. We hebben dus opnieuw ouderwijzers of een stapel

nodig.

Een knoop kan naar boven gebracht worden door hem telkens met zijn ouder op de zoekweg te roteren. Dit is echter niet voldoende om steeds een performantie van $O(m \log n)$ te verzekeren. Weliswaar wordt de knoop zelf wortel, maar de situatie voor de andere knopen op de zoekweg is nauwelijks verbeterd. Het is dan ook mogelijk om een reeks van m operaties te vinden die $\Theta(mn)$ duurt. (Voeg bijvoorbeeld de getallen $1, 2, 3, \ldots, n$ toe aan een ledige boom. Het resultaat is een gelinkte lijst, efficiënt opgebouwd in $O(n)$. De knopen in dezelfde volgorde opzoeken vereist echter $\Theta(n^2)$ bewerkingen, en bovendien is de boom na afloop weer precies dezelfde gelinkte lijst, waarna we van vooraf aan kunnen herbeginnen.)

We zullen de rotaties dus zorgvuldiger moeten aanpakken. Nog steeds gebeuren die van onder naar boven langs de zoekweg voor een knoop c, tot die wortel is. Maar we onderscheiden nu de volgende mogelijkheden (zie figuur 1.12):



Figuur 1.12. Bottom-up splay.

(1) *De ouder van c is wortel.* Dan roteren we beide knopen. Dit eenvoudig geval noemt men 'zig'.
(2) *Knoop c heeft nog een grootouder.* Er zijn dan *vier* gevallen, die uiteenvallen in twee groepen van twee, naar gelang dat ouder p een linker- of rechterkind is van grootouder g. Eens te meer zijn de operaties elkaars spiegelbeeld, en behandelen

we slechts één groep.
Stel dat p linkerkind is van g. Dan zijn er twee mogelijkheden:

a. *Knoop c is rechterkind van p.* We voeren dan twee rotaties uit, een in elke richting: eerst p en c naar links, en dan g en c naar rechts. Daarom heet dit geval 'zig-zag'.
b. *Knoop c is linkerkind van p.* Opnieuw roteren we tweemaal, telkens naar rechts: maar nu eerst g en p (de *bovenste* knopen), daarna p en c. (Bemerk dat dit het enige verschil is met de vorige methode.) Omdat de drie knopen op één lijn lagen noemt men dit geval 'zig-zig'.

Hoewel het effect moeilijk te zien is op kleine voorbeelden, wordt de diepte van de meeste knopen op de toegangsweg nu ongeveer gehalveerd. Slechts enkele ondiepe knopen zakken hoogstens twee niveau's. Het is instructief om deze strategie eens toe te passen op het voorbeeld van hierboven.

Het specifiek gedrag van splay trees wordt slechts zichtbaar bij grotere bomen: inefficiënte operaties worden gevolgd door rotaties die gunstig zijn voor de volgende operaties, terwijl er na efficiënte operaties minder goede of zelfs slecht uitvallende rotaties kunnen gebeuren. Maar steeds blijft de performantie van de reeks goed.

Met deze splay-operatie verlopen de woordenboekoperaties als volgt:

- Zoeken gebeurt zoals bij een gewone zoekboom. Daarna wordt de laatste knoop op de zoekweg wortel, via een splay-operatie. Deze wortel bevat dan ofwel de gezochte sleutel, ofwel zijn voorloper of opvolger.
- Ook toevoegen gebeurt zoals bij een gewone binaire zoekboom. Daarna wordt de nieuwe knoop wortel, via een splay-operatie.

  Een alternatieve manier die (geamortiseerd) iets trager uitvalt, zoekt eerst de toe te voegen sleutel, zoals hierboven. De wortel bevat daarna ofwel de sleutel, ofwel zijn voorloper of opvolger. Als de sleutel niet aanwezig is (duplicaten voegen we immers niet toe), slaan we die op in een nieuwe wortelknoop, met als linkerkind (rechterkind) de voorloper (opvolger) met zijn linkse (rechtse) deelboom, en als rechterkind (linkerkind) zijn rechtse (linkse) deelboom.
- Ook verwijderen kan gebeuren zoals bij een gewone zoekboom. Daarna wordt de *ouder* van de fysisch verwijderde knoop wortel, via een splay-operatie. Als de sleutel niet gevonden wordt, zal toch weer de laatste knoop op de zoekweg wortel worden.

  Een alternatieve methode die (geamortiseerd) iets trager uitvalt, zoekt eerst de knoop met de te verwijderen sleutel, via een splay-operatie. Als daarna de wortel de gezochte sleutel bevat verwijdert men hem, en worden zijn deelbomen samengevoegd. Samenvoegen van splay trees (een 'join'-operatie) gebeurt door eerst het grootste (kleinste) element in de linkse (rechtse) deelboom te zoeken en dus wortel te maken, via een splay-operatie. Deze wortel heeft dan geen rechterkind (linkerkind), en op die plaats komt de andere deelboom terecht.

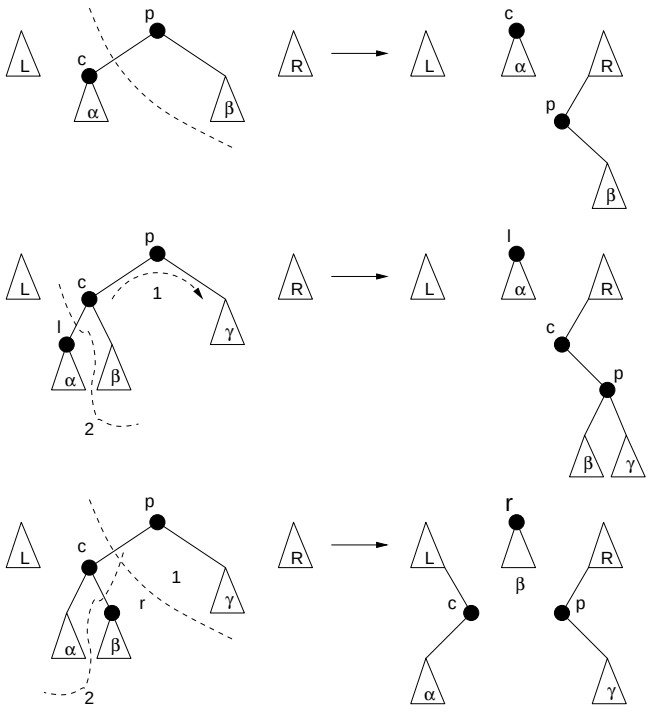
1.3.2 Top-down splay trees

De 'top-down' versie verricht de splay-operatie tijdens de afdaling, zodat de gezochte knoop wortel zal zijn als we hem bereiken. Er zijn dus geen ouderwijzers of stapel nodig, en in de praktijk blijkt deze versie eenvoudiger te implementeren en iets efficiënter.

Tijdens de afdaling wordt de boom in drie zoekbomen opgedeeld, zodanig dat alle sleutels in de linkse boom L kleiner zijn dan de sleutels in de middelste boom M, en alle sleutels in de rechtse boom R groter zijn dan die in M. Initieel is M de oorspronkelijke boom, en zijn L en R ledig. Het zoeken (naar een sleutel, naar de plaats voor een nieuwe sleutel, of naar een te verwijderen sleutel) begint dan bij de wortel van M, en men zorgt ervoor dat de huidige knoop op de zoekweg steeds wortel blijft van M.

Stel dat we op weg naar beneden bij knoop p terechtgekomen zijn (die dan wortel van M geworden is), en dat we nog verder moeten (en kunnen). Afhankelijk van de richting die we uit moeten zijn er twee groepen van mogelijke gevallen, met eens te meer gespiegelde operaties, zodat we slechts één groep moeten behandelen.

Stel dat we vanuit p naar zijn linkerkind c moeten. Dan zijn er twee hoofdgevallen (zie figuur 1.13):



Figuur 1.13. Top-down splay.

(1) *Linkerkind c is de laatste knoop op de zoekweg.* Dat doet zich voor ofwel als we c zoeken, ofwel omdat c minstens één kind mist, en we net die kant op moeten. (Dat is dus zeker het geval als c virtueel is.) Knoop p wordt dan het nieuwe kleinste element in R (de rechtse deelboom van p gaat mee), en de linkse deelboom van p wordt de nieuwe M (met c als wortel).
Dit eenvoudigste geval noemt men opnieuw 'zig'.

(2) *Linkerkind c is niet de laatste knoop op de zoekweg.* Er zijn dan nog twee gevallen, naar gelang dat we afdalen naar het linkerkind l of het rechterkind r van c:

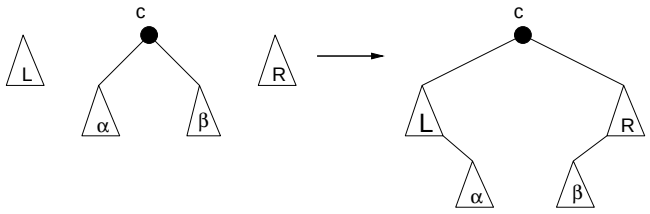
   a. *Naar het linkerkind l van c.* Dan *roteren* we eerst p en c naar rechts (de *bovenste* knopen eerst, zoals bij de bottom-up versie), daarna wordt c het nieuwe kleinste element van R (de nieuwe rechtse deelboom van c gaat mee). De linkse deelboom van c wordt de nieuwe M (met l als wortel).
   Aangezien de drie knopen p, c en l op één lijn lagen noemt men dit geval opnieuw 'zig-zig'.

   b. *Naar het rechterkind r van c.* Dan wordt p het nieuwe kleinste element van R (de rechtse deelboom van p gaat mee). Daarna wordt c het nieuwe grootste element in L (de linkse deelboom van c gaat mee), en de rechtse deelboom van c wordt de nieuwe M (met r als wortel).
   Hier lagen de drie knopen p, c en r niet op één lijn, zodat dit geval weer 'zig-zag' heet.
   Aangezien er geen rotatie gebeurt, kan men dit geval voor de eenvoud ook behandelen als een 'zig'. Dan wordt p nog steeds het nieuwe kleinste element van R (de rechtse deelboom van p gaat mee), maar de linkse deelboom van p wordt nu de nieuwe M (met c als wortel). Het voordeel van deze vereenvoudiging is dat er wat minder gevallen te onderscheiden zijn, het nadeel is dat we slechts één niveau in de boom zijn afgedaald in plaats van twee (er zijn dus meer iteraties nodig).

Als uiteindelijk de gezochte knoop c wortel van M geworden is, voltooien we de splay-operatie door de drie deelbomen samen te voegen tot één boom met wortel c (zie figuur 1.14). Alle sleutels in de linkse deelboom van c zijn groter dan de sleutels in L. We kunnen deze deelboom dus volledig onderbrengen in L, met zijn wortel op de meest rechtste plaats in L. En analoog komt de rechtse deelboom van c terecht in R, met zijn wortel op de meest linkse plaats in R. Tenslotte wordt deze gewijzigde L de nieuwe linkerdeelboom van c, en de gewijzigde R zijn nieuwe rechterdeelboom. De woordenboekoperaties verlopen nu als volgt:

- Zoeken maakt de knoop met de sleutel wortel (en als de sleutel niet gevonden wordt, dan de knoop met zijn voorloper of opvolger).

- Toevoegen gebeurt analoog met de alternatieve versie van bottom-up toevoegen. De voorloper of opvolger van de nieuwe sleutel wordt wortel (duplicaten worden niet toegevoegd), en de nieuwe knoop met de toegevoegde sleutel krijgt als linkerkind (rechterkind) de voorloper (opvolger) met zijn linkse (rechtse) deel-



Figuur 1.14. Samenvoegen na top-down splay.

boom, en als rechterkind (linkerkind) zijn rechtse (linkse) deelboom.

- Ook verwijderen gebeurt analoog met de alternatieve versie van bottom-up verwijderen. Eerst zoekt men de sleutel, en als hij gevonden wordt, en dus bij de wortel staat, dan verwijdert men die wortel en voegt zijn deelbomen samen (weer via een 'join'-operatie).

Na een top-down operatie is de boom niet noodzakelijk dezelfde als na de analoge bottom-up operatie, maar de geamortiseerde efficiëntie van alle operaties is opnieuw $O(\lg n)$.

1.3.3 Performantie van splay trees

De performantie-analyse van splay trees is niet zo eenvoudig, omdat de vorm van de bomen voortdurend verandert. Om aan te tonen dat elke reeks van m operaties op een splay tree met maximaal n knopen een performantie van $O(m \lg n)$ heeft, en dus een geamortiseerde efficiëntie van $O(\lg n)$ per operatie, gebruikt men een *potentiaalfunctie* Φ. Omdat de vorm van de boom kan veranderen, krijgt elke mogelijke vorm een reëel getal toegewezen, zijn potentiaal. Operaties op de boom kunnen zijn vorm en dus ook zijn potentiaal wijzigen. Het is de bedoeling dat efficiënte operaties, die minder tijd gebruiken dan de geamortiseerde tijd per operatie, de potentiaal verhogen. Inefficiënte operaties spreken deze reserve aan en doen de potentiaal dalen. De geamortiseerde tijd van een individuele operatie wordt gedefinieerd als de som van haar werkelijke tijd en de toename van de potentiaal. Als t_i de werkelijke tijd van de i-de operatie voorstelt, a_i haar geamortiseerde tijd, en Φ_i de potentiaal na deze operatie, dan is $a_i = t_i + \Phi_i - \Phi_{i-1}$. De geamortiseerde tijd van een reeks van m operaties is de som van de individuele geamortiseerde tijden:

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} (t_i + \Phi_i - \Phi_{i-1})$$

In deze som komen de meeste potentialen met tegengestelde tekens voor, zodat

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} t_i + \Phi_m - \Phi_0$$

waarbij Φ_0 de beginpotentiaal voorstelt. Als we de potentiaalfunctie zo kiezen dat de eindpotentiaal niet kleiner is dan de beginpotentiaal, dan is de totale geamortiseerde tijd een *bovengrens* voor de werkelijke tijd van de reeks operaties.

Een geschikte potentiaalfunctie vinden die de berekeningen eenvoudig houdt en toch een zinvol resultaat oplevert, is niet altijd eenvoudig, en vaak moet men verschillende mogelijkheden uitproberen. De eenvoudigste keuze voor dit geval geeft elke knoop i een gewicht s_i gelijk aan het aantal knopen in de deelboom waarvan hij wortel is (de knoop zelf inbegrepen), en neemt de som van de logaritmen van al deze gewichten:

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

(Met een andere geschikte keuze voor de gewichten kan men bijkomende eigenschappen aantonen.) Voor de eenvoud noteert men $\lg s_i$ als r_i, de *rang* van de knoop i. De potentiaal van een ledige boom onderstellen we nul.

De performantie-analyse van bottom-up en top-down splay trees is volledig analoog. We beperken ons daarom tot de bottom-up versie. De performantie van een splay-operatie op een knoop is evenredig met de diepte van die knoop, en dus met het aantal uitgevoerde (enkelvoudige) rotaties. (Een zig verricht één rotatie, zowel zig-zag als zig-zig verrichten er twee.)

We zullen trachten aan te tonen dat de geamortiseerde tijd voor het zoeken naar een knoop c gevolgd door een splay-operatie op die knoop gelijk is aan $O(1+3(r_w - r_c))$, waarbij r_w de rang van de wortel en r_c de rang van knoop c in de originele boom voorstelt. (In de uiteindelijke boom is de rang van c natuurlijk gelijk aan r_w.)

Als c de wortel is, dan moeten we niet afdalen in de boom en zijn er geen rotaties nodig. De werkelijke tijd is dan $O(1)$, en de potentiaal blijft gelijk, zodat dit triviaal geval alvast aan de eigenschap voldoet. Als we wel moeten afdalen, dan is het aantal enkelvoudige rotaties dat de splay-operatie zal uitvoeren gelijk aan de diepte van de knoop. De werkelijke tijd om één niveau te dalen en later de overeenkomstige (enkelvoudige) rotatie uit te voeren is $O(1)$. Omdat er een onbekend aantal zig-zags en/of zig-zigs gebeuren, eventueel gevolgd door één zig, bepalen we eerst de geamortiseerde tijd a voor de drie gevallen apart. Aangezien telkens dezelfde knoop c één of twee niveau's stijgt, trachten we resultaten te bekomen waarin enkel de rang van c vóór en na de operatie voorkomt. De geamortiseerde tijd van de volledige splay-operatie is immers de som van de afzonderlijke tijden, en omdat de rang van c na een operatie gelijk is aan de rang van c vóór de volgende operatie, hopen we dat de meeste termen in die som wegvallen.

Het gewicht en de rang van knoop i vóór een operatie noemen we s_i en r_i, na de operatie s'_i en r'_i. Vóór de operatie is p de ouder van c, en g de eventuele grootouder. Bij de berekeningen maken we gebruik van het feit dat de logaritmische functie concaaf is: voor positieve u en v geldt dat $\lg((u+v)/2) \geq (\lg u + \lg v)/2$. Dit kunnen we ook schrijven als

$$2\lg(u+v) \geq 2 + \lg u + \lg v.$$

- Een zig kan enkel de rang van c en p wijzigen, zodat

  $$a = 1 + r'_c + r'_p - r_c - r_p$$

  of ook, aangezien $r'_p < r_p$

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

- De dubbele rotatie van een zig-zag kan enkel de rang van c, p en g wijzigen, zodat

  $$a = 2 + r'_c + r'_p + r'_g - r_c - r_p - r_g$$

  of, aangezien $r'_c = r_g$

  $$a = 2 + r'_p + r'_g - r_c - r_p.$$

  Om $r'_p + r'_g$ af te schatten nemen we in onze formule $u = s'_p$ en $v = s'_g$. Dit levert op dat

  $$2\lg(s'_p + s'_g) \geq 2 + r'_p + r'_g.$$

  Maar $s'_c > s'_p + s'_g$, en dus $r'_c > \lg(s'_p + s'_g)$, zodat $2r'_c > 2 + r'_p + r'_g$. Dus wordt

  $$a < 2r'_c - r_c - r_p$$

  en omdat $r_p > r_c$ bekomen we uiteindelijk dat

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

- Ook de dubbele rotatie van een zig-zig kan enkel de rang van c, p en g wijzigen, zodat opnieuw

  $$a = 2 + r'_c + r'_p + r'_g - r_c - r_p - r_g$$

  en weer met $r'_c = r_g$ wordt dit

  $$a = 2 + r'_p + r'_g - r_c - r_p$$

  Om dit te vereenvoudigen stellen we $u = s'_g$, $v = s_c$ en maken we gebruik van $s'_c > s'_g + s_c$, en met dezelfde eigenschap als hierboven wordt

  $$2r'_c > 2\lg(s'_g + s_c) \geq 2 + r'_g + r_c$$

  zodat $r'_g < 2r'_c - r_c - 2$. We krijgen dan

  $$a < r'_p + 2r'_c - 2r_c - r_p$$

  Met $r'_c > r'_p$ en $r_p > r_c$ bekomen we tenslotte dat

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

De bovengrenzen voor de drie soorten operaties bevatten dezelfde positieve term $r'_c - r_c$ (want $r'_c > r_c$), maar met verschillende coëfficiënten. De totale geamortiseerde tijd is een som van dergelijke tijden, die we maar kunnen vereenvoudigen als de coëfficiënten gelijk zijn. Omdat het hier gaat om bovengrenzen, maken we alle coëfficiënten gelijk aan de grootste waarde, die van de zig-zig operatie. In de som vallen de meeste termen nu weg, behalve de rang van c vóór en na de volledige splay-operatie. Op het einde is c wortel, met als rang inderdaad r_w. Daarmee is de eigenschap aangetoond.

We kunnen nu de geamortiseerde tijden van de woordenboekoperaties op een (bottom-up) splay tree met n knopen bepalen:

- De geamortiseerde tijd voor zoeken gevolgd door een splay-operatie is $O(1 + 3 \lg n)$, want s_w is gelijk aan n, en voor een blad is s_c één.
- Toevoegen (we nemen de snelste versie) daalt eerst af net als zoeken, en wijzigt met de nieuwe knoop enkel de rang van de knopen $p_1, p_2, \ldots, p_k$ op de zoekweg. Als s_{p_i} het gewicht van knoop p_i vóór het toevoegen van de nieuwe knoop voorstelt, en s'_{p_i} het gewicht erna, dan is de potentiaaltoename gelijk aan

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

  Nu is zowel $s_{p_{i-1}} \geq s_{p_i} + 1$ als $s'_{p_i} = s_{p_i} + 1$ en dus ook $s_{p_{i-1}} \geq s'_{p_i}$, zodat deze potentiaaltoename niet groter is dan

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

  De geamortiseerde tijd van toevoegen is dus $O(1 + 4 \lg n)$.
- Een knoop fysisch verwijderen doet de rang van de knopen op de zoekweg dalen, zodat het effect nooit positief is.

De woordenboekoperaties (splay-operaties inbegrepen) zijn dus allemaal zeker $O(1 + 4 \lg n)$.

Tenslotte is de geamortiseerde tijd voor een reeks van m woordenboekoperaties de som van de geamortiseerde tijden voor de individuele operaties. Met n_i het aantal knopen bij de i-de operatie wordt die tijd $O(m + 4 \sum_{i=1}^{m} \lg n_i)$, en als de boom maximaal n knopen bevat is dat resultaat zeker $O(m + 4m \lg n)$ of eenvoudiger $O(m \lg n)$.

1.4 RANDOMIZED SEARCH TREES

De verwachtingswaarde van de performantie van de woordenboekoperaties op een gewone zoekboom is $O(\lg n)$ als elke toevoegvolgorde even waarschijnlijk is. Dat is

niet zeer realistisch. Bovendien geldt dit niet noodzakelijk wanneer men toevoegen afwisselt met verwijderen.

Randomized search trees ([36], [29]) maken gebruik van een random generator om het effect van de operatievolgorde te neutraliseren. Deze bomen blijven random, ook na elke operatie. De verwachtingswaarde van hun performantie hangt immers enkel af van de kwaliteit van de (hopelijk goede) random generator, en is dus steeds $O(\lg n)$.

Een treap (Aragon en Seidel, 1989) is de eenvoudigste boom van deze familie, en combineert een binaire zoekboom met een heap ('tree' + 'heap' = 'treap'). Elke knoop krijgt naast een sleutel (met eventueel bijbehorende informatie) ook nog een *prioriteit*, die bij het toevoegen door een random generator toegekend wordt. Een treap is een binaire zoekboom waarbij de prioriteit van de knopen ook nog aan de *heapvoorwaarde* voldoet: de prioriteit van een kind is hoogstens even hoog als die van zijn ouder. (Een treap moet echter niet voldoen aan de *structuur*voorwaarde van een binaire heap: het is niet noodzakelijk een complete binaire boom.)

Wanneer alle sleutels verschillend zijn, en ook alle prioriteiten, dan is er met deze gegevens slechts één treap mogelijk. (De knoop met de hoogste prioriteit moet immers steeds de wortel zijn, en zijn sleutel verdeelt de sleutels in twee groepen. Dit geldt recursief voor elk van de deelbomen.) Met als belangrijk gevolg dat de vorm van een treap niet afhangt van de toevoegvolgorde van de sleutels, maar van de random gegenereerde prioriteiten. Bij een goede random generator is elke reeks van n prioriteiten even waarschijnlijk, zodat een treap een random binaire zoekboom is.[2]

De woordenboekoperaties verlopen als volgt:

- Zoeken houdt natuurlijk geen rekening met de prioriteiten, en is dus identiek aan zoeken in een gewone binaire zoekboom.
- Toevoegen van een knoop gebeurt als blad, zoals bij een gewone zoekboom. Pas daarna houdt men rekening met de prioriteiten, en wordt de knoop indien nodig naar boven geroteerd, om aan de heapvoorwaarde te voldoen, desnoods tot bij de wortel. Stijgen vereist natuurlijk ouderwijzers of een stapel. Elke rotatie herstelt de heapvoorwaarde tussen een kind en zijn ouder, zonder gevolgen voor de heapvoorwaarde elders in de boom (behalve eventueel hoger op de zoekweg, bij de grootouder). In tegenstelling tot rood-zwarte bomen en zelfs splay trees is het aantal gevallen hier dus heel beperkt.
- Een te verwijderen knoop krijgt de laagst mogelijke prioriteit, zodat hij naar beneden kan geroteerd worden, en als blad uit de treap verdwijnt. Ook hier zijn er dus maar twee gevallen te onderscheiden.

De verwachtingswaarde van de efficiëntie van de woordenboekoperaties op treaps is $O(\lg n)$, wat beter is dan bij gewone zoekbomen, aangezien dat resultaat niet afhangt

[2] Strikt genomen moeten alle prioriteiten dus verschillend zijn, maar daarvoor zorgen heeft meer nadelen dan voordelen. Overigens is de hoogste prioriteit vaak het kleinste getal.

van de operatievolgorde. Hun performantie is niet zo goed als die van evenwichtige zoekbomen (want die halen dat resultaat in het slechtste geval), maar hun implementatie is veel eenvoudiger, onder meer omdat de prioriteit van een knoop nooit moet aangepast worden (in tegenstelling tot kleur of informatie over hoogte). Ook ten opzichte van splay trees kunnen treaps eenvoudiger en sneller uitvallen, omdat splay trees vaak veel rotaties uitvoeren (zelfs bij zoeken). Zoeken in een treap vereist immers geen rotaties, en men kan aantonen dat zowel toevoegen als verwijderen gemiddeld minder dan twee rotaties uitvoeren. Treaps kunnen echter geen goede performantie voor een reeks operaties garanderen.

1.5 SKIP LISTS

Conceptueel is een *skip list* een meerwegszoekboom, alhoewel de implementatie meestal lijkt op een verzameling gelinkte lijsten. Alle bladeren zitten op dezelfde diepte, zodat de boom evenwichtig is.

Vermits elke knoop een onbekend aantal sleutels kan hebben wordt hij geïmplementeerd als een gelinkte lijst met in elke lijstknoop plaats voor één sleutel en één kindwijzer. Zoals we gezien hebben bij B-trees bevat een knoop met k kinderen maar $k-1$ sleutels, zodat er een sleutelplaats op overschot is (bij conventie in de eerste lijstknoop). Deze wordt dan voorzien van een kopie van de voorganger in de ouder.

De laatste lijstknoop heeft normaal een nulpointer als opvolgerpointer. Deze wordt gebruikt linkerkind van de rechterbroer van zijn ouder. Hierdoor vormen de lijstknopen op elk niveau samen één gelinkte lijst. De laagste lijst, met de bladeren, omvat alle sleutels, de lijsten op hoger niveau bevatten dus maar een deel van de knopen (hoe hoger het niveau, hoe minder knopen). Zoeken in een hoger niveau komt dus overeen met het overslaan van een aantal knopen op het laagste niveau (vandaar de naam *skip* list). Bij toevoegen wordt aan de sleutel random een niveau toegekend. De sleutel wordt dan opgeslagen op dat niveau en alle onderliggende niveaus.

HOOFDSTUK 2
TOEPASSINGEN VAN DYNAMISCH PROGRAMMEREN

2.1 OPTIMALE BINAIRE ZOEKBOMEN

Stel dat de gegevens die in een binaire zoekboom moeten opgeslagen worden op voorhand gekend zijn, en niet meer wijzigen. Stel dat we ook weten met welke waarschijnlijkheid die gegevens gezocht zullen worden, en ook de afwezige gegevens, dan kunnen we trachten om de boom zo te schikken, dat de verwachtingswaarde van de zoektijd minimaal wordt.

De zoektijd wordt bepaald door de lengte van de zoekweg: voor een aanwezig gegeven is dat de diepte van de knoop, voor een afwezig gegeven de diepte van de ledige deelboom (een nullpointer). Daarom zullen we alle ledige deelbomen als bladeren van de zoekboom beschouwen. Alle aanwezige gegevens zijn dan in inwendige knopen opgeslagen.

De (gerangschikte) sleutels van de n aanwezige gegevens noemen we $s_1, s_2, \ldots, s_n$, de $n+1$ bladeren $b_0, b_1, \ldots, b_n$. Elk blad staat voor alle afwezige gegevens die in een deelboom op die plaats hadden moeten zitten: b_0 staat voor alle sleutels kleiner dan s_1, b_n voor alle sleutels groter dan s_n, en b_i voor alle sleutels groter dan s_i en kleiner dan s_{i+1}, voor $1 \leq i < n$.

De waarschijnlijkheid waarmee naar de i-de aanwezige sleutel s_i gezocht wordt noemen we p_i. Het heeft weinig zin om de waarschijnlijkheid te kennen waarmee elk afwezig gegeven gezocht wordt. Voor veel afwezige sleutels zal de zoekweg immers bij hetzelfde blad eindigen. We moeten dus enkel de waarschijnlijkheid q_i kennen waarmee alle afwezige sleutels voorgesteld door blad b_i kunnen gezocht worden. Omdat zoeken dan steeds een inwendige knoop of een blad vindt, is

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

Als we de zoektijd van een knoop of blad gelijk stellen aan het aantal knopen op de zoekweg (de diepte plus één), dan is de verwachtingswaarde van de zoektijd

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

We moeten nu een zoekboom vinden die deze uitdrukking minimaliseert.

Tegenvoorbeelden tonen aan dat een boom met minimale hoogte niet noodzakelijk optimaal is, en dat een inhalige methode die in de wortel van elke deelboom de sleutel met de grootste zoekwaarschijnlijkheid onderbrengt, ook niet altijd correct is. Alle mogelijke binaire zoekbomen onderzoeken is natuurlijk geen optie, aangezien hun aantal $\frac{1}{n+1}\binom{2n}{n} = \Omega(4^n/n^{3/2})$ bedraagt.

Gelukkig biedt dynamisch programmeren een uitkomst. Een optimalisatieprobleem komt in aanmerking voor een efficiënte oplossing via dynamisch programmeren, als het een optimale deelstructuur heeft, en de deelproblemen onafhankelijk maar overlappend zijn[1]. Laten we eens zien of dat hier het geval is:

- Als een zoekboom optimaal is, dan moet ook elk van zijn deelbomen optimaal zijn, want anders kunnen we een suboptimale deelboom vervangen door een beter exemplaar, wat een betere volledige boom oplevert. Een optimale oplossing bestaat dus uit optimale oplossingen voor deelproblemen. Zijn die deelproblemen ook onafhankelijk? Ja, want deelbomen hebben geen gemeenschappelijke knopen.

- Om te zien of de deelproblemen overlappend zijn, zodat veel optimale deelbomen meermaals in de oplossing gebruikt worden, moeten we eerst een oplossingsstrategie bedenken.

  Hoe kunnen we een optimale deelboom vinden? Elke deelboom bevat een reeks opeenvolgende sleutels $s_i, \ldots, s_j$ met bijbehorende bladeren $b_{i-1}, \ldots, b_j$. Een van die sleutels, bijvoorbeeld s_w ($i \leq w \leq j$), zal wortel zijn van de optimale deelboom. De linkse deelboom van s_w bevat dan de sleutels $s_i, \ldots, s_{w-1}$ met de bladeren $b_{i-1}, \ldots, b_{w-1}$, en de rechtse deelboom de sleutels $s_{w+1}, \ldots, s_j$ met de bladeren $b_w, \ldots, b_j$. (Als w gelijk is aan i, dan bevat de linkse deelboom de sleutels $s_i, \ldots, s_{i-1}$, wat natuurlijk betekent dat hij geen sleutels bevat, maar wel het blad b_{i-1}. Analoog, als w gelijk is aan j bevat de rechtse deelboom enkel het blad b_j.) Voor een optimale deelboom met wortel s_w moeten deze beide deelbomen zelf optimaal zijn. We vinden dus de optimale deelboom door achtereenvolgens elk van zijn sleutels $s_i, \ldots, s_j$ als wortel te kiezen, de zoektijd voor de boom te berekenen gebruik makend van de zoektijden van zijn (optimale) deelbomen, en het minimum bij te houden. Daarbij komen veel deelbomen meermaals voor, zodat het inderdaad de moeite loont om vooraf hun zoektijden te bepalen en op te slaan.

Concreter geformuleerd, komt elk deelprobleem neer op het bepalen van de kleinste verwachte zoektijd $z(i, j)$ voor een (deel)boom met sleutels $s_i, \ldots, s_j$ en bijbehorende bladeren $b_{i-1}, \ldots, b_j$. En dit moet gebeuren voor alle i en j zodat $1 \leq i \leq n+1$, $0 \leq j \leq n$, en $j \geq i-1$. (Als $j = i-1$ dan is de deelboom ledig, zie hoger.) De gezochte oplossing zal dan $z(1, n)$ zijn.

[1] Zie Grafen I in Algoritmen I.

Om $z_w(i,j)$ voor een deelboom met wortel s_w te vinden ($i \leq w \leq j$) maken we natuurlijk gebruik van de optimale zoektijden van zijn eigen deelbomen, $z(i, w-1)$ en $z(w+1, j)$. Deze zoektijden zijn echter berekend ten opzichte van de wortels van die deelbomen. Als kinderen van s_w wordt de diepte van al hun knopen (en bladeren) één groter. Hun bijdrage tot de zoektijd neemt dan toe met de som van de zoekwaarschijnlijkheden van al hun sleutels en bladeren. Voor een willekeurige deelboom met sleutels $s_i, \ldots, s_j$ is die som

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

Met s_w als wortel krijgen we dan dat

$$z_w(i,j) = p_w + (z(i, w-1) + g(i, w-1)) + (z(w+1, j) + g(w+1, j))$$

of eenvoudiger

$$z_w(i,j) = z(i, w-1) + z(w+1, j) + g(i,j)$$

Voor een optimale deelboom moet $z(i,j)$ minimaal zijn. We moeten dus achtereenvolgens elke sleutel van de deelboom tot wortel maken, en het minimum bepalen. De index w doorloopt daarvoor alle waarden tussen i en j (inbegrepen):

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

Dat minimum bepalen heeft enkel zin wanneer $i \leq j$, dus als de deelboom sleutels bevat. Wanneer $j = i - 1$ heeft de deelboom geen sleutels, maar enkel het blad b_{i-1}. In dat geval is $z(i, i-1) = q_{i-1}$. (Idem als $i = j + 1$: $z(j+1, j) = q_j = q_{i-1}$.)

Nu zijn we niet enkel geïnteresseerd in de minimale verwachte zoektijd, maar natuurlijk ook in de boom zelf. Daartoe volstaat het de index w van de wortel van elke optimale deelboom bij te houden. Voor de deelboom met sleutels $s_i, \ldots, s_j$ ($1 \leq i \leq j \leq n$) noemen we die index $r(i,j)$.

Een recursieve implementatie van $z_w(i,j)$ zou veel deeloplossingen meermaals bepalen. Ook bij de berekening van alle waarden $g(i,j)$ komen veel deelsommen meermaals voor. Daarom bepalen we de deeloplossingen bottom-up en slaan we de resultaten op in tabellen, zodat ze hogerop beschikbaar zijn. De implementatie van het volledige algoritme gebruikt drie tabellen:

- Een tweedimensionale tabel $z[1..n+1, 0..n]$ voor de waarden $z(i,j)$. Beide grenzen gaan niet van 1 tot n, omdat we plaats nodig hebben voor $z(1,0)$ (een deelboom met enkel blad b_0), en voor $z(n+1, n)$ (een deelboom met enkel blad b_n). Een driehoeksmatrix zou eigenlijk volstaan, want steeds is $j \geq i - 1$.
- Een tweedimensionale tabel $g[1..n+1, 0..n]$ voor de waarden $g(i,j)$, om efficiëntieredenen. Immers, voor de bepaling van $z(i,j)$ zoeken we het minimum

voor alle mogelijke indices w van de wortel, waarbij telkens dezelfde waarde $g(i,j)$ voorkomt. Die steeds opnieuw berekenen zou veel onnodig werk betekenen. Om deze tabel (nóg een driehoeksmatrix) in te vullen maken we gebruik van de betrekkingen $g(i,j) = g(i,j-1)+p_j+q_j$ (voor $i \leq j$) en $g(i,i-1) = q_{i-1}$.

- Een tweedimensionale tabel $r[1..n, 1..n]$ voor de indices $r(i,j)$ van de wortels van de optimale deelbomen. Ook hier volstaat een driehoeksmatrix, want steeds geldt dat $1 \leq i \leq j \leq n$.

Het algoritme moet ervoor zorgen dat de waarden $z(i,j)$ en $g(i,j)$ in de juiste volgorde bepaald worden. Voor beide is dat mogelijk door eerst alle elementen $z(i,i-1)$ en $g(i,i-1)$ op de diagonaal in te vullen (de triviale gevallen), en dan achtereenvolgens de elementen op elke evenwijdige diagonaal, in de richting van de tabelhoek linksboven. Immers, voor $z(i,j)$ zijn alle mogelijke waarden $z(i,i-1), z(i,i), \ldots, z(i,j-1)$ van linkse deelbomen nodig, en ook alle mogelijke waarden $z(i+1,j), \ldots, z(j,j), z(j+1,j)$ van rechtse deelbomen. En al die waarden staan op diagonalen onder deze van $z(i,j)$. Voor $g(i,j)$ volstaat één waarde $g(i,j-1)$ op de diagonaal er onder. Aangezien $z(i,j)$ gebruik maakt van $g(i,j)$ moet deze waarde eerst bepaald worden. De pseudocode 2.1 geeft wat meer details.

Rest nog de performantie. Aangezien de drie vernestelde herhalingen elk niet meer dan n iteraties kunnen uitvoeren, is dit algoritme zeker $O(n^3)$. Maar wat is de ondergrens? Het meeste werk gebeurt in de binnenste herhaling, waar alle mogelijke wortels voor elke deelboom getest worden. Een deelboom met sleutels $s_i, \ldots, s_j$ ($1 \leq i \leq j \leq n$) heeft $j-i+1$ mogelijke wortels, en elke test is $O(1)$. Dat werk is dus evenredig met $\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i+1)$, en omdat $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$, ook evenredig met n^3. De totale performantie is dus ook $\Theta(n^3)$. Door gebruik te maken van een bijkomende eigenschap kan men het aantal te testen wortels sterk reduceren, zodat de performantie $\Theta(n^2)$ wordt (Knuth, 1971, zie [26]).

2.2 LANGSTE GEMEENSCHAPPELIJKE DEELSEQUENTIE

Een deelsequentie van een string bekomt men door nul of meer stringelementen weg te laten. Elke deelstring is dus een deelsequentie, maar niet omgekeerd.

Een[2] langste gemeenschappelijke deelsequentie (LGD) van twee strings zoeken kan gebruikt worden om teksten te vergelijken (bijvoorbeeld om plagiaat op te sporen), of om gelijkaardige maar niet gelijke DNA-strings te identificeren.

Gegeven dus twee strings $X = \langle x_0, x_1, \ldots, x_{n-1}\rangle$ en $Y = \langle y_0, y_1, \ldots, y_{m-1}\rangle$. Een brutekrachtmethode test bijvoorbeeld alle 2^n deelsequenties van X, gaat na of ze een deelsequentie zijn van Y, en houdt de langste bij. Voor lange strings is dit natuurlijk uitgesloten.

[2] We schrijven *een* en niet *de* omdat er verschillende even lange LGD's kunnen zijn.

```
// Gegeven: n, en de reele tabellen p[1..n] en q[0..n]
// Minimale verwachtingswaarde van de zoektijden
double z[1..n+1][0..n];
int g[1..n+1][0..n];
// Indices van wortelelementen van optimale deelbomen
int r[1..n][1..n];
// Initialisatie van de diagonalen van z en g
for(int i=1 ; i<=n+1 ; i++)
   z[i][i-1]=g[i][i-1]=q[i-1];
// Invullen van de tabellen g,z en r
for(int k=1 ; k<=n ; k++)
   for(int i=1 ; i<=n-k+1 ; i++){
      int j=i+k-1;
      g[i][j]=g[i][j-1]+p[j]+q[j];
      // Index w van wortelelement zoeken met minimum
      z[i][j]=oneindig;
      for(int w=i ; w<=j ; w++){
         double t=z[i][w-1]+z[w+1][j]+g[i][j];
         if(t<z[i][j]){
            z[i][j]=t;
            r[i][j]=w;
         }
      }
   }
// Resultaat: z[1][n], en de wortelindices in tabel r
```

Pseudocode 2.1. Bepalen van een optimale binaire zoekboom

Gelukkig heeft het LGD-probleem een *optimale deelstructuur*: een optimale oplossing maakt gebruik van optimale oplossingen voor deelproblemen. Deze deelproblemen zijn paren prefixen van de twee strings. Het prefix van X met lengte i noemen we X_i. Het ledige prefix is dan X_0. Analoog voor Y.

Als $Z = \langle z_0, z_1, \ldots, z_{k-1} \rangle$ een LGD is van X en Y, dan zijn er drie mogelijkheden:

- Als $n = 0$ of $m = 0$ dan is $k = 0$.
- Als $x_{n-1} = y_{m-1}$, dan is $z_{k-1} = x_{n-1} = y_{m-1}$, en is Z_{k-1} een LGD van X_{n-1} en Y_{m-1}.
- Als $x_{n-1} \neq y_{m-1}$ dan is Z een LGD van X_{n-1} en Y (met $z_{k-1} \neq x_{n-1}$) of een LGD van X en Y_{m-1} (met $z_{k-1} \neq y_{m-1}$).

Deze deelproblemen zijn ook duidelijk *onafhankelijk*. Maar deelproblemen kunnen

overlappen, omdat ze door meer dan één (hoger) deelprobleem gebruikt worden. Immers, om de LGD van X en Y te vinden kan het nodig zijn om de LGD te vinden van zowel X en Y_{m-1} als van X_{n-1} en Y. Maar beide deelproblemen gebruiken als deel-deelprobleem de LGD van X_{n-1} en Y_{m-1}. Nog veel andere deelproblemen hebben gemeenschappelijke deel-deelproblemen.

Een recursieve vergelijking opstellen voor de lengte van een LGD is nu eenvoudig. Als $c[i, j]$ de lengte is van de LGD van X_i en Y_j, dan is

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

Een programma dat deze recursie implementeert om de oplossing $c[n, m]$ te vinden heeft een exponentiële performantie. Er zijn echter maar $\Theta(nm)$ verschillende deelproblemen, zodat dynamisch programmeren kan gebruikt worden. Beide voorwaarden daarvoor zijn immers voldaan. De waarden $c[i, j]$ worden dus opgeslagen in een tweedimensionale $(n + 1) \times (m + 1)$ tabel c, die rij per rij van links naar rechts berekend wordt. De eerste rij en de eerste kolom worden geïnitialiseerd met nullen.

Omdat we niet alleen de lengte van de LGD wensen, maar ook de deelsequentie zelf, gebruiken we een tweedimensionale $n \times m$ tabel b. In element $b[i, j]$ houden we de plaats bij van de c-waarde die gebruikt werd om $c[i, j]$ te bepalen. Vertrekkend vanuit $b[n, m]$ vinden we dan de LGD in omgekeerde volgorde: telkens wanneer $c[i, j]$ gebruikt maakt van $c[i - 1, j - 1]$ is immers $x_i = y_j$.

		a	l	o	y	s	i	u	s
	0	0	0	0	0	0	0	0	0
l	0	← 0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1
o	0	← 0	↑ 1	↑ ↖ 2	← 2	← 2	← 2	← 2	← 2
u	0	← 0	↑ 1	↑ 2	← 2	← 2	← 2	↖ 3	← 3
i	0	← 0	↑ 1	↑ 2	← 2	← 2	↖ 3	← 3	← 3
s	0	← 0	↑ 1	↑ 2	← 2	↖ 3	← 3	← 3	↖ 4

De performantie van het algoritme wordt duidelijk bepaald door het invullen van beide tabellen. De reconstructie van de LGD zelf is immers $O(n + m)$, aangezien bij elke iteratie tenminste een van de indices i of j gedecrementeerd wordt. De totale performantie is dus $\Theta(nm)$. Ook de vereiste plaats is $\Theta(nm)$. Wanneer men echter enkel de *lengte* van de LGD wenst, volstaat tabel c, en aangezien elke rij enkel waarden van de

vorige rij gebruikt, volstaan twee rijen. (Zelfs niet veel meer dan één rij.) De vereiste plaats is dan $\Theta(\min(n, m))$.

Deze beperkte informatie is echter niet voldoende om de LGD te reconstrueren. Er bestaat een vernuftig algoritme, met dezelfde performantie, dat slechts $O(n+m)$ plaats vereist [21].

HOOFDSTUK 3
UITWENDIGE GEGEVENSSTRUCTUREN

Wanneer de omvang van de gegevens te groot is voor het intern geheugen, moeten ze in een extern geheugen opgeslagen worden (meestal magneetschijven). Hoe moeten deze gegevens nu georganiseerd worden, om er efficiënte woordenboekoperaties en (eventueel) sequentiële operaties (volgorde) op uit te voeren?

Er moet natuurlijk een prijs betaald worden voor de grote en goedkope opslagcapaciteit van de schijf. De toegangstijd tot een gegeven wordt nu niet meer bepaald door electronische eigenschappen, maar door de mechanische performantie van de schijf: haar rotatiesnelheid en de tijd nodig voor het verplaatsen van de lees/schrijfkop. Gemiddeld is deze tijd typisch enkele milliseconden lang, en processoren kunnen in dezelfde tijd honderdduizenden instructies uitvoeren. Deze situatie verbetert er niet op, omdat processoren steeds sneller worden, terwijl wel de opslagcapaciteit van schijven spectaculair stijgt, maar niet hun toegangstijd.

Eens de informatie gelokaliseerd werd, kunnen er grote hoeveelheden gegevens snel overgebracht worden. Lezen van en schrijven naar een schijf gebeurt dan ook in vrij grote informatieblokken (pagina's).

De performantie van een programma dat gebruik maakt van een schijf, wordt dus voornamelijk bepaald door het aantal schijfoperaties. Het loont dan ook de moeite om eventueel het algoritme ingewikkelder te maken, als dat maar resulteert in minder schijfoperaties.

Net zoals inwendige gegevensstructuren maken uitwendige gegevensstructuren gebruik van boomstructuren of van hashing. Wanneer de volgorde van de gegevens belangrijk is, gebruikt men meestal een uitwendige versie van een evenwichtige zoekboom, de B-tree.[1] Als volgorde geen rol speelt, is een uitwendige versie van een hashtabel een efficiënt alternatief. Het artikel van Vitter [40] geeft een overzicht.

3.1 B-TREES

Een 'B-tree' (Bayer en McCreight, 1972) is een uitwendige evenwichtige zoekboom.[2]

[1] Met de huidige grootte van de inwendige geheugens, gebruikt men zelfs een B-tree als *inwendige* gegevensstructuur.

[2] De naam werd nooit verklaard. Misschien komt de 'B' van 'Boeing', de toenmalige werkgever van de

De efficiëntie van de meeste operaties op zoekbomen wordt begrensd door de hoogte van de boom, en dus zorgt men ervoor dat een B-tree onder alle omstandigheden perfect in evenwicht blijft, en bovendien een zeer kleine hoogte heeft. Aangezien het aantal sleutels n zeer groot is, en de minimale hoogte van een binaire boom $\lfloor \lg n \rfloor$, zullen we bomen met meer dan twee kinderen per knoop moeten gebruiken als we een zeer kleine hoogte willen. Omdat bovendien de informatie-overdracht van en naar een schijf in pagina's gebeurt, nemen we knopen die een volledige schijfpagina beslaan, en we geven ze zoveel kinderen als ze kunnen bevatten. Zo krijgen we knoopgraden gaande van 50 tot enkele duizendtallen, afhankelijk van de sleutels en de schijf.

Een knoop van een B-tree moet men eerst in het inwendig geheugen inlezen, vooraleer er mee kan gewerkt worden. Gewijzigde knopen moet men nadien natuurlijk op de schijf aanpassen. Om schijfoperaties uit te sparen voorziet men gewoonlijk geheugenruimte voor de meest recent gebruikte knopen. Aangezien elke operatie slechts een gering aantal knopen gebruikt (de hoogte en dus de afgelegde weg in de boom zijn immers heel klein), zal zeker de wortel en soms ook het volledige eerste niveau van de boom in het geheugen aanwezig blijven.

De op te slagen gegevens bestaan zoals gewoonlijk uit een sleutel met bijbehorende informatie. Dikwijls wordt de sleutel enkel vergezeld van een wijzer naar de bijbehorende informatie, maar voor de eenvoud zullen we onderstellen dat sleutel en informatie samen blijven. De sleutels moeten ook geordend kunnen worden, zoals bij elke zoekboom.

Een B-tree is dus een meerwegszoekboom waarvan alle bladeren dezelfde diepte hebben. Bij toevoegen en verwijderen van sleutels wordt het perfecte evenwicht van de boom behouden door het aantal kinderen in de knopen te manipuleren. (Zoals bij zijn voorloper, de 2-3 boom.) Ingrijpende wijzigingen aan de boomstructuur zijn dan ook zeldzaam. (Gelukkig maar, gezien de traagheid van schijfoperaties.) Om te vermijden dat de boom te hoog wordt, zorgt men ervoor dat elke (inwendige) knoop steeds genoeg kinderen heeft.

3.1.1 Definitie

Een B-tree van orde m, waarbij $m > 2$, wordt als volgt gedefinieerd:

- Elke inwendige knoop heeft hoogstens m kinderen.
- Elke inwendige knoop, behalve de wortel, heeft minstens $\lceil m/2 \rceil$ kinderen. De wortel heeft minstens twee kinderen, tenzij hij een blad is.
- Elke inwendige knoop met $k+1$ kinderen bevat k sleutels. De bladeren bevatten hoogstens $m-1$ en minstens $\lceil m/2 \rceil - 1$ sleutels. Als de wortel een blad is bevat hij minstens één sleutel, tenzij de B-tree ledig is.
- Alle bladeren bevinden zich op hetzelfde niveau.

uitvinders. Omwille van deze en nog andere bijdragen zou men ze gerust 'Bayer-trees' kunnen noemen.

Elke knoop bevat dan:

- Een geheel getal k dat het huidig aantal *sleutels* in de knoop aanduidt. (Niet elke knoop heeft kinderen.)
- Een tabel voor maximaal m 'wijzers' naar de kinderen van de knoop. Bij een blad blijft deze tabel ongedefinieerd.
- Een tabel voor maximaal $m - 1$ sleutels, die stijgend (niet dalend) gerangschikt zijn. Een tweede tabel, van dezelfde grootte, met hun bijbehorende informatie, of een verwijzing ernaar. De k geordende sleutels van een inwendige knoop verdelen het sleutelbereik in $k + 1$ deelgebieden. De sleutels van elk deelgebied zitten in een deelboom met als wortel een kind van deze knoop. De sleutels uit de deelboom van het i-de kind c_i liggen dus tussen de sleutels s_{i-1} en s_i. (Deze in de deelboom van c_0 zijn natuurlijk kleiner dan s_0, en deze in de deelboom van c_k zijn groter dan s_{k-1}.)
- Een logische waarde b die aanduidt of de knoop een blad is of niet. Door het perfecte evenwicht is er geen behoefte aan 'nullwijzers' voor de ontbrekende kinderen van een blad: de logische waarde duidt immers aan of we ons op het laagste niveau bevinden.

De eenvoudigste B-trees zijn de 2-3 bomen (orde 3) en de 2-3-4 bomen (orde 4).[3] Normaal is m echter veel groter, zodat de hoogte van de boom (zeer) klein blijft. Bemerk dat de knopen plaats moeten reserveren voor de maximale grootte van hun tabellen. De waarde van m hangt dus af van de grootte van een schijfpagina, van de grootte van de sleutel en van die van zijn bijbehorende gegevens.

3.1.2 Eigenschappen

Zoals bij elke zoekboom is de langste afgelegde weg bij woordenboekoperaties op een B-tree evenredig met zijn hoogte. Toegang tot elke knoop op die weg vereist echter een schijfoperatie. (En een tweede als die gewijzigd wordt.) Gelukkig blijkt die hoogte zeer klein, ook voor heel veel sleutels.

Stel dat de boom niet ledig is, en hoogte h heeft. Wat is dan het minimaal aantal sleutels n dat hij bevat? (Bemerk dat, in tegenstelling tot binaire zoekbomen, het aantal sleutels *niet* gelijk is aan het aantal knopen.) De wortel van een minimale boom heeft slechts één sleutel, en dus twee kinderen (als $h > 0$). Elk van die kinderen heeft minimaal $g = \lceil m/2 \rceil$ kinderen, en die op hun beurt ook elk minimaal g kinderen, enz. Het aantal *knopen* is dus ten minste

$$1 + 2 + 2g + \cdots + 2g^{h-1} = 1 + 2\sum_{i=0}^{h-1} g^i.$$

[3] Bij de originele definitie van een B-tree was m oneven. Nog andere definities vereisen dat m even is.

Elke knoop heeft minstens $g-1$ sleutels, behalve de wortel, die heeft er minstens één. Er komt dan

$$n \geq 1 + 2(g-1)\left(\frac{g^h - 1}{g-1}\right)$$

zodat

$$n \geq 2g^h - 1$$

of tenslotte, na het nemen van de logaritme met basis $g = \lceil m/2 \rceil$ (een constante)

$$h \leq \log_{\lceil m/2 \rceil} \frac{n+1}{2}.$$

De hoogte is dus $O(\lg n)$, zoals bij een rood-zwarte boom, maar de (verborgen) constante is een factor $\lg\lceil m/2 \rceil$ kleiner.

Men kan aantonen dat een B-tree met n uniform verdeelde sleutels gemiddeld ongeveer $n/(m \ln 2)$ schijfpagina's gebruikt, zodat elke pagina gemiddeld voor ongeveer 69 procent gevuld is ($\ln 2 \approx 0.69$).

3.1.3 Woordenboekoperaties

3.1.3.1 Zoeken

Zoeken gebeurt langs een weg vanuit de wortel in de richting van een blad, zoals bij een binaire zoekboom. Natuurlijk moet er nu in elke knoop op die weg een meerwegsbeslissing genomen worden.

Elke knoop op die weg moet eerst in het geheugen ingelezen worden (de wortel uiteraard niet meer). De sleutel wordt dan opgezocht in de gerangschikte tabel met sleutels. De snelste manier is natuurlijk binair zoeken, maar de winst is vrij onbelangrijk naast de schijfoperaties. Lineair zoeken in een gerangschikte tabel kan overigens ook wat efficiënter uitvallen.[4]

Vinden we de sleutel, dan stopt het zoeken, met als resultaat een verwijzing naar de knoop op de schijf (de opslag in het geheugen is immers maar tijdelijk), en de plaats van de sleutel in die knoop. Vinden we de sleutel niet, en is de knoop een blad, dan is de sleutel niet in de boom aanwezig. Bij een interne knoop echter moeten we verder zoeken in de juiste deelboom, waarvan de wortel een kind is van de huidige knoop. De wijzer naar dat kind staat bij dezelfde index in de kindtabel als waar de niet gevonden sleutel zou moeten staan in de sleuteltabel. Dat kind wordt dan ingelezen, en het zoekproces wordt op die nieuwe knoop herhaald.

Tijdens het afdalen moet elke knoop (behalve de wortel) ingelezen worden. Het aantal schijfoperaties is dus $O(h) = O(\log_{\lceil m/2 \rceil} n)$. De processortijd per knoop is $O(m)$,

[4] Recent gebruikt men interpolerend zoeken om het aantal cachefouten te minimaliseren. Interpolerend zoeken werkt echter slecht als de sleutels niet uniform verdeeld zijn, zodat men allerlei technieken aanwendt om dit probleem te omzeilen [19].

en dus in totaal $O(m \log_{\lceil m/2 \rceil} n)$. Bij binair zoeken zou dat $O(\lg n)$ worden. De verborgen constante is echter veel kleiner dan die voor de schijfoperaties.

3.1.3.2 Toevoegen

Door het aantal sleutels te wijzigen kunnen aanpassingen aan de boomstructuur nodig worden. Net zoals bij rood-zwarte bomen kunnen die bottom-up of top-down gebeuren. In het slechtste geval vereist de bottom-up versie dat de hoogte van de boom tweemaal doorlopen wordt, wat veel erger is dan bij een inwendige boom. Dit slechtste geval is echter zeldzaam (zie hieronder), en als men bovendien de meest recente gebruikte knopen in het inwendig geheugen bijhoudt, vervalt dit nadeel. We bespreken hier dan ook de bottom-up versie, omdat die het meest gebruikt wordt.[5]

De initialisatie van een ledige B-tree vereist wel wat meer werk dan die van een inwendige zoekboom. Men moet de wortelknoop in het geheugen aanmaken en gedeeltelijk invullen, en dan op de schijf kopiëren. Een ledige wortel is een blad, dat bovendien geen sleutels bevat. Zoals voor elke nieuwe knoop moeten we er eerst plaats voor reserveren op de schijf, vooraleer we kunnen kopiëren. Een verwijzing naar die plaats moet ook permanent bijgehouden worden. (Een andere knoop kan immers wortel worden.)

Toevoegen gebeurt steeds aan een blad want toevoegen aan een inwendige knoop zou een extra kind met een bijbehorende deelboom vereisen. Vanuit de wortel zoeken we dus het blad waarin de sleutel zou moeten zitten, en we voegen hem op de juiste plaats in de tabel toe (met zijn bijbehorende informatie).

Als dat blad dan m sleutels bevat, wordt het gesplitst bij de middelste sleutel (nummer $\lceil m/2 \rceil$). Daarbij wordt een nieuwe knoop op hetzelfde niveau aangemaakt (in het inwendig geheugen en op de schijf), waarin de gegevens rechts van de middelste sleutel terechtkomen. De middelste sleutel zelf (met zijn bijbehorende informatie) gaat naar de ouder, vergezeld van een verwijzing naar de nieuwe knoop. Een deel van de tabellen in de ouder moet dus opgeschoven worden.

Gewoonlijk heeft die ouder plaats voor deze extra sleutel. Als dat niet het geval is, moet de ouder op zijn beurt gesplitst worden. Als een te splitsen knoop kinderen heeft, moeten die samen met de sleutels over de originele en de nieuwe knoop verdeeld worden.

Soms tracht men het opsplitsen van knopen zo lang mogelijk uit te stellen, door gegevens over te brengen naar een broerknoop, en ze over beide knopen zo goed mogelijk te verdelen. Dit is natuurlijk enkel mogelijk als die broer plaats heeft voor extra sleutels. Omdat de inorder volgorde van de sleutels intact moet blijven, gebeurt dat overbrengen weer via een soort rotatie: een sleutel gaat van de knoop naar zijn ouder, die een

[5] De top-down versie (Guibas en Sedgewick, 1978) is interessant voor een B-tree met meerdere gelijktijdige gebruikers. De knopen op de gevolgde weg kunnen dan immers vroeger vrijgegeven worden, zodat ze door anderen kunnen gebruikt worden.

sleutel afstaat aan de broer. Die krijgt bovendien een kindwijzer van de knoop (als die kinderen heeft).

In het slechtste geval moet elke knoop op de (terug)weg naar de wortel gesplitst worden. Het kan ook nodig zijn om de wortel te splitsen. Er is dan echter geen ouder meer om de middelste sleutel in onder te brengen. Die moet dus aangemaakt worden, en wordt de nieuwe wortel. (Met slechts één sleutel, maar dat is toegelaten.) In tegenstelling tot binaire zoekbomen groeien B-trees dus bovenaan. (Om onderaan te groeien zou er een volledig niveau moeten bijkomen, wegens de perfecte vorm van de boom.) Alle nieuwe en alle gewijzigde knopen moeten op de schijf aangepast worden.

In het slechtste geval worden er dus $h+1$ knopen gesplitst. Dat is echter zeer zeldzaam. Immers, het totaal aantal splitsingen sinds de initialisatie van de boom is gelijk aan het aantal knopen, min $(h+1)$. (Ga eens na.) Dat is veel minder dan het aantal toegevoegde sleutels. Want een boom met p knopen bevat minstens $1+(p-1)(\lceil m/2 \rceil - 1)$ sleutels, zodat $p \leq 1 + (n-1)/(\lceil m/2 \rceil - 1)$. Per toegevoegde sleutel is het gemiddeld aantal splitsingen dan ook kleiner dan $1/(\lceil m/2 \rceil - 1)$.

Een knoop splitsen vereist drie schijfoperaties (als de te splitsen knoop en zijn ouder reeds ingelezen zijn), en een processortijd van $O(m)$, omdat alle kopieer- en opschuifbewerkingen begrensd zijn door het maximaal aantal kinderen in een knoop. In het slechtste geval moeten we tweemaal de volledige hoogte van de boom doorlopen, maar er gebeurt slechts een constant aantal schijfoperaties per niveau. Het totaal aantal schijfoperaties is dus $\Theta(h) = \Theta(\log_{\lceil m/2 \rceil} n)$. Per knoop is de processortijd $O(m)$ (want we moeten zoeken in of toevoegen aan een tabel met grootte $O(m)$, naast eventueel nog $O(m)$ voor opsplitsen), met dus een totaal van $O(mh) = O(m \log_{\lceil m/2 \rceil} n)$.

3.1.3.3 Verwijderen

De te verwijderen sleutel kan zowel in een blad als in een inwendige knoop zitten. Maar uiteindelijk moeten we steeds uit een blad verwijderen, want anders zou er samen met de sleutel een kind en de bijbehorende deelboom moeten verdwijnen.

Ook hier bespreken we een bottom-up versie: als het blad te weinig sleutels overhoudt, moet er soms een sleutel aan de ouder ontleend worden. In het slechtste geval gaat dit ontlenen door tot bij de wortel. Een sleutel ontlenen aan een wortel met slechts één sleutel maakt die wortel ledig, zodat die verwijderd moet worden: een B-tree krimpt dus ook bovenaan.

Wanneer de te verwijderen sleutel in een inwendige knoop zit, mogen we de structuur van die knoop niet wijzigen. Daarom vervangen we de sleutel door zijn voorloper of opvolger, die dan werkelijk verwijderd wordt (want te vinden in een blad). We moeten dus eerst helemaal afdalen, om dan achteraf nog een wijziging aan te brengen in een hoger gelegen knoop. Dat is echter zeldzaam, omdat de meeste sleutels van een B-tree in de bladeren zitten.

Wanneer een knoop te weinig sleutels overhoudt, kan men trachten een sleutel te ontlenen aan een broerknoop. Dat gebeurt met dezelfde rotatie als bij toevoegen: de sleutel van de broer gaat naar zijn ouder, een sleutel van de ouder gaat naar de knoop, die ook een kindwijzer van de broer overneemt (als die kinderen heeft). Omdat hierbij drie knopen moeten aangepast worden, brengt men soms meerdere sleutels over van een broer, zodat ze elk ongeveer even veel sleutels krijgen. Opnieuw verwijderen uit dezelfde knoop geeft dan niet meteen weer problemen.

Natuurlijk kan men enkel sleutels ontlenen aan een broer die sleutels kan missen. Als dat niet het geval is, kan men eventueel hetzelfde proberen bij de andere broer, als die bestaat. Maar als geen broer een sleutel kan missen, dan gebeurt het omgekeerde van splitsen: de knoop wordt samengevoegd met een broer, zodat er een knoop uit de boom verdwijnt. Hun ouder verliest dan een kind, zodat de sleutel tussen de twee broers moet verdwijnen: hij wordt ook toegevoegd aan de samengevoegde knoop. Met als mogelijk gevolg dat de ouder te weinig sleutels overhoudt.

Bij roteren of samenvoegen zijn drie knopen betrokken, zodat drie schijfoperaties vereist zijn (naast de leesoperaties). In het slechtste geval moeten we tweemaal de volledige hoogte van de boom doorlopen, maar er gebeurt slechts een constant aantal schijfoperaties per niveau. Het totaal aantal schijfoperaties is dus $\Theta(h) = \Theta(\log_{\lceil m/2 \rceil} n)$. Per knoop is de processortijd $O(m)$ (want we moeten zoeken in of verwijderen uit een tabel met grootte $O(m)$, naast eventueel nog $O(m)$ voor samenvoegen), met dus een totaal van $O(mh) = O(m \log_{\lceil m/2 \rceil} n)$.

3.1.4 Varianten van B-trees

3.1.4.1 B$^+$-tree

Een gewone B-tree heeft enkele nadelen. Zo moeten de bladeren plaats reserveren voor kindwijzers die toch niet gebruikt en dus niet ingevuld worden. Dat inwendige knopen gegevens kunnen bevatten maakt bijvoorbeeld verwijderen ingewikkelder, en zowel toevoegen als verwijderen moet uiteindelijk toch in een blad gebeuren. Tenslotte kan zoeken naar de opvolger van een sleutel $O(\log_{\lceil m/2 \rceil} n)$ schijfoperaties vereisen.

Een veel gebruikte B-tree variant is de B$^+$-tree (Knuth 1973), die alle gegevens (sleutels en hun bijbehorende informatie) in de bladeren opslaat. De inwendige knopen worden gebruikt als een *index* die toelaat om snel gegevens te lokaliseren. Met als gevolg dat bladeren en inwendige knopen een verschillende structuur krijgen, en ook in grootte mogen verschillen. Bovendien maakt men een *gelinkte lijst* van alle bladeren, in stijgende sleutelvolgorde (de 'sequence set').

Het scheiden van de index en de sequence set heeft belangrijke gevolgen. Omdat inwendige knopen uitsluitend dienen om de juiste weg te vinden, bevatten ze enkel sleutels (zonder bijbehorende informatie) en kindwijzers. Hun maximale graad kan daardoor groter worden. De bladeren moeten niet langer plaats reserveren voor kindwijzers, zodat ze meer gegevens kunnen bevatten. Het aantal gegevens in een blad wordt ook

tussen twee grenzen gehouden, die dus meestal verschillen van deze voor de inwendige knopen. Door de knopen beter te gebruiken wordt de hoogte van de boom kleiner, wat de woordenboekoperaties sneller maakt. De sequence set zorgt ervoor dat zoeken naar de opvolger van een sleutel hoogstens één schijfoperatie vereist. Ook sequentieel overlopen van alle gegevens is nu efficiënter, want elke knoop wordt slechts eenmaal bezocht, zodat er plaats voor slechts één knoop in het inwendig geheugen nodig is. Een B^+-tree is dus zeer geschikt voor zowel random als sequentiële operaties.

De woordenboekoperaties verlopen nagenoeg zoals bij een gewone B-tree. Zoeken gaat nu steeds van de wortel naar een blad, en stopt dus niet bij een inwendige knoop die de gezochte sleutel bevat. Toevoegen gebeurt in een blad. Als dat blad moet gesplitst worden, blijft de middelste sleutel in een van de twee knopen, en gaat er enkel een *kopie* van die sleutel naar de ouder. Ook verwijderen gebeurt steeds in een blad. Zolang dat blad genoeg sleutels overhoudt, moeten hoger gelegen knopen niet gewijzigd worden, zelfs als ze een kopie van de verwijderde sleutel bevatten. De enige taak van de sleutels in de inwendige knopen is immers te leiden naar het juiste blad. Het kan echter wel nodig zijn om deze knopen aan te passen, wanneer een blad gegevens moet ontlenen, of samengevoegd wordt met een buur.

3.1.4.2 Prefix B^+-tree

Wanneer de sleutels strings zijn, kunnen ze in de inwendige knopen teveel plaats innemen. De enige functie van een sleutel in een inwendige knoop is echter de sleutels in twee deelbomen van elkaar te onderscheiden. Daarom gebruikt een Prefix B^+-tree (Bayer en Unterauer, 1977) daarvoor een zo kort mogelijke string, vaak een prefix van een van de te onderscheiden strings. Met als gevolg meer plaats voor sleutels in de inwendige knopen, zodat de boom minder hoog wordt, wat de zoektijd ten goede komt.

3.1.4.3 B^*-tree

Bij een gewone B-tree stelt men soms het splitsen van een knoop uit, door gegevens over te brengen naar een buur. Wanneer die buur ook vol is, splitst een gewone B-tree de knoop in twee knopen die dan elk ongeveer half gevuld zijn. Een B^*-tree (Knuth, 1973) verdeelt de sleutels van de knoop en de volle buur over de *drie* knopen, zodat die elk voor ongeveer twee derden gevuld zijn. De wortel heeft echter geen buur, zodat men wacht met splitsen tot ook hier twee knopen voor twee derden kunnen gevuld worden. Dat betekent dat men toelaat dat de wortel tot vier derden van de normale grootte opgevuld wordt. Beter gevulde knopen betekent een minder hoge boom, en dus een kleinere zoektijd.

3.2 UITWENDIGE HASHING

Wanneer men niet geïnteresseerd is in de volgorde van de sleutels, bestaan er meer eenvoudige, efficiënte alternatieven voor B-trees: uitwendige versies van hashtabellen [13, 15, 40]. Hun woordenboekoperaties vereisen immers (gemiddeld) slechts $O(1)$ schijfoperaties.

Een interne hashtabel met chaining moet van dezelfde grootteorde zijn als het aantal sleutels dat ze moet bevatten, en met open adressering is ze best tweemaal zo groot. Beide types kunnen dus duidelijk niet ongewijzigd gebruikt worden voor een zeer groot aantal sleutels.

Aangezien de informatie-overdracht bij een schijf toch in pagina's gebeurt, zou men aan elk hashtabelelement een schijfpagina kunnen toewijzen. Alle sleutels met dezelfde hashwaarde zouden dan in dezelfde pagina ondergebracht worden. Het streefdoel van $O(1)$ schijfoperaties lijkt dan gehaald. Maar wat als er zoveel conflicten zijn dat een pagina vol geraakt? Als we dezelfde technieken toepassen als bij inwendige hashtabellen, pagina's expliciet linken bij chaining, of alternatieve pagina's berekenen bij open adresseren, dan stijgt in beide gevallen het aantal schijfoperaties. En natuurlijk is rehashing van alle sleutels in al die schijfpagina's uitgesloten.

Een boom is een meer flexibele structuur dan een tabel. Aangezien alle gegevens in schijfpagina's opgeslagen zijn, moeten we zo snel mogelijk bij de juiste pagina terechtkomen. Wanneer men een sleutel zoekt in een (binaire) trie (zie later, in 10.3), wordt die niet vergeleken met sleutels in de knopen, maar worden zijn opeenvolgende *bits* gebruikt om het zoekproces te sturen. Elke deelboom van een trie bevat alle sleutels met een gemeenschappelijke prefix (die leidt naar de wortel van de deelboom). Aangezien de bladeren van de trie hier schijfpagina's zijn, die veel sleutels kunnen bevatten, kan men alle sleutels van een deelboom in één pagina onderbrengen, zodat de weg erheen in de trie korter wordt. Pas wanneer een pagina vol geraakt, wordt ze gesplitst, en beide pagina's krijgen een nieuwe trieknoop als ouder. De sleutels van de volle knoop worden dan op grond van de eerstvolgende bit tussen beide pagina's verdeeld. Die bit zal op de zoekweg getest worden bij hun nieuwe ouder.

De vorm van een trie is onafhankelijk van de toevoegvolgorde, en wordt volledig bepaald door de bits van de sleutels. Die boom kan dus zeer onevenwichtig uitvallen. Daarom gebruikt men niet de bits van de sleutels, maar van hun *hashwaarden*. Een goede hashfunctie zorgt er immers voor dat die hashwaarden (en dus hun bits) zo random mogelijk zijn. Met als gevolg een meer evenwichtige boom.

Beide methoden voor uitwendige hashing elimineren ook nog het zoeken in de trie. Extendible hashing vervangt die door een rechtstreeks adresseerbare tabel, linear hashing gebruikt pagina's met opeenvolgende (logische) adressen.

3.2.1 Extendible hashing

Deze methode (Fagin, Nievergelt, Pippenger en Strong, 1978) elimineert het zoeken in de trie, door het langst mogelijke prefix uit de trie als index te gebruiken in een hashtabel, die verwijzingen naar de overeenkomstige pagina's bevat. Kortere prefixen komen dan overeen met meerdere tabelelementen (elke index heeft immers evenveel bits), die allemaal een verwijzing naar dezelfde pagina moeten bevatten. Het vinden van een pagina is nu inderdaad $O(1)$.

Laten we de implementatie wat nader bekijken. Voorlopig onderstellen we dat de hashtabel opgeslagen is in het inwendig geheugen. Deze tabel bevat 'wijzers' naar de schijfpagina's, die maximaal m (afhankelijk van de hardware) sleutels met bijbehorende gegevens bevatten. Net zoals bij een B-tree worden deze sleutels stijgend gerangschikt in een tabel opgeslagen.

Als hashwaarden gebruikt men gehele getallen, waarvan het bereik bepaald wordt door de breedte w van een processorwoord. De laatste d bits van die getallen worden als indices in de hashtabel gebruikt, zodat de tabel 2^d elementen bevat ($d < w$). Dat aantal bits d (de 'globale diepte') is de lengte van het langste prefix in de trie, en kan dus variëren. (In tegenstelling tot een gewone hashtabel zijn hier (veel) meer hashwaarden dan tabelelementen.) Alle sleutels waarvan de hashwaarde met dezelfde d bits eindigt komen dus bij hetzelfde tabelelement terecht, en worden in de overeenkomstige pagina opgeslagen.

Eenzelfde pagina kan sleutels met hashwaarden bevatten waarvan de laatste d bits verschillend zijn, aangezien meerdere tabelelementen naar dezelfde pagina mogen verwijzen. Daarom houden we per pagina het aantal bits k bij waarmee *al* haar hashwaarden eindigen (de 'lokale diepte'). De waarde van k kan zelfs nul zijn, maar is zeker niet groter dan d. Het aantal tabelelementen dat naar dezelfde pagina wijst is dan 2^{d-k}.

Extendible hashing voorziet enkel woordenboekoperaties (zoals gewone inwendige hashing):

- *Zoeken.* Zoeken van een sleutel komt neer op het berekenen van de hashwaarde, de pagina vinden via de hashtabel (en ze inlezen), en dan sequentieel (of eventueel binair) zoeken in de sleuteltabel. Zolang de hashtabel in het geheugen past, vereist dit slechts één schijfoperatie!

- *Toevoegen.* Toevoegen is analoog, en eenvoudig zolang de pagina niet vol is. Daarbij moet gemiddeld de helft van de gegevens in de tabel opgeschoven worden om de volgorde intact te houden, maar dat is verwaarloosbaar naast de tijd vereist voor een schijfoperatie.

  Een volle pagina moet echter gesplitst worden. Aangezien hier geen boomstructuur gebruikt wordt, kan splitsen ingrijpende gevolgen hebben voor de hashtabel.

  Omdat alle hashwaarden in de pagina met dezelfde k bits eindigen, splitst men door de sleutels onder te verdelen volgens de waarde van bit $k + 1$. Gegevens waarbij die bit één is worden overgebracht naar een nieuw gecreëerde pagina.

Daarna wordt de waarde van k één groter, zowel in de nieuwe als in de oude pagina. Ook de hashtabel moet aangepast worden:

- Als k kleiner was dan d dan moet de helft van de wijzers naar de oude pagina (we weten die staan) verwijzen naar de nieuwe pagina.
- Als k gelijk was aan d, dan was er maar één wijzer naar de oude pagina. Aangezien k nu groter wordt dan d, moet ook d met één toenemen en dus *verdubbelt* de grootte van de hashtabel. Elke index wordt dan één bit langer, zodat er twee nieuwe indices uit ontstaan. De tabelelementen bij beide indices moeten naar dezelfde pagina verwijzen als de oorspronkelijke index. Enkel bij de gesplitste pagina wijst het ene tabelelement naar de oude pagina, het andere naar de nieuwe.

Als de hashwaarden echter niet te onderscheiden zijn op grond van bit $k + 1$ dan komen al hun sleutels in dezelfde pagina terecht (oud of nieuw). We trachten ze dan te verdelen met de volgende bit, en indien nodig met nog meer bits, tot er minstens één sleutel in een van de betrokken pagina's terechtkomt. De grootte van de hashtabel kan daarbij telkens verdubbelen.

Deze methode werkt niet als er m gelijke hashwaarden in dezelfde pagina terechtgekomen zijn. Ook wanneer de hashwaarden in een pagina teveel identieke eindbits hebben, wordt de hashtabel onnodig groot. Voor beide problemen is het dus belangrijk dat de hashfunctie goed is, zodat de toegekende bits zo 'random' mogelijk zijn.[6]

Een belangrijke eigenschap van extendible hashing is dat de inhoud van de hashtabel enkel afhangt van de *waarde* van de opgeslagen sleutels, en niet van de volgorde waarin die sleutels toegevoegd werden. (Dat is niet zo verwonderlijk, want dat is een eigenschap van de structuur van tries.)

- *Verwijderen.* Verwijderen is lastiger dan toevoegen, net zoals bij een B-tree. Wanneer een pagina en haar ooit afgesplitste buur samen minder dan m sleutels bevatten, kan men ze samenvoegen. (Best *minder* dan m, om te vermijden dat de samengevoegde pagina snel weer moet gesplitst worden.) Er verdwijnt dan een pagina, en haar hashtabelelement moet dan naar de andere pagina verwijzen (waarvan de k-waarde met één vermindert). Zodra de hashtabel minstens twee verwijzingen naar elke pagina bevat (elke k is dan kleiner dan d), kan ze gehalveerd worden (d vermindert met één).

  Soms maakt men het zich gemakkelijk door pagina's met weinig elementen te tolereren, wat in de praktijk vaak goed werkt.

Wanneer er n uniform verdeelde sleutels opgeslagen zijn, kan men aantonen dat de verwachtingswaarde van het aantal pagina's $n/(m \ln 2) \approx 1.44n/m$ bedraagt, zodat elke pagina gemiddeld voor ongeveer 69 procent gevuld is. (Hetzelfde resultaat als bij B-trees.) De grootte van de hashtabel heeft een verwachtingswaarde van ongeveer

[6] Om een al te grote hashtabel te vermijden, legt 'bounded index exponential hashing' (Lomet, 1983) een maximumgrootte op aan die tabel, en verdubbelt daarna de grootte van de pagina's, waar nodig.

$3.92(n/m)\ \sqrt[m]{n}$. Voor praktische waarden van m is $\sqrt[m]{n} \approx 1$, zodat dit resultaat ongeveer $4(n/m)$ bedraagt.

Als de hashtabel te groot uitvalt voor het inwendig geheugen, zijn er een aantal mogelijkheden:

- De hashtabel kan op analoge manier in twee niveau's geïmplementeerd worden. We houden dan de wortel in het geheugen, en indexeren hem met een kleiner aantal eindbits om de delen van de eigenlijke hashtabel in het uitwendig geheugen te vinden. Zoeken vereist dan een extra schijfoperatie.
- Als alternatief kunnen we het maximaal aantal gegevens m in elke pagina vergroten, door er naast sleutels enkel wijzers naar de bijbehorende informatie in op te slaan. Er zijn dan minder pagina's nodig, en dus wordt de hashtabel kleiner. Maar door de wijzers vereist zoeken ook hier een extra schijfoperatie.

3.2.2 Linear hashing

Deze methode (Litwin 1980) heeft zelfs geen hashtabel meer nodig om de trie te elimineren, door pagina's met opeenvolgende adressen te gebruiken. De d eindbits van de hashwaarde worden niet meer als index in een hashtabel, maar rechtstreeks als adres van een pagina gebruikt. (Omdat het vereiste aantal pagina's meestal moeilijk te schatten valt, gebruikt men opeenvolgende *logische* adressen, en laat het aan het operating system over om deze om te zetten in fysische adressen.) Bij een hashtabel mag het aantal pagina's kleiner zijn dan 2^d, omdat verschillende wijzers naar dezelfde pagina mogen verwijzen, maar nu zijn er 2^d adressen, en dus even veel pagina's. Wanneer d toeneemt verdubbelt de hashtabel, maar niet het aantal pagina's. Hier zou het aantal pagina's moeten verdubbelen. Daarom gaat men anders te werk.

Wanneer een pagina vol is wordt er gesplitst, maar niet noodzakelijk de volle pagina. Pagina's worden in *sequentiële* volgorde gesplitst, of ze nu vol zijn of niet. Als een andere dan de volle pagina gesplitst wordt, krijgt deze laatste een overflow pagina toegewezen voor de overtollige gegevens. Als dan uiteindelijk deze pagina aan de beurt is om te splitsen, worden zowel haar gegevens als die in haar overflow pagina verdeeld over de twee pagina's, aan de hand van het $(d+1)$-de eindbit van hun hashwaarden. Met als gevolg dat de adressen van de oude en de nieuwe pagina voortaan $d+1$ bits tellen. De adressen van de nog niet gesplitste pagina's blijven echter d bits lang.

Ook linear hashing voorziet enkel woordenboekoperaties (zoals gewone hashing):

- *Zoeken.* De hashwaarde van de sleutel wordt berekend, maar we moeten weten hoeveel eindbits daarvan nodig zijn om de pagina te adresseren. Daartoe wordt het (logisch) adres van de volgende te splitsen pagina bijgehouden in een variabele p (zie onder, bij toevoegen). Het adres gevormd door de d eindbits wordt dan vergeleken met dat in p (wat ook d bits telt). Als het kleiner is, dan is de

gezochte pagina reeds gesplitst en moeten $d+1$ eindbits voor haar adres gebruikt worden. Anders volstaan de d bits.

De pagina zelf bevat net zoals bij extendible hashing een gerangschikte tabel van sleutels (met bijbehorende informatie), en houdt het huidig aantal sleutels bij (maximaal m). De sleutel wordt dan gelokaliseerd via binair of linear zoeken. Het is mogelijk dat er in plaats van in de (volle) pagina, in haar overflowpagina moet gezocht worden.

Bij extendible hashing vereist zoeken precies één schijfoperatie als de hashtabel in het inwendig geheugen zit, anders twee. Bij linear hashing vereist zoeken steeds één schijfoperatie, tenzij er ook in een of meerdere overflowpagina's moet gezocht worden. Voor heel veel gegevens is de performantie van extendible hashing dus minder goed dan het beste geval van linear hashing, maar beter dan het slechtste geval. Wanneer men ook bij extendible hashing overflowpagina's introduceert om splitsen uit te stellen, wordt de vergelijking moeilijker. (Bij linear hashing zijn overflowpagina's noodzakelijk, bij extendible hashing optioneel.)

- *Toevoegen.* Toevoegen is de pagina lokaliseren zoals bij zoeken, en als deze niet vol is, het gegeven tussenvoegen in de gerangschikte tabel.

  Als de pagina vol is moet er gesplitst worden. Het adres van de volgende te splitsen pagina zit in p (zie hoger). Omdat splitsen in sequentiële volgorde gebeurt, is p initieel nul, en wordt bij elke splitsing geïncrementeerd tot alle 2^d pagina's gesplitst zijn. Dan wordt d geïncrementeerd, wordt p opnieuw nul, en begint de volgende expansieronde. Wanneer pagina p gesplitst wordt, wordt het adres van de nieuwe pagina $p + 2^d$.

  Als echter de volle pagina niet gesplitst werd, dan wordt het gegeven toegevoegd aan een (eventueel nieuwe) overflowpagina.

- *Verwijderen.* Verwijderen is de pagina lokaliseren zoals bij zoeken, en het gegeven uit de tabel halen (eventueel uit een overflowpagina). Onderbezette pagina's verdwijnen in omgekeerde volgorde als waarin ze gecreëerd werden. (Het is dus niet noodzakelijk de pagina waaruit verwijderd werd die verdwijnt.) De gegevens van zo'n pagina komen dan terecht in de pagina waarvan ze origineel afgesplitst werd. Om te vermijden dat die bij toevoegen snel weer gesplitst wordt, moet hun gezamenlijk aantal gegevens kleiner zijn dan m.

HOOFDSTUK 4
MEERDIMENSIONALE GEGEVENSSTRUCTUREN

Vaak hebben gegevens meer dan één sleutel: ze zijn meerdimensionaal. Gegevensstructuren moeten dan toelaten om efficiënt te zoeken op al die sleutels, of in een bereik van een of meerdere sleutels. Voor eendimensionale gegevens is het aantal efficiënte gegevensstructuren beperkt. Efficiënt meerdimensionaal zoeken wordt moeilijker naarmate het aantal dimensies toeneemt ('the curse of dimensionality'). De meeste gegevensstructuren zijn dan ook bedoeld voor een klein aantal dimensies. (De belangstelling voor meer dimensies neemt echter toe, zie [8].) Het aantal alternatieven is groot, zodat de keuze niet steeds eenvoudig is. Voor een overzicht verwijzen we naar het artikel van Gaede en Günther [15], en hoofdstuk 18 uit het compendium van Atallah [4].

Hoewel de types van de sleutels vaak verschillend zijn, modelleert men deze gegevens en het zoeken erop meestal als een meerdimensionaal geometrisch probleem. (Met de impliciete veronderstelling dat er een ordening op elk sleuteltype bestaat.) Elk gegeven wordt dan beschouwd als een punt in een meerdimensionale (Euclidische) ruimte. Daardoor kan men gebruik maken van algoritmen en gegevensstructuren uit het vakgebied 'Computational Geometry', dat zich specifiek bezighoudt met geometrische problemen.

Een meerdimensionaal punt zoeken is een speciaal geval van zoeken naar alle punten gelegen in een meerdimensionale hyperrechthoek ('range search'). Daarom wordt meestal enkel dit algemener probleem behandeld. Bij de bespreking zullen we het aantal punten n noemen, en het aantal dimensies k. De eenvoudigste oplossing is natuurlijk alle gegeven punten sequentieel overlopen en nagaan of elk punt tot de (hyper)rechthoek behoort. Deze triviale methode valt enkel te overwegen wanneer n klein is, of wanneer men verwacht dat de zoekrechthoek een groot deel van de punten zal bevatten. Als de zoekopdrachten niet meteen moeten beantwoord worden, kan men ze verzamelen ('batching'), om ze dan samen te behandelen tijdens eenzelfde sequentiële doorgang.

4.1 INWENDIGE GEGEVENSSTRUCTUREN

4.1.1 Projectie

Deze methode gebruikt per dimensie een gegevensstructuur die alle punten gerangschikt bijhoudt volgens die dimensie. De punten worden dus geprojecteerd op elke dimensie.[1] Zoeken in een hyperrechthoek gebeurt door een dimensie te kiezen en in de overeenkomstige gegevensstructuur alle punten te zoeken die binnen zijn zijde voor die dimensie vallen. Deze punten worden dan sequentieel overlopen om te testen of ze in de hyperrechthoek liggen.

Wanneer de punten gelijkmatig verdeeld zijn in elke dimensie volstaat één gegevensstructuur, voor eender welke dimensie. Zoniet is het beter om telkens op alle dimensies te zoeken, en dan de kleinste van deze puntenverzamelingen sequentieel te overlopen.

Deze methode werkt het best wanneer de zoekrechthoek een zijde heeft die de meeste punten uitsluit. Voor uniform verdeelde punten, een nagenoeg kubusvormig zoekgebied, en een klein aantal gevonden punten, kan men aantonen dat de gemiddelde performantie $O(n^{1-1/k})$ is.

4.1.2 Rasterstructuur

Deze methode verdeelt de zoekruimte met behulp van een raster. Voor elk van de rastergebieden (een hyperrechthoek) houdt men dan een gelinkte lijst bij met de punten die erin liggen.

Alle punten opsporen in een zoekrechthoek komt dan neer op het vinden van alle rastergebieden die deze rechthoek overlappen, en nagaan welke van de punten in hun gelinkte lijsten binnen die rechthoek vallen.

Voor uniform verdeelde punten kiest men best een regelmatig raster. Dat kan men implementeren met een meerdimensionale tabel, zodat men de rastergebieden die een zoekrechthoek overlappen zeer snel kan vinden. De rest van de zoekoperatie is enkel efficiënt als er niet al te veel dergelijke rastergebieden zijn, en als hun gelinkte lijsten gemiddeld kort zijn. Men zal dus een compromis moeten vinden tussen de grootte van een rastergebied en de lengte van de lijsten. Een te fijne rasterverdeling betekent niet alleen dat er veel lijsten moeten doorzocht worden, maar ook dat er veel geheugen nodig is om ze op te slaan. Het aantal rastergebieden is dus best een *constante fractie* van het aantal punten n, zodat het gemiddeld aantal punten in elk rastergebied een kleine constante wordt. (Wanneer de grootte van de zoekrechthoek niet verandert, maar enkel zijn positie, kan men aantonen dat een rastergebied met dezelfde grootte nagenoeg optimaal is.)

Als er nogal wat rastergebieden ledig blijven, zodat de meerdimensionale tabel onnodig veel geheugen in beslag neemt, kan men kiezen voor een andere gegevensstructuur die

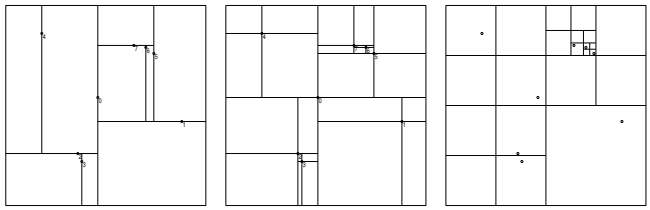
[1] Een andere naam voor deze techniek is 'inverted file': in plaats van coördinaten bij punten op te slaan, slaat men punten op bij coördinaten. Deze techniek is ook heel belangrijk voor zoeken in tekstcollecties (zie deel 3).

enkel de niet-ledige rastergebieden bevat. Die moet toelaten om snel de gebieden te vinden die de zoekrechthoek overlappen.

Een regelmatig raster is ongeschikt voor ongelijkmatig verdeelde punten. (In een extreem geval liggen ze allemaal in één rasterelement.) Ook voor een onregelmatig raster kan men een meerdimensionale tabel gebruiken, maar het verband tussen de positie van een rastergebied en die van het overeenkomstig tabelelement is wat ingewikkelder. Daarom houdt men per dimensie de gerangschikte positie van de raster(hyper)vlakken bij. Een zoekoperatie per dimensie geeft dan de indices in die dimensie van de gezochte rastergebieden.

Een vast raster (al dan niet regelmatig) veronderstelt statische gegevens. Bij dynamische gegevens kunnen er zowel overvolle als ledige rastergebieden ontstaan. Om de overvolle gebieden te vermijden zal men dus het raster moeten aanpassen, door de rastergrenzen te verplaatsen, of rastergrenzen toe te voegen. Maar dat kan dan weer veel ledige gebieden opleveren. Hiervoor zijn er twee mogelijke oplossingen. Ofwel gebruikt men een andere gegevensstructuur met enkel de niet-ledige rastergebieden (zoals hierboven bij een regelmatig raster). Ofwel maakt men het raster adaptief, zodat naburige ledige gebieden samengevoegd kunnen worden, en overvolle gebieden gesplitst. Een tabel is daarvoor niet geschikt, zodat men een boomstructuur moet gebruiken. Deze combinatie van een 2^k-wegsboom met een raster ligt aan de basis van de quadtrees.

4.1.3 Quadtrees



Figuur 4.1. 2D-boom, Point Quadtree, PointRegion Quadtree.

Meerdere verwante gegevensstructuren dragen deze naam. (Voor een uitgebreid overzicht, zie dat van Samet [33].) Ze verdelen de zoekruimte, of een gedeelte daarvan, telkens in 2^k (hyper)rechthoeken, waarvan de zijden evenwijdig zijn met het assenstelsel. (Oorspronkelijk werden ze gebruikt voor tweedimensionale toepassingen, vandaar hun naam. Bij drie dimensies noemt men ze 'octtrees'.) Deze hiërarchische verdeling wordt opgeslagen in een 2^k-wegsboom: elke knoop staat voor een gebied, dat onderverdeeld wordt in de 2^k deelgebieden van zijn kinderen. Meteen wordt duidelijk waarom quadtrees niet geschikt zijn voor veel dimensies: het aantal knopen neemt zeer snel toe met de hoogte. We zullen het hier houden bij twee dimensies.

We bespreken de twee belangrijkste soorten om *punten* op te slaan: 'point' quadtrees

en PR quadtrees ('Point Region'). Het belangrijkste verschil tussen beide is dat een point quadtree bij de opdeling van de zoekruimte rekening houdt met de opgeslagen punten, en een PR quadtree dat niet doet. Bij eendimensionale gegevensstructuren bestaat trouwens hetzelfde onderscheid tussen een binaire zoekboom en een binaire trie (zie later, in 10.3). Een PR quadtree zou dus beter een 'quadtrie' heten.

4.1.3.1 Point quadtrees

De originele quadtree (Finkel en Bentley, 1974) is een meerdimensionale uitbreiding van een binaire zoekboom. Elke inwendige knoop bevat een punt, waarvan de coördinaten de (deel)zoekruimte opdelen in vier rechthoeken, die dus niet noodzakelijk even groot zijn. Elk kind is de wortel van een deelboom die alle punten in de overeenkomstige rechthoek bevat. (Als de deelboom ledig is, is het een blad.)

Zoeken van een punt volgt een weg vanuit de wortel, vergelijkt telkens het zoekpunt met de punten bij de opeenvolgende knopen, en daalt eventueel af naar het kind met het gepaste deelgebied. Als het zoekpunt niet aanwezig blijkt (we vinden dan een leeg deelgebied), kan men het toevoegen als nieuw inwendig punt.

Zoals bij een binaire zoekboom is de vorm van een point quadtree afhankelijk van de toevoegvolgorde. Als elke toevoegvolgorde even waarschijnlijk is, dan zijn zoeken en toevoegen in een boom met n punten $O(\lg n)$. In het slechtste geval zijn ze echter $O(n)$.

Wanneer de punten op voorhand gekend zijn, kan men ervoor zorgen dat geen enkele deelboom meer dan de helft van de punten van die van zijn ouder bevat. Daartoe rangschikt men de punten lexicografisch (op x, bij gelijke x op y), en neemt als wortel de mediaan. Alle punten vóór de mediaan vallen dan in twee van zijn deelbomen, deze erachter in de andere twee. Bij de kinderen gebeurt hetzelfde, met dat verschil dat de punten reeds gerangschikt zijn. De hoogte van de boom wordt aldus $O(\lg n)$. Deze constructie is $O(n \lg n)$. Want rangschikken is $O(n \lg n)$, en het werk op elk van de $O(\lg n)$ niveau's is $O(n)$. De mediaan vinden is immers $O(1)$, en de punten verdelen over de deelbomen zodat ze gerangschikt blijven, is $O(n)$.

Zoeken naar de punten in een zoekrechthoek gebeurt recursief, door bij een knoop te testen of zijn punt in de rechthoek ligt, en daarna zijn deelbomen te onderzoeken die de rechthoek overlappen.

Verwijderen van een punt is moeilijk, want we moeten de deelboom onder de overeenkomstige knoop herstructureren. Een eenvoudige manier om dat te doen voegt al de punten van die deelboom opnieuw toe. (En best met de methode van hierboven.)

4.1.3.2 PR quadtrees

Zoals de naam suggereert is een point-region quadtree een gegevensstructuur voor ruimtelijke objecten ('gebieden' in twee dimensies), die ook kan gebruikt worden om punten op te slaan. (Zijn oorspronkelijke naam is trouwens 'region' quadtree, en hij wordt voornamelijk gebruikt voor beelden.) In tegenstelling tot een point quadtree vereist deze boom dat de zoekruimte een rechthoek is. Die is ofwel gegeven, of moet eerst bepaald worden, als kleinste rechthoek die alle punten bevat. Elke knoop verdeelt een deel van die ruimte in vier *gelijke* rechthoeken. Deze opdeling gaat door tot een deelgebied nog één punt bevat. De inwendige knopen bevatten dus geen punten. (Bemerk de analogie met een trie.) Het is zelfs niet nodig dat ze coördinaten bevatten: om plaats te sparen kan men die telkens op de zoekweg berekenen.

Zoeken van een punt gebruikt de opeenvolgende knopen vanuit de wortel om de rechthoek te vinden waarin het punt zou moeten liggen. (En die dus slechts één punt bevat.) Als het zoekpunt verschilt van dat enige punt, en we wensen toe te voegen, dan moet het gebied opgesplitst worden tot elk van de punten in een eigen gebied ligt. Bij verwijderen kan het omgekeerde gebeuren: onderverdelingen (en knopen) die verdwijnen.

In tegenstelling tot een point quadtree is de vorm van een PR quadtree onafhankelijk van de toevoegvolgorde. (Eens te meer, zoals een trie.) Opsplitsen van een deelgebied wil niet zeggen dat de punten erin onderverdeeld worden: een PR quadtree kan zeer onevenwichtig uitvallen. Daarom is het onmogelijk om zijn grootte en hoogte uit te drukken in functie van het aantal opgeslagen punten n. Er is echter wel een verband tussen zijn hoogte en de kleinste afstand a tussen twee punten. Immers, als de z de grootste zijde van de zoekruimte is, dan is de grootste zijde van een gebied op diepte d gelijk aan $z/2^d$. De maximale afstand tussen twee punten in dat gebied is $z\sqrt{k}/2^d$. Het gebied van elke inwendige knoop bevat minstens twee punten, en hun onderlinge afstand is tenminste a. Op elke diepte d is dus $z\sqrt{k}/2^d \geq a$, of $d \leq \lg(z/a) + (\lg k)/2$. De hoogte h is de maximale diepte van een inwendige knoop plus één, zodat voor een (tweedimensionale) quadtree $h \leq \lg(z/a) + 3/2$.

Op elk niveau bedekken de gebieden van de inwendige knopen de verzameling punten, en al deze gebieden bevatten punten: per niveau is het aantal inwendige knopen dus $O(n)$. Het totaal aantal inwendige knopen i in een boom met hoogte h is dan $O(hn)$. Elke inwendige knoop heeft vier kinderen, zodat het aantal bladeren $3i + 1$ is. Het aantal knopen van de boom is dus ook $O(hn)$. Bij de constructie van de boom gaat het meeste werk naar het verdelen van de punten over de deelgebieden van elke knoop. Dat werk is evenredig met het aantal te verdelen punten: op elk niveau is het dus $O(n)$. De constructietijd voor de boom is dus eveneens $O(hn)$.

4.1.4 k-d trees

Een k-d tree (Bentley, 1975, zie ook [5]) vermijdt de grote vertakkingsgraad van een point quadtree door een binaire boom te gebruiken. Elke inwendige knoop bevat een punt, dat de (deel)zoekruimte verdeelt in slechts één dimensie. Opeenvolgende knopen

kunnen opeenvolgende dimensies gebruiken om te splitsen, zodat de zoekruimte opnieuw in k-dimensionale rechthoeken verdeeld wordt. En deze opdeling wordt bepaald door de opgeslagen punten, zodat er rekening gehouden wordt met hun verdeling.

Om deze rechthoekige gebieden een zo regelmatig mogelijke vorm te geven, kan men ook in elke knoop beslissen volgens welke dimensie er opgedeeld wordt (die met de grootste spreiding). Een ideale opsplitsing bestaat uit een gelijkmatige verdeling van zowel het vlak als van de punten (wat zowel regelmatige gebieden als een evenwichtige boom oplevert), maar dat is niet voor elke collectie punten mogelijk.

De opdeling kan doorgaan tot elk gebied slechts één punt bevat, of men kan vroeger stoppen en per gebied een gelinkte lijst van punten bijhouden.

In het statisch geval, als men de n punten op voorhand kent, kan een evenwichtige k-d tree met een hoogte van $O(\lg n)$ opgebouwd worden in een tijd van $O(kn \lg n)$. Daartoe moeten beide deelbomen van elke knoop evenveel punten bevatten.

Voegt men de punten dynamisch toe, dan heeft de boom een *gemiddelde* hoogte van $O(\lg n)$ als elke toevoegvolgorde even waarschijnlijk is. (Net zoals bij een gewone binaire zoekboom - voor de vorm van de boom is er geen onderscheid tussen de dimensies.) In het slechtste geval echter is de boom een gelinkte lijst met lengte (hoogte) $O(n)$.

Door de afwisselende dimensies zijn er geen rotaties mogelijk om een dergelijke boom evenwichtig te maken. Het beste wat men kan doen is af en toe een deelboom evenwichtig *reconstrueren*. Om dezelfde reden kan er evenmin verwijderd worden. Als men te verwijderen knopen merkt ('lazy deletion'), kan men ze bij een eventuele reconstructie effectief verwijderen.

Zoeken naar de punten in een rechthoekig zoekgebied komt dus neer op het vinden van alle gebieden in de boom die het gegeven gebied overlappen. Voor een groot zoekgebied, of als het slecht gelegen is, kan het nodig zijn om heel de boom te doorlopen.

Een punt zoeken in of toevoegen aan een boom met hoogte h is $O(h)$, en dus afhankelijk van het evenwicht van de boom. De efficiëntie van zoeken in een gebied wordt naast dat evenwicht ook bepaald door het aantal gevonden punten r (dus resultaatafhankelijk). Voor een boom met perfect evenwicht kan men aantonen dat de performantie in het slechtste geval $O(r + \sqrt{n})$ bedraagt. Voor een random opgebouwde boom is de gemiddelde performantie $O(r + n^\alpha)$ (met $\alpha = (\sqrt{17} - 3)/2$).

Er bestaan nog andere gegevensstructuren voor zoeken in (hyper)rechthoek, maar een k-d boom is de enige die slechts $O(n)$ geheugenruimte vereist voor een behoorlijke performantie.

HOOFDSTUK 5
PRIORITEITSWACHTRIJEN II

5.1 SAMENVOEGBARE HEAPS: EEN OVERZICHT

De standaardoperaties ondersteund door een prioriteitswachtrij zijn toevoegen en het minimum (het element met de kleinste prioriteit) verwijderen. (Uiteraard moet men ook kunnen testen of de prioriteitswachtrij ledig is.) Vaak zijn bijkomende operaties op een willekeurig element mogelijk, zoals de prioriteit verminderen, of verwijderen. In beide gevallen gaat men er van uit dat het element reeds gelokaliseerd werd. Sommige algoritmen vereisen nog een andere operatie: samenvoegen van twee prioriteitswachtrijen.

De gebruikelijke implementatie van een prioriteitswachtrij is een binaire heap (eventueel een d-heap), opgeslagen in een tabel. Met n elementen is de performantie van al deze operaties, behalve samenvoegen, $O(\lg n)$. Samenvoegen is echter $O(n)$, omdat een van de tabellen moet gekopieerd worden. Wanneer prioriteiten verminderen veel frequenter gebeurt dan toevoegen of het minimum verwijderen (zoals bij de belangrijke algoritmen van Dijkstra en Prim), of wanneer efficiënt samenvoegen vereist is (zoals bij het algoritme van Edmonds voor een gerichte MOB), kunnen andere heapimplementaties voor een betere performantie zorgen. Aangezien een complete binaire boom, opgeslagen in een tabel, hiervoor een te rigide structuur blijkt, maken al deze implementaties gebruik van bomen met expliciete wijzers naar knopen. Samenvoegen is bij al deze structuren efficiënt, terwijl de andere operaties even efficiënt blijven, en sommigen verbeteren ook nog de efficiëntie van prioriteit verminderen. Er bestaan talrijke alternatieven, waarvan sommige vooral van theoretisch belang zijn. We overlopen even de bekendste:

- *Leftist heaps.* Zoals binaire heaps zijn dit binaire bomen met de heapvoorwaarde voor de opgeslagen sleutels. Het zijn echter geen complete binaire bomen, integendeel, ze trachten zo onevenwichtig mogelijk te zijn. Zoals de naam aangeeft is hun linkerkant diep, zodat de rechterkant ondiep blijft. De operaties zijn efficiënt omdat men ervoor zorgt dat al het werk in de rechterkant gebeurt.

  De fundamentele operatie is samenvoegen van twee heaps. Toevoegen is samenvoegen van de heap met een triviale heap, die enkel het nieuw element bevat. Het minimum wordt verwijderd door de wortel te verwijderen, en zijn deelbomen (ook heaps) dan samen te voegen. Al deze operaties zijn $O(\lg n)$. Een leftist heap construeren uit n elementen kan op dezelfde manier gebeuren als bij een

binaire heap, en is dus $O(n)$. Dat is echter niet optimaal, omdat een binaire heap weliswaar ook een leftist heap is, maar de slechtst mogelijke. Bovendien is deze constructie eenvoudiger in een tabel dan met wijzers.

Volgens Knuth [26] zijn leftist heaps reeds voorbijgestreefd, tenzij voor speciale gevallen.

- *Skew heaps.* Dit zijn zelforganiserende versies van leftist heaps, met een zeer eenvoudige implementatie. Ook hier is samenvoegen de fundamentele operatie. Het zijn opnieuw binaire bomen met de heapvoorwaarde, maar zonder vormbeperkingen. In het slechtste geval kunnen individuele operaties dus $O(n)$ zijn. Maar zoals bij splay trees is een reeks van m opeenvolgende operaties steeds $O(m \lg n)$, ook in het slechtste geval. De geamortiseerde efficiëntie van de operaties is dus $O(\lg n)$. Deze van samenvoegen en toevoegen is zelfs $O(1)$.

- *Binomial queues.* Deze prioriteitswachtrijen bestaan uit een bos van binomiaalbomen, elk met de heapvoorwaarde. Een binomiaalboom met hoogte h heeft $\binom{h}{d}$ knopen op diepte d, vandaar de naam.

  Samenvoegen, toevoegen, en het minimum verwijderen zijn in het slechtste geval weer $O(\lg n)$. Maar geamortiseerd is toevoegen $O(1)$. Bovendien kan men ervoor zorgen dat ze bijkomende operaties op een willekeurig (gelokaliseerd) element ondersteunen, zoals verminderen van de prioriteit, of verwijderen. Hun performantie is eveneens $O(\lg n)$. Meer details hieronder.

- *Fibonacci heaps.* Deze zijn een veralgemening van binomial queues. Ze verstoren de structuur van een binomial queue door knopen met te weinig deelbomen toe te laten. Ze gebruiken ook een snellere implementatie voor verminderen van de prioriteit. Hun binomiaalbomen worden enkel samengevoegd wanneer nodig ('lazy merging'), zodat ze soms veel meer bomen bevatten. De geamortiseerde performantie van samenvoegen, toevoegen en prioriteit verminderen is $O(1)$, die van het minimum verwijderen en een (gelokaliseerd) element verwijderen is $O(\lg n)$. Ze zijn dus vooral interessant wanneer verminderen van de prioriteit frequenter gebeurt dan verwijderen van het minimum. De verborgen constante van hun implementatie is echter te groot, zodat ze voornamelijk van theoretisch belang zijn.

- *Pairing heaps.* Een pairing heap is een zelforganiserende, gestroomlijnde versie van een binomial queue. Experimenten hebben aangetoond dat ze sneller zijn dan andere heapstructuren wanneer verminderen van de prioriteit vereist is [30, 38]. Ze worden dan ook gebruikt in de GNU C++ library en de LEDA library,MehlhornNaher2000. De analyse van hun performantie is lastig, en nog niet voltooid. Men weet reeds lang dat de geamortiseerde performantie van samenvoegen, toevoegen, minimum verwijderen, en prioriteit verminderen $O(\lg n)$ is. Voor minimum verwijderen kan deze bovengrens niet verbeterd worden, want ze is optimaal, maar voor de andere operaties moet ze blijkbaar te ruim zijn. Er worden toch vorderingen gemaakt [31]. De meest waarschijnlijke reden voor hun efficiëntie is hun eenvoud. Meer details hieronder.

- *Relaxed heaps.* Ook deze heaps zijn een veralgemening van binomial queues. Er zijn twee soorten relaxed heaps. De eerste, een 'rank-relaxed heap', heeft dezelfde geamortiseerde performantie als een Fibonacci heap. De tweede, een 'run-relaxed heap', laat overtredingen van de heapvoorwaarde toe, en houdt alle overtreders bij in een aparte gegevensstructuur. De performantie van minimum verwijderen en prioriteit verminderen blijft weliswaar dezelfde, maar ze geldt nu ook in het slechtste geval.

5.2 BINOMIAL QUEUES.

Bij leftist en skew heaps zijn de basisoperaties samenvoegen, toevoegen, en het minimum verwijderen, allemaal $O(\lg n)$. Bij binaire heaps echter is toevoegen gemiddeld $O(1)$. Kan een heap die samenvoegen toelaat ook deze performantie voor toevoegen halen?

5.2.1 Structuur

Een binomial queue (Vuillemin, 1978) bestaat uit een bos van binomiaalbomen, elk met de heapvoorwaarde. Een binomiaalboom wordt gekenmerkt door zijn hoogte: er is slechts één binomiaalboom B_h met hoogte h mogelijk. Hun definitie is recursief: B_0 bestaat uit één knoop, B_h bestaat uit twee B_{h-1} bomen, de tweede is de meest linkse deelboom van de wortel van de eerste. Binomiaalboom B_h bestaat dus uit een wortel met als kinderen (van rechts naar links) $B_0, B_1, \ldots, B_{h-1}$. Op diepte d zijn er dan $\binom{h}{d}$ knopen, de volledige boom heeft dus 2^h knopen.

Een prioriteitswachtrij met willekeurige grootte kan steeds voorgesteld worden door een bos binomiaalbomen, dat hoogstens één boom van elke hoogte bevat. Er zijn evenveel bomen als er enen zijn in de binaire voorstelling van het aantal elementen, en met 2^i komt B_i overeen. (Zo bijvoorbeeld wordt een prioriteitswachtrij met 13 elementen voorgesteld door $\langle B_3, B_2, B_0 \rangle$.)

5.2.2 Operaties

- Om het minimum te vinden moeten alle wortels van de bomen overlopen worden. Aangezien er hoogstens $\lg n$ bomen zijn is dat $O(\lg n)$. Als de plaats van het minimum bijgehouden wordt, kan dat gereduceerd worden tot $O(1)$.

- Samenvoegen telt bomen met dezelfde hoogte bij elkaar op. Twee bomen B_h worden opgeteld door de wortel met de grootste sleutel kind te maken van deze met de kleinste. Het resultaat is een boom B_{h+1}. Drie dergelijke bomen optellen (er kan bij het optellen immers een overdracht zijn) geeft één boom B_h en één boom B_{h+1}. Bij elke redelijke implementatie is binomiaalbomen samenvoegen

$O(1)$, en aangezien er $O(\lg n)$ bomen zijn, is samenvoegen van twee binomial queues $O(\lg n)$. Om dit efficiënt te maken moeten de binomiaalbomen gerangschikt op hoogte bijgehouden worden.

- Toevoegen is weer een speciaal geval van samenvoegen: een knoop met het toe te voegen element wordt een triviale binomial queue. De performantie is dus $O(\lg n)$. Men kan aantonen dat n toevoegoperaties op een ledige binomial heap in het slechtste geval $O(n)$ zijn, zodat toevoegen geamortiseerd $O(1)$ is, zoals gewenst.
- Om het minimum te verwijderen zoeken we de binomiaalboom B_k met het kleinste wortelelement. Die wordt verwijderd uit het bos, wat een nieuwe binomial queue oplevert. Verder verwijderen we de wortel in B_k, zodat we k binomiaalbomen $B_0, B_1, B_2, \ldots, B_{k-1}$ krijgen, die samen een tweede binomial queue vormen. Tenslotte worden beide binomial queues samengevoegd. De performantie van deze operatie is $O(\lg n)$, want B_k zoeken, beide binomial queues creëren, en ze samenvoegen, zijn elk ook $O(\lg n)$.

Men kan ervoor zorgen dat binomial queues bijkomende operaties ondersteunen. Een prioriteit verminderen kan gebeuren door naar de wortel toe te bewegen. Verwijderen is een combinatie van prioriteit verminderen en het minimum verwijderen. Beide operaties vereisen dat het element reeds gelokaliseerd werd, en zijn $O(\lg n)$.

5.2.3 Implementatie

Een binomial queue wordt voorgesteld als een tabel van binomiaalbomen. Voor de binomiaalbomen gebruikt men de standaard binaire boomvoorstelling van een algemene boom: elke knoop bevat een wijzer naar zijn linkerkind, en een naar zijn rechterbroer. De gelinkte lijst van kinderen van een knoop wordt gerangschikt volgens dalend aantal knopen, om samenvoegen van binomiaalbomen te vergemakkelijken.

5.3 PAIRING HEAPS.

Een pairing heap (Fredman, Sedgewick, Sleator, en Tarjan, 1986) is een algemene boom waarvan de sleutels voldoen aan de heapvoorwaarde. Voor de boom gebruikt men de standaard binaire boomvoorstelling: elke knoop bevat een wijzer naar zijn linkerkind, en een naar zijn rechterbroer. Wanneer verminderen van de prioriteit moet ondersteund worden, heeft elke knoop een extra wijzer nodig: een linkerkind naar zijn ouder, een rechterbroer naar zijn linkerbroer. (We blijven echter de terminologie van de algemene boom gebruiken.)

De operaties verlopen als volgt:

- Samenvoegen vergelijkt de wortelelementen van beide heaps. De wortel met het grootste element wordt het linkerkind van deze met het kleinste element. (Waarom het linkerkind?)

- Toevoegen is weer een speciaal geval van samenvoegen. (Toevoegen geeft een wijzer terug naar de nieuwe knoop, die later kan gebruikt worden als de prioriteit van die knoop moet verminderen.)

- Wanneer men de prioriteit van een (gelokaliseerde) knoop vermindert, moet men nagaan of de heapvoorwaarde ten opzichte van de ouder (als die bestaat) nog steeds voldaan is. Dat is niet zo eenvoudig, omdat de meeste knopen geen ouderwijzer hebben. Daarom wordt de knoop losgekoppeld van zijn ouder, zodat hij wortel wordt van een nieuwe heap. Deze wordt dan samengevoegd met de oorspronkelijke (gewijzigde) heap.

- Het minimum verwijderen gebeurt door de wortel te verwijderen, wat een collectie van c heaps oplevert. Daaruit kan een nieuwe heap gemaakt worden door $c - 1$ keer samen te voegen. In het slechtste geval is deze operatie $O(n)$, omdat $c = O(n)$. Door echter de volgorde van samenvoegen zorgvuldig te kiezen, kan men een geamortiseerde performantie van $O(\lg n)$ bekomen. (De naïeve manier die begint met een willekeurige heap en er de andere heaps een voor een aan toevoegt, is geamortiseerd $\Theta(n)$. Want een combinatie van toevoegen gevolgd door verwijderen van het minimum, die $\Omega(n)$ is, kan de oorspronkelijke heap herstellen, zodat men weer kan herbeginnen.)

  De standaardmanier gebruikt twee doorgangen: eerst worden de heaps in paren samengevoegd, te beginnen van links. (Als c oneven is blijft de laatste heap ongewijzigd.) Daarna begint men bij de meest rechtse (nieuwe) heap, en voegt er de anderen een voor een aan toe, van rechts naar links.

- Een willekeurige (gelokaliseerde) knoop verwijderen gebeurt door de knoop los te koppelen van zijn ouder, zodat hij wortel wordt van een nieuwe heap. Het minimum van die nieuwe heap wordt dan verwijderd (de wortel dus), en de rest van die heap wordt samengevoegd met de oorspronkelijke (gewijzigde) heap.

DEEL 2

GRAFEN II

HOOFDSTUK 6
TOEPASSINGEN VAN DIEPTE-EERST ZOEKEN

In Grafen I kwamen reeds enkele toepassingen van diepte-eerst zoeken aan bod: het opsporen van lussen en topologisch rangschikken. Dit hoofdstuk behandelt nog wat meer toepassingen, en ook later zullen we er nog gebruik van maken. Het belang van deze methode werd pas duidelijk met de publicaties van Hopcroft en Tarjan (tussen 1972 en 1974). Zoals vroeger is n het aantal knopen en m het aantal verbindingen.

6.1 ENKELVOUDIGE SAMENHANG VAN GRAFEN

6.1.1 Samenhangende componenten van een ongerichte graaf

Een ongerichte graaf heet *samenhangend*, wanneer er een weg bestaat tussen elk paar knopen. En een niet samenhangende ongerichte graaf bestaat dan uit (zo groot mogelijke) samenhangende componenten. Sommige graafalgoritmen verwachten dat de graaf samenhangend is, zodat men eerst die componenten moet bepalen, om ze daarna apart te behandelen.

Diepte-eerst zoeken vindt alle knopen die met de wortel van de diepte-eerst boom (en dus ook onderling) verbonden zijn. De ongerichte graaf is dus samenhangend wanneer die boom alle knopen bevat. Als er meerdere bomen zijn, vormt elke boom een samenhangende component. In plaats van diepte-eerst zoeken kan hier ook breedte-eerst zoeken gebruikt worden. Testen of een ongerichte graaf samenhangend is, en eventueel de componenten bepalen, is dus $\Theta(n + m)$ (ijle graaf) of $\Theta(n^2)$ (dichte graaf).

6.1.2 Sterk samenhangende componenten van een gerichte graaf

Een gerichte graaf met een weg tussen elk paar knopen, in *beide* richtingen, noemt men *sterk samenhangend* ('strongly connected'). Een gerichte graaf die niet sterk samenhangend is, maar toch samenhangend wanneer men de richting van de verbindingen even buiten beschouwing laat, heet *zwak samenhangend* ('weakly connected').

Een gerichte graaf die niet sterk samenhangend is, bestaat uit (zo groot mogelijke) *sterk*

samenhangende componenten. Sommige graafalgoritmen gaan ervan uit dat de graaf sterk samenhangend is, zodat men eerst die componenten zal moeten identificeren, om ze dan apart te behandelen. Soms stelt men daartoe een *componentengraaf* op, die een knoop heeft voor elke sterk samenhangende component, en een verbinding van knoop a naar knoop b als er in de originele graaf een verbinding bestaat vanuit een van de knopen van a naar een van de knopen van b. Bemerk dat deze componentengraaf geen lussen heeft. (Waarom?) De behandeling van de verschillende sterk samenhangende componenten gebeurt dan door de verbindingen van de componentengraaf te volgen.

De sterk samenhangende componenten kunnen opnieuw gevonden worden met diepte-eerst zoeken (Tarjan, 1972). Er bestaat een eenvoudiger algoritme dat tweemaal diepte-eerst zoeken gebruikt, en dus een constante factor trager is, maar gemakkelijker om uit te leggen en te verantwoorden (Kosaraju, 1978, en Sharir, 1981):

(1) Eerst stelt men de *omgekeerde graaf* op, door de richting van elke verbinding om te keren.
(2) Op deze graaf past men dan diepte-eerst zoeken toe, waarbij de knopen in *post-order* genummerd worden. (Hoe groter dat nummer, des te later werd de knoop afgewerkt.)
(3) Tenslotte voert men diepte-eerst zoeken uit op de oorspronkelijke graaf, waarbij men als startknoop steeds de resterende knoop met het hoogste postordernummer neemt. Het resultaat is een diepte-eerst bos, waarvan de bomen de gezochte sterk samenhangende componenten zijn.

Om aan te tonen dat dit inderdaad het geval is, merken we alvast op dat knopen uit verschillende bomen van dat bos niet tot dezelfde sterk samenhangende component kunnen behoren. Want een verbinding van de linkse boom naar de rechtse is onmogelijk.

Blijft de vraag of elk paar knopen van dezelfde boom in beide richtingen verbonden is. Daarvoor tonen we aan dat de wortel van die boom in beide richtingen verbonden is met elk van zijn knopen. Met als gevolg dat elk paar knopen van de boom in beide richtingen verbonden is, via de wortel:

- Via de boomtakken is er zeker een weg van de wortel w naar elk van de knopen u in de boom, en dus is er een weg van u naar w in de omgekeerde graaf.

- Nu blijkt dat w steeds een voorouder is van u in een diepte-eerst boom van de omgekeerde graaf. De twee knopen kunnen immers niet tot verschillende bomen behoren, want door het hogere postordernummer zou w in de rechtse boom moeten liggen, wat onmogelijk is aangezien er een weg bestaat van u naar w. Om dezelfde reden kunnen beide knopen evenmin behoren tot 'nevendeelbomen' in dezelfde boom. De ene knoop moet dus een voorouder zijn van de andere, en door het hogere postordernummer is dat w.

  Daaruit volgt dat er een weg van w naar u bestaat in de omgekeerde graaf, via de boomtakken. En dus is er een weg van u naar w in de oorspronkelijke graaf,

zodat de knopen in beide richtingen verbonden zijn.

Aangezien deze methode in essentie tweemaal diepte-eerst zoeken is, heeft ze een performantie van $\Theta(n+m)$ bij ijle en $\Theta(n^2)$ bij dichte grafen. Het omkeren van de graaf is $\Theta(n+m)$ of $\Theta(n^2)$, en wijzigt dit asymptotisch gedrag niet. (Hoeveel werk vereist zoeken naar overblijvende knopen met het grootste postordernummer?)

6.2 DUBBELE SAMENHANG VAN ONGERICHTE GRAFEN

Stel dat de graaf een ongericht (samenhangend) communicatienetwerk voorstelt, dan kan men zich afvragen of het uitvallen (wegnemen) van een verbinding of van een knoop niet een deel van het netwerk onbereikbaar maakt. Men zou het netwerk zo kunnen configureren dat er meerdere verbindingen of knopen moeten uitvallen vooraleer dat gebeurt.

Er zijn twee verwante definities van dubbele samenhang:

- Een *verbinding* die de ongerichte graaf doet uiteenvallen in twee gescheiden deelgrafen wanneer ze wordt weggenomen, noemt men een *brug* ('bridge'). Een graaf zonder bruggen heet *dubbel lijnsamenhangend* ('2-edge-connected'[1]). Wanneer men alle bruggen uit een graaf verwijdert bekomt men *dubbel lijnsamenhangende componenten* ('2-edge-connected components').

  Men kan aantonen dat er tussen elk paar knopen van een dubbel lijnsamenhangende graaf minstens twee wegen bestaan zonder gemeenschappelijke *verbindingen*. (Gemeenschappelijke knopen zijn dus wel toegelaten.) Ook het omgekeerde geldt: als er tussen elk paar knopen minstens twee dergelijke onafhankelijke wegen bestaan, dan is de graaf dubbel lijnsamenhangend.

- Een *knoop* die de graaf doet uiteenvallen in *tenminste* twee gescheiden deelgrafen wanneer hij weggenomen wordt (samen met al zijn verbindingen), noemt men een *scharnierpunt* ('articulation point'). Een graaf zonder scharnierpunten heet *dubbel samenhangend* ('biconnected'[2]). Een graaf met scharnierpunten kan onderverdeeld worden in bruggen en *dubbel samenhangende componenten* ('biconnected components'). Deze componenten zijn verbonden in scharnierpunten, eventueel via bruggen. Merk op dat twee componenten wel een gemeenschappelijke knoop kunnen hebben, maar nooit een gemeenschappelijke verbinding. (Waarom?) Elke verbinding behoort dus tot één component, of is een brug.

  Men kan aantonen dat er tussen elk paar knopen van een dubbel samenhangende graaf minstens twee wegen bestaan zonder gemeenschappelijke *knopen*. (En

[1] Ook wel 'edge-connected' genoemd, maar dat is niet consistent met de terminologie van meervoudige samenhang (zie later).

[2] Men spreekt niet van '2-vertex-connected', wat consequenter zou zijn.

dus zeker zonder gemeenschappelijke verbindingen.) Ook hier geldt het omgekeerde: als er tussen elk paar knopen twee dergelijke onafhankelijke wegen bestaan, dan is de graaf dubbel samenhangend.

De tweede vorm van samenhang is sterker dan de eerste. Immers, zonder dat de graaf uiteenvalt mag men bij een dubbel lijnsamenhangende graaf enkel een verbinding wegnemen, maar bij een dubbel samenhangende graaf een knoop samen met al zijn verbindingen.

Scharnierpunten, dubbel samenhangende componenten, bruggen en dubbel lijnsamenhangende componenten kunnen opnieuw efficiënt gevonden worden via diepte-eerst zoeken. Daartoe stelt men een diepte-eerst boom op, waarbij men de knopen in *preorder* nummert. (Er is maar één boom, omdat de graaf samenhangend ondersteld wordt.) Bovendien bepaalt men voor elke knoop u de *laagst genummerde knoop* die vanuit u kan bereikt worden via een weg bestaande uit nul of meer dalende boomtakken gevolgd door één terugverbinding. In principe kan dit gebeuren tijdens een tweede postorderdoorgang van de boom, maar beide kunnen ook gecombineerd worden. (Waarom postorder?)

Als alle kinderen van een knoop op die manier een knoop kunnen bereiken die hoger in de boom ligt dan hemzelf, dan is hij zeker geen scharnierpunt. De wortel is een speciaal geval, omdat er natuurlijk geen knoop hoger in de boom ligt. De wortel is een scharnierpunt als hij meer dan één kind in de boom heeft, want de enige verbinding tussen deze kinderen loopt via de wortel. Dit zal men apart moeten testen. Een knoop kan een scharnierpunt zijn voor enkele, of zelfs voor elk van zijn kinderen. (Hoe vindt men een brug?)

Als we voor de eenvoud de test op de wortel achterwege laten, dan zou het opsporen van scharnierpunten er kunnen uitzien zoals in code 6.1. Voor de duidelijkheid gebruiken we een aparte tabel om na te gaan of een knoop reeds ontdekt werd, hoewel de tabel met preordernummers daar ook voor kan dienen.

Bemerk dat eenzelfde knoop meermaals scharnierpunt kan zijn. Als blijkt dat knoop a een scharnierpunt is voor zijn kind b, dan behoren de meest recent ontdekte verbindingen tot een nieuwe dubbel samenhangende component. Immers, geen van de wegen die vanuit b gevonden werden, leidt naar een knoop hoger in de boom dan a (eventueel wel tot bij a). Het volstaat dus om al de ontdekte verbindingen op een *stapel* bij te houden, en ze eraf te halen bij het ontdekken van een scharnierpunt a, tot en met de boomtak (a, b). (Als er in de deelboom met wortel a nog andere scharnierpunten aanwezig zijn, werden de verbindingen van de bijbehorende componenten reeds op dezelfde manier van de stapel gehaald.)

Dit algoritme is in essentie diepte-eerst zoeken, zodat de uitvoeringstijd $\Theta(n + m)$ bedraagt voor ijle grafen en $\Theta(n^2)$ voor dichte.

```
void rec_zoek_scharnierpunten (int i) {
//Voor alle knopen bereikbaar vanuit knoop i
//bereken de preordernummers pre[]
//en laagst bereikbare knoopnummers laagst[]
   ontdekt[i] = true;
   pre[i] = num++;
//Initialisatie: elke knoop bereikt minstens zichzelf
   laagst[i] = pre[i];
   for (alle buren j van knoop i) {
      if (!ontdekt[j]) {
         // Boomtak, en dus een kind van i
         ouder[j] = i;
         rec_zoek_scharnierpunten(j);
         if (laagst[j] < laagst[i])
            laagst[i] = laagst[j]; // Nieuw minimum
         else if (laagst[j] >= pre[i])
            meld dat knoop i scharnierpunt is voor j;
      }
      else
         if (j != ouder[i])
         // Eventueel een terugverbinding
            if (pre[j] < laagst[i])
         //Dan zeker pre[j]<pre[i] (=> terugverbinding)
               laagst[i] = pre[j]; // Nieuw minimum
   }
}
void zoek_scharnierpunten () {
   // Initialisatie
   for (int i = 0 ; i < n ; i++) {
      ontdekt[i] = false;
      ouder[i] = -1; // Nog geen ouder
   }
   // Zoek scharnierpunten
   int num = 0; // Initialisatie preordernummering
   // Onderstel graaf samenhangend
   rec_zoek_scharnierpunten (0);
}
```

Code 6.1. Zoeken van scharnierpunten.

6.3 EULERCIRCUITS

Een Eulercircuit is een gesloten omloop in een (multi)graaf[3] die alle *verbindingen*

[3] Een multigraaf laat meerdere verbindingen tussen twee knopen toe, en ook verbindingen van een knoop

eenmaal bevat (met de juiste richting in een gerichte graaf). Niet elke graaf heeft een Eulercircuit. Een Eulercircuit kan niet efficiënt gevonden worden via diepte-eerst zoeken, maar wel door een rechttoe-rechtaanzoeken. Merkwaardig is wel dat er (heel waarschijnlijk) geen efficiënte oplossing bestaat voor het duale probleem, een Hamiltoncircuit, dat alle *knopen* eenmaal bevat. (Het is immers NP-compleet, zie deel 4.)

6.3.1 Ongerichte grafen

Een Eulergraaf is een (multigraaf) met een Eulercircuit. De volgende drie eigenschappen zijn equivalent:

(1) Een samenhangende (multi)graaf G is een Eulergraaf.
(2) De graad van elke knoop van G is even.
(3) De verbindingen van G kunnen onderverdeeld worden in lussen. (Deze verdeling is een *partitie*: elke verbinding behoort tot één enkele lus.)

Eigenschap twee volgt uit één. Want telkens als knoop k voorkomt op een Eulercircuit draagt dat twee bij tot zijn graad. Aangezien elke verbinding precies eenmaal op dat circuit voorkomt, moet elke knoopgraad even zijn.

Eigenschap drie volgt uit twee. Stel dat G n knopen heeft. Er zijn dan minstens $n - 1$ verbindingen, want G is samenhangend. Geen enkele knoopgraad is echter gelijk aan één, zodat er minstens n verbindingen zijn. Dus bevat G minstens één lus. Als we die lus uit G verwijderen blijft er een niet noodzakelijk samenhangende graaf H over waarvan alle knoopgraden nog steeds even zijn. Elk van de samenhangende componenten van H kan op dezelfde manier onderverdeeld worden in lussen. Tot uiteindelijk de volledige graaf in lussen onderverdeeld is.

Eigenschap één volgt uit drie. Stel dat L een van die lussen van G is. Als L een Eulercircuit is, dan is G een Eulergraaf. Zoniet bestaat er een andere lus L' die een gemeenschappelijke knoop k heeft met L. Lus L' kan bij knoop k tussengevoegd worden in lus L, zodat een grotere lus ontstaat. Als we dit proces herhalen bekomen we uiteindelijk een Eulercircuit, want elke verbinding behoort tot één lus.

De derde eigenschap beschrijft bijna een constructie van een Eulercircuit (het algoritme van Hierholzer, 1873). Om de eerste lus L te vinden beginnen we bij een willekeurige knoop, en volgen verbindingen tot we weer bij die knoop terechtkomen. (Werkt dat altijd?) De volgende lus L' begint bij een van de knopen van L waarvan nog niet alle verbindingen doorlopen zijn. En ook daar volgen we verbindingen tot we weer bij de vertrekknoop terugkeren. Om niet steeds de groeiende lus te moeten doorzoeken naar een knoop met nog ongebruikte verbindingen, testen we ze in volgorde langs de lus. En we blijven nieuwe lussen genereren zolang niet alle verbindingen van een knoop opgebruikt zijn, vooraleer een volgende knoop te zoeken.

---

met zichzelf ('selfloop').

Bemerk dat wanneer men geen gesloten circuit wenst (een ‘Eulerweg’), de knoopgraden van (enkel) de begin- en eindknoop oneven mogen (moeten) zijn.

6.3.2 Gerichte grafen

Een Eulercircuit in een gerichte (multi)graaf is slechts mogelijk als de graaf een (sterk samenhangende) Eulergraaf is. De voorwaarden voor een gerichte Eulergraaf zijn analoog als voor een ongerichte graaf: de ingraad van elke knoop moet gelijk zijn aan zijn uitgraad. (De graaf heet dan ‘pseudosymmetrisch’.) Ook de constructie van het Eulercircuit verloopt analoog.

HOOFDSTUK 7
KORTSTE AFSTANDEN II

In de cursus Gegevensstructuren I hebben we gezien hoe kortste afstanden gezocht worden met het algoritme van Dijkstra. Dit algoritme eist echter niet-negatieve gewichten. Wanneer de gewichten van graafverbindingen negatief kunnen zijn, wordt het bepalen van kortste afstanden ingewikkelder. Lussen met negatief gewicht mogen echter niet voorkomen, zoniet zijn kortste afstanden niet gedefinieerd.

Om algoritmen voor gerichte grafen te kunnen toepassen op ongerichte grafen, maakt men deze gericht door elke verbinding te vervangen door twee tegengesteld gerichte verbindingen, met hetzelfde gewicht. Een ongerichte verbinding met negatief gewicht wordt dan echter een gerichte lus met negatief gewicht. De oplossing voor dit probleem is vrij ingewikkeld (zie bijvoorbeeld [23]). Voor de eenvoud zullen we de grafen dus gericht onderstellen.

7.1 KORTSTE AFSTANDEN VANUIT ÉÉN KNOOP

Als gewichten negatief kunnen zijn, dan werkt de eenvoudige inhalige strategie van het algoritme van Dijkstra niet meer, en moet het algoritme op zijn beslissingen kunnen terugkomen. De aangewezen methode is dan die van Bellman-Ford.

7.1.1 Het algoritme van Bellman-Ford

Dit algoritme (Bellman, 1958 en Ford, 1956, 1962) maakt gebruik van dynamisch programmeren.[1] Het kan ook wel beschouwd worden als een combinatie van breedte-eerst zoeken (niet-gewogen grafen) met kortste afstanden (gewogen grafen).

De methode berust op het feit dat als de graaf geen negatieve lussen heeft, de kortste wegen evenmin lussen hebben, en dus hoogstens $n - 1$ verbindingen bevatten. We kunnen dan een recursief verband opstellen tussen kortste wegen met maximaal k verbindingen en kortste wegen met maximaal $k - 1$ verbindingen. Als $d_i(k)$ het gewicht is van de kortste weg met hoogstens k verbindingen vanuit de startknoop naar knoop i, en g_{ji} het gewicht van de verbinding (j, i), dan geldt dat

[1] Het werk van Bellman in de jaren 1950 zorgde er in belangrijke mate voor dat dynamisch programmeren een algemene algoritmische methode werd. Het Bellman-Fordalgoritme was een van de eerste toepassingen.

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

De gezochte kortste afstanden zijn dan natuurlijk de waarden $d_i(n-1)$, voor elke knoop i.

Om $d_i(k)$ te bepalen voor elke knoop i hebben we telkens de afstanden $d_j(k-1)$ nodig, voor verschillende (of zelfs alle) waarden van j. Deze oplossing heeft duidelijk een optimale deelstructuur, de deeloplossingen zijn onafhankelijk, en overlappend. Alle voorwaarden voor dynamisch programmeren zijn dus voldaan.

Het algoritme bestaat dus uit $n-1$ iteraties, waarbij de kortste wegen telkens één verbinding langer mogen worden. Voor elke knoop moet dan onderzocht worden of zijn kortste afstand verbeterd kan worden via de knopen waarvan hij buur is. Daartoe worden alle voorlopige afstanden in een tabel bijgehouden. Om de kortste wegen te kunnen reconstrueren volstaat het de voorloperknoop op elke voorlopige kortste weg op te slaan.

In principe moet elke iteratie alle m verbindingen testen: de performantie is dus $O(nm)$. Dat is aanzienlijk slechter dan het algoritme van Dijkstra. Gelukkig zijn gewichten gewoonlijk positief.

Er zijn twee goede implementaties:

- Meestal is het niet nodig om in elke iteratie alle verbindingen te onderzoeken. Want als een iteratie de voorlopige kortste afstand tot een knoop niet aangepast heeft, is het zinloos om bij de volgende iteratie de verbindingen vanuit die knoop te onderzoeken. Daarom plaatst men enkel de knopen waarvan de afstand door de huidige iteratie gewijzigd werd in een *wachtrij*, en worden enkel de buren van deze knopen bij de volgende iteratie getest. (Dit is duidelijk verwant met breedte-eerst zoeken.) Een knoop moet dus niet meer nagaan of de afstand van knopen waarvan hij buur is verbeterd werd ('pull-based'), maar wordt als het ware door die knopen van de wijziging verwittigd ('push-based'). Deze verbetering heeft natuurlijk geen invloed op de asymptotische waarde voor de performantie.

  Wanneer een knoop uit de wachtrij verwijderd wordt, en de afstand van één van zijn buren wordt aangepast, dan moet die buur in de wachtrij gestopt worden, ook als hij er reeds vroeger uit verwijderd werd. Als die buur reeds in de wachtrij zit, zullen zijn buren toch met zijn nieuwe afstand getest worden, en kan hij gewoon blijven zitten. Elke knoop kan aldus hoogstens $n-1$ maal in de wachtrij terechtkomen. (Na elke iteratie. De eerste iteratie test enkel de buren van de startknoop. De startknoop zelf komt nooit meer in de wachtrij, als er geen negatieve lus is.)

  Het is niet altijd eenvoudig om vooraf na te gaan of een grote graaf lussen met negatief gewicht heeft. Met een dergelijke lus zou dit algoritme nooit stoppen. Men kan zich daartegen beveiligen door het aantal iteraties bij te houden, en als de wachtrij niet ledig is na n iteraties, dan heeft de graaf een negatieve lus. (Iteratie $n-1$ past normaal de laatste afstanden aan, en deze knopen worden nog in iteratie n onderzocht. Die mogen dan geen kortere afstanden meer opleveren.)

- Een tweede implementatie (Pape, 1974) blijkt in de praktijk zeer snel, maar kan in bepaalde gevallen zeer inefficiënt uitvallen. De werkwijze is vergelijkbaar met de vorige, maar in plaats van in een wachtrij worden nog te onderzoeken knopen in een *deque* bijgehouden. Een knoop wordt zoals bij een gewone wachtrij vooraan weggenomen, maar toevoegen gebeurt zowel voor- als achteraan. Als de afstand van een knoop wordt aangepast, en als die knoop reeds vroeger in de deque zat, dan voegt men vooraan toe, anders achteraan.

  De intuïtie waarop deze heuristiek steunt is de volgende. Als knoop i ooit in de deque zat werd hij verwijderd en werden daarbij al zijn buren onderzocht. Stel dat de afstand van sommige buren werd aangepast, zodat ze in de deque terechtkwamen (met i als voorlopige ouder). Als die nog steeds in de deque zitten op het ogenblik dat de afstand van i opnieuw aangepast wordt, dan is het best mogelijk dat hun afstand nogmaals zou wijzigen met die nieuwe afstand voor i. Maar als men i achteraan toevoegt, dan worden zijn buren eerder verwijderd, en wordt nog hun oude afstand gebruikt om al hun buren te onderzoeken. Het is dus beter om de buren van i zo snel mogelijk aan te passen, en i daarom vooraan toe te voegen. Want daardoor moeten er minder knopen opnieuw onderzocht worden.

  In de praktijk blijkt inderdaad dat deze implementatie minder knopen onderzoekt dan andere, en voor ijle grafen is het een van de snelste methoden. In het slechtste geval is haar performantie echter $O(\min(nmC, m2^n))$, waarbij C (de absolute waarde van) het grootste gewicht van een verbinding voorstelt.

Het algoritme van Bellman-Ford laat niet enkel negatieve gewichten toe, maar is ook meer flexibel en gedecentraliseerd dan dat van Dijkstra. Wanneer de voorlopige kortste afstand tot een knoop wijzigt, kan de afstand van zijn buren mogelijks verbeteren. Die knoop kan dan zijn buren van de wijziging verwittigen. Dat is ideaal voor een netwerk. Het algoritme van Dijkstra daarentegen vereist een *globale* kennis van het netwerk, omdat telkens de randknoop met kortste afstand gekozen wordt. Het Bellman-Ford algoritme wordt dan ook door netwerkrouters gebruikt om de optimale afstand (gewichten zijn transmissiesnelheden) tot elke andere netwerkknoop te bepalen. (Elke knoop wordt op zijn beurt startknoop van het algoritme.) In een netwerk gebeuren deze meldingen en eventuele aanpassingen niet in een aantal iteraties, maar asynchroon. Toch kan men aantonen dat het resultaat hetzelfde blijft.

Aangezien elke knoop een vector van 'afstanden' tot alle andere knopen bijhoudt, heet deze methode het 'distance vector protocol'. Omdat elke knoop slechts een lokale kennis van het netwerk heeft, kunnen veranderende verbindingsgewichten voor problemen zorgen. (Vertraging op overbelaste verbindingen, of zelfs uitgevallen verbindingen.) Daarom gebruikt men liever het 'path vector protocol', waarbij elke knoop de volledige weg naar elke andere knoop bijhoudt. Dat vereist dan wel veel meer geheugen.

7.2 KORTSTE AFSTANDEN TUSSEN ALLE KNOPENPAREN

Voor dichte grafen is er het algoritme van Floyd-Warshall, met performantie $\Theta(n^3)$, dat ook negatieve gewichten toelaat. Voor grote ijle grafen met negatieve gewichten kan men beter het algoritme van Johnson gebruiken, met een performantie van $O(nm \lg n)$.

7.2.1 Het algoritme van Johnson

Dit algoritme (Johnson, 1977) maakt gebruik van de algoritmen van Bellman-Ford en van Dijkstra.

Om toch het algoritme van Dijkstra vanuit elke knoop te kunnen gebruiken, krijgen alle verbindingen een nieuw positief gewicht. Het spreekt natuurlijk vanzelf dat kortste wegen ook met de nieuwe gewichten kortste wegen moeten blijven.

Daartoe breidt men eerst de graaf uit met een nieuwe knoop s, die verbindingen van gewicht nul krijgt naar alle andere knopen. Als de oorspronkelijke graaf geen negatieve lussen had, dan geldt dit eveneens voor de nieuwe graaf, want knoop s heeft enkel uitgaande verbindingen. We kunnen dus het algoritme van Bellman-Ford toepassen op de nieuwe graaf, om vanuit s de kortste afstand d_i te bepalen tot elke originele knoop i. (Bemerk dat al deze afstanden nul zijn wanneer de originele graaf geen negatieve gewichten heeft.)

Deze afstanden gebruiken we om de nieuwe gewichten te bepalen. Als g_{ij} het oorspronkelijk gewicht van verbinding (i, j) was, dan wordt haar nieuw gewicht $\widehat{g}_{ij} = g_{ij} + d_i - d_j$. Aangezien d_i en d_j kortste afstanden zijn, geldt $d_j \leq d_i + g_{ij}$. En dus is $\widehat{g}_{ij}$ nooit negatief.

Blijft de vraag of kortste wegen met de nieuwe gewichten dezelfde zijn als met de originele gewichten. Stel eens dat w_{ij} het originele gewicht is van de kortste weg tussen knopen i en j, en dat $\widehat{w}_{ij}$ het nieuw gewicht is van diezelfde weg. Als er met de nieuwe gewichten een kortere weg tussen i en j zou zijn, met (nieuw) gewicht $\widehat{w}'_{ij}$, en origineel gewicht w'_{ij}, dan zou $\widehat{w}'_{ij} < \widehat{w}_{ij}$ zijn, zodat $w'_{ij} + d_i - d_j < w_{ij} + d_i - d_j$ en dus $w'_{ij} < w_{ij}$, wat betekent dat de eerste weg met de originele gewichten toch niet de kortste was.

Het algoritme van Dijkstra kan dus toegepast worden op elke originele knoop (in de nieuwe graaf), en zal de correcte kortste *wegen* tussen alle knopenparen vinden. (Bemerk dat geen van die wegen via s kan gaan.) Voor de kortste *afstanden* moeten natuurlijk opnieuw de originele gewichten gebruikt worden.

De performantie van deze methode is eenvoudig te bepalen. De graaf uitbreiden is $\Theta(n)$, Bellman-Ford is $O(nm)$, de gewichten aanpassen is $\Theta(m)$, en n maal Dijkstra tenslotte is $O(n(n+m) \lg n)$. De laatste term is de belangrijkste, zodat de methode ook

$O(n(n+m)\lg n)$ is, te vereenvoudigen tot $O(nm \lg n)$. (Als de d-heap bij Dijkstra vervangen wordt door een Fibonacciheap, zou de performantie $O(nm + n^2 \lg n)$ worden, maar zoals eerder vermeld, zijn de verborgen constanten te groot om competitief te zijn.) Voor ijle grafen is dit beter dan Floyd-Warshall.

7.3 TRANSITIEVE SLUITING

Een *sluiting* is een algemene methode om een of meerdere verzamelingen op te bouwen. Daartoe past men herhaaldelijk een of meerdere regels toe van de vorm 'als een verzameling deze gegevens bevat, dan moet ze ook de volgende gegevens bevatten', tot er niets meer aan de verzameling(en) kan toegevoegd worden. (Om die reden noemt men een sluiting ook wel een *'fixed point'*-bepaling, omdat verdere toepassing van de regels niets meer verandert: $f(x) = x$.) Uiteraard moet(en) deze verzameling(en) geïnitialiseerd worden (afhankelijk van het probleem). Bemerk dat de regels wel bepalen wat er in de verzameling(en) moet komen, maar niet specificeren wat er niet in mag. Daarom legt men de bijkomende voorwaarde op dat er niets anders in mag dan wat de regels vermelden. (Dat heet dan ook een *'least'* fixed point bepaling, omdat ze de kleinste x zoekt waarvoor $f(x) = x$.) Een belangrijk speciaal geval is een *transitieve sluiting*, waarbij de regels de vorm 'als (a, b) en (b, c) aanwezig zijn dan moet ook (a, c) aanwezig zijn' hebben.

De transitieve sluiting van een gerichte graaf is een nieuwe gerichte graaf met dezelfde knopen, en een verbinding van knoop i naar knoop j als er een weg van i naar j bestaat in de oorspronkelijke graaf. (Hoe vindt men de transitieve sluiting van een ongerichte graaf?)

De (enige) verzameling die in ons geval moet opgebouwd worden is deze met de verbindingen van de nieuwe graaf. Er zijn drie manieren om de transitieve sluiting te bepalen:

- De eenvoudigste manier gebruikt diepte-eerst of breedte-eerst zoeken om alle knopen op te sporen die vanuit een startknoop bereikbaar zijn. Dit wordt herhaald met elke knoop als startknoop. Voor ijle grafen is de performantie dan $\Theta(n(n+m))$, voor dichte grafen wordt dat $\Theta(n^3)$. Deze methode is vooral interessant voor ijle grafen, want voor dichte grafen is de derde methode, hoewel ook $\Theta(n^3)$, door zijn eenvoud sneller.
- Als men verwacht dat de transitieve sluiting een *dichte* graaf zal zijn, dan zijn veel knopen onderling bereikbaar en dus heeft de graaf een gering aantal sterk samenhangende componenten. Die componenten kunnen in $\Theta(n+m)$ bepaald worden, en alle knopen in dezelfde component zijn onderling bereikbaar. Daarna construeert men de *componentengraaf* (zie hoger). Deze componentengraaf zal heel wat kleiner zijn dan de oorspronkelijke, en kan opgesteld worden in $O(n+m)$. Het probleem herleidt zich nu tot het bepalen van de transitieve sluiting

van de (lusloze) componentengraaf. Als dan blijkt dat component j bereikbaar is vanuit component i, dan zijn alle knopen van j bereikbaar vanuit alle knopen van i.

- Voor dichte grafen is het algoritme van Warshall aangewezen (Warshall, 1962). De methode is analoog met die van Floyd-Warshall voor kortste afstanden, die er trouwens uit afgeleid werd.

  Dus wordt er opnieuw een reeks opeenvolgende $n \times n$ matrices $T^{(0)}, T^{(1)}, \ldots, T^{(n)}$ bepaald die nu echter *logische waarden* bevatten. Het element $t_{ij}^{(k)}$ duidt aan of er een weg bestaat tussen i en j met als *mogelijke* intermediaire knopen $1, 2, \ldots, k$. Verbinding (i, j) behoort dan tot de transitieve sluiting als $t_{ij}^{(n)}$ waar is: $T = T^{(n)}$ is de gezochte burenmatrix.

  De opeenvolgende matrices worden als volgt bepaald:

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

  en

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

  waarbij $\vee$ en $\wedge$ de gebruikelijke symbolen zijn voor de logische operaties 'of' resp. 'en'. Ook deze berekening kan ter plaatse gebeuren, en vereist slechts één matrix. De efficiëntie is eveneens $\Theta(n^3)$, met mogelijks nog een kleinere verborgen constante dan bij Floyd-Warshall, en bovendien vereisen logische waarden minder plaats dan getallen.

HOOFDSTUK 8
STROOMNETWERKEN

Een stroomnetwerk ('flow network') is een gerichte graaf met twee speciale knopen, een *producent* ('source') en een *verbruiker* ('sink'). De producent levert een niet nader bepaald materiaal met een gelijkmatig debiet, en de verbruiker verwerkt dat materiaal met hetzelfde debiet. Het materiaal 'stroomt' van producent naar verbruiker via de gerichte verbindingen van de graaf. Daarbij wordt ondersteld dat elke knoop van de graaf bereikbaar is vanuit de producent, en ook dat de verbruiker vanuit elke knoop kan bereikt worden. De graaf mag lussen bevatten.

Elke gerichte verbinding kan beschouwd worden als een soort pijpleiding voor het materiaal, en met als gewicht een bepaalde *capaciteit*, die het grootste debiet voorstelt dat er doorheen kan stromen. De knopen zijn enkel kruispunten van pijpleidingen, en kunnen geen materiaal opslaan (producent en verbruiker zijn uitzonderingen). Alles wat er in een knoop toestroomt, moet er dus ook weer uit wegstromen. De stroom in de knopen heet daarom *conservatief*.

Een dergelijk stroomnetwerk kan gebruikt worden als model voor vloeistoffen in pijpleidingen, onderdelen op transportbanden, stroom in elektrische netwerken, informatie in communicatienetwerken, enz. Maar men kan er ook tal van minder evidente problemen mee modelleren en oplossen. Deze problemen liggen op de grens tussen computerwetenschappen en 'operations research', een vakgebied dat wiskundige modellen voor complexe systemen bestudeert, om optimale oplossingen voor die systemen te zoeken. Veel problemen uit dat vakgebied kunnen opgelost worden met 'lineair programmeren', een belangrijke algemene methode. Wanneer men ze echter als stroomnetwerk kan modelleren zijn er snellere oplossingen mogelijk. Stroomnetwerken kunnen daarom beschouwd worden als een belangrijke algoritmische methode, naast de verdeel-en-heers methode, de inhalige methode, en dynamisch programmeren.

8.1 MAXIMALESTROOMPROBLEEM

Het eenvoudigste probleem in stroomnetwerken is het *maximalestroomprobleem* ('maximum flow problem'). Dat bestaat erin om zoveel mogelijk materiaal van producent naar verbruiker te laten stromen, zonder de capaciteit van de verbindingen te overschrijden.

De klassieke methode om dit probleem op te lossen is die van Ford-Fulkerson (1956). Het is een *methode*, geen algoritme, omdat ze meerdere implementaties toelaat, met

verschillende performanties. De methode is iteratief, en bij elke iteratie neemt de nettostroom vanuit de producent toe, tot uiteindelijk het maximum bereikt wordt. Dat betekent niet dat de stroom in de verbindingen telkens toeneemt: soms vindt men alternatieve wegen, en moet de stroom in bepaalde verbindingen zelfs dalen.

De capaciteit van elke verbinding (i, j) is een positief getal $c(i, j)$, en de stroom $s(i, j)$ die er door loopt wordt beperkt door die capaciteit: $0 \leq s(i, j) \leq c(i, j)$. Als we de verzameling van alle knopen K noemen, dan is de totale nettostroom f uit de producent p

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

In feite moet de som genomen worden over alle $j \in K$ waarvoor er een verbinding (p, j) en/of een verbinding (j, p) bestaat. Om de notatie wat te vereenvoudigen zullen we onderstellen dat er een verbinding bestaat tussen *elk* paar knopen. Een niet bestaande verbinding krijgt dan capaciteit nul, zodat de stroom erdoor steeds nul is. De waarde f noemt men de *netwerkstroom*, en die moet maximaal worden.

Het geheel van alle stromen voor alle mogelijke knopenparen, in beide richtingen, heet een *stroomverdeling*. De methode van Ford-Fulkerson start met een initiële stroomverdeling die overal nul is. Bij elke iteratie gebruikt ze de huidige stroomverdeling en de verschillende capaciteiten om een overzicht op te stellen van de mogelijke stroomtoename tussen elk paar knopen. Dit overzicht is een nieuw stroomnetwerk, met dezelfde knopen, maar niet noodzakelijk dezelfde verbindingen en capaciteiten: het *restnetwerk* ('residual network'). In dat restnetwerk wordt dan een weg van producent naar verbruiker gezocht die stroom tussen beide toelaat: de *vergrotende weg* ('augmenting path'). Met die extra stroom wordt de stroomverdeling in het oorspronkelijk netwerk aangepast (en de netwerkstroom vergroot). Als er geen vergrotende weg meer gevonden wordt, stopt de iteratie, en zal de netwerkstroom maximaal zijn.

We zullen eerst het restnetwerk en de vergrotende weg wat nader bekijken:

- Voor elke mogelijke stroomverdeling in een stroomnetwerk, bestaat er een overeenkomstig restnetwerk. Dat is eveneens een stroomnetwerk, met dezelfde knopen, maar met enkel de verbindingen die meer stroom kunnen toelaten. Meer stroom van knoop i naar knoop j betekent dat de capaciteit van de verbinding (i, j) nog niet volledig gebruikt werd ($s(i, j) < c(i, j)$), en/of dat er stroom loopt over de verbinding (j, i), die kleiner kan gemaakt worden. In het restnetwerk krijgt de verbinding (i, j) dan ook een (rest)capaciteit $c_r(i, j) = c(i, j) - s(i, j) + s(j, i)$, die uiteraard positief is. Bemerk dat de restcapaciteit zelfs groter kan zijn dan de originele capaciteit, als $s(i, j) < s(j, i)$. Meer nog, als het netwerk geen verbinding (i, j) bevat ($c(i, j) = 0$), maar wel de verbinding (j, i) waarover een positieve stroom loopt, dan heeft het restnetwerk toch een verbinding (i, j)! De verbindingen van het restnetwerk vormen dus niet noodzakelijk een deelverzameling van de originele verbindingen.
- Een vergrotende weg in een restnetwerk is een enkelvoudige weg (zonder lus) van producent naar verbruiker. Bij definitie heeft elke verbinding op die weg

een positieve (rest)capaciteit, en kan dus nog meer stroom doorlaten. Tussen producent en verbruiker is dan via deze weg een extra stroom mogelijk gelijk aan de kleinste restcapaciteit op die weg. De stroom in de overeenkomstige verbindingen van het eigenlijke stroomnetwerk wordt daarmee aangepast. Een stroomtoename van knoop i naar knoop j kan verwezenlijkt worden door een toename van $s(i,j)$, of door een afname van $s(j,i)$, of door een combinatie van de twee (als beide verbindingen bestaan).

Om aan te tonen dat de methode van Ford-Fulkerson correct is, hebben we het begrip *snede* ('cut') in een netwerk nodig. Zoals bekend is een snede (P,V) van een samenhangende graaf een verzameling verbindingen die de knopenverzameling in twee niet-ledige stukken P en V verdeelt. Dit houdt in dat een verbinding (i,j) in (P,V) zit als en slechts als $i \in P$ en $j \in V$ of, omgekeerd, $j \in P$ en $i \in V$. Bij de sneden die voor ons nuttig zijn behoort p tot P en v tot V. De capaciteit $c(P,V)$ van de snede (P,V) wordt gedefinieerd als de som van alle capaciteiten $c(i,j)$, met i in P en j in V. De nettostroom $f(P,V)$ van de snede (P,V) is de som van alle voorwaartse stromen $s(i,j)$, min de som van alle achterwaartse stromen $s(j,i)$, met i in P en j in V. Aangezien $s(i,j) \leq c(i,j)$ is $f(P,V) \leq c(P,V)$, voor elke snede (P,V).

De conservatieve eigenschap heeft als belangrijk gevolg dat de netwerkstroom f gelijk is aan de nettostroom $f(P,V)$ van *elke mogelijke* snede. Hoewel dit wellicht intuïtief duidelijk is, kan het gemakkelijk aangetoond worden. De stroom van het netwerk is bij definitie de nettostroom vanuit de producent p:

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

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

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

Als we de som maken van deze betrekkingen voor alle knopen van P (p inbegrepen), dan bekomen we dat

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

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

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

Aangezien de nettostroom van een snede nooit groter kan zijn dan haar capaciteit, kan de netwerkstroom niet groter zijn dan de capaciteit van elke snede, en dus niet groter

dan de minimale snedecapaciteit. De belangrijke 'maximale stroom minimale snede'-stelling ('max-flow min-cut') zegt dat deze maximale netwerkstroom bereikt wordt als het overeenkomstig restnetwerk geen vergrotende weg meer heeft. De volgende drie eigenschappen zijn immers equivalent:

(1) De netwerkstroom f is maximaal.
(2) Er is geen vergrotende weg meer te vinden in het restnetwerk.
(3) De netwerkstroom f is gelijk aan de capaciteit van *een* snede in de oorspronkelijke graaf.

Eigenschap twee volgt uit één, want als er nog een vergrotende weg overblijft, dan kan de netwerkstroom f zeker groter gemaakt worden.

Eigenschap drie volgt uit twee. Beschouw een snede (P, V) in het stroomnetwerk waarvan P alle knopen bevat die *in het restnetwerk* bereikbaar zijn vanuit de producent p. Aangezien er geen vergrotende weg meer is, behoort de verbruiker v zeker niet tot P. De restcapaciteit van elke verbinding (i, j), met $i \in P$ en $j \in V$, moet dan nul zijn. Zoniet zou knoop j bereikbaar zijn vanuit p in het restnetwerk, en dus niet in V liggen. In het stroomnetwerk zelf moet dan $f(P, V) = c(P, V)$, want als

$$\sum_{i \in P} \sum_{j \in V} c_r(i, j) = \sum_{i \in P} \sum_{j \in V} (c(i, j) - s(i, j) + s(j, i)) = 0$$

dan is

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

Tenslotte volgt de eerste eigenschap uit de derde, want we weten al dat de netwerkstroom gelijk is aan de nettostroom doorheen elke snede, en de nettostroom van de snede hierboven heeft zijn maximum bereikt. Overigens moet dat de snede zijn met de kleinste capaciteit, vandaar de naam van deze stelling.

De methode van Ford-Fulkerson is dus correct. Ze specificeert echter niet hoe een vergrotende weg gezocht moet worden. Er zijn dus meerdere implementaties mogelijk:

- *Performantie afhankelijk van de capaciteiten.* De performantie van een aantal implementaties wordt niet enkel bepaald door de complexiteit van de graaf (n en m), maar ook door de grootte van de capaciteiten. Daarom noemt men hun performantie *pseudopolynomiaal*.
  - Stel dat alle capaciteiten geheel zijn, en dat C de grootste capaciteit voorstelt. (Irrationale getallen kunnen niet voorgesteld worden in een computer, en rationale getallen kunnen door vermenigvuldiging met een gepaste factor geheel gemaakt worden.) De maximale netwerkstroom is dan $O(nC)$.

Bij elke iteratie van Ford-Fulkerson zal de stroomtoename langs een vergrotende weg ook geheel zijn (en dus minstens één), zodat het aantal iteraties $O(nC)$ is.

Een restnetwerk bepalen is $O(m)$, en daarin een vergrotende weg vinden met diepte-eerst of breedte-eerst zoeken is ook $O(m)$. De totale performantie wordt dan $O(nmC)$. Dat is enkel goed voor kleine waarden van C. Om de performantie voor grote C te verbeteren moet het aantal iteraties dalen, want sneller zoeken naar een vergrotende weg is onmogelijk.

- Als men steeds de vergrotende weg neemt die de *grootste stroomtoename* mogelijk maakt, dan kan men aantonen dat het aantal iteraties $O(m \lg C)$ wordt (Edmonds en Karp, 1972). Om die weg te vinden volstaat een kleine wijziging aan het algoritme van Dijkstra, zodat de iteratiestap $O(m \lg n)$ wordt. De totale performantie wordt dan $O(m^2 \lg n \lg C)$.
- Een eenvoudige en snellere variant van de vorige implementatie heet 'capacity scaling' (Ahuja en Orlin, 1991). Deze methode berust op de volgende observaties:
  - Een vergrotende weg vinden die een stroomtoename van minstens c eenheden toelaat, of besluiten dat er geen dergelijke weg bestaat, kan in $O(m)$.
  - Stel dat er geen enkele vergrotende weg meer gevonden wordt met een stroomtoename van tenminste c. Dan is de minimale snedecapaciteit van het restnetwerk lager dan mc. We halveren nu de ondergrens, en gaan op zoek naar vergrotende wegen met een stroomtoename van minstens $c/2$. Er zijn hoogstens $2m$ dergelijke wegen. Als we beginnen met ondergrens $c = 2^{\lfloor \lg C \rfloor}$, deze in elke fase halveren, en eindigen met één, dan wordt de maximale stroom inderdaad bereikt met $O(m \lg C)$ iteraties.

  De totale performantie wordt dus $O(m^2 \lg C)$.

- *Performantie onafhankelijk van de capaciteiten.* Als de vergrotende weg steeds het *minimum aantal verbindingen* heeft, dan kan men aantonen dat de lengte van de vergrotende wegen na hoogstens m iteraties stijgt, en aangezien de maximale lengte $n-1$ is, volstaan $O(nm)$ iteraties (Dinitz, 1970 en ook Edmonds en Karp, 1972). De iteratiestap moet nu breedte-eerst zoeken gebruiken en is dus $O(m)$, zodat de totale performantie $O(nm^2)$ wordt. Dit resultaat is onafhankelijk van de capaciteiten, zodat deze performantie echt polynomiaal is.

Alle algoritmen die een maximale stroom zoeken via vergrotende wegen hebben als nadeel dat die stroomtoename langs de *volledige* weg van p naar v moet gebeuren, wat in het slechtste geval $O(n)$ vereist. Stroomtoename langs een weg kan echter opgedeeld worden in stroomtoename langs zijn verbindingen. De recentere 'preflow-push'-methode vindt de maximale stroom door stroomtoename langs *individuele* verbindingen (een 'push'-operatie), in plaats van langs een vergrotende weg. Met als gevolg dat de stroom tijdens de uitvoering van het algoritme niet noodzakelijk conservatief is: er

kan meer stroom binnenkomen in een knoop dan er buiten gaat (de stroom is nu tijdelijk een 'preflow'). Knopen met een dergelijk positief overschot ('excess') heten 'actief', en zolang er actieve knopen zijn, voldoet de oplossing niet (want die moet conservatief zijn). De basisoperatie van deze methode is dan ook een actieve knoop selecteren, en trachten om zijn overschot weg te werken via zijn buren. Als er geen actieve knopen meer zijn, voldoet de stroom aan de vereisten, en blijkt bovendien maximaal. Meer details vallen buiten het opzet van deze inleidende cursus. We vermelden nog enkele performanties, ter vergelijking met die van Ford-Fulkerson.

Een eenvoudige implementatie van de preflow-push-methode haalt een performantie van $O(n^2m)$. Betere implementaties verschillen in de manier waarop ze actieve knopen selecteren:

- Het 'FIFO preflow-push'-algoritme (Goldberg, 1985) selecteert de actieve knopen met een wachtrij, en is $O(n^3)$.
- Het 'highest-label preflow-push'-algoritme (Goldberg en Tarjan, 1986) neemt de actieve knoop verst van v, en is $O(n^2\sqrt{m})$, wat in alle gevallen beter is dan $O(n^3)$.
- Het 'excess-scaling'-algoritme (Ahuja en Orlin, 1989) duwt stroom van een actieve knoop met voldoend groot overschot naar een knoop met een voldoend klein overschot, en is $O(nm + n^2 \lg C)$.

De beste implementaties zitten dus dicht bij de vermoedelijke ondergrens van $\Omega(nm)$.

8.2 VERWANTE PROBLEMEN

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

- Stel dat een stroomnetwerk *meerdere producenten en verbruikers* heeft, en dat men de gezamenlijke nettostroom van alle producenten wil maximaliseren. Dat is gemakkelijk op te lossen door er eerst een gewoon stroomnetwerk van te maken. Daartoe voert men een fictieve 'totaalproducent' en een 'totaalverbruiker' in. Vanuit de totaalproducent komen er verbindingen met *onbeperkte* capaciteit naar alle producenten, en vanuit elke verbruiker komt er een soortgelijke verbinding naar de totaalverbruiker. De totaalproducent levert dan zoveel materiaal als nodig voor alle producenten, en de totaalverbruiker verbruikt alles wat bij alle verbruikers samen aankomt. Het is intuïtief duidelijk dat elke stroomverdeling in het origineel netwerk overeenkomt met een stroomverdeling in het tweede en vice versa, en dat dit ook geldt voor een maximale stroomverdeling. In het tweede netwerk wordt die op de gewone manier gevonden.

- Ook aan de netwerk*knopen* kunnen er capaciteitsbeperkingen opgelegd worden. Om opnieuw een gewoon stroomnetwerk te bekomen ontdubbelt men elke knoop, en wordt een verbinding met de knoopcapaciteit voorzien tussen de originele knoop en zijn dubbelganger. De inkomende verbindingen van de originele knoop blijven bij die knoop, de uitgaande verbindingen komen terecht bij de dubbelganger. Het is duidelijk dat een maximale stroom in dit nieuwe netwerk ook een maximale stroom in het originele netwerk is, die rekening houdt met de knoopcapaciteiten.

- Soms vereist het probleemmodel een *ongericht* stroomnetwerk. De stroom in een verbinding kan daarbij in de ene of de andere richting lopen, maar mag haar capaciteit niet overschrijden. (Dit gelijkt dan best op vloeistoffen in pijpleidingen.) Om een gewoon stroomnetwerk te bekomen vervangt men elke verbinding door een *paar* gerichte verbindingen, één in elke richting, en beide verbindingen krijgen de originele capaciteit.

  Elke stroom in het ongericht netwerk komt overeen met een even grote stroom in het gericht netwerk (in de gepaste richting), en vice versa: de positieve nettostroom tussen elk paar knopen in het gericht netwerk wordt de stroom in hun ongerichte verbinding. En dus komt een maximale stroom in het gerichte netwerk overeen met een maximale stroom in het ongerichte.

- Soms wenst men naast bovengrenzen (capaciteiten) ook *ondergrenzen* op te leggen aan de stroom in de verbindingen. Dit probleem wordt in twee fasen opgelost: eerst onderzoekt men of een netwerkstroom wel mogelijk is (want dat is niet zeker), en indien ja, transformeert men die daarna in een maximale stroom. Beide fasen vereisen de oplossing van een gewoon stroomnetwerk, met enkel bovengrenzen voor de stromen.

- Een andere variant is een stroomnetwerk met *meerdere soorten* materiaal die door de verbindingen kunnen 'stromen' ('multicommodity network'). Voor elke soort is er één producent, en ook één verbruiker. In elke knoop (behalve producenten en verbruikers) is de stroom voor elke soort *apart* conservatief (een soort kan niet omgezet worden in een andere). Maar de gezamenlijke stroom van alle materialen door een verbinding mag haar capaciteit niet overschrijden.

- Tenslotte is het realistisch om naast een capaciteit ook een *kost per stroomeenheid* aan elke verbinding toe te kennen. Het *minimalekostprobleem* zoekt niet alleen de maximale stroom, maar bovendien die met minimale kost. (Een variant zoekt een stroomverdeling met minimale kost voor een gegeven hoeveelheid te transporteren stroom.) Eigenlijk is dit probleem fundamenteler dan het maximalestroomprobleem, dat er trouwens een speciaal geval van is, net als het probleem van de kortste afstanden in een graaf. Het heeft nog veel meer toepassingen, maar het is dan ook moeilijker.

8.3 MEERVOUDIGE SAMENHANG IN GRAFEN

De definities van enkelvoudige en dubbele samenhang en lijnsamenhang kunnen veralgemeend worden, zowel voor ongerichte als gerichte grafen:

- Wanneer er tussen elk paar knopen van een graaf k onafhankelijke wegen bestaan (in beide richtingen voor gerichte grafen), zonder gemeenschappelijke *knopen*, dan is deze graaf *k-voudig samenhangend* ('k-connected'). Bij een samenhangende (of sterk samenhangende) graaf is k dus gelijk aan één, en bij een dubbel samenhangende graaf gelijk aan twee.
- Wanneer er tussen elk paar knopen van een graaf k onafhankelijke wegen bestaan (in beide richtingen voor gerichte grafen), zonder gemeenschappelijke *verbindingen*, dan is deze graaf *k-voudig lijnsamenhangend* ('k-edge-connected').

Eenvoudige vormen van samenhang en lijnsamenhang (enkelvoudig, dubbel, zelfs drievoudig) kunnen efficiënt via diepte-eerst zoeken onderzocht worden. Voor meervoudige samenhang en lijnsamenhang doet men beroep op stroomnetwerken. Er is immers een nauw verband tussen maximale netwerkstromen en minimale sneden: als men een maximale netwerkstroom vindt, heeft men meteen een minimale snede gevonden (max-flow min-cut stelling). Als alle capaciteiten één zijn, dan is de capaciteit van een snede (P, V) gelijk aan het aantal verbindingen van knopen in P naar knopen in V. Elke vergrotende weg laat een stroomtoename van één toe, en satureert meteen een weg van p naar v in het stroomnetwerk, die één van de voorwaartse verbindingen over een minimale snede gebruikt. De maximale netwerkstroom loopt aldus over (gerichte) wegen zonder gemeenschappelijke verbindingen. Het aantal onafhankelijke wegen is gelijk aan de minimale snedecapaciteit, en dus gelijk aan de maximale netwerkstroom f. In netwerken met eenheidscapaciteit ($C = 1$) kan die maximale netwerkstroom, met bijbehorende onafhankelijke wegen, gevonden worden in $O(nm)$.

De fundamentele eigenschap van samenhang in een graaf wordt gegeven door de stelling van Menger (1927). Daarvan bestaan vier versies: zowel voor gerichte als ongerichte grafen, en zowel voor meervoudige samenhang ('k-connectivity') als meervoudige lijnsamenhang ('k-edge connectivity'). Dit is bijvoorbeeld de versie voor een meervoudig lijnsamenhangende gerichte graaf:

> Het minimum aantal verbindingen dat moet verwijderd worden om een knoop v van een gerichte graaf onbereikbaar te maken vanuit een andere knoop p is gelijk aan het maximaal aantal lijnonafhankelijke wegen (zonder gemeenschappelijke *verbindingen*) van p naar v.

Aan beide versies voor meervoudige samenhang voegt men de beperking toe dat v geen buur mag zijn van p. (Waarom?)

Deze versie van de stelling volgt uit de eigenschappen van een stroomnetwerk met eenheidscapaciteiten, zoals hierboven. Beide aantallen zijn immers gelijk aan de capaciteit van een minimale snede tussen p en v:

- De maximale stroom loopt over een maximaal aantal lijnonafhankelijke wegen van p naar v. Dat aantal is gelijk aan de maximale stroom, en dus ook gelijk aan de capaciteit van een minimale snede.
- Een minimale verzameling verbindingen M die v onbereikbaar maakt vanuit p vormt een snede (P, V). Want definieer P als de verzameling van alle knopen bereikbaar vanuit p via wegen die geen verbinding uit M bevatten (dus ook p), en V als de verzameling van alle andere knopen (dus ook v). Alle verbindingen van deze snede behoren tot M. (Waarom?) Bovendien kan M geen andere verbindingen bevatten, want M is minimaal. Dus is M een snede, en wel een minimale. En de capaciteit van die snede is gelijk aan het aantal verbindingen in M.

Om een verband te leggen tussen onafhankelijke wegen zonder gemeenschappelijke *knopen* en vergrotende wegen ontdubbelt men alle knopen, en er komt een nieuwe verbinding van capaciteit één tussen elke knoop en zijn dubbelganger. Inkomende verbindingen blijven bij de originele knoop, uitgaande verbindingen komen bij de dubbelganger. Nu kan er slechts één vergrotende weg gebruik maken van elke knoop. Om de knopen te vinden die v onbereikbaar maken vanuit p wanneer men ze wegneemt, kan men enkel eenheidscapaciteiten toewijzen aan de verbindingen tussen knopen en dubbelgangers, en de capaciteit van alle andere verbindingen onbeperkt groot maken.

Hoe men van ongerichte grafen een stroomnetwerk maakt kwam reeds vroeger aan bod: elke ongerichte verbinding wordt vervangen door twee tegengesteld gerichte verbindingen, beide met eenheidscapaciteit. Elke collectie onafhankelijke wegen in de ongerichte graaf komt dan zeker overeen met een collectie onafhankelijke gerichte wegen in de gerichte graaf, en ook omgekeerd. (Dat is niet waar als de onafhankelijke gerichte wegen antiparallelle verbindingen zouden bevatten. Maar die kunnen vrij eenvoudig geëlimineerd worden.)

Als de originele graaf n knopen en m verbindingen had, dan blijft het aantal knopen van de getransformeerde graaf $O(n)$, en zijn aantal verbindingen $O(m)$.

De stelling van Menger spreekt slechts over één paar knopen. De stelling van Whitney (1932) past die eigenschap toe op *elk* paar knopen van een graaf om voorwaarden op te stellen voor zijn samenhang en lijnsamenhang (en heeft dus ook vier versies):

> Een graaf (al dan niet gericht) is k-voudig samenhangend (k-voudig lijnsamenhangend) als en enkel als er tenminste k knopen (verbindingen) moeten verwijderd worden om hem te doen uiteenvallen.

Om de graad k van samenhang of lijnsamenhang van een graaf te vinden, kan men dus een maximalestroomprobleem oplossen voor elk knopenpaar (van een eventueel

getransformeerde graaf), en het minimum van al die oplossingen nemen. Aangezien er $\Theta(n^2)$ knopenparen zijn is de performantie $O(n^3m)$.

Gelukkig blijken $\Theta(n)$ stroomproblemen te volstaan. Want stel dat r en s producent en verbruiker zijn van het stroomnetwerk met de kleinste minimale snede (R, S) onder alle $\Theta(n^2)$ knopenparen. Dan blijkt dat we hetzelfde resultaat zouden gevonden hebben voor een stroomnetwerk met een *willekeurige* knoop x uit R als producent, en een willekeurige knoop y uit S als verbruiker. Immers, (R, S) is ook een snede voor dat stroomnetwerk. Want als men de verbindingen uit (R, S) wegneemt wordt y zeker onbereikbaar vanuit x, anders zou s ook bereikbaar blijven vanuit r. En dat stroomnetwerk kan geen kleinere snede hebben, omdat (R, S) bij onderstelling de kleinste minimale snede was. Het probleem is natuurlijk dat we de snede (R, S) niet kennen, en dus ook x en y niet kunnen kiezen! Maar als we alle knopen in een willekeurige volgorde $x_1, x_2, \ldots, x_n$ zetten (cyclisch, zodat x_1 volgt op x_n), dan zijn er altijd twee *opeenvolgende* knopen x_i en x_{i+1} zodat x_i in R ligt, en x_{i+1} in S. We moeten dus slechts n stroomnetwerken oplossen, voor alle paren opeenvolgende knopen, en het minimum van al die oplossingen bijhouden.

HOOFDSTUK 9
KOPPELEN

Een *koppeling* ('matching') in een ongerichte graaf is een deelverzameling van de verbindingen waarin elke knoop hoogstens eenmaal voorkomt. De eindknopen van deze verbindingen heten 'gekoppeld'. En een maximale koppeling is natuurlijk een koppeling met het grootste aantal verbindingen (en dus gekoppelde knopen).

Soms hebben die verbindingen ook nog een gewicht, dat een voorkeur voor bepaalde koppelingen kan uitdrukken. Het *gewogen koppelingsprobleem* ('weighted matching problem') zoekt dan de koppeling met het grootste totale gewicht. Voor een overzicht verwijzen we naar het artikel van Galil[16].

9.1 KOPPELEN IN TWEELEDIGE GRAFEN

Een *tweeledige graaf* ('bipartite graph') is een ongerichte graaf waarbij de knopen in twee deelverzamelingen L en R kunnen verdeeld worden, zodat alle verbindingen steeds een knoop uit L met een knoop uit R verbinden. Een dergelijke graaf kan bijvoorbeeld gebruikt worden om uit te voeren taken toe te wijzen aan uitvoerders (personen of machines). De verbindingen duiden aan welke taken (knopen uit R) een uitvoerder (een knoop uit L) aankan.

9.1.1 Ongewogen koppeling

Als elke uitvoerder slechts één taak tegelijk kan verrichten, dan zorgt een maximale ongewogen koppeling ervoor dat er zoveel mogelijk taken tegelijk uitgevoerd worden.

Er blijkt een nauw verband te bestaan tussen deze koppelingen en de maximale stroom in netwerken, zodat ze met dezelfde efficiënte methode kunnen opgelost worden.

De tweeledige graaf moet echter eerst getransformeerd worden tot een stroomnetwerk. Daartoe voert men een producent in, die met alle knopen van L verbonden wordt (in die richting). Analoog verbindt men alle knopen van R met een nieuwe verbruiker (ook in die richting). De oorspronkelijke verbindingen van de graaf krijgen eveneens een richting, van L naar R. Tenslotte geeft men alle verbindingen (ook de nieuwe) een capaciteit gelijk aan één.

Het is eenvoudig in te zien dat er voor elke koppeling met k verbindingen een overeenkomstige *gehele stroomverdeling* in dat nieuwe stroomnetwerk bestaat, met als netwerkstroom k. (Een gehele stroomverdeling in een stroomnetwerk bestaat enkel uit gehele nettostromen, zodat ook de netwerkstroom geheel is.) Daarbij stuurt de producent één eenheid stroom naar elke knoop van L die tot de koppeling behoort. Die eenheid loopt natuurlijk via de koppeling naar de overeenkomstige knoop van R, en vandaar naar de verbruiker.

Het omgekeerde geldt ook: voor elke gehele stroomverdeling in het stroomnetwerk, met (uiteraard gehele) netwerkstroom k, is er een koppeling met k verbindingen. Aangezien de stroomverdeling geheel is, en de capaciteit van de verbindingen gelijk aan één, is de stroom in elke verbinding één of nul. Als er dus een eenheid stroom van de producent in een knoop van L aankomt, moet die over precies één verbinding naar de overkant. Ook kan elke knoop van R slechts stroom ontvangen van één enkele knoop uit L, en deze verbindingen bepalen de overeenkomstige koppeling. Vermits de netwerkstroom gelijk is aan k, zijn er evenveel verbindingen in deze koppeling.

Een maximale koppeling komt dan ook overeen met een maximale gehele stroomverdeling, en het ligt dus voor de hand om het probleem op te lossen met de methode van Ford-Fulkerson. Deze methode levert weliswaar een stroomverdeling op die de capaciteiten respecteert, maar is die ook geheel? Ja, want de capaciteiten zijn geheel, dus is de stroom in elke vergrotende weg geheel, en de stromen bestaan uit sommaties van dergelijke (deel)stromen.

Het aantal verbindingen k van deze maximale koppeling is niet groter dan het aantal knopen in de kleinste van de twee verzamelingen L en R, en dus zeker $O(n)$. Met breedte-eerst zoeken voor de vergrotende wegen is het maximaal koppelingsprobleem dan $O(km)$, en dus $O(nm)$.

9.1.2 Gewogen koppeling

9.2 STABIELE KOPPELING

Gegeven één of twee verzamelingen van elementen. Elk element heeft een gerangschikte voorkeurslijst van andere elementen. Deze elementen moeten gekoppeld worden, rekening houdend met hun voorkeuren, en zodanig dat de koppeling stabiel is. Een koppeling is onstabiel wanneer ze twee niet met elkaar gekoppelde elementen bevat, die liever met elkaar zouden gekoppeld zijn dan in hun huidige toestand te blijven.

Dit algemeen scenario vormt de basis voor drie verwante problemen:

- *Stable marriage.* Hier zijn er twee verzamelingen, met dezelfde grootte. Hun elementen worden traditioneel mannen en vrouwen genoemd. Elke man (vrouw) heeft een voorkeurslijst die alle vrouwen (mannen) bevat. Elke man moet gekoppeld worden aan een vrouw, zodat de koppeling stabiel is.

Dit mag een kunstmatig probleem lijken, maar het heeft praktische toepassingen. (Onder meer in netwerkrouters, zie bijvoorbeeld [9] en [24].)

- *Hospitals/residents.* Ook hier zijn er twee verzamelingen. Hun elementen worden traditioneel hospitalen en stagiairs genoemd ('hospitals/residents' of ook wel 'colleges/students'). Elk hospitaal biedt een aantal stageplaatsen aan. Het totaal aantal stageplaatsen moet niet noodzakelijk gelijk zijn aan het aantal stagiairs. Elk hospitaal en elke stagiair heeft opnieuw een voorkeurslijst van alle elementen uit de andere verzameling. Elk hospitaal moet gekoppeld worden aan een aantal stagiairs (niet meer dan er plaatsen zijn), zodat de koppeling stabiel is.

  Een oplossing voor dit probleem wordt sinds 1952 in de VS gebruikt om studenten geneeskunde toe te wijzen aan hospitalen, via het National Intern (later Resident) Matching Program (NRMP). Hoewel de aanvaarding van toewijzingen op vrijwillige basis gebeurt, wordt het door de meerderheid van de betrokkenen gebruikt.

- *Stable roommates.* Dit probleem zoekt een stabiele koppeling tussen de elementen van dezelfde verzameling (met even cardinaliteit). Elk element heeft een voorkeurslijst die alle andere elementen bevat.

Al deze problemen kan men veralgemenen door onvolledige voorkeurslijsten toe te laten.

9.2.1 Stable marriage

9.2.1.1 Het Gale-Shapley-algoritme

Elk stable marriage probleem heeft tenminste één stabiele koppeling. Het Gale-shapley-algoritme[1] (Gale en Shapley, 1962) vindt gegarandeerd zo'n oplossing, met de merkwaardige eigenschap dat alle mannen de beste partner krijgen die ze kunnen hebben in een stabiele koppeling (niet noodzakelijk gevonden met dit algoritme). (Er is een symmetrische versie van het algoritme, waarbij deze eigenschap geldt voor de vrouwen.)

Een voorgestelde oplossing controleren is eenvoudig: alle paren niet gekoppelde elementen moeten hun partners verkiezen boven elkaar. Het volstaat om elk element uit een van de verzamelingen te testen met de elementen uit de andere verzameling die hoger op zijn voorkeurslijst staan dan zijn partner. Met geschikte gegevensstructuren heeft dit een performantie van $\Theta(n^2)$.

Het Gale-Shapley-algoritme bestaat uit een reeks 'aanzoeken' van de mannen aan de vrouwen. (Dit is de man-georiënteerde versie. Als men de rollen omkeert, bekomt men de vrouw-georiënteerde versie.) Iedereen is op elk ogenblik ofwel 'verloofd' ofwel vrij. Een man kan afwisselend verloofd of vrij zijn, maar een vrouw, eens verloofd, blijft verloofd, ook al kan ze van partner veranderen. Een aanzoek gebeurt enkel door een

[1] Het algoritme is de basis voor de Nobelprijs die Shapley kreeg in 2012.

vrije man. Wanneer een man een aanzoek doet aan een vrije vrouw, zal ze zich steeds met hem verloven. Als een man een aanzoek doet aan een verloofde vrouw, vergelijkt ze hem met haar partner, en verwerpt de laagst geklasseerde. Als ze haar verloving verbreekt ten gunste van de aanzoeker, wordt haar vorige partner opnieuw vrij. Elke man doet aanzoeken aan de vrouwen, in de volgorde van zijn voorkeurslijst, tot hij zich kan verloven. Wanneer die verloving ooit verbroken wordt (door de vrouw), zet hij zijn lijst verder en doet een aanzoek aan de volgende vrouw. Het algoritme stopt wanneer iedereen verloofd is, en dat gebeurt zeker vóór een man het einde van zijn lijst bereikt.

Bemerk dat de volgorde waarin vrije mannen aanzoeken niet gespecificeerd wordt. Toch zal blijken dat de oplossing steeds dezelfde is.

9.2.1.2 Eigenschappen van de oplossing

- Het algoritme stopt altijd, en de oplossing is steeds stabiel.
  - Geen enkele man wordt afgewezen door alle vrouwen. Immers, enkel een verloofde vrouw kan een man afwijzen, en eens verloofd, blijft ze dat (eventueel met opeenvolgende mannen). Een man kan niet afgewezen worden door de laatste vrouw op zijn lijst, want dan zouden immers alle vrouwen verloofd zijn, en polygamie is hier niet toegestaan. Er kan dus geen (vrije) man overblijven.
  - In elke iteratie gebeurt er een aanzoek, en geen enkele man doet dat tweemaal aan dezelfde vrouw. Er zijn dus maximaal n^2 aanzoeken, zodat het algoritme zeker stopt.
  - De oplossing is stabiel. Want als er een man m bestaat die vrouw v verkiest boven zijn partner, dan moet v hem ooit afgewezen hebben. Dat gebeurde omdat ze verloofd was, of zich verloofde, met een man die ze verkoos boven m. En als ze later een andere partner aanvaardde, stond die nog hoger op haar lijst. Kortom, m verkiest een andere vrouw v, maar niet omgekeerd. Er bestaat dus geen ongekoppeld paar dat de stabiliteit van de koppeling in gevaar kan brengen.
- Elke mogelijke aanzoekvolgorde van de mannen geeft dezelfde oplossing. Daarbij krijgt elke man de best mogelijke partner die hij kan hebben in een stabiele koppeling (niet noodzakelijk gevonden met Gale-Shapley).

  Onderstel dat man m niet zijn best mogelijke stabiele partner v krijgt, maar een vrouw lager op zijn lijst. Dan werd m ooit door v afgewezen. Onderstel verder dat m de eerste man is die door een stabiele partner wordt afgewezen tijdens de uitvoering van het algoritme. Dat impliceert dat v de *best mogelijke* stabiele partner van m is. Wanneer m door v wordt afgewezen, gebeurt dat omdat ze een andere man m' verkiest. Die man was reeds haar partner, of wordt haar nieuwe partner, maar wanneer m' de partner wordt van v werd hij nog niet door een stabiele partner afgewezen, want we onderstelden dat dit de eerste keer gebeurt bij m. Dat betekent dat alle mogelijke andere stabiele partners van m' lager op

zijn lijst staan dan v. Omdat v een stabiele partner van m is, bestaat er een andere stabiele koppeling K waarin m en v partners zijn, en waarin de partner van m' bijvoorbeeld v' is. Dus staat ook deze v' lager dan v op de lijst van m'. Dat betekent dat de koppels (m, v) en (m', v') uit K niet stabiel zijn, want zowel v als m' verkiezen elkaar boven hun partners. De koppeling K blijkt dus toch niet stabiel, zodat onze veronderstelling niet klopt.

Het is helemaal niet vanzelfsprekend dat de mannen, die toch elkaars concurrenten zijn, een stabiele koppeling bekomen die gelijktijdig optimaal is voor hen allemaal. Deze man-georiënteerde versie heet dan ook man-optimaal. Voor een vrouw-georiënteerde versie moet men de rol van mannen en vrouwen verwisselen, en die is dan vrouw-optimaal. Het is mogelijk dat beide optimale oplossingen identiek zijn, maar meestal is dat niet zo.

- In de man-georiënteerde versie krijgt elke vrouw de slechtst mogelijke partner die ze kan hebben in een stabiele koppeling.

  Onderstel dat vrouw v niet haar slechtst mogelijke stabiele partner m krijgt, maar een man m' hoger op haar lijst. Omdat m een stabiele partner van v is, bestaat er een andere stabiele koppeling K waarin m en v partners zijn, en waarin de partner van m' bijvoorbeeld v' is. Zowel v als v' zijn dus stabiele partners van m'. En omdat Gale-Shapley aan elke man de best mogelijke stabiele partner toewijst (zie hierboven), staat v hoger dan v' op de lijst van m'. Maar ook m' staat hoger dan m op de lijst van v. Dat betekent dat de koppels (m, v) en (m', v') uit K niet stabiel zijn, want zowel v als m' verkiezen elkaar boven hun partners. De koppeling K blijkt dus toch niet stabiel, zodat onze veronderstelling niet klopt.

9.2.1.3 Implementatie

Het algoritme vertrekt van de gegeven voorkeurslijsten van alle deelnemers. Die volstaan echter niet om snel te weten te komen aan welke van twee mannen een vrouw de voorkeur geeft. Daartoe moeten *ranglijsten* voor elke vrouw opgesteld worden, die de volgorde van elke man aangeven in haar voorkeurslijst. Die lijsten opstellen is $\Theta(n^2)$.

Ook het vinden van een vrije man om het volgende aanzoek te doen moet efficiënt kunnen gebeuren. Daartoe volstaat een lijst van vrije mannen. Na initialisatie worden er enkel mannen van die lijst verwijderd, omdat een afgewezen man meteen een nieuw aanzoek kan doen.

Het algoritme stopt wanneer de laatst overgebleven vrouw haar eerste aanzoek krijgt. Het totaal aantal aanzoeken is $O(n^2)$, want $n - 1$ vrouwen kunnen elk maximaal n aanzoeken krijgen, plus één aanzoek aan de laatste vrouw. In het slechtste geval wordt dat aantal ook gehaald. Het Gale-Shapley-algoritme is dus $\Theta(n^2)$.

9.2.1.4 Uitbreidingen

Er zijn enkele eenvoudige uitbreidingen mogelijk van het stable marriage probleem. We behandelen ze apart, maar ze kunnen ook gecombineerd worden.

- *Verzamelingen van ongelijke grootte.* Het aantal mannen n_m is dus verschillend van het aantal vrouwen n_v. De meeste resultaten voor gelijke aantallen blijven echter behouden, in licht gewijzigde vorm. Een koppeling M wordt nu als onstabiel beschouwd, wanneer er een man m en vrouw v bestaan zodat:

  (1) m en v geen partners zijn.
  (2) m ofwel ongekoppeld blijft, ofwel v verkiest boven zijn partner.
  (3) v ofwel ongekoppeld blijft, ofwel m verkiest boven haar partner.

  Hierbij onderstelt men dat iemand liever gekoppeld wordt dan alleen te moeten blijven. Men kan aantonen dat elke stabiele koppeling nu uit $\min(n_m, n_v)$ koppels bestaat, terwijl de rest van de grootste groep ongekoppeld blijft. Toegepast op de kleinste groep vindt het Gale-Shapley-algoritme een stabiele koppeling, die weer optimaal is voor iedereen uit die groep. Voor iedereen uit de grootste groep die een partner krijgt is de koppeling weer de slechtst mogelijke.

- *Onaanvaardbare partners.* De voorkeurslijsten moeten niet langer de volledige andere groep bevatten: onaanvaardbare partners mogen weggelaten worden. De stabiele koppelingen kunnen nu gedeeltelijk zijn, zodat niet noodzakelijk iedereen een partner krijgt. Een koppeling M wordt nu als onstabiel beschouwd, wanneer er een man m en vrouw v bestaan zodat:

  (1) m en v geen partners zijn, maar wel aanvaardbaar zijn voor elkaar.
  (2) m ofwel ongekoppeld blijft, ofwel v verkiest boven zijn partner.
  (3) v ofwel ongekoppeld blijft, ofwel m verkiest boven haar partner.

  Men kan aantonen dat mannen en vrouwen nu in twee groepen onderverdeeld worden: zij die partners krijgen in alle stabiele koppelingen, en zij die nooit partners krijgen. Het (aangepaste) Gale-Shapley-algoritme vindt beide groepen.

- *Gelijke voorkeuren.* De voorkeurslijsten mogen nu ex aequo's bevatten. Er zijn dan drie mogelijkheden om stabiliteit te definiëren:

  (1) *Superstabiliteit.* Een koppeling is onstabiel als er een man en een vrouw bestaan die geen partners zijn, en die elkaar *minstens* evenzeer verkiezen als hun partners. Er zijn nu gevallen waarbij geen (super)stabiele koppeling mogelijk is. (Met als triviaal voorbeeld totale onverschilligheid: alle personen op elke lijst krijgen dezelfde voorkeur.)
  (2) *Sterke stabiliteit.* Een koppeling is onstabiel als er een man en een vrouw bestaan die geen partners zijn, waarvan de ene de andere strikt verkiest boven de partner, en de andere de eerste minstens even graag heeft als de partner. Ook hier bestaat er niet altijd een (sterk) stabiele koppeling.

(3) *Zwakke stabiliteit.* Een koppeling is onstabiel als er een man en een vrouw bestaan die geen partners zijn, en die elkaar *strikt* verkiezen boven hun partners. Dit geval kan getransformeerd worden tot het standaardprobleem door gelijke voorkeuren arbitrair te ordenen. Het is gemakkelijk te zien dat een stabiele oplossing hiervoor ook (zwak) stabiel is voor het origineel probleem. Het Gale-Shapley-algoritme kan dus opnieuw gebruikt worden, en zal steeds een oplossing vinden. Die is niet noodzakelijk uniek, omdat ze afhangt van de arbitraire ordening.

9.2.2 Hospitals/Residents

Dit is een asymmetrische veralgemening van het stable marriage probleem: enerzijds zijn er stagiairs, en anderzijds hospitalen die elk een of meer stageplaatsen aanbieden. Elk hospitaal moet stabiel gekoppeld worden aan een aantal stagiairs (niet méér dan het aantal beschikbare plaatsen). Twee uitbreidingen worden hier meteen voorzien: het totaal aantal plaatsen moet niet noodzakelijk gelijk zijn aan het aantal stagiairs, en de voorkeurslijsten van zowel hospitalen als stagiairs mogen onvolledig zijn. Gelijke voorkeuren worden echter niet toegelaten.

Een dergelijke koppeling is onstabiel wanneer er een hospitaal h en een stagiair s zijn, zodat:

(1) h en s geen partners zijn, maar wel aanvaardbaar zijn voor elkaar.
(2) s ofwel ongekoppeld blijft, ofwel h verkiest boven zijn toegewezen hospitaal.
(3) h ofwel onbezette stageplaatsen overhoudt, ofwel s verkiest boven tenminste een van zijn toegewezen stagiairs.

Deze koppeling van meerdere stagiairs aan één hospitaal kan getransformeerd worden in een gewone een-eenduidige koppeling. Daartoe wordt elk hospitaal h met p plaatsen vervangen door p identieke hospitalen $h_1, h_2, \ldots, h_p$ met elk één stageplaats, en allemaal met dezelfde voorkeurslijst als h. Wanneer h voorkomt in de lijst van een stagiair wordt die vervangen door de reeks $h_1, h_2, \ldots, h_p$. Men kan aantonen dat een stabiele koppeling voor dit probleem overeenkomt met een stabiele koppeling voor het origineel probleem. Het algoritme van Gale-Shapley, toegepast op het getransformeerd probleem, geeft dus stabiele koppelingen voor het origineel probleem.

De symmetrie tussen mannen en vrouwen van het stable marriage probleem is hier echter niet aanwezig, zodat er twee versies zijn: een hospitaal- en een stagiairversie. De oplossing van de hospitaalversie (die gebruikt wordt door de NRMP) is opnieuw gelijktijdig optimaal voor alle hospitalen. En weer heeft niet-determinisme geen invloed: de oplossing is steeds dezelfde.

Omdat stagiairs die (levens)partners zijn meestal dezelfde hospitalen verkiezen, bestaat er een versie voor dergelijke paren. Elk stagiairpaar (s_1, s_2) mag dan een gezamenlijke voorkeurslijst van hospitalen indienen. Die bestaat uit geordende paren hospitalen (h_1, h_2), waarbij h_1 de voorkeur is van partner s_1, en h_2 die van s_2 (h_1 en h_2 mogen

dus gelijk zijn). Het is nu niet langer zeker dat er stabiele oplossingen bestaan, en zelfs als dat zo is, zijn ze soms moeilijk te vinden.

9.2.3 Stable roommates

Dit is een veralgemening van het stable marriage probleem: er is slechts één groep met n personen (n even), die elk een voorkeurslijst opmaken van al de anderen als potentiële kamergenoten. Een koppeling verdeelt de groep in paren, en is onstabiel wanneer twee personen elkaar verkiezen boven hun toegewezen kamergenoten.

Er zijn gevallen waarvoor geen stabiele koppeling mogelijk is. Testen of er een bestaat, en zo ja, er een vinden, is $O(n^2)$.

DEEL 3

STRINGS

INLEIDING

Een ‘string’ is een sequentie van elementen die behoren tot een eindige verzameling (het ‘alfabet’). Het meest voor de hand liggende voorbeeld is natuurlijk een tekststring bestaande uit karakters (letters, cijfers, leestekens, speciale tekens), maar een string kan ook een reeks bits zijn, of een DNA-sequentie (opgebouwd uit nucleotiden voorgesteld door de letters A,C,G,T).

In dit deel komen drie onderwerpen aan bod:

- *Gegevensstructuren voor strings.* Voor gegevens met sleutels waarvan de individuele elementen (zoals bits, bytes, of karakters) vlot toegankelijk zijn, bestaan er efficiënte gegevensstructuren, die daarvan gebruik maken. Deze sleutels kunnen dan beschouwd worden als strings van sleutelelementen. Uiteraard zijn deze gegevensstructuren zeer geschikt om ‘echte’ strings op te slaan.

  We bespreken digitale zoekbomen,tries, en ternaire zoekbomen.

- *Zoeken naar deelstrings.* Een van de belangrijkste operaties op strings is het zoeken naar een deelstring in een gegeven string. We bespreken enkele efficiënte algoritmen hiervoor, die elk het probleem op een wezenlijk verschillende manier aanpakken. Via zoeken naar deelstrings gespecificeerd door een regexp introduceren we (abstracte) automaten, een model dat ook daarbuiten vaak gebruikt wordt. Daarna komen suffixbomen en suffixtabellen aan bod, waarmee niet alleen zeer efficiënt naar deelstrings kan gezocht worden, maar die nog veel andere toepassingen hebben. Tenslotte raken we even het onderwerp van tekstzoekmachines aan.

- *Vergelijkbare strings.* Gelijkenissen opsporen tussen strings (en dus ook teksten) gebeurt onder meer bij DNA-onderzoek, of bij het opsporen van plagiaat. We beperken ons hier tot het bepalen van de langste gemeenschappelijke deelsequentie van twee strings, en tot het zoeken naar korte deelstrings met fouten.

HOOFDSTUK 10
GEGEVENSSTRUCTUREN VOOR STRINGS

10.1 INLEIDING

Net zoals men sleutels efficiënt kan rangschikken door gebruik te maken van de afzonderlijke sleutelelementen (bits, of cijfers, of karakters), bestaan er efficiënte gegevensstructuren die een zoeksleutel lokaliseren door zijn elementen één voor één te testen. Deze tegenhanger van radix sort heet dan ook 'radix search' (of 'digital search'). De sleutels bij radix search kunnen beschouwd worden als *strings* van sleutelelementen, al dan niet van dezelfde lengte.

De toegangstijd tot de gegevens bij radix search is competitief met die van hashing en binaire zoekbomen. In het slechtste geval is die meestal goed, zonder de complexiteit van evenwichtige zoekbomen te vereisen. Het voornaamste nadeel van radix search is dat de methode soms veel geheugen vereist. En natuurlijk moeten de individuele sleutelelementen efficiënt toegankelijk zijn.

Er bestaan verschillende boomstructuren die radix search toepassen. Zo zijn er digitale zoekbomen, die echter het nadeel hebben dat ze de volgorde van de opgeslagen sleutels niet ondersteunen. Tries doen dit wel, net als hun voornaamste variant, patricia(tries). Een ternaire zoekboom is een alternatieve voorstelling van een meerwegstrie, die minder geheugen gebruikt en toch efficiënt blijft.

We zullen steeds onderstellen dat geen enkele sleutel een prefix is van een andere. (Wat impliceert dat de sleutels verschillend zijn.)

10.2 DIGITALE ZOEKBOMEN

Net zoals bij gewone zoekbomen worden de sleutels opgeslagen in de knopen. Ook zoeken en toevoegen verlopen analoog: beginnend bij de wortel, vergelijken we de zoeksleutel met de sleutel in de huidige knoop, en als ze verschillen zoeken we verder in een van de deelbomen van deze knoop. (Als de deelboom ledig blijkt voegen we eventueel toe.) Er is echter één belangrijk verschil: de juiste deelboom wordt niet bepaald door de zoeksleutel te vergelijken met de sleutel in de knoop, maar enkel door het volgende element van de zoeksleutel (van links naar rechts). Bij de wortel gebruiken

we het eerste sleutelelement, een niveau dieper het tweede sleutelelement, enz.

Hoewel dit principe toepasbaar is op willekeurige sleutelelementen, zullen we ons voor de eenvoud beperken tot bits, en dus tot *binaire* digitale zoekbomen (Coffman en Eve, 1970). In dat geval wordt in een knoop op diepte i bit $(i + 1)$ van de zoeksleutel gebruikt om te beslissen of we afdalen naar het linker- dan wel naar het rechterkind.

Bemerk dat overlopen in inorder van een (binaire) digitale zoekboom de opgeslagen sleutels niet noodzakelijk in volgorde oplevert. Weliswaar zijn alle sleutels in de linkerdeelboom van een knoop op diepte i kleiner dan deze in zijn rechterdeelboom, want de i beginbits van al deze sleutels zijn gelijk, en het onderscheid werd gemaakt op grond van bit $(i + 1)$. Ook de sleutel in de knoop zelf heeft dezelfde i beginbits, maar toch kan hij om het even waar vallen tussen de sleutels van beide deelbomen.

Elke sleutel in een digitale zoekboom bevindt zich ergens op de weg vanuit de wortel bepaald door de opeenvolgende bits van deze sleutel. Met als direct gevolg dat de langste weg in de boom, en dus zijn hoogte, beperkt wordt door het aantal bits van de langste opgeslagen sleutel. Voor een groot aantal sleutels met relatief kleine bitlengte is de performantie in het slechtste geval veel beter dan die van een gewone binaire zoekboom, en vergelijkbaar met die van een rood-zwarte boom. (Voor een miljoen gehele getallen van 32 bits is de maximale hoogte van een rood-zwarte boom ongeveer 40, en die van een binaire digitale zoekboom 32.)

Voor gelijkmatig verdeelde sleutels is de kans op een volgend nul- of éénbit steeds gelijk, zodat er gemiddeld evenveel sleutels aan beide zijden van elke knoop zullen terechtkomen. De gemiddelde weglengte tot elke knoop in een digitale zoekboom met n knopen is dan ook $O(\lg n)$, zodat zoeken of toevoegen gemiddeld $O(\lg n)$ (sleutel)vergelijkingen vereist. Het aantal vergelijkingen is trouwens nooit meer dan het aantal bits van de zoeksleutel.

De performantie van (binaire) digitale zoekbomen is dus vergelijkbaar met die van rood-zwarte bomen, terwijl hun implementatie nagenoeg even eenvoudig uitvalt als die van gewone zoekbomen. De (dikwijls beperkende) voorwaarde is een efficiënte toegang tot de bits van de sleutels. Bovendien zijn enkel woordenboekoperaties mogelijk.

10.3 TRIES

10.3.1 Tries

Een trie[1] (de la Briandais, 1959) is een digitale zoekstructuur die wél de volgorde van de opgeslagen sleutels behoudt.

10.3.1.1 Binaire tries

De zoekweg in een binaire trie wordt net als bij digitale zoekbomen volledig bepaald door de opeenvolgende bits van de zoeksleutel. Het belangrijkste verschil met digitale zoekbomen is dat tries de sleutels niet opslaan in inwendige knopen, maar enkel in de *bladeren*. Met als gevolg dat overlopen van de boom in inorder de opgeslagen sleutels nu wel gerangschikt oplevert: het waren immers de sleutels in de inwendige knopen van een digitale zoekboom die deze volgorde konden verstoren. Nog een ander belangrijk gevolg is dat de zoeksleutel niet meer moet vergeleken worden met een sleutel in elke knoop op de zoekweg.

Zoeken en toevoegen gebruiken dus enkel de opeenvolgende bits van de zoeksleutel om de juiste weg te volgen:

- Komen we uiteindelijk bij een ledige deelboom terecht (nullwijzer), dan bevat de boom de zoeksleutel niet, en kunnen we die eventueel op deze plaats toevoegen (in een nieuw blad).
- Komen we in een blad terecht, dan bevat dit blad de enige sleutel in de boom die gelijk *kan* zijn aan de zoeksleutel, aangezien ze dezelfde beginbits hebben. (Bepaald door de zoekweg, en dus uniek in de boom.) Een vergelijking van beide sleutels geeft uitsluitsel.

  Stel dat de zoeksleutel niet aanwezig blijkt, en dat we hem willen toevoegen. De sleutel in het blad en de zoeksleutel hebben minstens dezelfde beginbits, bepaald door de weg van de wortel tot bij dat blad. Als hun volgende bit verschilt, dan wordt dat blad vervangen door een knoop met twee kinderen (bladeren), een voor elke sleutel. Als echter nog meer bits van beide sleutels gelijk zijn, dan wordt het blad vervangen door een reeks opeenvolgende (inwendige) knopen, zoveel als er nog gemeenschappelijke bits zijn. En de weg tussen die knopen komt overeen met deze gemeenschappelijke bits. Bij de eerste verschillende bit krijgt de laatste knoop van die reeks twee kinderen (bladeren), een voor elke sleutel.

Uit dat laatste geval blijkt meteen een vervelende eigenschap van tries: wanneer opgeslagen sleutels veel gelijke beginbits hebben, zijn er veel knopen met slechts één kind. Het aantal knopen is dan ook hoger dan het aantal sleutels. Een trie met n gelijkmatig verdeelde sleutels heeft gemiddeld $n/\ln 2 \approx 1.44n$ inwendige knopen[2].

[1] Trie wordt uitgesproken als het Engelse ‘try’, hoewel men eigenlijk dezelfde uitspraak als ‘tree’ voor ogen had, zoals in ‘re*trie*val’ (Fredkin, 1960).

[2] De originele trie van de la Briandais voorzag een knoop voor *elk* sleutelelement, tot bij het einde van elke sleutel, zodat het niet meer nodig was om de sleutels zelf op te slaan. Het aantal knopen was dan ook nog groter.

Uit de beschrijving van de operaties blijkt waarom de bits van een sleutel geen prefix mogen zijn van dat van een andere sleutel. Stel bijvoorbeeld dat de langere sleutel reeds in de boom zit. Wanneer we de kortere sleutel zoeken, of willen toevoegen, zullen er uiteindelijk geen bits meer overblijven om hem van de eerste te onderscheiden.

Een belangrijke eigenschap van tries is dat hun structuur *onafhankelijk* is van de volgorde waarin de sleutels werden toegevoegd. Voor elke verzameling (verschillende) sleutels is er slechts één trie mogelijk! (Eenvoudig aan te tonen door inductie op de deelbomen.) Bemerk dat de structuur van de meeste zoekbomen (ook de digitale) zowel afhangt van de opgeslagen sleutels als van de volgorde waarin ze werden toegevoegd.[3]

In tegenstelling tot digitale (en andere) zoekbomen gebeuren er geen sleutelvergelijkingen langs een zoekweg, maar worden enkel de opeenvolgende bits van de zoeksleutel getest. Pas op het einde is er eventueel één sleutelvergelijking. Dat is interessant wanneer de sleutels lange strings zijn.

In een trie opgebouwd uit n gelijkmatig verdeelde sleutels vereist zoeken of toevoegen van een willekeurige sleutel gemiddeld $O(\lg n)$ *bit*operaties. Dat aantal is trouwens nooit hoger dan de bitlengte van de zoeksleutel. Net zoals bij een digitale zoekboom wordt de hoogte van een trie beperkt door de bitlengte van de langste opgeslagen sleutel. Dat is meteen de bovengrens voor het aantal operaties bij zoeken of toevoegen, in het slechtste geval.

10.3.1.2 Meerwegstries

Om de hoogte van een trie met lange sleutels te beperken kan men in elke knoop meerdere (opeenvolgende) sleutelbits tegelijk behandelen, wat trouwens ook radix sort sneller maakt. (Patriciatries bieden een alternatieve oplossing, zie onder.)

Stel dus dat een sleutelelement nu m verschillende waarden kan aannemen. Elke inwendige knoop van een trie krijgt dan (potentieel) m kinderen, zodat hij een m-wegsboom wordt ('multiway trie').

Zoeken en toevoegen werken analoog als bij een binaire trie. In elke knoop van een zoekweg moet nu een m-wegsbeslissing genomen worden, op grond van het volgende sleutelelement. Om deze operatie $O(1)$ te houden, kan men de wijzers naar de kinderen opslaan in een *tabel*, en deze indexeren met het sleutelelement. Toevoegen van een sleutel kan opnieuw de aanmaak vereisen van een opeenvolgende reeks inwendige knopen met slechts één kind. Met tabellen in elke knoop betekent dit natuurlijk veel plaatsverlies.

Aangezien de kinderen van de knopen geordend zijn volgens de waarde van de sleutelelementen, kunnen de sleutels opnieuw in 'alfabetische' volgorde uit de boom opgehaald worden.

[3] Aangezien extendible hashing steunt op een onderliggende trie, was ook deze gegevensstructuur enkel afhankelijk van de opgeslagen sleutels.

Ook de performantie van meerwegstries is analoog met die van binaire tries. Zoeken of toevoegen van een willekeurige sleutel in een boom met uniform verdeelde sleutels vereist gemiddeld $O(\log_m n)$ testen (op *sleutel*elementen), en het aantal testen is nooit groter dan de lengte van de zoeksleutel. Opnieuw wordt de boomhoogte, en dus het aantal operaties in het slechtste geval, beperkt door de lengte van de langste opgeslagen sleutel (gerekend in sleutelelementen). Een m-wegstrie voor n gelijkmatig verdeelde sleutels heeft gemiddeld slechts $n/\ln m$ inwendige knopen, maar het aantal wijzers in die knopen wordt wel $mn/\ln m$.

Tot nog toe hebben we steeds uitgesloten dat een sleutel een prefix zou zijn van een andere. Want hoe moet men in een binaire boom aanduiden dat de bits van een kortere sleutel op een langere zoekweg 'opgebruikt' zijn? Men zou dit eventueel kunnen oplossen door informatie in de inwendige knopen op te slaan, omdat elk van die knopen overeenkomt met een sleutelprefix. Bij meerwegsbomen kan dat probleem opgelost worden door elke sleutel af te sluiten met een speciaal sleutelelement, dat in geen enkele sleutel voorkomt (zoals het 'nullcharacter' bij C-strings). Elke knoop moet dan een extra kind krijgen dat overeenkomt met dat afsluitelement.

Het grootste nadeel van meerwegstries is dat ze veel geheugen gebruiken. De knopen dicht bij de wortel hebben meestal wel veel kinderen, maar deze op de laagste niveaus daarentegen heel weinig. Mogelijke verbeteringen zijn:

- In plaats van bij elke knoop een tabel met m wijzers te voorzien, waarvan de meeste vaak toch nullwijzers zijn, slaat men de kindwijzers soms op in een gelinkte lijst (zoals bij de originele tries van de la Briandais), ten koste van enig snelheidsverlies. Elke knoop van die lijst bevat een sleutelelement en een wijzer, en de lijst is gerangschikt volgens deze sleutelelementen. (Waarom?)

  Wanneer een sleutelelement veel waarden kan aannemen (m groot), gebruikt men soms verschillende mechanismen, afhankelijk van het niveau in de boom. Op de eerste niveaus kunnen er veel kinderen verwacht worden, zodat een tabel zin heeft. Op de laagste niveaus zullen er normaal weinig kinderen zijn, zodat een gelinkte lijst aangewezen is. En op intermediaire niveaus maakt men gebruik van hashtabellen of efficiënte zoekbomen.

- Ook kan men een trie enkel voor de eerste niveaus gebruiken, en dan overschakelen op een andere gegevensstructuur (zoals trouwens ook bij extendible hashing gebeurt). Zo kan men bijvoorbeeld stoppen wanneer een deelboom niet meer dan s sleutels bevat. Deze sleutels kan men dan in een (korte) lijst opslaan, die nadien sequentieel doorzocht wordt. Met als gevolg dat het aantal inwendige boomknopen met een factor s daalt, tot ongeveer $n/(s \ln m)$.

10.4 VARIABELELENGTECODERING

Klassiek slaat men data op in de vorm van gegevensvelden met een vaste grootte: een

letter wordt opgeslagen als een byte van 8 bits, een integer als een woord van 32 bits, en zo verder. Dit gebeurt ook vaak in gegevensstromen. In sommige gevallen is het echter nuttig om een variabele lengte te voorzien. Twee belangrijke redenen hiervoor zijn:

(1) Verhoogde flexibiliteit. Een 32-bitsnotatie van een getal beperkt het aantal mogelijkheden op een manier die initieel misschien geen problemen oplevert, maar na zekere tijd wel. Voorbeelden van dergelijke problemen zijn de beperkte capaciteit van IPv4-adressen en de beperkingen van ASCII die tot Unicode leidden.
(2) Compressie. In dit geval heeft men een stroom van letters van een bepaalde bitlengte. Men zoekt nu een code[4], die ervoor zorgt dat veelgebruikte letters een kort codewoord krijgen zodat het gewogen gemiddelde van de lengte van de codewoorden kleiner is dan dat van de oorspronkelijke letters. Dit gebeurt bijvoorbeeld in de tweede fase van ZIP-compressie (of juister, in het DEFLATE-algoritme gebruikt door ZIP).

In beide gevallen hebben we dus een alfabet waarbij we waarbij we niet elke letter door evenveel bits voorstellen. Dit alfabet kan in principe oneindig groot zijn, bijvoorbeeld als we een geheel getal willen voorstellen zonder dat we kennis hebben van een beperking in grootte van het getal, of eindig (het alfabet gebruikt in ZIP heeft 288 letters).

Variabelelengtecodering heeft een belangrijk nadeel. Als we een blok gegevens hebben, bijvoorbeeld een schijfblok dat een aantal gehele getallen bevat, moeten we het gehele blok decoderen voor we weten waar elk getal begint. Dit sluit bijvoorbeeld binair zoeken uit in dat blok, ook als de getallen geordend werden opgeslagen. Variabelelengtecodering wordt dan ook alleen gebruikt als de voordelen opwegen tegen dit nadeel.

Bij het decoderen is het praktisch als we een zogenaamde *prefixcode* hebben. Dit is een codering waarbij een codewoord nooit het prefix van een ander codewoord kan zijn. Dit heeft als voordeel dat we weten dat we het einde van een codewoord bereikt hebben zonder dat we het begin van het volgende codewoord moeten analyseren.

- We nemen volgende code voor een natuurlijk getal: we schrijven het in het 128-delig stelsel en slaan elk cijfer op in een aparte byte. Alleen tellen we bij het laatste cijfer 128 op (zodat de laatste byte een 1-bit heeft op de meest significante plaats). Dit is een typische prefixcode.
- Als we geschreven taal beschouwen als een string met als alfabet alle mogelijke woorden en leestekens, dan is de normale weergave een variabelelengtecodering die *geen* prefixcode is. Immers, een woord in gewone letters kan een prefix zijn van een ander woord, zodat we moeten wachten tot we een leesteken tegenkomen (een spatie is een leesteken) om te weten dat een woord gedaan is.

[4] Voor alle duidelijkheid onderscheiden we het begrip *code* van het begrip *codewoord*. Een code is een afbeelding die elke letter van het alfabet afbeeldt op een codewoord.

Een trie is een uitstekend instrument voor de decodering van een invoerstroom gecodeerd met een prefixcode, of toch van een eindige prefixcode. Eerst slaan we alle codewoorden in de trie op. Aan het begin van een codewoord vertrekken we aan de wortel en per ingelezen symbool (dat is vaak een bit, maar zoals bij ons voorbeeld kan het ook een byte zijn) gaan we een niveau naar omlaag in de trie. Als we bij een blad komen weten we dat ons codewoord volledig is. De bijbehorende data kunnen opgeslagen zijn in dat blad.

10.4.1 Universele codes

Universele codes zijn prefixcodes die onafhankelijk zijn van de gekozen brontekst. Ze worden vaak toegepast als onderdeel van andere compressiealgoritmes.

10.4.1.1 De Eliascodes en de Fibonaccicode

De meest eenvoudige manier om deze universele codes te introduceren, is als codering voor de verschillende positieve gehele getallen. In de onderstaande tabel geven we drie verschillende courante voorbeelden. Hoewel dit op het eerste zicht niet zo duidelijk is, blijken deze codes interessante statistische eigenschappen te hebben die meestal leiden tot goede compressies. Je kan ze inderdaad beschouwen als Huffmancodes voor een oneindig alfabet.

Elias' gammacode voor het getal n (behalve nul) genereer je op de volgende manier: stel het getal voor met zo weinig mogelijk bittekens (stel k bittekens; het eerste teken is dan '1') en laat dit voorafgaan door $k-1$ nulbits. Merk op dat dit een prefixcode is: aan het aantal nulbits vóór de eerste éénbit weet het decodeerprogramma hoeveel bittekens er daarna nog zullen volgen. Het getal n wordt dus voorgesteld door $2\lfloor \log_2 n \rfloor + 1$ bittekens[5].

Bij *Elias' deltacode* voor het getal n, gebruik je de laatste $k-1$ bittekens van het getal (dus zonder de eerste éénbit) en laat je dit voorafgaan door Elias' gammacode voor k. Je ziet ook hier gemakkelijk in dat de prefixeigenschap voldaan is. We hebben hier $\lfloor \log_2 n \rfloor + 2\lfloor \log_2(\log_2 n + 1) \rfloor + 1$ bittekens nodig om het getal n te coderen.

De Fibonaccicode is gebaseerd op de beroemde reeks van Fibonacci

1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...

waarin elk getal de som is van de twee vorige. Deze rij heeft de eigenschap dat elk getal n op precies één manier kan geschreven worden als de som van verschillende Fibonaccigetallen zodanig dat er nooit twee getallen in de reeks worden gebruikt die onmiddellijke buren zijn van elkaar. Een aantal voorbeelden:

12 = 8 + 3 + 1, 26 = 21 + 5, 72 = 55 + 13 + 3 + 1

[5] $\lfloor x \rfloor$ is x naar beneden afgerond tot een geheel getal

Om nu de Fibonacci*code* van het getal te bepalen, overloop je de Fibonaccireeks van klein naar groot en gebruik je een éénbit voor elk getal dat in de berekende som voorkomt, en een nulbit voor alle andere getallen. Om af te sluiten plaats je een extra éénbit na de laatste éénbit die je op die manier bekwam. Ook dit is een prefixcode: het einde van elk codewoord bestaat uit twee opeenvolgende éénbits, een paar dat niet midden in een codewoord kan optreden. Het is gemakkelijk in te zien dat voor een getal n waarvoor $F_k \leq n < F_{k+1}$, waarbij F_k het $k$de Fibonaccigetal voorstelt, we $k + 1$ bits nodig hebben.

	Elias' gammacode	Elias' deltacode	Fibonaccicode
1	1	1	11
2	010	0100	011
3	011	0101	0011
4	00100	01100	1011
5	00101	01101	00011
6	00110	01110	10011
7	00111	01111	01011
8	0001000	00100000	000011
9	0001001	00100001	100011
...			
23	000010111	001010111	01000011
45	00000101101	0011001101	001010011
...			

10.5 HUFFMANCODERING

Wanneer we een tekst in een computerbestand opslaan, slaan we elk letterteken van deze tekst doorgaans op als een afzonderlijke byte van 8 bittekens, steunend op de ASCII-code. In de meeste tekstbestanden komen bepaalde letters echter meer voor dan andere. Wanneer we minder bittekens zouden gebruiken voor letters die veel voorkomen (en vice versa) dan kan wellicht de totale lengte van het bestand gevoelig worden ingekort.

10.5.1 Opstellen van de decoderingsboom

Bij Huffmancodering wordt een prefixcode toegepast waarbij elke letter een apart codewoord heeft dat over het hele bericht dezelfde blijft (er bestaan methodes waarbij de codering onderweg verandert. Daar gaan we hier niet op in). We gaan in deze paragraaf uit, zonder veel verlies van algemeenheid, dat we bitcodes gebruiken en dat de op te stellen trie dus binair is. Om een optimale code op te stellen moeten we weten hoe vaak elk codewoord gaat gebruikt worden. Om de aandacht te vestigen gaan we uit van een bestand waarbij we voor elke letter kunnen tellen hoe vaak ze voorkomt. Uitgaande

van een alfabet $\Sigma = \{s_i | i = 0, \ldots, d-1\}$ krijgen we dus een reeks van frequenties f_i. We zoeken dus een trie met n bladeren die de optimale code oplevert[6]

Het heeft uiteraard geen zin om een trie te bekijken die niet volledig is, d.w.z. een trie met een inwendige knoop met maar één kind. Immers, als we zo een knoop wegknippen en zijn kind rechtstreeks aan zijn ouder hangen (of wortel maken, als de verdwenen knoop de wortel was), dan krijgen alle bladeren onder de verdwenen knoop een korter codewoord en dus hebben we daarmee een betere trie. Nemen we een willekeurige volledige binaire trie met d bladeren, elk met een letter uit Σ. We gaan nu elk van de knopen een gewicht toekennen:

- Elk blad krijgt als gewicht de frequentie van de overeenkomstige letter.
- Elke inwendige knoop krijgt als gewicht de som van de gewichten van zijn kinderen. Dit is ook de som van de gewichten van de bladeren die onder de knoop hangen.

Stel dat het bestand gecodeerd wordt met de bijbehorende code zodat we de trie kunnen gebruiken om te decoderen. Voor elke bit gaan we één niveau omlaag in de trie en komen we bij een nieuwe knoop. Het totaal aantal bits in het bestand is dus het aantal keren dat we bij een knoop aankomen die niet de wortel is (naar de wortel springen we op het einde van een codewoord, zonder een bit te gebruiken). Kijken we naar een vaste knoop k, hoe vaak komen we daar langs? Elke keer als we naar een blad gaan onder die knoop. Het aantal keer dat we naar een bepaald blad gaan wordt gegeven door zijn gewicht, het aantal keer dat we langs knoop k gaan wordt dus ook gegeven door *zijn* gewicht. Het aantal bits in het gecodeerde bestand is dus de som van de gewichten van alle knopen samen, met uitzondering van de wortel. Maar de wortel heeft als gewicht n, waarin n het aantal letters in het bestand is. We zoeken dus een trie met een minimaal gewicht (het heeft geen belang of we daarin ja dan neen de wortel meetellen).

Stel nu dat we een trie hebben met daarin een knoop k met gewicht w_k op diepte d_k en een knoop ℓ met gewicht w_ℓ op diepte w_ℓ, zodanig dat k niet onder ℓ hangt en ℓ niet onder k. We kunnen een nieuwe trie maken door k, inclusief de bijbehorende deelboom, van plaats te verwisselen met ℓ. Verandert daarmee het gewicht van de trie? Even de factuur opmaken.

- Er waren d_k knopen boven k in de trie. Deze verliezen gewicht w_k, maar krijgen gewicht w_ℓ bij.
- Analoog zijn er d_ℓ knopen die een gewicht w_ℓ verliezen, maar een gewicht w_k bijkrijgen.

[6] Op het eerste gezicht lijkt dit erg op het zoeken van de optimale zoekboom zoals we in paragraaf 2.1 beschreven hebben. Dynamisch programmeren is hier echter niet bruikbaar. Bij een zoekboom met d sleutels kunnen we de verzameling sleutels op $d+1$ manieren verdelen over linker- en rechterdeelboom. Hier hebben we echter niet de sleutels maar de bijbehorende data (de letters), en die kunnen we op 2^d manieren verdelen over linker- en rechterdeeltrie.

Alles samengeteld krijgen we dus een gewichtsverandering van

$$(d_k - d_\ell)(w_\ell - w_k).$$

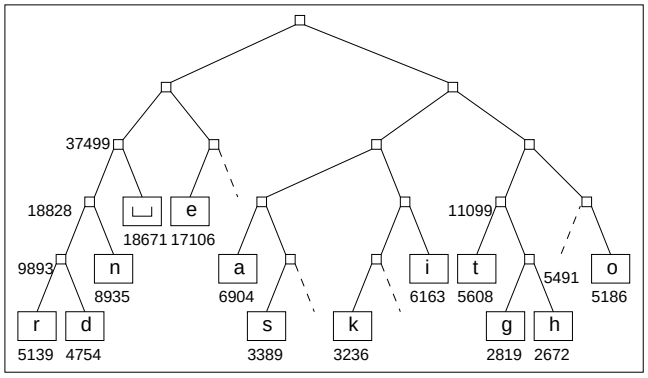
Er is nog een andere manier om tot dit resultaat te komen: de bladeren onder k[7] krijgen allemaal een nieuw codewoord: het prefix van lengte d_k wordt vervangen door een van lengte d_ℓ. In de codering van het hele bestand zijn dus w_k codewoorden vervangen door een codewoord met lengteverschil $d_\ell - d_k$ en, op analoge manier, w_ℓ codewoorden vervangen met een lengteverschil van $d_k - d\ell$.

Heeft ℓ een *groter* gewicht maar (na de wijziging) een *kleinere* diepte dan k dan hebben we een betere trie gevonden en dan was de oude trie zeker niet optimaal. Zijn de dieptes gelijk dan mogen we de verwisseling doorvoeren zonder dat het resultaat slechter is.

We kunnen dus een optimale trie opstellen met de volgende eigenschappen:

(1) Geen enkele knoop heeft een groter gewicht dan een knoop op een kleinere diepte (elke optimale trie voldoet daaraan).
(2) Geen enkele knoop heeft een groter gewicht dan een knoop links van hem op dezelfde diepte (want dan kunnen we de twee knopen verwisselen).

We hebben dit principe geschetst in Figuur 10.1.



Figuur 10.1. Een Huffmantrie

We veronderstellen even dat we nooit twee knopen gaan tegenkomen met gelijk gewicht om de uitleg te vereenvoudigen. We bouwen de boom nu op van onder naar boven. Op elk ogenblik hebben we een *bos* van deelbomen die we verder aan mekaar moeten hangen. Dit doen we door twee bomen uit het bos te halen, ze te verenigen onder een nieuwe knoop en de nieuwe boom terug in het bos te steken. In het begin bestaat ons bos uit alle bladeren.

[7] We zeggen dat een blad onder zichzelf hangt.

De diepte van de trie die we gaan construeren is h. We weten niet hoe groot h is. Wel weten we dat alle knopen op niveau h bladeren zijn, dat het een even aantal is en dat, als we alle bladeren sorteren naar gewicht (kleinste eerst), de bladeren op dat niveau alle vooraan staan, en wel in een volgorde die van rechts naar links gaat in de trie. We kunnen dus alvast beginnen met deze bladeren twee aan twee samen in een boom te steken, elke keer de twee lichtste overblijvende bladeren nemend. Deze boom steken we terug in ons geordende bos, ook weer gesorteerd naar gewicht. Vermits deze boom (correct taalgebruik: zijn wortel) op niveau $h-1$ in het resultaat komt, staat hij verder naar achter in ons gesorteerd bos dan de overblijvende knopen van niveau h.

Op het ogenblik dat we klaar zijn met niveau h steken alle knopen/bomen van niveau $h-1$ in het bos. Immers, ofwel hebben we ze geconstrueerd door knopen uit het vorige niveau samen te nemen ofwel zijn het bladeren die we al hadden. Bovendien hebben we ze nog niet verwerkt, want hun gewicht is groter dan dat van de knopen op niveau h, en bovendien staan ze helemaal vooraan. We kunnen dus verder gaan met elke keer de twee lichtste bomen samen te nemen in een nieuwe boom. Hoe weten we dat we klaar zijn met niveau h? het antwoord is dat we dat niet weten, maar dat we dat ook niet moeten weten, vermits we bij overgang naar niveau $h-1$ gewoon verder gaan met hetzelfde als wat we al deden. Dit geldt voor elke overgang naar een hoger niveau. Uiteindelijk houden we maar één boom over. Dat is onze trie.

Het is duidelijk dat we in de praktijk ons bos niet volledig sorteren, vermits we telkens alleen de lichtste boom nodig hebben. Merken we verder nog op dat we de Huffmantrie hebben opgesteld aan de hand van het bestand en dat we deze nodig hebben voor de decodering. In toepassingen moet men dus ofwel de Huffmantrie zelf op een of andere manier bij het bestand voegen ofwel gebruik maken van een vaste, vooraf afgesproken Huffmantrie.

10.5.2 Patriciatries

Veel trieknopen hebben maar één kind, wat onnodig veel geheugen vereist. Bovendien zijn er twee soorten knopen: inwendige knopen met kinderen (wijzers) maar zonder sleutel, en bladeren met sleutel maar zonder kinderen.

Patriciatries ('**P**ractical **a**lgorithm **t**o **r**etrieve **i**nformation **c**oded **i**n **a**lphanumeric', Morrison 1968)[8] vermijden die problemen door enkel knopen met meer dan één kind te behouden.

Bij binaire tries wordt het daarmee mogelijk om slechts één soort knopen te gebruiken. Sleutels worden nu wel in inwendige knopen opgeslagen, en wijzers naar bladeren worden vervangen door wijzers naar inwendige knopen, hoger in de boom.

We behandelen eerst tries die de laatste techniek niet toepassen. Bij een gewone meerwegstrie kunnen knopen voorkomen met maar één kind. We kunnen zo'n knoop weglaten en het kind in de plaats zetten, maar dat heeft twee gevolgen:

[8] Origineel kwam het woord 'trie' hierin niet voor.

(1) Om de juiste weg te vinden in het kind moeten we een karakter van de string overslaan. We moeten dus in het kind een aanduiding geven dat de ouder ontbreekt of, juister, hoeveel voorouders ontbreken (er kan een hele rij zijn van onder elkaar staande knopen die maar een kind hadden in de oorspronkelijke trie). Voor de eenvoud gaan we de index van het te testen karakter opslaan, de testindex.
(2) Omdat we nu een karakter niet getest hebben kan het zijn dat dit karakter in de zoekstring niet overeenkomt met het karakter dat in de verdwenen knoop naar zijn kind leidde.

We noemen een knoop expliciet als hij nog voorkomt in de resulterende boom en impliciet als hij alleen wordt aangeduid door een indexsprong aangegeven in de nakomeling.

Door de wijziging in de structuur veranderen ook de operaties. Een ledige trie is een speciaal geval: we gaan er dus even van uit dat de trie niet leeg is.

Bij het zoeken gaan we naar beneden in de boom en nemen het karakter aangegeven door de indexsprong om de weg in de volgende knoop te bepalen. Leidt dit naar een nulpointer dan zit de string niet in de trie. Komen we echter in een blad dan weten we niet of we de string gevonden hebben: karakters die we onderweg hebben overgeslagen kunnen verschillen van de overeenkomstige karakters van de string in het blad zodat we dit moeten controleren. In de praktijk vergelijken we gewoon de zoekstring met de string in het blad. Die laatste moet dus expliciet worden opgeslagen. Bij veel implementaties heeft elke inwendige knoop ook een verwijzing naar een (willekeurig) onderliggend blad. Op die manier kunnen we, als we in een inwendige knoop komen met impliciete voorouders meteen controleren of onze zoekstring overeenkomt met de letters van de impliciete knopen.

Ook toevoegen wordt nu ingewikkelder. Bij een gewone trie gaan we verder tot we een nulpointer tegenkomen, waaraan we dan een blad toevoegen, of tot we een blad tegenkomen, waardoor we weten dat de string al in de trie zit. Nu kan het echter zijn dat we het blad moeten toevoegen aan een impliciete knoop. Bij de versie zonder verwijzingen naar een willekeurig blad moeten we altijd verder gaan tot in een blad (als we al weten dat de nieuwe string niet overeenkomt met het pad dat we volgen nemen we elke keer een willekeurig kind), als we wel zo'n wijzer hebben kunnen we meteen controleren of we in tussenliggende impliciete knopen wel de goede kant opgaan. In elk geval krijgen we een *verschilindex* die de eerste plaats aanduidt waar de nieuwe string verschilt van de meest gelijkende string in de trie (deze met het langste gemeenschappelijke prefix). Als de boom leeg is dan zetten we de verschilindex op nul.

Als de nieuwe string nog niet in de trie zit hebben we drie mogelijkheden:

(1) Zoeken eindigt in een expliciete knoop waar de testindex gelijk is aan de verschilindex, maar die geen kind heeft voor het karakter in de string aangeduid door de verschilindex. Deze is geen blad, want dan zou een prefix van de nieuwe string al in de trie zitten. We kunnen dus zondermeer een blad toevoegen voor de nieuwe string.

(2) We komen uit in een expliciete knoop waar de testindex groter is dan de verschilindex. Er moet een expliciete knoop worden toegevoegd met als testindex de verschilindex. Deze krijgt twee kinderen: de oude expliciete knoop (via het oude karakter) en het nieuwe blad (via het nieuwe karakter).
(3) We geraken opnieuw tot in het blad. Als we een blad opvatten als een knoop met oneindig grote testindex dan is dit een speciaal geval van het vorige.

Omdat we bij toevoegen soms van een expliciete knoop naar een willekeurig onderliggend blad moeten kan het dus handig zijn een wijzer naar een blad op te nemen in een expliciete knoop. We maken echter alleen een expliciete knoop aan bij toevoegen, en dan maken we ineens ook een kindblad aan: de wijzer kan dus naar dit nieuwe blad wijzen.

Bij een binaire patriciatrie heeft elke expliciete inwendige knoop twee kinderen, zodat geval (1) uit het toevoegscenario niet kan voorkomen. We moeten dus altijd zowel een blad als een expliciete inwendige knoop toevoegen. Men smelt deze samen in één structuur, zodat de noodzaak van bladpointers verdwijnt.

Bij afdalen in de boom kijkt men alleen naar het inwendigeknoopgedeelte. Men weet dat men in een blad aankomt als men bij afdalen stuit op een knoop met een testindex die niet groter is dan de vorige. Voor n sleutels zijn er dus n knopen vereist, die $2n$ wijzers bevatten. Daarvan verwijzen er n naar een sleutel, en $n-1$ naar een kindknoop. (Eén wijzer wordt dus niet gebruikt.)

Als we een string toevoegen wordt er dus één structuur bijgemaakt, een versmelting van een inwendige knoop met een blad. Aanvankelijk is de inwendige knoop de directe ouder van het blad, maar latere wijzigingen kunnen daar nog knopen tussenvoegen.

In tegenstelling tot een trie hangt de structuur van een binaire patriciatrie wél af van de volgorde waarin de sleutels werden toegevoegd. De structuur van de knopen blijft weliswaar onafhankelijk van deze volgorde, maar welk blad versmolten is met een inwendige knoop hangt af van de volgorde: het is steeds het *tweede* blad onder de inwendige knoop, behalve bij de wortel. Dit komt omdat een expliciete inwendige knoop wordt samengeplakt met het blad dat tot het ontstaan van de inwendige knoop leidde.

In principe kan men een lege patriciatrie voorstellen door als wortel een kindloze 'inwendige' knoop te nemen met testindex 0. Als we echter het formalisme voor een binaire patriciatrie nemen is dit een probleem omdat elke knoop plaats voor een sleutel bevat. Men moet dus, bij zoeken en toevoegen, onderscheid maken tussen een lege boom (de wortel heeft twee nulpointers en is een inwendige knoop, of de wortel is afwezig) en een niet-lege boom. Bij een boom met één sleutel is de wortel een blad met testindex -1.

In een (binaire) patriciatrie met n gelijkmatig verdeelde sleutels vereist zoeken of toevoegen van een willekeurige sleutel gemiddeld $O(\lg n)$ bitvergelijkingen, en nooit meer dan de bitlengte van de zoeksleutel. De hoogte van de boom en dus het maxi-

maal aantal bitvergelijkingen worden opnieuw beperkt door de lengte van de langste opgeslagen sleutel.

Gewoonlijk hangt de zoektijd in een gewone trie af van de sleutellengte (de eerste onderscheidende bit kan immers willekeurig ver in de sleutel liggen). Dat is trouwens ook zo bij alle zoekmethoden gebaseerd op sleutelvergelijkingen. En zelfs bij hashing, omdat een goede hashfunctie in principe alle sleutelelementen gebruikt. Patriciatries testen echter meteen (en enkel) de belangrijke karakters, zodat de zoektijd *niet* toeneemt met de sleutellengte. Wel neemt uiteraard de tijd om te vergelijken met de string in het blad toe. Ze zijn dus zeer geschikt voor lange sleutels (zoals de suffixen van een lange tekst, hun originele toepassing trouwens). Ook zijn er toepassingen waarbij men zoekt in een trie terwijl men zeker weet dat de string er wel degelijk inzit en men op zoek is naar data die in het blad zijn opgeslagen.

Aangezien een patriciatrie uiteindelijk een alternatieve voorstelling is van een gewone trie, blijft de volgorde van de opgeslagen sleutels intact.

Kortom, patriciatries combineren de voordelen van digitale zoekbomen en tries:

- Zoals digitale (en gewone) zoekbomen gebruiken ze niet meer geheugen dan nodig. Er zijn n bladeren en hoogstens $n - 1$ inwendige knopen, want elke inwendige knoop heeft minstens twee kinderen. Voor binaire patriciatries is het geheugengebruik optimaal; als m groot is kunnen inwendige knopen veel plaats innemen omdat ze een tabel met m pointers bevatten. Het kan helpen om de de tabel te vervangen door een andere structuur zoals een gelinkte lijst of een binaire zoekboom, maar dit gaat ten koste van tijdefficiëntie.
- Ze zijn even efficiënt als tries: zoeken vereist gemiddeld $O(\lg n)$ karaktervergelijkingen, gevolgd door één sleutelvergelijking.
- Net zoals tries (en gewone zoekbomen) respecteren ze de volgorde van de sleutels, zodat bijkomende operaties mogelijk zijn.

10.6 TERNAIRE ZOEKBOMEN

Deze zoekbomen zijn al lang bekend (Clampett, 1964), maar hun praktisch belang werd pas vrij recent aangetoond (Bentley en Sedgewick, 1997 [6]). Een ternaire zoekboom is eigenlijk een alternatieve voorstelling van een meerwegstrie. De snelste implementatie van een m-wegstrie gebruikt immers een tabel met m kindwijzers in elke knoop, wat onnodig veel geheugen vereist. Om dat te vermijden gebruikt men een ternaire boom waarvan elke knoop een *sleutelelement* bevat.

Zoeken vergelijkt telkens een zoeksleutelelement met het element in de huidige knoop. Is het zoeksleutelelement kleiner (groter), dan zoeken we verder in de linkse (rechtse) deelboom, met *hetzelfde* zoeksleutelelement. Is het element echter gelijk aan dat in de

knoop, dan zoeken we verder in de middelste deelboom, met het *volgende* zoeksleutelelement (als dat bestaat). In tegenstelling tot een digitale zoekboom of een trie wordt het te testen sleutelelement dus *niet* bepaald door de diepte van de knoop. We vinden de opeenvolgende elementen van zoeksleutels enkel wanneer we de middelste wijzers volgen.

Om ervoor te zorgen dat geen enkele sleutel een prefix is van een andere, gebruiken we opnieuw een geschikt afsluitelement, zoals bij een meerwegstrie.

Een zoeksleutel wordt gevonden wanneer we met zijn afsluitelement bij een knoop met datzelfde element terechtkomen. De boom bevat de zoeksleutel niet wanneer we bij een nullwijzer terechtkomen, of wanneer we het einde van de zoeksleutel bereiken vooraleer een afsluitelement in de boom gevonden werd.

Toevoegen begint zoals gewoonlijk met zoeken, gevolgd door het aanmaken van een reeks opeenvolgende knopen voor alle volgende elementen van de zoeksleutel, net zoals bij (oorspronkelijke) tries.

De tijd vereist om te zoeken of toe te voegen in een standaard ternaire zoekboom is evenredig met de sleutellengte. Het aantal knopen hangt enkel af van de opgeslagen sleutels, en is dus onafhankelijk van hun toevoegvolgorde. Met elk (verschillend) sleutelprefix komt immers één knoop overeen.

Net zoals tries behouden ternaire zoekbomen de volgorde van de opgeslagen sleutels. Overlopen in inorder geeft dus die volgorde, en ook zoeken naar een voorloper of opvolger van een sleutel is mogelijk.

Ternaire zoekbomen hebben een aantal voordelen:

- Aan onregelmatig verdeelde zoeksleutels, die in de praktijk vaak voorkomen, passen ze zich goed aan:
  - Soms is het aantal mogelijke sleutelelementen groot (bijvoorbeeld vreemde alfabetten, of de Unicode standaard, met duizenden karakters), en is het gebruik van deze elementen helemaal niet uniform (een bepaalde groep sleutels kan bijvoorbeeld slechts een klein gedeelte van het alfabet gebruiken). Zeker in dit geval zouden standaard meerwegstries met hun tabellen veel te veel plaats gebruiken. Bovendien hoeven we ons niet af te vragen welk deel van het alfabet er gebruikt wordt.
  - In de praktijk hebben sleutels vaak een gestructureerd formaat, dat verschilt naargelang de toepassing: een deel van de sleutel bestaat bijvoorbeeld enkel uit letters, en een ander deel enkel uit cijfers. De overeenkomstige deelbomen van de ternaire zoekboom gaan zich dan automatisch gedragen als zoekbomen met 26 knopen (voor de letters), of met 10 knopen (voor de cijfers).
- Zoeken naar *afwezige* sleutels is meestal zeer efficiënt, zelfs voor lange sleutels. Een hashtabel heeft alle sleutelelementen nodig voor haar hashfunctie, een

zoekboom met n sleutels heeft $\Omega(\lg n)$ sleutelvergelijkingen nodig, en zelfs patriciatries vereisen $O(\lg n)$ bitoperaties voor een willekeurige afwezige sleutel. Ternaire zoekbomen vergelijken dikwijls slechts enkele sleutelelementen, en volgen slechts enkele wijzers.

- Ternaire zoekbomen laten meer complexe zoekoperaties toe, zoals zoeken naar sleutels waarvan bepaalde elementen niet gespecificeerd zijn ('don't care'-karakters), of alle sleutels opsporen die in niet meer dan één element verschillen van de zoeksleutel.

Er zijn enkele verbeteringen mogelijk aan ternaire zoekbomen:

- Het aantal knopen kan beperkt worden door de sleutels in bladeren op te slaan zodra men ze kan onderscheiden (zoals bij tries), en door inwendige knopen met slechts één kind te elimineren via een sleutelelementindex (zoals bij patriciatries). Net zoals bij patriciatries wordt de zoektijd dan onafhankelijk van de sleutellengte, wat interessant is voor lange sleutels.
- Nog een eenvoudige maar effectieve verbetering vervangt de wortel door een meerwegstrieknoop, zodat we een tabel van ternaire zoekbomen bekomen. Als het aantal mogelijke sleutelelementen m niet te groot is, kunnen we een tabel van m^2 ternaire zoekbomen gebruiken, zodat er een zoekboom overeenkomt met elk eerste paar sleutelelementen. Dit heeft natuurlijk enkel zin als de eerste sleutelelementen gelijkmatig verdeeld zijn.

Ternaire zoekbomen met deze verbeteringen behoren tot de efficiëntste woordenboekstructuren voor strings, en ze laten bovendien nog andere operaties toe.

HOOFDSTUK 11
ZOEKEN IN STRINGS

Algemeen gebruikte symbolen in dit hoofdstuk:

Symbool	Betekenis
Σ	Het gebruikte alfabet
$\Sigma*$	De verzameling strings van eindige lengte van letters uit Σ
d	Aantal karakters in Σ
P	Patroon (naald)
p	Lengte van P
T	Tekst (hooiberg)
t	Lengte van T

Een fundamenteel zoekprobleem voor strings tracht een bepaalde string (het 'patroon') in een (langere) string (de 'tekst') te lokaliseren. We nemen hier aan dat men *alle* plaatsen zoekt waar dat patroon voorkomt. (Bemerk dat patronen op twee of meer van die plaatsen kunnen overlappen.) Evidente toepassingen vindt men bij tekstverwerkers en tekstzoekmachines, maar deze technieken worden bijvoorbeeld ook gebruikt bij het zoeken naar patronen in DNA. Waar bij veel toepassingen efficiëntie niet van zeer groot belang is (zo is het zoeken in tekst door een tekstverwerker, waarbij een eenvoudige string gezocht wordt in een tekst van een paar tienduizenden woorden niet direct een groot probleem), zijn er toepassingen die zeer kritisch zijn in dat opzicht. Denken we bijvoorbeeld aan een virusscanner die signaturen van virussen moet zoeken in elk programma dat loopt.

Ook moeten we niet alleen denken aan 'tekst' in de klassieke betekenis van dat woord. Bij een virusscanner bestaat de 'tekst' uit uitvoerbare code. Zoeken in binaire strings komt voor bij beeldverwerking, of bij systeemprogrammatuur. Het groot belang van deze problemen heeft heel wat onderzoek en resultaten opgeleverd (zie bijvoorbeeld [20] en [12]), zodat we in dit kort bestek slechts een idee kunnen geven van enkele belangrijke algoritmen of interessante ideeën.

Voor het vervolg maken we gebruik van de terminologie van tekststrings, en we onderstellen dat zowel het patroon P (met lengte p) als de tekst T (met lengte t) in het (inwendig) geheugen opgeslagen zijn. Soms vindt men in de literatuur termen die niet meer doen denken aan klassieke tekst: zo spreekt men soms van de *naald* en de *hooiberg*.

In een aantal gevallen is het nuttig om simultaan naar verschillende naalden in dezelfde

hooiberg te zoeken. Een voorbeeld daarvan is onze virusscanner, maar er zijn er vele andere. Klassiek is het hoofdletteronafhankelijk zoeken in 'echte' tekst. Soms, als men niet zeker is van de juiste spelling, wil men benaderend zoeken. In sommige gevallen kan men gebruik maken van klassieke methodes die men enigszins aanpast (zo is bijvoorbeeld het Karp-Rabinalgoritme gemakkelijk aan te passen als men een vrij klein aantal verschillende strings van dezelfde lengte moet zoeken). Dergelijke problemen doen een aantal vragen rijzen. Eén ervan is de vraag wat de beste methode is om de verzameling van gezochte strings te beschrijven; een tweede vraag is wat goede methodes zijn om strings van die taal te vinden. De tak van de informatica die zich met dit soort vragen bezig houdt is de theorie van *formele talen*.

11.1 FORMELE TALEN

Een *formele taal* over een alfabet is een verzameling eindige strings over dat alfabet. Formele talentheorie bestudeert de vraag hoe men een taal kan beschrijven (een eindige taal kan men in principe beschrijven door een opsomming van alle elementen te geven, maar een taal kan oneindig groot zijn), wat voor soort informatieverwerkende eenheid men nodig heeft om de taal te herkennen en dergelijke vragen meer. Het begrip taal moet hier ruim worden opgevat. Naast Nederlands als menselijke taal staat dan het Nederlands als formele taal dat bestaat uit alle correcte zinnen opgevat als strings over het alfabet van fonemen. Merk op dat Nederlands als menselijke taal vaag gedefinieerd is en dat dit dus ook geldt voor de formele taal (is een zin die te lang is om ooit door iemand begrepen te worden een correcte zin?). Maar een formele taal kan ook een verzameling elementen uit een andere taal zijn (de verzameling van alle mogelijke identifiers in een Javaprogramma, bijvoorbeeld). Of iets dat nauwelijks aan het begrip 'taal' doet denken, zoals de verzameling van alle strings bestaande uit een rij 'a's gevolgd door een even lange rij 'b's.

Een algemene inleiding tot formeletalentheorie valt buiten het bestek van deze cursus, zodat we ons moeten beperken tot een aantal zaken die voor de praktijk zeer belangrijk zijn.

11.1.1 Generatieve grammatica's

Eén methode om bepaalde talen te beschrijven maakt gebruik van generatieve grammatica's. Hierbij gaat men uit van een startsymbool dat men kan transformeren tot een zin van de taal (een element van de taal wordt in deze context vaak een *zin* genoemd) met behulp van zgn. substitutieregels. Hiervoor heeft men, buiten de karakters van het alfabet Σ, ook nog zgn. *niet-terminale symbolen*, of kortweg niet-terminalen, nodig. Deze worden aangeduid met de notatie $\langle \ldots \rangle$, waarin $\ldots$ vervangen wordt door de naam van het niet-terminale symbool. De verzameling van alle strings van letters uit Σ vermengd met niet-terminalen noemen we Ξ en, uiteraard, de bijbehorende verzameling strings $\Xi*$.

Een belangrijk geval hierbij is dat van de zgn. *contextvrije talen*, waarbij men grammatica's van een bepaalde vorm kan gebruiken. Beginnend van het startsymbool (of, correct gezegd, met de string met als enige letter het startsymbool) heeft men op elk ogenblik een string uit $\Xi*$. Als men geen niet-terminale symbolen meer heeft dan heeft men een zin uit de taal, anders kan men één niet-terminaal vervangen door een string uit $\Xi*$ gegeven door een passende substitutieregel. Men spreekt van contextvrije talen omdat de substitutie onafhankelijk is wat voor en achter de betreffende niet-terminaal staat. Als er dus een regel is dat je een bepaalde niet-terminaal $\langle$**NT**$\rangle$ mag vervangen door string α, dan mag je dat altijd en overal doen waar $\langle$**NT**$\rangle$ voorkomt. Voor alle duidelijkheid: als $\langle$**NT**$\rangle$ verschillende keren in een string voorkomt dan mag je $\langle$**NT**$\rangle$ één keer vervangen en de andere kopies van $\langle$**NT**$\rangle$ laten staan. De meest gebruikelijke vorm om de grammatica uit te drukken is *BNF*, wat meestal staat voor *Backus-Naur form*. Vaak gebruikt men BNF, of een variant ervan, om een formele definitie van een bestandsformaat zoals XML te geven.

Hier is er voor elke niet-terminaal één regel, waarbij de verschillende mogelijkheden gescheiden worden door een OF-teken, |. De regel voor het startsymbool staat dan eerst. Een klein voorbeeld:

$$\begin{array}{lcl} \langle \textbf{S} \rangle & ::= & \langle \textbf{AB} \rangle | \langle \textbf{CD} \rangle \\ \langle \textbf{AB} \rangle & ::= & a \langle \textbf{AB} \rangle b | \epsilon \\ \langle \textbf{CD} \rangle & ::= & c \langle \textbf{CD} \rangle d | \epsilon \end{array}$$

Hierin zijn de kleine letters elementen van Σ, terwijl ϵ de lege string voorstelt. Deze grammatica definieert als formele taal de verzameling van alle strings ofwel bestaande uit een rij 'a's gevolgd door een even lange rij 'b's ofwel bestaande uit een rij 'c's gevolgd door een even lange rij 'd's. De afleiding van bijvoorbeeld de string "$cccddd$" wordt gegeven door

$$\langle \textbf{S} \rangle \Rightarrow \langle \textbf{CD} \rangle \Rightarrow c\langle \textbf{CD} \rangle d \Rightarrow cc\langle \textbf{CD} \rangle dd \Rightarrow ccc\langle \textbf{CD} \rangle ddd \Rightarrow cccddd.$$

Merk op dat de grammatica voor een taal niet uniek is.

11.1.2 Reguliere uitdrukkingen

Een tweede belangrijke manier om bepaalde formele talen uit te drukken maakt gebruik van *reguliere uitdrukkingen*. In plaats van 'reguliere uitdrukking' spreken we ook vaak kortweg over een *regexp*. Een regexp is een string over ons alfabet $\Sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_{d-1}\}$ aangevuld met de symbolen '$\emptyset$', 'ϵ', '$*$', '(', ')' en '$\underline{|}$' gedefinieerd door

$$\begin{array}{lcl} \langle \textbf{Regexp} \rangle & ::= & \langle \textbf{basis} \rangle | \langle \textbf{samengesteld} \rangle \\ \langle \textbf{basis} \rangle & ::= & \sigma_0 | \ldots | \sigma_{d-1} | \emptyset | \epsilon \\ \langle \textbf{samengesteld} \rangle & ::= & \langle \textbf{plus} \rangle | \langle \textbf{of} \rangle | \langle \textbf{ster} \rangle \\ \langle \textbf{plus} \rangle & ::= & (\langle \textbf{Regexp} \rangle \langle \textbf{Regexp} \rangle) \\ \langle \textbf{of} \rangle & ::= & (\langle \textbf{Regexp} \rangle \underline{|} \langle \textbf{Regexp} \rangle) \\ \langle \textbf{ster} \rangle & ::= & (\langle \textbf{Regexp} \rangle)* \end{array}$$

Om verwarring te vermijden hebben we hierboven een onderlijnd ofteken $\underline{|}$ gebruikt voor de ofoperator binnen de regexp terwijl we | gebruikten voor de of van de grammatica. In de rest van de tekst zullen we | ook gebruiken voor de ofoperator van regexps.

Elke regexp definieert een (eventueel oneindig grote) formele taal. Voor een regexp R noteren we deze taal als Taal(R). Een taal die door een regexp gedefinieerd kan worden heet een reguliere taal. De definitie van regexp en reguliere taal verloopt recursief. Op het eerste niveau hebben we de primitieve reguliere uitdrukkingen/talen.

(1) $\emptyset$ is een regexp, met als taal de lege verzameling.
(2) De lege string, voorgesteld als ϵ, is een regexp met als taal Taal$(\epsilon) = \{\epsilon\}$.
(3) Voor elke $a \in \Sigma$ is "a" een regexp[1], met als taal Taal("a") $= \{$"a"$\}$.

Regexps kunnen gecombineerd worden. Er zijn, zoals blijkt uit de gegeven grammatica, drie operaties: concatenatie, of en de *Kleenesluiting*. Hiermee komen drie operaties op talen overeen:

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

De $\cdot$-operatie toegepast op twee verzamelingen van strings geeft de verzameling $A \cdot B$ van alle strings bestaand uit een string uit A gevolgd door een string uit B. $A*$ is de verzameling van alle opeenvolgingen van een eindig aantal strings uit A.

De definitie levert alle nodige regexps. Voor bepaalde vaak voorkomende gevallen gebruikt men in de praktijk echter gewoonlijk een verkorte notatie:

- In de praktijk schrijft men niet altijd alle haakjes. Zoals in gewone algebraïsche uitdrukkingen hebben de operatoren dan een *prioriteit*. Haakjes hebben de grootste prioriteit, gevolgd door de ster (een soort machtsverheffing), dan concatenatie (een soort vermenigvuldiging) en tenslotte of (een soort optelling). Op deze manier wordt $a|bc*$ een geldige regexp, alhoewel het even nadenken is wat ermee bedoeld wordt.

  Daarmee is onze notatie $\Sigma*$ verklaard: $\Sigma*$ is de Kleenesluiting van Σ opgevat als de taal van eenletterwoorden.

- *Minstens eenmaal herhalen.* De 'positieve sluiting' $r+$ staat voor $rr*$. Deze operator heeft dezelfde prioriteit als de herhaling.

- *Optionele uitdrukking.* De notatie $r?$ staat voor $r|\epsilon$, en duidt een optionele string uit r aan.

[1] In de praktijk schrijft men vaak a (het symbool voor de letter) in plaats van "a", de correcte weergave voor de string van lengte 1. Dit is erg slordig en de meeste programmeertalen staan het niet toe (denk aan het verschil tussen 'a' en "a" in C++/Java), maar we zullen ons in deze cursus ook hieraan bezondigen en de aanhalingstekens weglaten als we een string letterlijk uitschrijven.

- *Unies van symbolen.* Met de symbolen a, b, c en z uit Σ staat $[abc]$ voor $a|b|c$, $[a-z]$ voor $a|b|\cdots|z$, en $[a-zA-Z\,0-9]$ voor een willekeurig alfamerisch teken.
- De regexp $\emptyset$ heeft weinig nut voor handgemaakte regexps. Immers, Kleenesluiting of concatenatie met een andere regexp leveren weer de lege taal en de ofoperatie met een andere regexp levert de taal van die andere regexp. Hij wordt dan ook dikwijls weggelaten.

Reguliere uitdrukkingen worden veel gebruikt in situaties waarbij een mens manueel een regexp opstelt zoals bijvoorbeeld het programma `grep` of een tekstverwerker/editor met aangepaste zoekfunctie. Het is echter ook mogelijk dat de regexp berekend wordt door een programma aan de hand van een bepaald probleem (overigens: hier kan het zijn dat de regexp $\emptyset$ wel een rol speelt).

Dat regexps verbonden kunnen worden met graafproblemen blijkt uit volgende stelling:

Stelling 1 *Zij G een gerichte multigraaf met verzameling takken Σ. Als a en b twee knopen van G zijn dan is de verzameling $P_G(a,b)$ van paden beginnend in a en eindigend in b een reguliere taal over Σ.*

Bewijs: We bewijzen dit door inductie op het aantal verbindingen m van G. Als $m = 0$ dan is de stelling triviaal. Voor $a \neq b$ is $P(a,b) = \emptyset$, want de verzameling paden is leeg. Als $a = b$ dan hebben we $P(a,b) = \{\epsilon\}$. Breiden we nu een multigraaf G waarvoor de stelling bewezen is uit naar de multigraaf G' door één verbinding toe te voegen, laat ons zeggen een verbinding v_{xy} van knoop x naar knoop y, waarbij mogelijks $x = y$. Alle paden van a naar b zijn van één van de twee volgende vormen:

(1) De paden die v_{xy} niet bevatten. Deze vormen de reguliere taal $P_G(a,b)$
(2) De paden die v_{xy} één of meer keer bevatten. Deze verzameling wordt gegeven door

$$P_G(a,x) \cdot \{v_{xy}\} \cdot (P_G(y,x)) \cdot \{v_{xy}\}) * \cdot P_G(y,b).$$

Deze is bekomen uit reguliere talen door $\cdot$- en $*$-operaties en is dus regulier.

Als unie van twee reguliere talen is $P_{G'}(a,b)$ regulier. Hiermee is de stelling bewezen.∎

Wat geldt voor twee knopen geldt ook voor twee *verzamelingen* knopen. Hebben we een multigraaf G en zijn A en B verzamelingen van knopen G, dan is $P_G(A,B)$, de verzameling paden die in A beginnen en eindigen in B, een reguliere taal. Immers, ze is de eindige unie van verzamelingen $P_G(a,b)$, waarin $a \in A$ en $b \in B$.

Een belangrijk idee voor sommige toepassingen is het *substitutieprincipe:* als we in een regexp elke letter van het basisalfabet vervangen door een reguliere uitdrukking over hetzelfde alfabet of over een ander alfabet, dan krijgen we uiteraard opnieuw een

regexp. Vaak zal men verschillende exemplaren van een bepaalde letter vervangen door dezelfde regexp, maar dat hoeft niet.

Een toepassing van dit principe die belangrijk kan zijn bij graafproblemen vervangt verbindingen door etiketten. Bij geëtiketteerde grafen heeft elke verbinding een etiket. Zo'n etiket kan opgevat worden als een letter van het alfabet van mogelijke etiketten, of als een string over een bepaald alfabet, eventueel van lengte 1. Dit begrip kunnen we uitbreiden tot etiketten van paden: het etiket van een pad is de opeenvolging van de etiketten van de verbindingen. Het etiket van een pad is dus een string over het alfabet van etiketten. Volgens het substitutieprincipe is het zo, dat als we twee verzamelingen A en B van knopen van een geëtiketteerde graaf hebben, dat de verzameling van de etiketten van al de paden van A naar B een reguliere taal is.

Als we een reguliere uitdrukking hebben dan kunnen we er een contextvrije grammatica voor bedenken. Inderdaad:

- De primitieve regexps leiden tot de volgende grammatica's

Regexp	Grammatica		
$\emptyset$	$\langle \mathbf{S} \rangle$	$::=$	$\langle \mathbf{S} \rangle$
ϵ	$\langle \mathbf{S} \rangle$	$::=$	ϵ
a	$\langle \mathbf{S} \rangle$	$::=$	a

  De eerste mogelijkheid lijkt bizar, maar de grammatica heeft geen eindige derivatie en definieert dus inderdaad de lege taal.

- Zijn R en S twee regexps en hebben de bijbehorende contextvrije grammatica's de startsymbolen $\langle \mathbf{R} \rangle$ en $\langle \mathbf{S} \rangle$. Dan kunnen we grammatica's voor $(R|S)$ en RS construeren door de grammatica's voor R en S onder mekaar te schrijven en ze te laten vooraf te gaan door de regel $\langle \mathbf{T} \rangle ::= \langle \mathbf{R} \rangle | \langle \mathbf{S} \rangle$ dan wel $\langle \mathbf{T} \rangle ::= \langle \mathbf{R} \rangle \langle \mathbf{S} \rangle$. We veronderstellen uiteraard dat de lijsten van niet-terminalen niet overlappen. Een grammatica voor $R*$ wordt gegeven door de grammatica voor R te laten vooraf gaan door $\langle \mathbf{T} \rangle ::= \langle \mathbf{R} \rangle \langle \mathbf{T} \rangle | \epsilon$.

  Alle reguliere talen zijn dus contextvrij. Later in de cursus zullen we een voorbeeld zien van een contextvrije taal die niet regulier is.

11.2 VARIABELE TEKST

11.2.1 Een eenvoudige methode

De meest voor de hand liggende methode test of P vanaf positie j in T voorkomt door de overeenkomstige karakters van beide te vergelijken ($P[i]$ met $T[j+i]$, voor

$0 < i \leq p$), en te stoppen zodra er een verschil gevonden wordt. (En natuurlijk ook op het einde van P.) In beide gevallen herneemt men deze test bij de volgende tekstpositie $j + 1$.

Als de strings 'random' zijn, en het alfabet niet te klein is, dan zal $P[0]$ vaak verschillen van $T[j]$, zodat de test op veel beginposities j reeds na één karaktervergelijking stopt. De *gemiddelde* uitvoeringstijd van dit algoritme is dan ook $O(t)$. (Het is eenvoudig om na te gaan dat het gemiddeld aantal karaktervergelijkingen per beginpositie van P hoogstens twee is.) In het slechtste geval echter is de uitvoeringstijd $O(tp)$, en dat doet zich voor als de eerste $O(p)$ karakters van P op $O(t)$ beginposities met de tekst overeenkomen. Bij gewone tekst is een dergelijk geval zeldzaam, maar bij bijzondere situaties kan dit best voorkomen.

11.2.2 Zoeken met de prefixfunctie: Knuth-Morris-Pratt

11.2.2.1 De prefixfunctie

Nemen we een string P en i met $i \leq p$. We zeggen dat we een string Q voor i op P kunnen leggen als $i \geq Q.size()$ en als Q overeenkomt met de even lange deelstring van P eindigend[2] voor i. De prefixfunctie $q()$ van een string P bepaalt voor elke stringpositie i, $1 \leq i \leq p$, de lengte van het langste prefix van P met lengte kleiner dan i dat we voor i kunnen leggen. Duidelijk is $q(i)$ steeds kleiner dan i, en is $q(1)$ gelijk aan nul. $q(0)$ is niet gedefinieerd; eventueel kan men bij implementatie $q(0)$ op -1 zetten om bepaalde lusstopvoorwaarden te vereenvoudigen.

De waarde van $q(i + 1)$ kan vrij gemakkelijk bepaald worden als de waarden voor alle vorige posities reeds gekend zijn. $q(i + 1)$ kan zeker niet groter zijn dan $q(i) + 1$, met gelijkheid alleen als de karakters $P[q(i)]$ en $P[i]$ overeenkomen. Komen ze niet overeen, dan trachten we een korter (maar zo lang mogelijk) prefix te verlengen dat we voor positie i kunnen leggen. Maar een prefix korter dan $q(i)$ dat we voor i kunnen leggen, kunnen we ook voor $q(i)$ leggen. Dus kan het niet langer zijn dan $q(q(i))$. Kunnen we dit met één karakter verlengen om een prefix te krijgen dat we voor $i + 1$ kunnen leggen? Alleen als de karakters $P[q(q(i))]$ en $P[i]$ overeenkomen: dan wordt $q(i + 1)$ gelijk aan $q(q(i)) + 1$. Anders moeten we het vorige herhalen, nu met het prefix van lengte $q(q(q(i)))$, enzovoorts. Deze herhaling kan eventueel doorgaan tot blijkt dat geen enkel prefix dat we voor i kunnen leggen kan verlengd worden, ook niet dat met lengte nul. In dat geval is $q(i + 1) = 0$.

Alle prefixwaarden gebruikt bij het berekenen van $q(i + 1)$ horen bij posities kleiner dan $i + 1$, en die zijn gekend ondersteld. De prefixwaarden kunnen dus voor stijgende indices berekend worden, aangezien $q(1)$ steeds nul is.

Hoe efficiënt is deze berekening? Er moeten p prefixwaarden berekend worden, en voor elke waarde is er mogelijks een herhaling die vorige prefixwaarden overloopt. Op

[2] Voor alle duidelijkheid: i wijst naar de plaats voorbij de deelstring, niet naar de laatste letter van de deelstring.

het eerste gezicht is dat een $O(p^2)$ methode, maar een zorgvuldige analyse leert dat ze slechts $\Theta(p)$ is.

Het algoritme levert een dubbele lus op. De buitenste lus wordt p keer uitgevoerd; het aantal berekeningen daarin (buiten de binnenlus) is constant. Hoe vaak wordt nu de binnenste lus uitgevoerd? Hiervoor kijken we naar q: bij elke uitvoering van de buitenste lus wordt q met hoogstens 1 verhoogd, terwijl de binnenlus q elke keer met minstens 1 vermindert. In totaal wordt de binnenlus dus hoogstens $p - q(p)$ keer uitgevoerd, terwijl $q(p) \geq 0$. Aangezien er $p - 1$ buitenste herhalingen zijn, is de totale performantie inderdaad $\Theta(p)$.

11.2.2.2 Een eenvoudige lineaire methode

Met deze prefixfunctie alleen kan reeds een eenvoudig lineair zoekalgoritme geconstrueerd worden. Daartoe stelt men een string samen bestaande uit P gevolgd door T, en gescheiden door een speciaal karakter dat in geen van beide voorkomt. Daarna bepaalt men de prefixfunctie van deze nieuwe string, wat dus een lineaire tijd $\Theta(t + p)$ vereist. Telkens wanneer de prefixwaarde van een positie i in deze string gelijk is aan p, werd P gevonden, beginnend bij index $i - p$ in T. Nu kan geen enkel prefix langer zijn dan p, omwille van dat speciaal scheidingskarakter. Met als gevolg dat we enkel de eerste p waarden van de prefixfunctie moeten bewaren, en de methode gelukkig slechts $\Theta(p)$ plaats vereist. (Naast die voor P en T, natuurlijk. Is er geen plaats nodig voor de samengestelde string?)

11.2.2.3 Het Knuth-Morris-Prattalgoritme

Ook dit lineair algoritme (Knuth, Morris, en Pratt, 1977) maakt gebruik van de prefixfunctie, maar door het aantal binnenste iteraties te beperken is het gewoonlijk sneller dan de vorige methode.

Stel dat P op een bepaalde beginpositie vergeleken wordt met T, en dat er geen overeenkomst meer is tussen $P[i]$ en $T[j]$. Als bij deze foutpositie $i = 0$ dan wordt P één positie naar rechts geschoven, en begint het vergelijken met T weer bij $P[0]$. Als echter $i > 0$ dan werd er een correct prefix van P met lengte i gevonden, dat we voor j op T kunnen leggen. Deze kennis kunnen we gebruiken: als we P verschuiven met een stap s kleiner dan i dan hebben we overlapping tussen het begin van P en het prefix van P dat we al in T gevonden hebben. Deze overlapping heeft lengte $i - s$. Nu moeten de overlappende delen overeenkomen, anders heeft vergelijking geen zin. Dit impliceert dat het prefix van P met lengte $i - s$ een prefix is van P dat we voor i kunnen leggen. De kleinste waarde van s waarvoor dit kan is $i - s = q(i)$. We kunnen P dus $i - q(i)$ plaatsen opschuiven en dan verder gaan door $T[j]$ te vergelijken met $P[q(i)]$.

Tot hier is er nauwelijks een verschil met de eenvoudige lineaire methode van hierboven. Maar we weten niet alleen dat er een prefix is van P dat we op T kunnen

leggen, we weten ook dat $P[i]$ en $T[j]$ verschillen. Als $P[q(i)] = P[i]$ dan treedt er onmiddellijk een fout op en dat weten we zonder zelfs nog eens naar $T[j]$ te kijken.

Opschuiven met een stap s is dus alleen zinvol als het prefix van P met lengte $i - s$ een prefix is van P dat we voor i kunnen leggen *en* als $P[i - s] \neq P[i]$. Gegeven i kunnen we de kleinst mogelijke waarde s berekenen die hieraan voldoet. In de praktijk berekent men een functie q' zodat $i - q'(i)$ de kleinste zinvolle s-waarde geeft. Door deze verstrenging van de voorwaarde zal de verschuiving vaak groter zijn dan bij de vorige methode. We weten natuurlijk niet of $T[j]$ gelijk is aan $P[q'(i)]$: dit wordt de eerstvolgende karaktervergelijking.

Aangezien de bijkomende vereiste voor het nieuwe prefix enkel met P te maken heeft, kunnen deze nieuwe prefixwaarden $q'(i)$ voor elke positie in P op voorhand bepaald worden. We kunnen ook $q'(p)$ zo definiëren dat $p - q'(p)$ de sprong is na het vinden van P.

Het aantal karaktervergelijkingen van dit algoritme is $\Theta(t)$. Want na elke 'verschuiving' van P wordt hoogstens één karakter van T getest dat vroeger reeds getest werd. (Als het opnieuw niet overeenkomt, wordt er doorgeschoven.) Het totaal aantal karaktervergelijkingen is dus hoogstens gelijk aan de lengte van T, plus het aantal verschuivingen. Elke verschuiving gebeurt over minstens één positie, zodat het aantal verschuivingen $O(t)$ is. Het aantal karaktervergelijkingen is dus inderdaad $\Theta(t)$. (Elk karakter van T wordt minstens eenmaal getest.) Aangezien de voorbereiding (berekenen van de prefixfunctie, van de nieuwe prefixfunctie, en van de foutfunctie) $\Theta(p)$ is, wordt de totale performantie $\Theta(t + p)$.

11.2.2.4 Varianten

Basis van de prefixmethode en van het Knuth-Morris-Prattalgoritme is dat men, op het ogenblik dat men discrepantie ontdekt heeft, al informatie heeft over een gedeelte van het patroon. Hierop zijn heel wat varianten, zoals het Boyer-Moorealgoritme. Eén van de elementen daarbij is dat men het patroon van achter naar voor onderzoekt, waardoor men informatie krijgt die verder naar achter in de tekst ligt. Als men bij Knuth-Morris-Pratt een aantal karakters van de naald heeft vergeleken met de tekst (bijvoorbeeld 5), dan kan de sprong nooit meer dan 5 bedragen, want daarmee legt men het patroon voorbij het onderzochte gebied. Begint men achteraan, dan kan de sprong wel groter zijn: als bijvoorbeeld het eerst onderzochte karakter van de tekst niet voorkomt in het patroon, dan kan men onmiddellijk voorbij dit karakter springen. Ook kan men niet alleen gebruik maken van het feit *dat* men een verschil heeft tussen patroon en tekst, maar ook van de *waarde* van het verschilkarakter in de tekst. Dit laatste is de basis van het Boyer-Moorealgoritme.

Men kan ook proberen zo snel mogelijk een fout te vinden. Als het patroon bijvoorbeeld letters bevat die nauwelijks voorkomen in de tekst, dan gaat men eerst deze letters controleren.

Dergelijke varianten zijn pas echt interessant bij specifieke toepassingen, waarbij de eigenschappen van het probleem goed aansluiten bij sommige ideeën en slecht bij andere.

11.2.3 Het Boyer-Moorealgoritme

Deze methode (Boyer en Moore, 1977) heeft nog het meest gemeen met de eenvoudige methode, omdat ze tekst T en patroon P op verschillende beginposities in de tekst karakter per karakter vergelijkt. Enkele belangrijke verschillen geven haar echter een merkelijk betere performantie, zodat ze voor vele toepassingen dan ook de voorkeur geniet.

Een eerste opvallend verschil is dat Boyer-Moore het patroon *van achter naar voor* overloopt bij het vergelijken met de tekst. Als we bij die vergelijking een fout vinden, en dan P één positie naar rechts ten opzichte van T zouden verschuiven, dan zou dat in essentie neerkomen op de eenvoudige methode, met in het slechtste geval een performantie van $O(nm)$.

Maar Boyer-Moore maakt gebruik van twee heuristieken, die grotere verschuivingen mogelijk maken:

(1) De eerste heuristiek, die van het *verkeerde karakter*, is de meest krachtige, die het algoritme dikwijls een sublineaire performantie geeft, beter dus dan $O(t + p)$. In het slechtste geval echter blijft nog altijd een performantie van $O(nm)$ mogelijk.
(2) Het is de tweede heuristiek, die van het *juiste suffix*, die voor een lineaire performantie van $O(t + p)$ zorgt, ook in het slechtste geval.

11.2.3.1 De heuristiek van het verkeerde karakter

Stel eens dat we patroon en tekst op een bepaalde beginpositie in de tekst vergelijken. We beginnen dan achteraan in P, en onderstel dat we reeds een fout vinden bij de eerste vergelijking. Noem het tekstkarakter op die plaats f (het verkeerde karakter). P naar rechts schuiven ten opzichte van T heeft slechts zin als we tegenover f in T hetzelfde karakter in P kunnen positioneren. Zoniet krijgen we daar zeker een fout.

Stel nu dat we de *meest rechtse* positie in P kennen van elk karakter in het gebruikte alfabet. (Deze informatie kunnen we vooraf bepalen en in een tabel opslaan.) Het volstaat dan om die positie voor f op te zoeken, en P zover naar rechts te schuiven dat die positie tegenover f in T terechtkomt. En als f niet in P voorkomt dan kunnen we P helemaal voorbij f in T schuiven, een verschuiving over p posities. Na de verplaatsing van P begint het vergelijken opnieuw achteraan. (Bemerk dat we de meest rechtse plaats van f in P moeten nemen, zoniet zouden we een correcte beginpositie voor P kunnen overslaan.) We zien dus dat er verschuivingen over afstanden behoorlijk groter dan één mogelijk zijn, zodat veel tekstkarakters niet eens getest worden. (Als

het verkeerde tekstkarakter nooit in P voorkomt, en voor niet al te kleine alfabetten is dat best mogelijk, dan worden er in totaal slechts $\lfloor t/p \rfloor$ tekstkarakters getest!)

Natuurlijk treedt de eerste fout niet altijd achteraan het patroon op, maar bijvoorbeeld bij patroonpositie i ($0 \leq i < p$). Stel dat j de meest rechtse positie van het verkeerde tekstkarakter f in P is (j is -1 als f er niet in voorkomt). We schuiven dan P naar rechts over $i - j$ posities. Maar wat als $i - j$ negatief is? Want het is best mogelijk dat de meest rechtse f in P rechts van positie i staat. (Kan i gelijk zijn aan j?) Voor de eenvoud houdt Boyer-Moore dan geen rekening met het verkeerde karakter. Net als de eenvoudige methode zou men P één plaats naar rechts kunnen opschuiven, maar Boyer-Moore gebruikt ook nog een tweede heuristiek, die steeds een verschuiving van minstens één oplevert.

Het voorbereidend werk voor de eerste heuristiek vereist dus een tabel van gehele getallen met grootte d, die in $\Theta(p + d)$ kan ingevuld worden.

Er bestaan verschillende varianten van deze eerste heuristiek. We overlopen de belangrijkste:

- *Uitgebreide heuristiek van het verkeerde karakter.* Om een negatieve verschuiving te vermijden gebruikt deze heuristiek de meest rechtse positie j in P van het verkeerde tekstkarakter f, *links* van de patroonpositie i waar de fout gevonden werd. De verschuiving $i - j$ is dan natuurlijk steeds positief. Het nadeel is dat we nu j niet alleen voor elk mogelijk karakter f moeten bepalen, maar ook nog voor elke mogelijke patroonpositie i. Om die snel te kunnen opzoeken kunnen we een tweedimensionale tabel gebruiken, met d rijen en p kolommen. Hoewel deze extra plaats zelden een probleem vormt, kan dat wel zo zijn voor de tijd nodig om ze in te vullen (ten opzichte van de zoektijd).

  Een compromis tussen tijd en plaats houdt voor elke f een dalend gerangschikte lijst bij met alle posities waar f voorkomt in P. Door P eenmaal van links naar rechts te doorlopen kunnen we die opstellen in $\Theta(p+d)$. Ook de totale gebruikte plaats is nu $\Theta(p)$. Maar in plaats van j in $O(1)$ te vinden op rij f en kolom i van de tweedimensionale tabel, moeten we nu de lijst voor f doorzoeken. Het is in het algemeen moeilijk uit te maken of deze extra zoektijd opweegt tegen de mogelijks grotere verschuivingen.

- *Variant van Horspool.* Een betere oplossing gebruikt dezelfde tabel als de originele methode (Horspool, 1980). Meer bepaald, voor elk karakter van het alfabet bevat de tabel de meest rechtse positie j van dat karakter in P, links van positie $p - 1$. Als het karakter daar niet voorkomt is $j = -1$. Wanneer er bij het vergelijken een fout optreedt bij patroonpositie i en tekstpositie k, dan moet P opgeschoven worden. Tegenover het tekstkarakter $T[k+p-1]$ (dat met $P[p-1]$ vergeleken werd) mag enkel een nieuw patroonkarakter terechtkomen dat gelijk is aan dat tekstkarakter, anders is er meteen weer een fout. Nu ligt dat patroonkarakter links van $P[p - 1]$ (als het bestaat), en zijn positie j vinden we in de tabel met als index tekstkarakter $T[k + p - 1]$. De verschuiving van P is dan

$p-1-j$, en die is steeds positief! (Bemerk dat dit tekstkarakter niet noodzakelijk verkeerd is.)

Deze verschuiving is bovendien onafhankelijk van de foutieve patroonpositie i, en meestal groter dan bij de oorspronkelijke versie. Ook als het patroon in de tekst voorkomt (er is dan geen verkeerd karakter), kan de verschuiving toch groter zijn dan één.

Omdat de eerste heuristiek Boyer-Moore zo snel maakt, en omdat de tweede nu niet meer nodig is om een positieve verschuiving te garanderen, gebruikt de *Boyer-Moore-Horspool methode* enkel de eerste heuristiek in deze versie. In het slechtste geval is deze methode dan $O(pt)$, maar voor de eenvoud wil men dit risico soms lopen.

- *Variant van Sunday.* Een kleine wijziging aan de vorige oplossing maakt die nog wat interessanter (Sunday, 1990). Deze variant gebruikt bijna dezelfde tabel als de originele methode. Maar als er een fout optreedt tussen patroonpositie i en tekstpositie k, dan zorgt men ervoor dat bij verschuiving het gepaste patroonkarakter tegenover tekstkarakter $T[k+p]$ terechtkomt (dat net *voorbij* $P[p-1]$ viel op de vorige beginplaats van P, en dus nog niet getest werd). De verschuiving is hier uiteraard ook steeds positief, en meestal zelfs nog iets groter dan hiervoor.

  Net als bij de vorige variant is de verschuiving onafhankelijk van de foutieve patroonpositie i. Dat heeft als belangrijk gevolg dat de volgorde waarin de karakters van P met deze van T vergeleken worden geen rol meer speelt. Voor deze versies van de eerste heuristiek is het dus niet meer noodzakelijk dat P van achter naar voor doorlopen wordt. De beste volgorde is deze die zo snel mogelijk fouten tussen P en T ontdekt. Stel dat men de kans kent waarmee elk patroonkarakter in de tekst voorkomt, dan kan men de karakters van P testen volgens stijgende kansen (van karakters met gelijke kansen neemt men zoals bij Boyer-Moore het meest rechtse).

11.2.3.2 De heuristiek van het juiste suffix

In wat volgt bespreken we de originele Boyer-Mooremethode, die onder meer P met T vergelijkt van achter naar voor.

De eerste heuristiek werkt doorgaans zeer goed, en is grotendeels verantwoordelijk voor de snelheid van Boyer-Moore. Maar er zijn gevallen mogelijk waarbij het verkeerde karakter vaak, zo niet altijd, aan de rechterkant van de foutpositie i voorkomt, zodat er telkens over slechts één positie kan verschoven worden. Dit gebeurt bijvoorbeeld bij kleine alfabetten en met een tekst die veel nagenoeg gelijke deelstrings bevat. (Een typisch voorbeeld is DNA met vier karakters in het alfabet, of zelfs proteïnen met een alfabet van twintig karakters, waarbij sterk gelijkende (maar niet gelijke) deelketens voorkomen.) Deze gevallen zijn trouwens niet alleen ongunstig voor de originele versie van de eerste heuristiek, maar ook voor zijn varianten. Wanneer nagenoeg alle beginposities van P in T moeten getest worden, en er op elke beginpositie veel testen

nodig zijn vooraleer er een fout gevonden wordt, dan zou Boyer-Moore met enkel de eerste heuristiek in het slechtste geval een performantie van $O(pt)$ hebben.

Daarom gebruikt men een tweede heuristiek. Als we immers bij positie i in P een verkeerd karakter f in T vinden, dan weten we eigenlijk meer dan hetgeen we in de eerste heuristiek gebruiken. We hebben namelijk een *suffix* van P in T gevonden, met lengte $p-i-1$ (vandaar de naam, 'the good suffix heuristic'.) En als we P verschuiven, dan moet het gedeelte dat tegenover dit juiste suffix in T terechtkomt ermee overeenkomen, zoniet heeft de verschuiving geen zin. De tweede heuristiek geeft ons dus een tweede mogelijke verschuiving naast die van de eerste heuristiek, en Boyer-Moore neemt de *grootste* van de twee. We zullen zien dat de tweede verschuiving steeds positief is, zodat er met een eventueel negatieve eerste verschuiving geen rekening wordt gehouden.

We moeten dus te weten komen of dit juiste suffix s nog ergens in P voorkomt, en waar. Als het bovendien meermaals voorkomt, dan moeten we het meest rechtse nemen, zoniet zouden we een correcte beginpositie voor P kunnen overslaan. (Bemerk dat de suffixen op twee of meer van die plaatsen elkaar kunnen overlappen.)

Als i de foutpositie in P is, dan begint het juiste suffix s, met lengte $p-i-1$, op positie $i+1$. Gevraagd de meest rechtse positie k in P ($k \leq i$), waar een deelstring s' gelijk aan s begint. Of met andere woorden, waar een suffix van P met lengte $p-i-1$ begint. Bemerk dat we hier een overeenkomst zoeken tussen twee deelstrings van P (al ligt s ook in T). Deze informatie kan dus in een voorbereidende fase gevonden worden, waar T niet aan te pas komt.

Stel eens dat we, analoog met de prefixfunctie, voor elke index j in P de lengte $s(j)$ van het grootste (echte) *suffix* van P kennen, dat daar *begint*. Als $s(j)$ kleiner is dan de lengte $p-i-1$ van het juiste suffix, dan komt j zeker niet in aanmerking voor de gezochte positie k. Maar ook niet als $s(j)$ groter is dan $p-i-1$, want het kortere juiste suffix komt dan achteraan dit langere suffix voor, en is dus rechts van j in P terug te vinden. Kortom, enkel de indices j waar $s(j)$ gelijk is aan $p-i-1$ zijn kandidaten voor k. En aangezien we de meest rechtse positie nodig hebben, moet k de grootste van die j-waarden zijn. De gezochte tweede verschuiving voor foutpositie i wordt dan $i+1-k$. Deze verschuivingen voor de p mogelijke foutposities i worden in een tabel v opgeslagen, zodat men ze even snel kan vinden als de verschuivingen voor de eerste heuristiek.

De suffixfunctie bepalen gebeurt op analoge manier als de prefixfunctie, en is dus ook $\Theta(p)$. Wanneer we echter voor elke verschuiving $v[i]$ de grootste j in de suffixtabel van P moeten zoeken, waarvoor $s(j) = p-i-1$, vereist het opstellen van tabel v een tijd van $O(p^2)$. We zullen zien dat het beter kan.

Maar eerst moeten we nog enkele speciale gevallen afhandelen. Want het juiste suffix komt niet noodzakelijk elders in het patroon voor, en soms is er zelfs geen juiste suffix:

- *Het patroon P werd gevonden.* De 'foutieve' patroonpositie i is dan -1. Het juiste suffix is nu P zelf, maar dat komt uiteraard slechts eenmaal in P voor.

Aangezien er geen verkeerd karakter is, kan de (originele) eerste heuristiek geen verschuiving geven (tenzij een veilige één). Toch mogen we P niet zomaar over p posities verschuiven, want een nieuwe P in T kan de vorige gedeeltelijk overlappen. En de grootst mogelijke overlapping komt overeen met de eerstvolgende positie waarop P zou kunnen voorkomen. De maximale overlapping is het langst mogelijke suffix van P, korter dan p, dat overeenkomt met P beginnend bij positie 0. De lengte van dit suffix is natuurlijk $s(0)$. De overeenkomstige verschuiving is dan $p - s(0)$, die we voor de uniformiteit in een extra tabelelement $v[-1]$ opslaan.

Bemerk dat deze verschuiving steeds positief is (want $s(0)$ is kleiner dan p), en dat ze gelijk is aan p als er geen overlapping is.

- *Er is geen juist suffix.* Als de fout optreedt bij patroonpositie $p-1$, dan is er geen juist suffix, zodat we hiervoor geen verschuiving kunnen bepalen. Maar de verschuiving van de (originele) eerste heuristiek is nu zeker positief: het verkeerde tekstkarakter f kan onmogelijk rechts van positie $p-1$ in P liggen. De tweede verschuiving $v[p-1]$ kunnen we dan de minimale waarde van één geven.

- *Het juiste suffix komt niet meer in P voor.* Er werd dan geen index j gevonden waarvoor $s(j) = p - i - 1$. Het is duidelijk dat we P dan over meer dan i posities naar rechts mogen verschuiven. Maar niet noodzakelijk over p posities, want het is nog altijd mogelijk dat er een suffix van het juiste suffix overeenkomt met een prefix van P. En als er meerdere dergelijke suffixen zijn, dan moeten we het grootste nemen. Want dit geeft de kleinste verschuiving, en zo lopen we niet het risico om een correcte beginplaats voor P over te slaan. De lengte van het gezochte suffix is dus weer $s(0)$, zodat de overeenkomstige verschuiving $v[i]$ ($0 < i < p$) opnieuw $p - s(0)$ wordt. Net als hierboven is die steeds positief, en gelijk aan p als er geen suffix en dus geen overlapping bestaat.

Het opstellen van de tabel v leek traag omdat men voor elke foutpositie i de indices j waarvoor $s(j) = p - i - 1$ lineair moest zoeken in de suffixtabel van P, om dan de grootste index te gebruiken voor de verschuiving $v[i]$.

We kunnen echter ook de suffixtabel eenmaal overlopen, voor elke index j de foutpositie $i = p - s(j) - 1$ en de verschuiving $v[i] = i + 1 - j$ bepalen, en de kleinste verschuiving voor elke i bijhouden. Die kleinste verschuiving hoort bij de grootste j-waarde. Als we dus j van klein naar groot aflopen, zullen we die kleinste waarde het laatste vinden: we moeten dus niet controleren of we een kleinere waarde vinden, we overschrijven gewoon wat er staat.

Omdat we een minimum moeten bepalen voor deze foutposities, moeten we hun verschuiving initialiseren. Maar ook de verschuiving voor foutposities waarvoor geen juist suffix te vinden is moet ingevuld worden. We lossen dat op door heel de tabel v te initialiseren met $p - s(0)$:

- Voor de foutposities $0 \leq i < p$ waarbij het juiste suffix wel nog voorkomt, wordt de waarde overschreven.

- Voor de foutposities $0 \leq i < p$ waarbij het juiste suffix niet meer voorkomt, is $p - s(0)$ de vereiste verschuiving (zie hoger). Bij het overlopen van de suffixtabel worden deze waarden niet meer gewijzigd.
- Voor i gelijk aan $p - 1$, waarbij er dus geen juist suffix bestaat, moet $v[p - 1]$ de waarde één krijgen. Dat zal gebeuren bij het overlopen van de suffixtabel, omdat reeds met $j = p - 1$ de foutpositie $p - s(j) - 1 = p - 1$ overeenkomt, met verschuiving $p + 1 - p$. Omdat de initiële waarde $p - s(0) \geq 1$ zal dit nieuw minimum ingevuld worden, en kleinere j-waarden kunnen het niet meer wijzigen.

Na initialisatie van de tabel met verschuivingen, overloopt deze verbeterde versie slechts eenmaal de suffixtabel, en per index j is het werk $O(1)$: ze is dus $\Theta(p)$. Het voorbereidend werk voor de tweede heuristiek wordt dus ook $\Theta(p)$, en vereist $\Theta(p)$ plaats.

De originele methode van Boyer-Moore maakte gebruik van deze versie van de tweede heuristiek. Later bleek echter dat een lineaire performantie in het slechtste geval enkel verzekerd is als het gevonden juiste suffix s' nog aan een bijkomende voorwaarde voldoet. (Men spreekt dan van de '*strong* good suffix heuristic'.) Het karakter dat links van s' ligt in P (als er een is) moet namelijk *verschillend* zijn van het karakter $P[i]$ dat links van s ligt in P. Want anders zal er, na verschuiving van P, op die plaats zeker geen overeenkomst met T gevonden worden (i was immers de plaats van de fout). De verschuivingen kunnen hierdoor groter worden. (Bemerk de analogie met de methode van Knuth-Morris-Pratt.) De aanpassingen aan het algoritme zijn gering.

Wanneer P *niet* in T voorkomt, kan men aantonen dat het algoritme nu zeker $O(t + p)$ is. Wanneer P echter wel voorkomt, blijft het slechtste geval nog steeds $O(tp)$. (Bijvoorbeeld voor het extreem geval waarbij alle karakters van zowel P als T gelijk zijn.) Een kleine aanpassing kan ook dit verhelpen. We hebben gezien dat wanneer P voorkomt in T, de verschuiving $p - s(0)$ moet zijn, om het grootst mogelijke overlappende suffix $s(0)$ van P niet te missen. Daarna begint de vergelijking van P en T weer achteraan in P. Wanneer deze vergelijking echter bij index $s(0)$ van P is gekomen, kan ze stoppen, want het suffix werd reeds gevonden. Daarmee is het algoritme nu in alle gevallen lineair. (In het extreem geval van hierboven bijvoorbeeld, is $s(0)$ gelijk aan $p - 1$, zodat P over slechts één positie verschoven wordt. Op de nieuwe positie van P moet enkel het laatste karakter $P[p - 1]$ met T vergeleken worden, want we weten dat de resterende $p - 1$ karakters overeenkomen. Er gebeurt dus slechts één karaktertest per positie in T.)

11.2.4 Onzekere algoritmen

In de volgende subparagraaf bespreken we het Karp-Rabinalgoritme. Voor we hieraan beginnen is het passend een paar woorden te zeggen over onzekere algoritmen.

Een aantal algoritmen leveren een onzeker resultaat op. We hebben het hier niet over algoritmen die een goede benadering van het juiste antwoord geven, zoals een nume-

riek algoritme met een afrondingsfout of een routeplanner die een goede, maar niet de best mogelijke, route berekent. Het gaat hier over algoritmen die met een zekere waarschijnlijkheid een geheel foutief resultaat opleveren. Een dergelijk algoritme wordt een *Monte Carloalgoritme* genoemd. Op het eerste gezicht is een dergelijk algoritme geheel nutteloos. Er zijn echter twee soorten toepassingen voor dit soort algoritmen:

(1) (Het voorbeeld stamt uit de tijd van de dollekoeienepidemie.) Elk rund dat voor consumptie geslacht wordt moet getest worden op BSE, ook al zijn er weinig koeien (veel minder dan 1‰) besmet. Je hebt een test die € 200 kost en altijd een juist resultaat geeft. Je hebt ook een Monte Carlotest die € 10 kost, maar die in 1% van de gevallen een fout antwoord geeft. Die fout is echter niet symmetrisch: de test geeft nooit een vals negatief antwoord (de test zegt dat het rund niet besmet is, terwijl dat wel zo is) maar alleen valse positieve antwoorden (de test zegt dat het rund BSE heeft terwijl dat niet zo is).

(2) Je leven hangt af van de juiste uitkomst van een algoritme. Je hebt de keuze tussen

- Een klassiek, deterministisch algoritme, dat 2 milliseconden nodig heeft.
- Een Monte Carloalgoritme dat 1 keer op 2^{100} een foutief antwoord geeft, maar de berekening in 1 milliseconde uitvoert.

Het algoritme zal draaien op een computer waarbij gemiddeld één keer op duizend jaar een geheugenfout optreedt. Als zo'n fout optreedt is er één kans op een miljoen dat dit de uitkomst van een berekening beïnvloedt. Welk algoritme kies je?

Bij het tweede geval gaat het om een algoritme dat een zeer kleine foutkans heeft. Bij dergelijke algoritmes is het vaak zo dat men de foutkans willekeurig klein kan maken[3].

In het eerste geval daarentegen hebben we een goedkope test die in een aantal gevallen zekerheid oplevert. Men spreekt van een *waar-scheve* (Eng.: *true-biased*) of *onwaar-scheve* (Eng.: *false-biased*) test. Een voorbeeld daarvan is de zogenaamde *Bloom*filter. We veronderstellen dat we een klasse van objecten hebben met een hashfunctie en dat we een verzameling van zulke objecten willen bijhouden. Een Bloomfilter houdt nu de logische (bitsgewijze) *of* bij van de hashwaarden van alle elementen. Willen we weten of een element tot de verzameling behoort dan berekenen we de hashwaarde en nemen de logisch *en* met de waarde van de Bloomfilter. Als het resultaat verschilt van de hashwaarde zelf dan behoort het element zeker niet tot de verzameling, anders weten we het niet, zodat de test waar-scheef is. Een Bloomfilter is nuttig als het opzoeken in de verzameling zelf computationeel duur is, bijvoorbeeld omdat de verzameling op harde schijf is opgeslagen omdat ze zo groot is. Maar ook als er een klassieke, vrij efficiënte representatie is van de verzameling, bijvoorbeeld in een rood-zwarte boom, is een Bloomfilter efficiënt als de hashwaarde al gekend is. Opzoeken in een rood-zwarte boom is zelf $O(\lg n)$ en de Bloomfilter sluit met één enkele operatie veel zoekwerk uit.

[3] Soms reserveert men de naam Monte Carloalgoritme voor algoritmes met een parameter zodat men het algoritme herhaalde malen kan uitvoeren met *onafhankelijke* foutkans. Bij deze algoritmes kan men de fout verkleinen door het algoritme herhaaldelijk uit te voeren met verschillende parameterwaarden.

Merk op dat ook bij een Bloomfilter men de kans dat dat hij een verkeerd antwoord geeft willekeurig klein kan maken door het aantal bits in de hashfunctie te vergroten.

11.2.5 Het Karp-Rabinalgoritme

Deze methode (Karp en Rabin, 1987) herleidt het vergelijken van strings tot het vergelijken van *getallen*. Dit heeft alleen maar zin als getallen sneller kunnen vergeleken worden dan strings, en als die snelheidswinst niet verloren gaat bij het berekenen van al die getallen.

Aan elke mogelijke string met die even lang is als P kent men een uniek geheel getal toe. En in plaats van P en de even lange deeltekst op een bepaalde positie te vergelijken, vergelijkt men de overeenkomstige getallen. Gelijke strings betekent gelijke getallen, en omgekeerd, als de getallen gelijk zijn dan kunnen we besluiten dat ook de strings gelijk zijn.

Nu zijn er d^p verschillende strings met lengte p, zodat die getallen heel groot kunnen worden. Om dergelijke getallen voor te stellen moet men tabellen gebruiken, en die vergelijken is niet efficiënt, want dat vereist meer dan $O(1)$ primitieve operaties. Alleen al het vergelijken op elke beginpositie zou de uitvoeringstijd van deze methode in *alle* gevallen reeds groter maken dan de gemiddelde tijd van de eenvoudige methode.

Daarom beperkt men zich tot getallen die wel efficiënt kunnen vergeleken worden: deze die men in één processorwoord (met lengte w bits) kan voorstellen. Aangezien er nu (meestal) veel minder getallen dan strings met lengte p zijn ($2^w \ll d^p$)[4], zullen meerdere strings met hetzelfde getal moeten overeenkomen. Dat is natuurlijk hetzelfde principe als bij *hashing*.

Bij een hashtabel waren er conflicten, en ook hier vinden we een verwant verschijnsel: gelijke strings betekent nog altijd gelijke getallen, maar als de getallen gelijk zijn, dan is het niet meer zeker dat de strings gelijk zijn. Om een snel algoritme te bekomen, laat men dus toe dat het zich af en toe vergist.

Hoe definiëren we nu deze getallen? Om minstens een vergelijkbare (gemiddelde) performantie als die van de eenvoudige methode te halen, moet het getal voor elk van de $O(t)$ deelstrings van de tekst in $O(1)$ kunnen berekend worden. Een hashwaarde voor een string met lengte p berekenen in $O(1)$ is niet realistisch. We kunnen ons doel enkel bereiken als we de hashwaarde voor de deelstring op beginpositie $j+1$ kunnen afleiden uit die voor de deelstring op positie j, en dat in $O(1)$. Het berekenen van de eerste hashwaarde, voor positie één, mag dan gerust langer duren.

Er zijn verschillende definities voor de hashwaarde mogelijk die aan deze vereiste voldoen. We bespreken de meest gebruikelijke. Omdat een stringelement d verschillende waarden kan aannemen, beschouwt men een string als een (geheel) getal in een d-tallig stelsel. Elk stringelement wordt daarbij voorgesteld door een 'cijfer' gelegen tussen 0

[4] Als $d^p < 2^w$ dan kunnen we gewoon de string zelf in een processorwoord steken.

en $d-1$ (inbegrepen). Het (groot) getal dat P voorstelt wordt dan:

$$H(P) = \sum_{i=0}^{p-1} P[i]d^{p-i-1}$$

Deze berekening vereist slechts p optellingen en evenveel vermenigvuldigingen (als ze gebeurt met de welbekende methode van Horner).

Om nu een beperkte hashwaarde te bekomen, neemt men de rest bij deling door een nader te bepalen getal r ($0 < r \leq 2^w$). Deze hashwaarde noemt men in deze context ook wel de 'fingerprint' van P:

$$H_r(P) = H(P) \bmod r$$

Eerst het groot getal $H(P)$ berekenen, en dan pas de rest bepalen is echter niet efficiënt. Gelukkig blijkt de rest bij deling door r van een som van twee getallen gelijk aan de rest van de som van hun resten:

$$(a+b) \bmod r = (a \bmod r + b \bmod r) \bmod r$$

Deze eigenschap geldt ook voor een verschil en een product. Dat betekent dat we bij elke berekening die gebruik maakt van deze bewerkingen, de rest van het eindresultaat ook kunnen vinden door met de resten van willekeurige tussenresultaten verder te werken. Tussenresultaten die te groot zouden worden kunnen we dus steeds kleiner dan r maken.

De berekening van fingerprints (Horner) maakt enkel gebruik van optellen (aftrekken) en vermenigvuldigen, en de initiële getallen zijn kleiner dan d. We kunnen er dus voor zorgen dat alle tussenresultaten net als r in een processorwoord passen, zodat elke deelbewerking slechts $O(1)$ vereist. De fingerprint voor P berekenen wordt dan $\Theta(p)$.

Wanneer echter het berekenen van de fingerprint van elke deelstring van T ook $\Theta(p)$ zou vergen, dan hadden we een $\Theta(tp)$ algoritme. Er is echter een eenvoudig verband tussen het (groot) getal voor de deelstring T_{j+1} bij beginpositie $j+1$ en dat voor deelstring T_j bij beginpositie j:

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

De fingerprint voor T_{j+1} wordt dan:

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

Om het rechterlid te berekenen, mogen we weer resten van tussenresultaten gebruiken. De waarde van $d^{p-1} \bmod r$ moet slechts eenmaal berekend worden (ook met tussentijdse resten indien nodig). Kortom, de nieuwe fingerprint kan in $O(1)$ uit de vorige

bepaald worden[5].

Dan blijft er enkel nog de eenmalige berekening van $H_r(T_0)$, maar deze is volledig analoog aan die van $H_r(P)$, en vereist ook $\Theta(p)$:

$$H_r(T_0) = \left(\sum_{i=0}^{p-1} T[i] d^{p-i-1} \right) \bmod r$$

Het berekenen van $H_r(P)$, $H_r(T_0)$, en $d^{p-1} \bmod r$ vereist dus $\Theta(p)$ operaties. De berekening van alle andere fingerprints $H_r(T_j)$ ($0 < j \leq t - p$) vergt in totaal $\Theta(t)$ operaties, zodat dit een $\Theta(t + p)$ algoritme is.

Daarbij hielden we geen rekening met eventuele *string*vergelijkingen ter controle. Want om zeker te spelen moet dit algoritme bij gelijke fingerprints nog altijd de strings P en T_j zelf vergelijken. Strings met lengte p vergelijken vereist natuurlijk $O(p)$ operaties, zodat we in het slechtste geval, als de fingerprints op elke positie gelijk zijn, opnieuw een performantie van $O(tp)$ krijgen.

Hoe kiezen we nu r? Niet te groot zodat alle getallen (fingerprints en tussenresultaten) in een processorwoord passen, en zodanig dat de kans op vergissingen klein is. Men kan een vaste r gebruiken, of r random kiezen:

- *Vaste r.* Kies voor r een zo groot mogelijk *priemgetal*, zodat $rd \leq 2^w$. Net als bij hashing nemen we een priemgetal om te vermijden dat verwante deelstrings dezelfde fingerprints zouden opleveren. We nemen r zo groot mogelijk, om zoveel mogelijk verschillende fingerprints te bekomen (we zijn hier niet beperkt door de grootte van een hashtabel). De laatste voorwaarde tenslotte zorgt ervoor dat er geen 'overflow' optreedt bij het vermenigvuldigen van tussenresultaten met d, en dat het aantal modulobewerkingen beperkt blijft. (Ga eens na.)

  Het verband tussen de fingerprints op twee opeenvolgende beginposities wordt dan:

  $$H_r(T_{j+1}) = \Big(\big((H_r(T_j) + r(d-1) - T[j](d^{p-1} \bmod r)) \bmod r \big) d + T[j+p] \Big) \bmod r$$

  (Een term $r(d-1)$ werd zonder gevolg voor het eindresultaat toegevoegd om een negatief tussenresultaat te vermijden[6].)

[5] Het laatst opgetelde 'cijfer' heeft slechts een beperkte invloed op het resultaat. Ideaal heeft elk karakter invloed op alle bits. Een mogelijke verbetering vermenigvuldigt de waarde van $H(T_0)$ nog met d, zodat bij de berekening van de overige getallen de volgorde van vermenigvuldigen en optellen wordt omgekeerd:

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

[6] In de wiskunde heeft a mod b steeds het teken van b, in programmeertalen zoals C++ echter steeds het teken van a.

Er zijn ook implementaties die de relatief trage modulobewerkingen vermijden door de impliciete modulo $r = 2^w$-operaties van de processor te gebruiken. Omdat r dan geen priem meer is, moet men de basis (radix) van het talstelsel zorgvuldig kiezen. Die radix mag zeker geen macht van twee zijn (zoals 128 of 256 bij tekststrings), en is best relatief priem met r. Goede waarden bij tekststrings zijn 127, 131 of 257.

- *Random* r. Een vaste r kan natuurlijk nadelig zijn in bepaalde situaties, zodat er heel veel vergissingen gebeuren ('demon seed'). De veiligste implementaties kiezen een 'random' priem r uit een bepaald bereik, zodat de kans op vergissingen niet langer bepaald wordt door de waarschijnlijkheidsverdelingen van het patroon en/of de tekst. (Een oplossing verwant aan universele hashing bij hashtabellen.)

  Deze foutkans kan willekeurig klein gemaakt worden door een groot bereik te nemen. Daarbij gebruikt men de fundamentele eigenschap dat het aantal priemgetallen kleiner dan of gelijk aan k ongeveer $k/\ln k$ bedraagt, voor grote k. Door k groot genoeg te kiezen (maar niet te groot, zodat r niet te groot wordt) zorgt men ervoor dat slechts een klein deel van al die priemgetallen een fout kan veroorzaken. De kans dat r een van die priemen is, en dus de foutkans, wordt dan klein. Zo is bijvoorbeeld voor k gelijk aan t^2 de kans op één enkele fout $O(1/t)$.

  Om hiervan een Las Vegasalgoritme te maken kan men bijvoorbeeld:

  - Overgaan op de eenvoudige methode wanneer de hoger vermelde $O(t)$ test een fout signaleert. Met de foutkans van hierboven geeft dat een *gemiddelde* uitvoeringstijd van $(1 - 1/t)(\Theta(t + p) + O(t)) + (1/t)(\Theta(t + p) + O(t) + O(tp)) = \Theta(t + p)$ voor een foutloos algoritme.
  - Herbeginnen met een nieuwe random priem r wanneer dezelfde test een fout signaleert. Met telkens een foutkans van $O(1/t)$ moeten we het algoritme (en de test) gemiddeld $O(t/(t - 1))$ maal uitvoeren (de waarschijnlijkheidsverdeling is immers geometrisch), zodat de totale *gemiddelde* uitvoeringstijd $\Theta(t + p)$ wordt. De variatie op de uitvoeringstijd is hier veel kleiner dan bij het vorige geval.

  De kans op fouten kan nog veel kleiner gemaakt worden door meerdere fingerprints tegelijk te gebruiken (met elk een eigen random gekozen r). Een fout treedt slechts op wanneer alle fingerprints *tegelijk* een fout opleveren, onafhankelijk van elkaar.

Deze interessante en relatief eenvoudige methode kan ook uitgebreid worden voor *tweedimensionale* patroonherkenning, zelfs met patronen van onregelmatige vorm, waar andere methoden het moeilijk mee hebben. Ze is dus geschikt voor bepaalde vormen van beeldverwerking. Mits een kleine aanpassing is het ook de aangewezen methode om *tegelijk* naar meerdere strings te zoeken in een tekst. (Bijvoorbeeld bij het opsporen van plagiaat.)

11.2.6 Zoeken met automaten

De automatentheorie bestudeert abstracte machines, om de grenzen van de mogelijkheden van computers vast te leggen, om de complexiteit van algoritmen te catalogeren, om formele talen te herkennen, maar ook als model voor talrijke processen (zie bijvoorbeeld [22]).

Het uitgangspunt voor de ontwikkeling van de automatentheorie was de volgende vraag: *wat voor soort berekeningen kan een eindig neuraal netwerk juist maken in een omgeving die wijzigt met de tijd?*

Deze vraag werd eerst gesteld in de context van neurale netwerken, maar automaten geven een antwoord voor algemene informatieverwerkende eenheden met een eindig geheugen. Om te beginnen moest hiervoor een abstracte modellering gevonden worden voor het begrip geheugen. Bij een klassieke computer worden bits op een of andere elektromagnetische manier opgeslagen, maar een geheugen kan ook bestaan uit mechanische onderdelen, zoals tandwielen die een bepaalde positie innemen. Elk mechanisme dat de toestand van de eenheid verandert kunnen we als geheugen beschouwen, ook als die verandering onomkeerbaar is: een vuurpijl heeft een geheugen dat opslaat of de pijl ooit is afgevuurd, want je kan hem maar één keer afvuren.

Hiervoor voert men het begrip *staat* in. Staat is, bij deterministische automaten, synoniem voor toestand. Een staat beschrijft dus de gehele toestand van de eenheid: er zijn evenveel staten als er mogelijkheden zijn. Een geheugenmodule van 32 KB heeft dus bijvoorbeeld 256^{32K} mogelijke staten. Bij zgn. niet-deterministische automaten is de interpretatie van een staat enigszins anders. Hier komen we later op terug.

De veranderende omgeving wordt gemodelleerd door de automaat te voorzien van een invoerkanaal. Tijd wordt als een discrete grootheid beschouwd en op elk ogenblik is er dan ook een zekere invoer beschikbaar. De verzameling van alle mogelijke invoerwaarden heet het *alfabet* of voluit het *invoeralfabet*. Dit kan enigszins misleidend te zijn, daar het niet hoeft te gaan om een klassiek karakteralfabet zoals ASCII. Bij het menselijk brein gaat het om de gecombineerde invoer van alle zintuigen, inclusief zenuwsignalen die een interne toestand van het menselijk lichaam registreren. Het alfabet kan dus gigantisch zijn, maar is wel eindig (er wordt bijvoorbeeld verondersteld dat zintuigen een eindige precisie hebben).

Het is nu niet meer nodig om expliciet begrippen zoals berekeningen of programma's te bekijken: al wat een automaat doet als het een karakter als invoer krijgt is veranderen van staat en eventueel uitvoer genereren. Wat de nieuwe staat is hangt enkel af van de invoer en van de oude staat. Wat de uitvoer is hangt enkel af van de nieuwe staat.

In de context van het zoeken van strings beschouwen we de hooiberg als de reeks karakters of symbolen die de automaat als invoer krijgt. In tegenstelling tot vroeger gaan we nu niet meer op zoek naar één enkele deelstring, maar hebben we een verzameling van gezochte strings. Telkens de automaat het einde van een van de gezochte strings ontmoet, geeft ze een uitvoersignaal. Merk op dat gezochte strings mekaar kunnen overlappen op alle mogelijke manieren.

Een kleine variatie is een automaat die alleen strings uit de verzameling kan herkennen: we geven een willekeurige string als invoer en op het einde zegt de automaat of de string tot de gezochte verzameling behoort. Men zegt dat de automaat de verzameling *herkent*.

Deterministische automaten. Een deterministische automaat met een beperkt (eindig) aantal toestanden (voortaan DA) is een abstract model voor een schakeling die of een programma dat zich in één van een aantal toestanden kan bevinden, in elke toestand een symbool invoert, en afhankelijk van dat symbool van toestand kan veranderen.

Formeel, bestaat een DA uit

- Een (eindige) verzameling *invoersymbolen* Σ (het invoeralfabet).
- Een (eindige) verzameling *toestanden* S.
- Een *begintoestand* s_0 behorend tot S.
- Een verzameling *eindtoestanden* F, die een deelverzameling is van S.
- Een *overgangsfunctie* $p(t, a)$ die de nieuwe toestand geeft wanneer de automaat in toestand t symbool a ontvangt. Elke toestand heeft precies één overgang voor *elk* invoersymbool. (Soms vereenvoudigt men wat door onverwachte overgangen - fouten - niet expliciet te vermelden.)

Een DA kan voorgesteld worden door een gerichte geëtiketteerde multigraaf G, de *overgangsgraaf*, met als knopen de toestanden en als verbindingen de overgangen, met als etiket het overeenkomstig invoersymbool.

Een DA start steeds in zijn begintoestand, en maakt de gepaste toestandsovergangen bij elk ingevoerd symbool. Daarmee komt een weg overeen in de overgangsgraaf, en voor elke string van invoersymbolen is er precies één weg.

Wanneer een DA zich na het invoeren van een string in een eindtoestand bevindt, zegt men dat de string *herkend* werd door de DA. De etiketten op de weg tussen knoop s_0 en de eindknoop zijn dan de opeenvolgende symbolen van deze string. De verzameling van alle strings die door een DA herkend worden is de taal herkend door die automaat.

We zien onmiddellijk dat een taal die door een DA herkend wordt regulier is. Immers, de taal is niets anders dan de verzameling van etiketten van $P_G(\{s_0\}, F)$. Dit is de helft van de stelling van Kleene, die zegt dat de verzameling van reguliere talen gelijk is aan de verzameling talen die herkenbaar zijn door een DA.

Een snelle (software)implementatie van de overgangsfunctie van een DA is een tweedimensionale overgangstabel met (bijvoorbeeld) als rijen de toestanden en als kolommen de symbolen. (Soms vindt men deze tabel nog niet snel genoeg, wordt de toestand bijgehouden in de programmateller, en de overgangsfunctie geïmplementeerd via

meerkeuze-opdrachten.) Wanneer Σ en het aantal toestanden groot zijn, kan deze tabel te veel plaats vereisen. Er bestaan vernuftige methoden om de tabel te comprimeren. De overgangen vanuit elke toestand kunnen ook opgeslagen worden in een gelinkte lijst. Dit is dan de ijle representatie van de overgangsgraaf; deze implementatie is compacter, maar uiteraard trager.

Niet-deterministische automaten. Weinig termen in de informatica zijn zo ongelukkig gekozen als de term 'niet-deterministische automaat', kortweg NA. Een NA is namelijk een deterministisch mechanisme waarin toeval (in de zin van *randomness*) geen enkele rol speelt. Het NA-formalisme is gewoon een speciale manier om een DA voor te stellen.

Eén term in de informatica die wél even ongelukkig gekozen is als 'niet-deterministische automaat' is de term 'toestand van een niet-deterministische automaat' (Engels: *NA state*). We zullen deze term dan ook niet gebruiken. Daarmee wijkt de cursus af van de algemene literatuur, maar het vereenvoudigt wel de uiteenzetting. In de plaats daarvan zullen we de term *statenbit* gebruiken.

Een NA heeft een aantal statenbits, die de waarden 'aan' of 'uit' kunnen aannemen. De staat kan dan worden aangeduid door de verzameling statenbits die aan staan. De beginstaat van de NA wordt aangeduid met een speciale statenbit, de *beginbit*. Deze staat aan in het begin terwijl alle andere bits uit staan. De eindstaten van de NA worden aangeduid door de *eindbits*. De NA is in een eindstaat als een of meer eindbits aanstaan; hiervoor heeft de waarde van de andere bits geen belang.

De overgang van een staat naar de volgende werkt bit per bit. Een statenbit die aan staat reageert op een invoersymbool door een signaal naar nul of meer statenbits te sturen. Een statenbit die één of meer signalen binnenkrijgt zet zichzelf aan, een statenbit die geen signaal binnenkrijgt dooft uit (of blijft uit). Is i een statenbit en a een letter uit het alfabet, dan is $s(i, a)$ de verzameling statenbits die rechtstreeks een signaal van i krijgen als de inkomende letter a is.

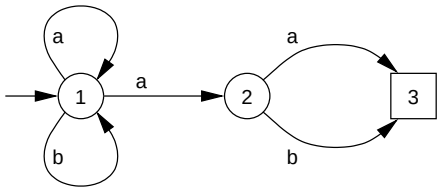
Er zijn ook nog zogenaamde ϵ-overgangen. Als er een ϵ-overgang is van statenbit i naar statenbit j en i krijgt een signaal binnen, dan stuurt i direct, zonder vertraging, een signaal naar j, zodanig dat ze beide aangezet worden. De naam ϵ-overgang komt voort uit de idee dat i een signaal stuurt als i de lege string ϵ binnenkrijgt. Eigenlijk zijn ϵ-overgangen niet nodig voor niet-determinisme. We kunnen een ϵ-overgang van i naar j vervangen door overgangen te leggen van de voorgangers van i naar j zelf. De mogelijkheden van niet-deterministische automaten met of zonder ϵ-overgangen zijn dus dezelfde. Maar deze overgangen maken het ontwerp, en vooral de berekeningen, eenvoudiger.

De overgangstabel voor een NA zal dus *verzamelingen* van statenbits moeten bevatten in plaats van enkelvoudige staten, en een extra kolom voor het 'invoersymbool' ϵ.

Vertrekkend vanuit de begintoestand, kunnen in een NA meerdere statenbits aan staan bij het inlezen van symbolen (of zelfs zonder dat er een invoersymbool aan te pas

komt). Bij elk nieuw invoersymbool moet nu de verzameling van statenbits bepaald worden die vanuit de huidige verzameling statenbits voor dat symbool kunnen bereikt worden. Met elke string van invoersymbolen kunnen er dus meerdere wegen in de overgangsgraaf overeenstemmen.

Een NA herkent dus een string als, op het einde van die string, er een of meer eindbits aan staan. In termen van de overgangsgraaf wordt een string herkend wanneer er *een* weg bestaat tussen de beginbit s_0 en een eindtoestand, met als etiket de string. Een ϵ-overgang in het pad heeft als etiket de lege string. Die 'zie' je dus niet in het etiket van het pad. Figuur 11.1 toont een niet-deterministische automaat (zonder ϵ-overgangen), met zijn overgangstabel, die alle strings herkent beschreven door de regexp $(a|b)*a\,(a|b)$.



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

Figuur 11.1. Niet-deterministische automaat voor $(a|b)*a(a|b)$.

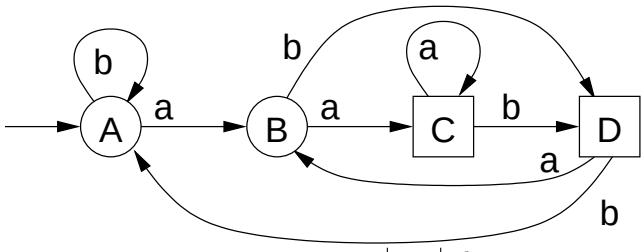
11.2.6.1 De deelverzamelingconstructie

Een NA is een alternatieve voorstelling van een DA. Voor een aantal problemen is het gemakkelijker om een NA op te stellen dan een DA. Dit is bijvoorbeeld het geval als we een reguliere uitdrukking willen omzetten in een automaat. Het is echter duidelijk dat ze niet erg geschikt is om een efficiënte implementatie van een zoek- of herkenningsoperatie te geven. Bij het DA-formalisme moeten we, bij elke binnenkomende letter, de nieuwe staat opzoeken in een tabel. Dit is in principe een efficiënte operatie, tenminste als deze tabel niet onhandelbaar groot is. Bij een NA moeten we bij elke binnenkomende letter alle statenbits die aanstaan afgaan, en de daarbijhorende bits die een signaal krijgen aanduiden. Als de NA k bits bevat, dan is dit in het slechtste geval een $O(k^2)$ operatie.

Vandaar dat we, als we een NA hebben, deze meestal gaan omzetten naar het DA-formalisme. Elke staat van de NA komt overeen met een verzameling statenbits die aanstaan en is dus impliciet gedefinieerd. Het omzetten naar het DA-formalisme met expliciete staten staat bekend als de *deelverzamelingconstructie*[7] ('subset construc-

[7] Elke staat komt overeen met een deelverzameling statenbits. Als je de onderstaande constructie implementeert moet je je afvragen hoe je zo'n deelverzameling representeert. Zoals altijd moet je daarvoor kijken welke operaties je gebruikt, zodat deze efficiënt kunnen uitgevoerd worden.

tion'). Figuur 11.2 toont de DA, met zijn overgangstabel, geconstrueerd uit de NA van figuur 11.1. Als er k statenbits zijn dan zijn er 2^k mogelijke deelverzamelingen,



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

Figuur 11.2. Deterministische automaat geconstrueerd uit die van figuur 11.1.

zodat het aantal toestanden van de DA in principe enorm groot kan uitvallen. In de praktijk blijkt een NA slechts een klein aantal van deze deelverzamelingen te bereiken, en is het aantal toestanden van de DA vergelijkbaar met dat van de NA (er zijn wel meer overgangen). Het zou dus zeker niet efficiënt zijn elke deelverzameling te gaan bekijken en vast te stellen welke zijn buren zijn volgens het NA-model: de meeste deelverzamelingen zijn niet bereikbaar vanuit de begintoestand en zijn dus niet belangrijk.

In de plaats daarvan gaan we de impliciet gegeven multigraaf met 2^k knopen doorlopen met een klassieke methode zoals diepte-eerst of breedte-eerst zoeken, vertrekkend van de beginstaat. Knopen die we zo niet bereiken zijn overbodig voor de DA. Op het eerste zicht kennen we de beginstaat van de NA: daarin staat de beginbit b_0 aan en alle andere bits uit. Zoals we zullen zien is dit echter niet de beginstaat van de DA als er ϵ-overgangen vanuit b_0 zijn.

Buren in de impliciete multigraaf kunnen we niet opzoeken in een burenlijst; we hebben twee hulpoperaties nodig:

- De deelverzameling van statenbits bereikbaar via ϵ-overgangen (nul of meer) vanuit een verzameling statenbits T noemt men ϵ-*sluiting* (T). (Een speciaal geval is ϵ-*sluiting* (t), wanneer T slechts één statenbit t bevat.)
- De overgangsfunctie $p(t, a)$ kunnen we uitbreiden voor een *verzameling* van statenbits T tot $p(T, a)$. Het resultaat is dan de deelverzameling van alle statenbits rechtstreeks bereikbaar vanuit een of andere toestand t uit T, voor het invoersymbool a.

Het bepalen van ϵ-*sluiting* (T) komt neer op het zoeken van alle statenbits bereikbaar

via een ϵ-overgang vanuit T. Pseudocode 11.1 gebruikt daarbij een stapel voor de statenbits waarvan de ϵ-overgangen nog moeten onderzocht worden. *

```
// Verzameling voor het resultaat, initieel ledig
Set esluiting;
// Voor statenbits met nog te bepalen overgangen
Stack DEZstack;
for (elke statenbit t uit T) {
   esluiting.voegtoe(t);
   DEZstack.push(t);
}
while (!st.ledig()) {
   Statenbit t = DEZstack.topEnPop();
   for (elke statenbit u bereikbaar uit t voor epsilon)
      if (!esluiting.bevat(u)) {
         esluiting.voegtoe(u);
         DEZstack.push(u);
      }
}
```

Pseudocode 11.1. Bepaling van ϵ-*sluiting* (T).

Om de DA te construeren moeten we de verzameling van zijn toestanden D (met begin- en eindtoestanden), en zijn overgangstabel M opstellen. De begintoestand van de DA is ϵ-*sluiting* (b_0), want bij aanvang gaan eventuele ϵ-overgangen vanuit b_0 onmiddellijk andere statenbits aanzetten. Om de overgang van de DA vanuit toestand T voor invoersymbool a te bepalen, moeten we alle statenbits vinden bereikbaar vanuit de statenbits van T voor a, en bovendien nog alle van daaruit bereikbare statenbits via ϵ-overgangen. De nieuwe DA-toestand is dus ϵ-*sluiting* $(s(T, a))$. Een eindtoestand van de DA bevat tenminste één eindtoestand van de NA.

Pseudocode 11.2 toont hoe dit in zijn werk gaat.

Opnieuw wordt een stapel[8] gebruikt voor toestanden waarvan de overgangen nog moeten onderzocht worden. Dit algoritme stopt omdat het aantal mogelijke DA-toestanden (deelverzamelingen van S) eindig is, en omdat elke DA-toestand slechts eenmaal op de stapel terechtkomt.

Zowel de bepaling van de epsilonsluiting als de deelverzamelingsconstructie zijn voorbeelden van een *sluiting*.

[8] Het gaat dus om diepte-eerst zoeken. Dit is meestal iets efficiënter qua geheugenverbuik dan breedte-eerst zoeken.

```
typedef Staat Set<Statenbit>;
Map<Staat,int> volgnummer; //volgnummers DA-staten
Stack<Staat> DEZstack;     //DA-toestanden met
                           // nog te bepalen overgangen
Staat T = esluiting(s0);   //Begintoestand van DA
int nieuwestaatnummer=0;
volgnummer.voegtoe(T,nieuwestaatnummer++);
DEZstack.push(T);
while (!DEZstack.ledig()) {
   T = DEZstack.topEnPop();
   for (elk invoersymbool a) {
      Staat U = esluiting(s(T,a)); // Mogelijks leeg
      if (!volgnummer.bevat(U)) {  //Nieuwe toestand
         volgnummer.voegtoe(U,nieuwestaatnummer++);
         DEZstack.push(U);
      }
      //Overgangstabel DA invullen
      M[volgnummer[T],a] = volgnummer[U];
   }
}
```

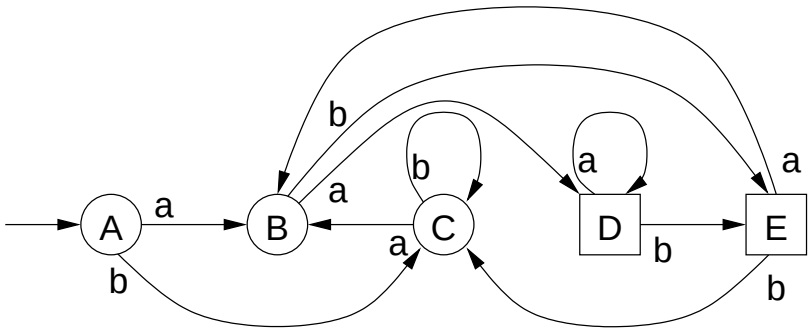
Pseudocode 11.2. Deelverzamelingsconstructie.

11.2.6.2 Automaten voor regexps

De automatentheorie leert dat de taal gedefinieerd door een regexp (een 'reguliere taal') kan herkend worden door een DA of NA. (En omgekeerd: met elke DA of NA komt een regexp overeen.) Er zijn verschillende manieren om een automaat te construeren die deze taal herkent. De *constructie van Thompson* gebruikt de structuur van de regexp als leidraad om een NA op te bouwen (Thompson, 1968). De methode werkt bottom-up. Eerst worden er primitieve NA's gedefinieerd die overeenstemmen met de basiselementen van een regexp. (Om die te vinden moet de structuur van de regexp ontrafeld worden.) Een complexe regexp wordt samengesteld uit eenvoudiger uitdrukkingen met behulp van operatoren. Analoog werd een mechanisme bedacht om een NA samen te stellen uit eenvoudiger NA's, voor elke operator.

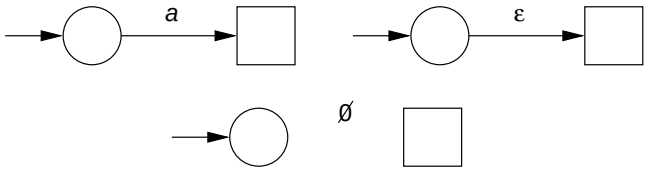
De primitieve elementen van een regexp zijn de symbolen uit Σ, en ϵ. Een NA die een dergelijk element herkent is triviaal, en bestaat uit een begin- en eindbit waartussen één overgang bestaat, gemerkt met het element (figuur 11.4). De basisconstructie voor $\emptyset$ wordt zeer zelden gebruikt. Als een symbool meerdere malen voorkomt in de regexp worden er evenzoveel triviale NA's gemaakt. Als het symbool a dus tienmaal voorkomt, zijn er tien identieke NA's voor a, elk met een unieke begin- en eindbit.

Blijft nog het samenstellen van NA's voor elk van de drie basisoperatoren. De andere



		a	b
$\rightarrow$	$A = \{1, 2, 3, 4, 8\}$	B	C
	$B = \{2, 3, 4, 5, 7, 8, 9, 10, 11\}$	D	E
	$C = \{2, 3, 4, 6, 7, 8\}$	B	C
$\leftarrow$	$D = \{2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 14\}$	D	E
$\leftarrow$	$E = \{2, 3, 4, 6, 7, 8, 13, 14\}$	B	C

Figuur 11.3. Deterministische automaat geconstrueerd uit de NA van figuur 11.6.

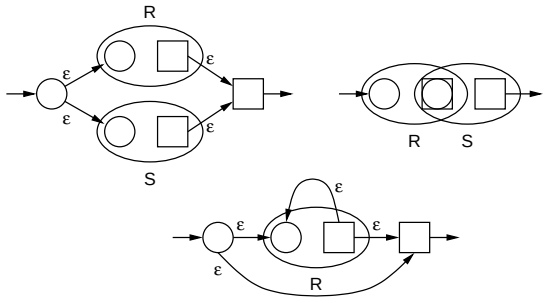


Figuur 11.4. Constructie van Thompson: basiselementen.

operatoren zoals de positieve sluiting kunnen ofwel via basisoperatoren gesimuleerd worden, ofwel kan een aparte constructie op analoge manier gedefinieerd worden. Stel dat r en s regexps zijn waarvoor reeds de overeenkomstige NA's R en S geconstrueerd werden:

- De NA voor $r|s$ voert een nieuwe begin- en eindbit in, en verbindt de beginbit met beide beginbits van R en S via ϵ-overgangen. Analoog worden de eindbits van R en S via ϵ-overgangen met de nieuwe eindbit verbonden (figuur 11.5, linksboven).

  Het is zonder meer duidelijk dat voor een invoerstring die overeenkomt met r, de nieuwe NA een weg doorheen R zal volgen tot in de overeenkomstige eindbit (en zich dan meteen in de gemeenschappelijke eindbit kan bevinden, via de ϵ-overgang). Idem voor een invoerstring overeenkomend met s. De nieuwe NA herkent dus zeker alle strings corresponderend met $r|s$, maar ook geen andere.



Figuur 11.5. Constructie van Thompson: unie, concatenatie en Kleenesluiting.

- De NA voor $r \cdot s$ gebruikt als beginbit deze van R en als eindbit deze van S, en identificeert de eindbit van R met de beginbit van S(figuur 11.5, rechtsboven). Een variant is om de twee bits te verbinden via een ϵ-overgang.

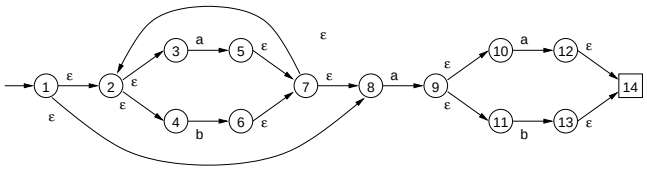
  Een string gegenereerd door $r \cdot s$ zal dus eerst R doorlopen en meteen daarna S, zodat hij herkend wordt door de nieuwe NA. Andere strings zullen niet herkend worden.

- De NA voor $r*$ voert een nieuwe begin- en eindbit in. Zowel de nieuwe beginbit als de eindbit van R worden via ϵ-overgangen verbonden met zowel de beginbit van R als de nieuwe eindbit (figuur 11.5, onder).

  Door de overbrugging wordt de ledige invoerstring opgevangen, door de terugkoppeling een eventuele herhaling. Andere mogelijkheden zijn er niet.

- Haakjes spelen hier geen rol. De NA voor (r) is dan ook dezelfde als deze voor r.

Een NA die op deze manier geconstrueerd werd, zal dus alle strings gegenereerd door de regexp herkennen, en geen andere. (Er zijn wel enkele vereenvoudigingen mogelijk bij deze constructie, maar ze combineren kan automaten opleveren die niet langer de juiste strings herkennen.) Figuur 11.6 toont de NA die met deze constructie bekomen wordt voor $(a|b)*\,a(a|b)$.



Figuur 11.6. Constructie van Thompson: NA voor $(a|b)*\,a(a|b)$.

Stelling 2 *(Kleene, 1953) Een taal kan herkend worden door een eindige deterministische automaat als en slechts als ze regulier is.*

Bewijs: We hebben al opgemerkt dat een taal die herkend wordt door een automaat zeker regulier is, omdat taal bestaande uit de etiketten van de paden vertrekkend uit de beginstaat en eindigend in een eindstaat regulier is.

Is omgekeerd een taal regulier, dan wordt ze beschreven door een regexp en voor deze regexp kunnen we met de bovengeschreven methode een DA construeren. ∎

Gevolg: Niet alle contextvrije talen zijn regulier. Als voorbeeld nemen we de taal bestaande uit strings beginnend met een aantal 'a's gevolgd door evenveel 'b's. Een eenheid die deze taal kan herkennen moet, als ze aan de eerste 'b' komt, zich herinneren hoeveel 'a's er zijn geweest. Dit aantal is niet begrensd en kan dus niet in een vooraf bepaald eindig geheugen geplaatst worden.

De NA uit de constructie heeft een aantal eigenschappen die ons nog van pas zullen komen:

- De NA heeft slechts één eindbit. (Er is ook slechts één beginbit, maar dat moet bij elke automaat.) Er zijn geen interne overgangen naar de beginbit, en ook niet vanuit de eindbit. Dat geldt eveneens voor de begin- en eindbits van elke intermediaire NA. Wanneer kleinere automaten dan samengevoegd worden tot grotere, zijn de aanhechtingspunten ondubbelzinnig, en worden er daarbij geen onvoorziene wegen gecreëerd.
- Het aantal bits van de NA is niet groter dan tweemaal het aantal elementen in de regexp. Immers, elke stap in de constructie correspondeert met een alfabetsymbool of een operator en introduceert hoogstens twee nieuwe bits.
- Vanuit elke bit van de NA vertrekken er niet meer dan twee overgangen: ofwel één overgang voor een symbool uit Σ, ofwel hoogstens twee overgangen voor ϵ.

We hebben nu een NA voor de regexp r. De efficiëntie van deze constructie is $O(|r|)$, waarbij $|r|$ de lengte van r voorstelt (het aantal alfabetsymbolen en operatoren). De syntactische ontleding van een regexp r is immers $O(|r|)$, en met elk gevonden symbool of operator komt één constructiestap overeen die duidelijk $O(1)$ is. Het aantal bits van deze NA is ook $O(|r|)$.

Een NA is echter een abstract model. Om hiermee een herkenner te implementeren, zijn er een aantal mogelijkheden:

- Een DA kan geïmplementeerd worden met een programma of een digitale schakeling. We kunnen dus de NA *transformeren* in een DA via de deelverzamelingsconstructie, en die implementeren in soft- of hardware. Eens de DA geconstrueerd, heeft de herkenning van een string s met lengte $|s|$ een uitvoeringstijd

van $\Theta(|s|)$. Er zijn echter (zeldzame) gevallen waarbij de DA een exponentieel aantal toestanden heeft, zodat de vereiste geheugenruimte (en de tijd om die in te vullen) $O(2^{|r|})$ bedraagt.

- We kunnen de NA ook rechtstreeks *simuleren*. Dat komt neer op het bepalen van alle bits van de NA die aanstaan na het inlezen van elk symbool zoals bij de deelverzamelingsconstructie gebeurt. Daarbij worden alle overgangen bepaald die de DA voor de string zou maken, met als gevolg dat sommige overgangen meermaals bepaald kunnen worden, maar ook dat niet gebruikte overgangen nooit bepaald worden.

  Deze simulatie gebruikt dezelfde functies $p(T, a)$ en ϵ-*sluiting* (T) als de deelverzamelingsconstructie. De grootte van de deelverzamelingen is $O(|r|)$. Met goede gegevensstructuren voor de deelverzamelingen kan men ervoor zorgen dat de bepaling van beide functies $O(|r|)$ wordt. (Suggesties?) Omdat er vanuit elke statenbit hoogstens twee overgangen zijn, vereist een efficiënte voorstelling van de overgangstabel slechts $O(|r|)$ plaats.

  Voor het herkennen van een string s moeten er $|s|$ overgangen bepaald worden, zodat de totale uitvoeringstijd $O(|s| \times |r|)$ bedraagt.

- De eerste herkenner is snel, maar vereist veel voorbereidend werk (in principe $O(2^{|r|})$ tijd en plaats, ook al valt dat meestal goed mee). De tweede herkenner is $O(|r|)$ maal trager, maar vraagt minder voorbereidend werk (slechts $O(|r|)$ tijd en plaats). Daarom wordt de eerste methode voornamelijk gebruikt wanneer de te herkennen strings lang zijn.

  Er is er echter ook een *hybridische methode*, die de snelheid van de eerste en het plaatsgebruik van de tweede tracht te combineren. Ook hier gebeurt de (gedeeltelijke) deelverzamelingsconstructie bij elke string, maar reeds bepaalde overgangen worden bijgehouden in een *cachegeheugen*[9], zodat ze later opnieuw kunnen gebruikt worden. In plaats van de volledige DA te construeren, gebeurt er vaak slechts een deel daarvan, omdat enkel de werkelijk gebruikte overgangen bepaald worden, en dan slechts eenmalig. Daarom heet deze methode lui ('lazy transition evaluation').

  Telkens wanneer er een overgang moet gebeuren, zoekt men eerst in de cache. (Het is duidelijk dat dit efficiënt moet gebeuren. Suggesties?) Enkel een niet gevonden overgang wordt bepaald en opgeslagen. Om het plaatsgebruik te beperken, worden oude overgangen uit een volle cache eerst verwijderd.

  Als het zoeken in de cache efficiënt gebeurt, blijkt herkennen van een string met deze methode bijna even snel als met de volledig geconstrueerde DA. Met de constructie erbij is deze methode vaak zelfs veel sneller, omdat niet gebruikte overgangen nooit bepaald worden. En haar plaatsgebruik is slechts die van de NA vermeerderd met de cache.

[9] Niet te verwarren met de cachegeheugens van de processor.

Al deze methoden vertrekken vanuit een NA. Het is echter ook mogelijk om *rechtstreeks* een DA te construeren vanuit de regexp (McNaughton en Yamada, 1960). We gaan daar niet op in.

11.2.6.3 Minimalisatie van een automaat

De constructie van Thompson is eenvoudig, maar geeft wellicht de indruk dat er onnodig veel statenbits gebruikt worden, wat nog grotere gevolgen heeft voor het aantal toestanden van de equivalente DA. Zo bijvoorbeeld toont figuur 11.3 de DA die bekomen wordt uit de NA van figuur 11.6. Bemerk dat zelfs deze kleine DA één toestand méér bevat dan de DA van figuur 11.2, ook al herkennen ze dezelfde strings. Gelukkig leert de automatentheorie dat elke DA kan getransformeerd worden in een equivalente DA, die dezelfde taal herkent, maar met een minimaal aantal toestanden.

Bij deze minimalisatie tracht men *equivalente* toestanden te groeperen: elke groep bevat alle toestanden die hetzelfde gedrag vertonen (wat herkennen betreft), en die dus niet van elkaar te onderscheiden zijn. Twee toestanden zijn equivalent wanneer men vanuit beide toestanden, na invoer van om het even welke string, ofwel in twee gewone toestanden terechtkomt, ofwel in twee eindtoestanden (niet noodzakelijk dezelfde). Equivalentie van toestanden is een equivalentierelatie: de groepen equivalente toestanden vormen een partitie van de verzameling toestanden van de DA.

Om na te gaan of twee toestanden equivalent zijn zou men alle mogelijke strings moeten uitproberen. Daarom gaat men omgekeerd te werk, en spoort alle paren toestanden op die *niet* equivalent zijn, en die dus in verschillende groepen thuishoren. Stel dat toestand t voor invoersymbool a overgaat naar toestand v, en toestand u voor diezelfde a naar w. Als v en w niet equivalent zijn, dan zijn t en u dat evenmin. Want dan bestaat er minstens één string s (eventueel ledig) waarmee v en w kunnen onderscheiden worden, zodat t en u met string as te onderscheiden zijn.

Voor elk paar toestanden dat voorlopig nog niet te onderscheiden valt gaat men na naar welke paren toestanden ze overgaan, en dat voor elk invoersymbool. En als een van die paren reeds kon onderscheiden worden, geldt dit ook voor het geteste paar. Deze testen worden herhaald tot er niets meer verandert. (Bemerk dat dit opnieuw een sluiting is.) Natuurlijk moet men met minstens één paar niet-equivalente toestanden beginnen, maar dat is eenvoudig: elke eindtoestand onderscheidt zich van elke gewone toestand (voor de ledige string). Wanneer men geen niet-equivalente toestanden meer kan vinden, zijn de overblijvende paren equivalent. (Dit lijkt vanzelfsprekend, maar het moet en kan aangetoond worden.)

Aangezien er $O(t^2)$ paren toestanden zijn, en elke iteratie minstens één nieuw paar niet-equivalente toestanden vindt, zou een eenvoudige implementatie van deze methode $O(|\Sigma|t^4)$ zijn. Een betere implementatie stelt voor elk paar toestanden een gelinkte lijst op met paren toestanden die ervan afhangen: als een paar niet-equivalent bevonden wordt, geldt dat meteen voor alle paren op zijn lijst. Het eenmalig opstellen van de lijsten is $O(|\Sigma|t^2)$, en elke lijst worden nadien slechts eenmaal overlopen, wanneer het

overeenkomstig paar toestanden niet-equivalent blijkt. De totale lengte van alle lijsten is ook $O(|\Sigma|t^2)$, omdat elk paar toestanden tot hoogstens $|\Sigma|$ lijsten kan behoren. Alle lijsten eenmaal overlopen is dus $O(|\Sigma|t^2)$, zodat de methode eveneens $O(|\Sigma|t^2)$ wordt.

De partitie kan zelfs nog sneller gevonden worden (Hopcroft, 1971). Dat gebeurt door iteratieve verfijning: telkens wordt nagegaan of de toestanden van eenzelfde groep nog steeds equivalent zijn, en indien niet wordt de groep opgesplitst. Dit proces stopt wanneer er geen groep meer kan opgesplitst worden: alle toestanden in dezelfde groep zijn dan equivalent. Initieel zijn er slechts twee groepen: de eerste met alle eindtoestanden, de tweede met de rest.

Net als hierboven zoekt men niet naar paren equivalente toestanden in een groep, maar naar niet-equivalente paren. Ook hier gaat men voor elk paar in een groep na naar welk paar ze overgaan, voor elk invoersymbool. Als ze naar toestanden in verschillende (voorlopige) groepen overgaan, zijn ze niet-equivalent.

Een goede implementatie van deze methode is $O(|\Sigma|t \lg t)$, maar die is helemaal niet vanzelfsprekend: de volgorde waarin groepen getest worden, en de manier waarop informatie wordt bijgehouden is (zoals zo vaak) cruciaal.

Eens de partitie gekend is, wordt de gereduceerde DA opgesteld. Daartoe kiest men in elke groep een toestand als *vertegenwoordiger*. Deze vertegenwoordigers worden de toestanden van de nieuwe DA. De begintoestand van de nieuwe DA is de vertegenwoordiger van de groep die de begintoestand van de originele DA bevat. Elke groep van de partitie bestaat volledig uit eindtoestanden van de originele DA, of bevat geen enkele eindtoestand. (Waarom?) De eindtoestanden van de nieuwe DA zijn dan de vertegenwoordigers van de groepen met originele eindtoestanden.

Overgangen tussen toestanden moeten overgangen tussen groepen worden, en dus tussen hun vertegenwoordigers. Tenslotte verwijdert men nog alle toestanden uit de nieuwe DA die onbereikbaar zijn vanuit de begintoestand. Men kan aantonen dat de bekomen DA uniek is, op naamgeving van de toestanden na.

Als we deze minimalisatie bijvoorbeeld toepassen op de DA van figuur 11.3 op pagina 136, dan zijn er initieel twee groepen: $\{A, B, C\}$ en $\{D, E\}$. Voor zowel a als b blijft D in dezelfde groep, maar E niet. De groep eindtoestanden moet dus gesplitst worden (en kan niet meer verder splitsen). In de andere groep vertonen A en C hetzelfde gedrag, maar B niet. Deze groep wordt dus gesplitst in $\{A, C\}$ en B. Ook daarna blijft het gedrag van A en C gelijk, zodat er niets meer te splitsen valt. Als we A tot vertegenwoordiger van zijn groep kiezen, dan wordt de overgangstabel

		a	b
$\rightarrow$	$A = \{A, C\}$	B	A
	B	D	E
$\leftarrow$	D	D	E
$\leftarrow$	E	B	A

die, op andere namen voor de toestanden na, dezelfde automaat voorstelt als in fi-

guur 11.2.

11.2.7 De Shift-AND-methode

Dit is een eenvoudige, bitgeoriënteerde methode, die zeer efficiënt werkt voor *kleine* patronen (zoals gewone woorden), en eenvoudig kan uitgebreid worden om patronen te zoeken waarin *fouten* geslopen zijn. Ze maakt gebruik van dynamisch programmeren.

Deze methode (Baeza-Yates en Gonnet, 1992) houdt voor elke positie j in de tekst T bij welke prefixen van het patroon P overeenkomen met de tekst, *eindigend* op positie j. Daartoe gebruikt men een (eendimensionale) tabel R van p logische waarden. Het i-de element van deze tabel komt overeen met het prefix van lengte i.

Als we met R_j de waarde van tabel R aanduiden die hoort bij tekstpositie j, dan is het i-de element $R_j[i-1]$ waar als de eerste i karakters van P overeenkomen met de i tekstkarakters eindigend in j. Zodat P zelf eindigt op positie j als $R_j[p-1]$ waar is, en dus voorkomt in de tekst bij beginpositie $j-p+1$.

Het volgende tekstkarakter $T[j+1]$ kan sommige van die prefixen verlengen tot aan de nieuwe positie $j+1$. De nieuwe tabel R_{j+1} kan dus uit de vorige R_j afgeleid worden, als volgt:

$$
\begin{aligned}
R_{j+1}[0] &= \begin{cases} \text{waar als } P[0] = T[j+1] \\ \text{niet waar als } P[0] \neq T[j+1] \end{cases} \\
R_{j+1}[i] &= \begin{cases} \text{waar als } R_j[i-1] \text{ waar is en } P[i] = T[j+1] \\ \text{niet waar in de andere gevallen} \end{cases} \qquad \text{voor } 1 \leq i < p
\end{aligned}
$$

Op het eerste gezicht vraagt de methode veel werk, aangezien we deze tabel bij elke tekstpositie moeten bepalen. Telkens p tabelelementen berekenen op t plaatsen zou inderdaad een performantie van $\Theta(tp)$ opleveren.

De bewerkingen voor elk tabelelement zijn echter analoog. (Behalve voor het eerste element, maar dat kan wat aangepast worden.) Aangezien de elementen logische waarden zijn, en er processorinstructies bestaan die op alle bits van processorwoorden *tegelijk* inwerken, gebruikt men de afzonderlijke bits van een processorwoord om die waarden voor te stellen. Dat vereist natuurlijk dat p niet groter is dan de breedte w van een processorwoord.

Bij de berekening van R_{j+1} moeten we weten of het nieuwe tekstkarakter $T[j+1]$ gelijk is aan $P[i]$, voor elke mogelijke waarde van i. En als we de bits gelijktijdig willen bepalen, moet ook deze informatie in de bits van een processorwoord opgeslagen worden. Telkens p testen uitvoeren bij elke tekstpositie is nefast, maar aangezien P niet verandert, kunnen we dat woord op voorhand bepalen, voor elk mogelijk tekstkarakter. We stellen dus een tabel S op met d woorden. (De grootte van het alfabet.) Een bit i van woord $S[s]$ is waar als het karakter s op plaats i in P voorkomt.

Om bit i van R_{j+1} te berekenen hebben we dus bit $(i-1)$ van R_j nodig, samen met bit i van het woord in S dat overeenkomt met $T[j+1]$. En om alle bits van R_{j+1}

gelijktijdig te berekenen voeren we eerst een *schuifoperatie* naar rechts op R_j uit (bit i wordt bit $i+1$), gevolgd door een bit-per-bit EN-operatie met $S[T[j+1]]$:

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

Bij de schuifoperatie onderstellen we dat er vooraan (bij index één) een 'waar'-bit wordt ingeschoven. De waarde van $R_{j+1}[0]$ wordt dan volledig bepaald door de eerste bit van $S[T[j+1]]$, anders gezegd, ze is waar als $T[j+1]$ gelijk is aan $P[0]$.

Elk van die operaties kan met één processorinstructie gebeuren, zodat het aantal operaties om R_{j+1} uit R_j te berekenen slechts $O(1)$ bedraagt in plaats van $O(p)$.

Blijft nog het bepalen van woord R_0, dat geen voorloper heeft. Dat kan apart gebeuren, maar we kunnen dezelfde schuif- en EN-operaties gebruiken als we een extra tabel R_{-1} invoeren, waarvan elke bit onwaar is.

Terzijde merken we op dat een schuifoperatie meestal een nul vooraan inschuift, en als we een logische 'waar' door één willen voorstellen, dan moeten we die bit inverteren. Om dat te vermijden keert men soms de betekenis van 'waar' om, zodat de methode dan beter 'Shift-OR' genoemd wordt, haar oorspronkelijke naam trouwens.

Hoewel er in totaal $\Theta(tp)$ bitoperaties gebeuren (in alle gevallen), worden dat $\Theta(t)$ woordoperaties (processorinstructies) als $p \leq w$. Dat geldt ook nog als p een klein veelvoud van w is, omdat er dan slechts enkele woorden vereist zijn om elke tabel op te slaan. Het opstellen van tabel S vergt $\Theta(d)$ voor de initialisatie, en $\Theta(p)$ voor het invullen. De totale performantie wordt dan $\Theta(t+p)$ (als we d kunnen verwaarlozen). Voor kleine patronen is dit dus een snelle methode, die bovendien weinig geheugen gebruikt. Voor grote patronen (zoals in de bio-informatica waar p enkele duizendtallen kan bedragen) moet men andere methoden gebruiken.

11.3 DE SHIFT-AND-METHODE: BENADERENDE OVEREENKOMST

Deze methode kan vrij gemakkelijk aangepast worden om fouten (gewijzigde, en/of ingelaste, en/of verwijderde karakters) in het gevonden patroon toe te laten (Wu en Manber, 1992).

Beginnen we met een eenvoudig geval, waarbij slechts één extra karakter op een willekeurige plaats in P mag ingelast worden. Anders gezegd, we zoeken alle deelstrings in T niet langer dan $m+1$ die P als deelsequentie bevatten. (In tegenstelling tot de karakters van een string zijn die van een sequentie niet noodzakelijk opeenvolgend.)

We gebruiken dezelfde tabellen R en S, maar nu zijn er twee mogelijkheden om een overeenkomst te vinden voor elk prefix, namelijk exact, of met één inlassing. Daarom voeren we een analoge tabel R_j^1 in, die alle prefixen aanduidt die in de tekst te vinden

zijn eindigend bij positie j, met *hoogstens* één inlassing. (We zullen R voortaan dan ook noteren als R^0.) Dat betekent dat $R^1_j[i]$ waar is als de eerste i karakters van P overeenkomen met i van de $i+1$ karakters (uiteraard in dezelfde volgorde) die in de tekst eindigen bij positie j. (Bemerk dat dit ook een exacte overeenkomst inhoudt.) Dan zal $R^0_j[m]$ een exacte overeenkomst en $R^1_j[m]$ een overeenkomst met hoogstens één inlassing aanduiden als ze waar zijn (en dat kan gelijktijdig).

De afleiding van R^0_{j+1} uit R^0_j blijft uiteraard gelijk:

$$R^0_{j+1} = \text{Schuif}(R^0_j) \text{ EN } S[T[j+1]]$$

Voor de bepaling van $R^1_{j+1}[i]$ maken we een onderscheid tussen een inlassing aan het einde van het prefix (na patroonpositie i), of eventueel op een andere plaats. Er zijn dus twee gevallen:

(1) De eerste i patroonkarakters komen exact overeen tot tekstpositie j. Als we dan tekstkarakter $T[j+1]$ na die i karakters inlassen, dan bekomen we een overeenkomst van i patroonkarakters met één inlassing tot tekstpositie $j+1$.
(2) De eerste $i-1$ patroonkarakters komen overeen tot tekstpositie j met hoogstens één inlassing, en $T[j+1]$ is gelijk aan $P[i]$. In dit geval gebeurt de eventuele inlassing dus elders in het prefix en niet aan het einde.

Beide gevallen moeten gecombineerd worden met een OF-operatie, want ze kunnen elk apart maar ook samen voorkomen. We krijgen dan

$$R^1_{j+1}[i] = R^0_j[i] \text{ OF } (R^1_j[i-1] \text{ EN } T[j+1] = P[i])$$

In het eerste geval is R^1_{j+1} enkel een kopie van R^0_j. Geval twee vereist een schuifoperatie op R^1_j gevolgd door een bit-per-bit EN-operatie met $S[T[j+1]]$. Als we onderstellen dat de schuifoperatie de hoogste prioriteit heeft, dan geeft dat

$$R^1_{j+1} = R^0_j \text{ OF } (\text{Schuif}(R^1_j) \text{ EN } S[T[j+1]])$$

Dus ook R^1 berekenen voor elke tekstpositie vereist slechts enkele processorinstructies. (Opnieuw wordt hier een waar-bit ingeschoven. Ga eens na of dat correct is.)

Stel dat we nu toelaten dat er één karakter verwijderd wordt (en geen inlassing meer). Opnieuw gebruiken we de tabellen S, R^0 en R^1, en deze laatste duidt nu prefixen met hoogstens één verwijdering aan. Dat betekent dat $R^1_j[i]$ waar is als er $i-1$ van de eerste i karakters van P overeenkomen met de $i-1$ karakters (uiteraard in dezelfde volgorde) die in de tekst eindigen bij positie j. En weer zijn er twee mogelijke gevallen voor een overeenkomst van de i eerste karakters van P eindigend op tekstpositie $j+1$, met hoogstens één verwijdering:

(1) De eerste $i-1$ patroonkarakters komen exact overeen tot tekstpositie $j+1$. Hier werd dus $P[i]$ verwijderd. Dan is bit $R^0_{j+1}[i-1]$ uit de nieuwe tabel R^0 waar.

(2) De eerste $i-1$ patroonkarakters met hoogstens één verwijdering komen overeen tot tekstpositie j, en $T[j+1]$ is gelijk aan $P[i]$. De verwijdering gebeurt opnieuw ergens binnenin en niet aan het einde.

Die moeten opnieuw gecombineerd worden met een OF-operatie, zodat

$$R^1_{j+1}[i] = R^0_{j+1}[i-1] \text{ OF } (R^1_j[i-1] \text{ EN } T[j+1] = P[i])$$

Geval één vereist enkel een schuifoperatie op de nieuwe tabel R^0_{j+1} (die dus reeds bepaald moet zijn). Het tweede geval is volledig hetzelfde als bij inlassing. We krijgen nu

$$R^1_{j+1} = \text{Schuif}(R^0_{j+1}) \text{ OF } (\text{Schuif}(R^1_j) \text{ EN } S[T[j+1]])$$

De laatste mogelijke fout is een vervanging (zonder enige inlassing of verwijdering). We laten dan toe dat er één karakter van P vervangen wordt door een tekstkarakter. Vervanging kan ook gesimuleerd worden met een verwijdering en een inlassing, maar vaak wil men dat als slechts één fout beschouwen. Als $R^1_j[i]$ waar is betekent dat nu dat $i-1$ van de eerste i karakters van P overeenkomen met $i-1$ van de i karakters (uiteraard in dezelfde volgorde) die in de tekst eindigen bij positie j. Eens te meer zijn er twee gevallen:

(1) De eerste $i-1$ patroonkarakters komen exact overeen tot bij tekstpositie j. Karakter $P[i]$ werd dus vervangen door $T[j+1]$. (Als beide karakters gelijk zijn wordt dat aangeduid in R^0_{j+1}.)
(2) De eerste $i-1$ patroonkarakters komen met hoogstens één vervanging overeen tot bij tekstpositie j, en $P[i]$ is gelijk aan $T[j+1]$. De vervanging gebeurt weer binnenin.

Er komt dan

$$R^1_{j+1}[i] = R^0_j[i-1] \text{ OF } (R^1_j[i-1] \text{ EN } T[j+1] = P[i])$$

Voor geval één moeten we R^0_j naar rechts schuiven. Geval twee is hetzelfde als dat bij inlassing en verwijdering. Samen geeft dat

$$R^1_{j+1} = \text{Schuif}(R^0_j) \text{ OF } (\text{Schuif}(R^1_j) \text{ EN } S[T[j+1]])$$

Als we nu hoogstens één fout toelaten, om het even dewelke, dan moeten we de drie gevallen combineren:

$$R^1_{j+1} = R^0_j \text{ OF Schuif}(R^0_{j+1}) \text{ OF Schuif}(R^0_j) \text{ OF } (\text{Schuif}(R^1_j) \text{ EN } S[T[j+1]])$$

Om te kunnen beginnen introduceert men ook hier een tabel R^1_0. (Hoe moet die er uitzien?)

Het geval van hoogstens één fout kan uitgebreid worden tot het algemeen geval dat *hoogstens* f fouten toelaat. Elk van die fouten kan een inlassing, verwijdering of vervanging zijn. (Twee strings die uit elkaar af te leiden zijn via f dergelijke operaties liggen op 'edit-distance' f van elkaar.)

In plaats van de tabel R^1 komen er nu f tabellen $R^1, R^2, \ldots, R^f$, waarbij R^k alle mogelijke overeenkomende prefixen met *hoogstens* k fouten aanduidt. Het komt er dus weer op aan om R^k_{j+1} af te leiden, als R^k_j reeds gekend is. Bovendien kan er een fout bijkomen, zodat we ook R^{k-1}_j en R^{k-1}_{j+1} nodig hebben. De overgangen tussen de tabellen voor posities j en $j+1$ moeten dus bepaald worden volgens stijgende k.

Er zijn nu vier mogelijke gevallen om een overeenkomst te vinden voor de eerste i karakters met hoogstens k fouten, eindigend bij tekstpositie $j+1$. Bij de eerste drie treedt er een fout op aan het einde van P, bij geval vier komen alle eventuele fouten elders in P voor:

(1) Er bestaat een overeenkomst van de eerste i patroonkarakters met hoogstens $k-1$ fouten tot bij tekstpositie j. Hier wordt tekstkarakter $T[j+1]$ ingelast na $P[i]$.
(2) Er bestaat een overeenkomst van de eerste $i-1$ patroonkarakters met hoogstens $k-1$ fouten tot bij tekstpositie $j+1$. Dat betekent de verwijdering van $P[i]$.
(3) Er bestaat een overeenkomst van de eerste $i-1$ patroonkarakters met hoogstens $k-1$ fouten tot bij tekstpositie j. Dat komt neer op een vervanging van $P[i]$ door $T[j+1]$.
(4) Er bestaat een overeenkomst van de eerste $i-1$ patroonkarakters met hoogstens k fouten tot bij tekstpositie j, en $P[i]$ is gelijk aan $T[j+1]$.

Alles samen geeft dat (voor $k > 0$):

$$\begin{aligned} R^k_{j+1} &= R^{k-1}_j \text{ OF Schuif}(R^{k-1}_{j+1}) \text{ OF Schuif}(R^{k-1}_j) \text{ OF } (\text{Schuif}(R^k_j) \text{ EN } S[T[j+1]]) \\ &= R^{k-1}_j \text{ OF Schuif}(R^{k-1}_{j+1} \text{ OF } R^{k-1}_j) \text{ OF } (\text{Schuif}(R^k_j) \text{ EN } S[T[j+1]]) \\ R^k_0 &= \overbrace{\underbrace{11\ldots1}_{k}00\ldots\ldots0}^{m} \end{aligned}$$

Elke R^k ($k > 0$) kan dus bepaald worden met twee schuifoperaties, één EN- en drie OF-operaties.

Het voorbereidend werk van de methode bestaat uit het opstellen van tabel S (zoals bij de exacte overeenkomst) en het initialiseren van de $f+1$ tabellen R_0 wat $O(f\lceil p/w \rceil)$ operaties vereist. Daarna gebeuren er $O(f\lceil p/w \rceil)$ operaties per tekstkarakter, zodat de totale performantie nu $O(tf\lceil p/w \rceil)$ wordt, en dit voor een woordbreedte van w bits. Voor niet al te grote p wordt dat dan $O(tf)$.

11.3.0.1 Verdere uitbreidingen

Daarmee zijn de mogelijkheden van deze versatiele methode nog niet uitgeput. Met soms verrassend kleine aanpassingen kan ze uitgebreid worden voor meer ingewikkelde zoekopdrachten:

- *Verzameling van karakters.* In plaats van een welbepaald karakter in het patroon kunnen we een verzameling van toegelaten karakters specificeren. Zo kunnen we bijvoorbeeld zoeken naar het patroon 'ABC[0-9]XYZ', met hoogstens k fouten. (We gebruiken hier de notatie voor regexps. Elk cijfer in het midden van het patroon is toegelaten.)

  De aanpassing is hier zeer eenvoudig. Om elk mogelijk cijfer te aanvaarden op de vierde positie van de zoekstring volstaat het om in elke rij van de tabel S die overeenkomt met een cijfer, een waar-bit te voorzien bij index vier.

  Het *complement* van een karakter is een speciaal geval van een verzameling karakters en kan dus op dezelfde manier behandeld worden.

- *Jokers.* Eén enkele joker ('wild card', 'don't care') is een karakter dat overeenkomt met elk mogelijk karakter. Dit kan dus eveneens als een speciaal geval van een verzameling karakters beschouwd worden.

  Om een niet nader bepaald aantal willekeurige karakters op willekeurige plaatsen van een patroon toe te laten zijn ietwat meer wijzigingen aan de methode nodig, waar we niet verder op ingaan.

- *Een onbekend aantal fouten.* Stel dat we alle plaatsen in de tekst willen zoeken waar het patroon met *zo weinig mogelijk* fouten voorkomt. Natuurlijk kan men eerst een exacte overeenkomst zoeken, als er geen is, een overeenkomst met één fout, dan eventueel met twee fouten, enz. Maar het kan ook met minder werk, door telkens dubbel zoveel fouten plus één toe te laten. Men zoekt dus eerst zonder fouten, dan eventueel met één fout, daarna met drie fouten, met zeven fouten, enz. (Een variant van binair zoeken.) Als blijkt dat het minimaal aantal fouten k is, dan bedraagt de uitvoeringstijd $O(n+2n+4n+\cdots+2^b n)$, waarbij 2^b de kleinste macht van twee is, groter dan k (of $b = \lfloor \lg k \rfloor + 1$). Dat betekent dat er in het slechtste geval viermaal meer operaties worden uitgevoerd dan wanneer k op voorhand gekend was. In de praktijk blijkt dat meestal slechts twee- of driemaal zoveel.

- *Een combinatie van exacte en benaderende overeenkomst.* We kunnen ook eisen dat er in bepaalde delen van het patroon geen fouten mogen optreden. Stel dat we de nummerplaat 'ABC123' zoeken, waarbij we zeker zijn van de letters, maar één van de cijfers verkeerd kan zijn.

  De methode moet dan aangepast worden door een gedeelte van het patroon af te schermen tegen het optreden van fouten. Dat gebeurt met een tabel M van m logische waarden (een masker) die waar zijn op de patroonposities waar een fout

mag optreden, en onwaar elders. De berekening van R^k_{j+1} moet nu zo gebeuren dat inlassingen, verwijderingen en vervangingen enkel mogelijk zijn op de posities waar het masker M een waar-bit bevat. We krijgen dan

$$R^k_{j+1} = (\,(R^{k-1}_j \text{ OF Schuif}(R^{k-1}_{j+1} \text{ OF } R^{k-1}_j)\,) \text{ EN } M) \\ \text{OF } (\text{Schuif}(R^k_j) \text{ EN } S[T[j+1]])$$

Bemerk dat het masker gebruikt wordt bij de termen voor een eventuele fout achteraan, en niet bij de term voor de exacte overeenkomst met het huidige tekstkarakter.

- *Fouten met verschillende kost.* De 'edit distance' onderstelt dat inlassingen, verwijderingen en vervangingen evenveel kosten. Maar soms willen we bijvoorbeeld minder verwijderingen toelaten dan vervangingen, of misschien zelfs helemaal geen verwijderingen.

  Als de kosten van de verschillende operaties kleine gehele getallen zijn, dan kan de berekening van R^k_{j+1} hiermee rekening houden door de bijdrage van een bepaalde soort fout tot het totaal aantal fouten groter dan één te maken.

  Stel eens dat de kost van een inlassing en een verwijdering drie bedraagt, en die van een vervanging slechts één. Zeven fouten (eigenlijk fouteenheden) kunnen dan bijvoorbeeld bestaan uit één verwijdering en vier vervangingen, of uit één inlassing, één verwijdering en één vervanging. Voor de vervanging verandert er niets, maar aangezien beide andere fouten driemaal zoveel kosten moet een inlassing of verwijdering die tot een overeenkomst met hoogstens k fouten leidt nu afkomstig zijn van een overeenkomst met hoogstens $k-3$ fouten. De aanpassing aan de berekening van R^k_{j+1} is eenvoudig: vervang $k-1$ door $k-3$ in de termen die overeenkomen met inlassen en verwijderen. De methode blijft hiermee zelfs even snel.

Mits meer ingrijpende aanpassingen, kan de methode ook nog gebruikt worden om naar een aantal patronen *tegelijk* te zoeken, en om een patroon te zoeken beschreven door een *regexp*. We gaan hier niet verder op in.

HOOFDSTUK 12
INDEXEREN VAN VASTE TEKST

Sommige zoekoperaties op strings gebeuren op een (grote) *vaste* tekst T (met lengte t), waarin frequent gezocht moet worden naar een (veranderlijk) patroon P (met lengte p). Een belangrijk voorbeeld daarvan zijn databanken voor genomen. In dat geval loont het de moeite om voorbereidend werk te doen op de *tekst*, zodat de zoekoperaties nadien efficiënter verlopen.

De efficiënte zoekalgoritmen hierboven verrichten voorbereidend werk op het *patroon*, en hun performantie in het slechtste geval is $O(t + p)$. We zullen zien dat tengevolge van $O(t)$ voorbereidend werk op de tekst, deze zoektijd kan gereduceerd worden tot $O(p)$, onafhankelijk dus van de tekstlengte.

Als voorbereidend werk slaat men alle *suffixen* van de tekst in een gegevensstructuur op. Wanneer immers het patroon in de tekst voorkomt, moet het een *prefix* zijn van een van die suffixen. We hebben dus een gegevensstructuur nodig die toelaat om snel dergelijke prefixen te zoeken, die bovendien in een redelijke tijd kan geconstrueerd worden, en niet al te veel geheugen vereist. Een snelle constructie is niet zo evident, aangezien er t suffixen zijn, met in totaal $O(t^2)$ karakters. (Bedenk dat t zeer groot kan zijn.) In wat volgt gebruiken we de notatie suff_i om het suffix aan te duiden dat begint op lokatie i.

12.1 SUFFIXBOMEN

Een voor de hand liggende keuze voor deze gegevensstructuur lijkt een *meerwegstrie*, aangezien de zoektijd dan inderdaad beperkt wordt door de sleutellengte p, onafhankelijk van het aantal opgeslagen suffixen t. (Weer vermijden we dat een suffix een prefix zou kunnen zijn van een ander suffix door de tekst af te sluiten met een speciaal karakter.)

Een eenvoudige methode om deze suffixtrie te construeren voegt de suffixen één voor één toe. De constructietijd is dan echter $O(t^2)$. Bovendien neemt een meerwegstrie veel te veel geheugen in beslag, vooral omdat er lange reeksen opeenvolgende knopen met slechts één kind kunnen voorkomen. Aangezien de gemiddelde lengte van de suffixen $t/2$ bedraagt, en de suffixen gewoonlijk weinig gemeenschappelijke prefixen hebben, vereist de trie $O(|\Sigma|t^2)$ geheugen.

Aangezien alle opgeslagen strings suffixen zijn van dezelfde tekst, werd een nieuwe gegevensstructuur bedacht, de *suffixboom* ('suffix tree', Weiner, 1973). Deze boom is verwant aan een (meerwegs) patriciatrie, die trouwens oorspronkelijk ook gebruikt werd om alle suffixen van een lange tekst op te slaan. Zoals we reeds gezien hebben worden bij patricia knopen met slechts één kind geëlimineerd. Daarmee wordt het aantal inwendige knopen gereduceerd tot $O(t)$ en de vereiste geheugenruimte tot $O(|\Sigma|t)$.

Bovendien kan deze suffixboom geconstrueerd worden in $O(t)$. De eerste lineaire constructiealgoritmen (Weiner, 1973, en McCreight, 1976) doorliepen de tekst van achter naar voor, maar een recenter algoritme (Ukkonen, 1995) werkt van voor naar achter, zodat het ook voor 'online' toepassingen (zoals datacompressie) bruikbaar werd.

We beschrijven de wijzigingen tegenover patricia:

(1) Bij patricia moesten we de strings opslaan bij de bladeren. Nu kunnen we de tekst als referentie nemen. In plaats van het suffix suff_i op te slaan volstaan we met de beginindex i. Wel moeten we natuurlijk de tekst zelf bijhouden.
(2) Bij patricia hielden we de testindex bij in elke knoop. Omdat we daarmee karakters in de zoekstring oversloegen moesten we op het einde naar een blad gaan om de overgeslagen karakters te verifiëren. Dit is hier erg onhandig omdat we prefixen van opgeslagen strings zoeken. Hier kunnen we echter op eenvoudige wijze alle karakters van de verdwenen knopen registreren. Deze vormen een substring van de tekst, zodat we in elke knoop alleen een begin- en een eindindex moeten opnemen. Op die manier kunnen we, als we bij het zoeken aankomen in een expliciete inwendige knoop, onmiddelijk verifiëren of verdergaan nog zin heeft.
(3) Voorts kunnen we in elke inwendige knoop een staartpointer opnemen die de opbouw en sommige toepassingen veel sneller maakt.

Dit laatste vereist enige uitleg. We noemen de *staart* van een string de string bekomen door het eerste karakter (als dat er is; de staart van de lege string is de lege string) te verwijderen. De staart van een string s noteren we als

$$staart(s).$$

Dit zit natuurlijk ook in de boom en dus heeft de trie een zeer repeterend karakter. Immers, in een trie is er een expliciete inwendige knoop horend bij de niet-lege string α (die het pad naar de inwendige knoop aanduidt; we spreken dan ook van de *padstring* voor de knoop), als en slechts als de trie twee strings bevat van de vorm $\alpha\beta$ en $\alpha\gamma$, zodat de eerste letter van β verschilt van de eerste letter van γ. Maar in een suffixboom zitten $staart(\alpha\beta)$ en $staart(\alpha\gamma)$ dan ook in de trie, en dus is er een expliciete inwendige knoop voor $staart(\alpha)$. De staartpointer van de knoop met prefix α wijst nu naar die tweede inwendige knoop.

We weten dat een trie problemen oplevert als er een string is die een prefix is van een ander. Dit kunnen we oplossen met een afsluitend karakter, maar deze oplossing

kunnen we niet toepassen bij de opbouw van een suffixboom. We starten immers met een lege suffixboom en voegen elke keer één karakter van T toe, zodat we na k iteraties suffixen hebben van de string $T[0] \ldots T[k-1]$, zonder afsluitteken. Stel nu dat er een suffix α is dat een prefix is van een ander suffix $\alpha\beta$. Dan is het volgende suffix $staart(\alpha)$ en dat is een prefix van $staart(\alpha\beta)$, dat ook in de suffixboom zit. Met andere woorden: na k iteraties is er een kleinste index i (die gelijk kan zijn aan k zelf) zodanig dat het incomplete ***suff***$_i$ een prefix is van een vorig suffix. Dan geldt

(1) Alle suffixen ***suff***$_j$ waarvoor $j < i$ geen prefix zijn en dus aangeduid worden door een blad.
(2) Alle suffixen ***suff***$_j$ met $j \geq i$ prefix zijn van een andere string en dus niet zichtbaar zijn in de suffixboom. ***suff***$_i$ noemen we het *actieve suffix*.

We gaan nu verder door $T[k]$ toe te voegen. We moeten alle strings in de suffixboom met dit karakter verlengen en de 1-karakterstring "$T[k]$" toevoegen, tenzij dit karakter al eerder in de string zat, want dan is deze string een prefix dat we niet te zien krijgen.

(1) Voor suffixen die aangeduid worden door een blad moeten we niets doen. Immers, in het blad staat de beginindex van de string in T en daardoor gaat de verlenging automatisch.
(2) Nu gaan we de strings af die een prefix waren, in orde van dalende lengte. Er zijn er nul of meer die door dit extra karakter geen prefix meer zijn. Die moeten een blad krijgen, waarna we naar het volgende suffix gaan. Vinden we een suffix dat zelfs met dat extra karakter een prefix is dan kunnen we stoppen, want alle kortere suffixen zijn ook een prefix.

Als het laatste karakter van T nergens anders voorkomt in de tekst (een stopkarakter) dan hebben alle suffixen een blad gekregen en zitten dus expliciet in de suffixboom.

Als de operatie bestaand uit het toevoegen van een blad en verder springen naar de volgende impliciet aanwezige suffix $O(1)$ is, dan is ons hele algoritme $O(t)$, want er zijn t bladeren. Om dit te bereiken houden we een pointer bij naar de laatst toegevoegde inwendige knoop en een pointer naar het zogenaamde *actieve punt*, dit is de laatste expliciete inwendige knoop die we tegenkomen als we het actieve suffix zouden zoeken in de suffixboom (het actieve suffix kan eindigen in een expliciete of een impliciete inwendige knoop, maar uiteraard niet in een blad). De behandeling van het actieve suffix gaat als volgt:

(1) Onderzoek of de impliciete of expliciete inwendige knoop waar het suffix eindigt een uitgang heeft voor $T[j]$. Als dit zo is dan is het verlengde actieve suffix nog altijd een prefix en blijft het actieve suffix. Wel kan het zijn dat we het actieve punt moeten verleggen.
(2) Is er geen uitgang, maak dan een eventuele impliciete knoop expliciet en hang een blad aan de knoop. Het volgende suffix wordt nu actief. Via de staartpointer van het actieve punt springen we naar dit volgende suffix.

(3) Als we een impliciete knoop I expliciet gemaakt hebben moeten we de staartpointer nog invullen. Deze moet wijzen naar de knoop waar het nieuwe actieve suffix (dat nog niet verlengd is) eindigt. Ofwel is die knoop al expliciet ofwel een impliciete knoop J, maar dan heeft hij maar één uitgang, namelijk dezelfde als I. I had geen uitgang voor $T[j]$, dus J ook niet. Dus zal ook J expliciet gemaakt worden bij het verwerken van het volgende actieve suffix.

Het invullen van de staartpointer is een vrij delicate operatie. Immers, de staartpointer van een nieuw aangemaakte knoop kan wijzen naar een knoop die zelf nog moet aangemaakt worden, maar ook naar een knoop die al langer bestaat. In beide gevallen gebeurt het invullen bij de afhandeling van het volgend actieve suffix (maar dat gebeurt dan wel voor we naar het volgende karakter in de tekst gaan).

Bij het overspringen naar het volgende actieve suffix moeten we dus gebruik maken van de staartpointer van de *laatste oude expliciete knoop* op het pad van het huidig suffix; als dit de wortel is (het huidige actieve suffix heeft dan lengte 1) blijven we uiteraard in de wortel. Als we dus een nieuwe expliciete knoop aanmaken verleggen we het actieve punt *niet*. Als we echter bij het toevoegen van het karakter aankomen in een oude expliciete knoop dan verleggen we het actieve punt *wel*, want de staartpointer is al ingevuld.

Een suffixboom heeft heel wat toepassingen, waarvan de eenvoudigste zijn:

- *Het klassieke deelstringprobleem.* Gevraagd alle beginposities van P in T. We construeren een suffixboom voor T, en zoeken daarna de (eventueel impliciete) knoop die overeenkomt met P. Als die niet bestaat, dan komt P niet in T voor. Indien wel, dan zijn de gezochte beginposities de indices bij alle bladeren die opvolgers zijn van deze knoop. Als P k maal voorkomt, kunnen die dus gevonden worden in $O(p + k)$ (de deelboom heeft hoogstens $k - 1$ inwendige knopen).

  Als we enkel één plaats willen kennen waar P in T voorkomt, kunnen we vooraf in elke inwendige knoop een index van een van de bladeren uit zijn deelboom opslaan. Dit voorbereidend werk is $O(t)$: vermits we altijd een blad aanmaken als we een expliciete inwendige knoop maken kunnen we de index uit dit blad nemen. Dit komt overeen met de verwijzing naar een willekeurige bladknoop bij een patricia meerwegstrie. Daarmee wordt de operatie zelf $O(p)$.

- *Langste gemeenschappelijke deelstring.* Gegeven een verzameling van k (verschillende) strings $S = \{s_1, s_2, \ldots, s_k\}$, met totale lengte t. Gezocht de langste gemeenschappelijke deelstring van al die strings.

  Een *veralgemeende suffixboom* voor een verzameling strings bevat al de suffixen van al de strings uit deze verzameling. Zijn bladeren bevatten niet alleen een beginpositie van een suffix, maar ook de string waartoe die behoort. Hetzelfde suffix kan zelfs tot meerdere strings behoren, zodat al deze informatie bij het overeenkomstig blad moet opgeslagen worden. Bovendien moeten de takken niet enkel een begin- en eindindex in een string aanduiden, maar ook nog in welke

string. Toch bedraagt de constructietijd van een veralgemeende suffixboom voor een verzameling strings met totale lengte t slechts $O(t)$.

Om het probleem op te lossen construeert men dus een veralgemeende suffixboom voor de strings van S. Elke (expliciete) inwendige knoop van de boom komt overeen met een prefix van een suffix, en deze deelstring komt voor in elke string die vermeld wordt bij een blad dat opvolger is van die knoop. De boom wordt daarna systematisch overlopen om de lengte van al de prefixen (de diepte van de knopen, gerekend in aantal karakters) en het aantal *verschillende* strings te bepalen waarin ze voorkomen, en dus meteen ook het langste prefix dat in alle strings voorkomt. De boom heeft $O(t)$ knopen, kan dus in $O(t)$ doorlopen worden, zodat de oplossing ook in $O(t)$ gevonden wordt.

Het is handig om voor elke string in S een *verschillend* afsluitkarakter te nemen. Op die manier kan men aan het laatste karakter van een suffix zien tot welke string hij behoort en hoort elk blad bij maar één string.

Een suffixboom kan nog veel meer interessante operaties efficiënt ondersteunen, wanneer het langste gemeenschappelijk prefix van twee willekeurige suffixen kan gevonden worden in $O(1)$. (Dat komt neer op het vinden van de laagste gemeenschappelijke voorloper van de twee bladeren.) Daarvoor is bijkomend voorbereidend werk nodig, dat slechts $O(t)$ tijd vraagt.

12.2 SUFFIXTABELLEN

Een suffixtabel ('suffix array', Manber en Myers, 1990) is een eenvoudiger alternatief voor een suffixboom. Ze ondersteunt dezelfde operaties, met dezelfde performantie, en bovendien vereist ze minder geheugen.

In principe is een suffixtabel niets anders dan een tabel met de ('alfabetisch') gerangschikte suffixen van de tekst T. Net zoals bij de bladeren van een suffixboom worden deze suffixen niet opgeslagen: de tabel bevat enkel hun beginindices. Bemerk dat een suffixtabel geen informatie over het gebruikte alfabet bevat, in tegenstelling tot een suffixboom.[1], maar vermindert de benodigde geheugenruimte. Voor (zeer) grote alfabetten is dat een voordeel, en een van de belangrijkste redenen om een suffixtabel te gebruiken.

Een suffixtabel construeren is niet ingewikkeld, tenzij men dat efficiënt wil doen. De te rangschikken suffixen bevatten immers $O(t^2)$ karakters. De klassieke oplossing construeert eerst de suffixboom (als daar tijdelijk voldoende plaats voor is), en bekomt

[1] Zoals bij een meerwegstrie vereist een efficiënte meerwegsbeslissing in elke knoop een tabel met de grootte $|\Sigma|$ van het alfabet. De boom vereist dan $O(|\Sigma|t)$ plaats, kan geconstrueerd worden in $O(t)$, en een prefix van lengte p vinden is $O(p)$. Sommige toepassingen van suffixbomen gebruiken strings met gehele getallen als 'karakters', zodat de geheugenvereisten buitensporig kunnen worden. Alternatieve implementaties van de meerwegsbeslissing, vergelijkbaar met wat we bij ternaire bomen gezien hebben, verlengen dan de constructietijd tot $\min(O(t \log t), O(t \log |\Sigma|))$, en de zoektijd tot $\min(O(p \log t), O(p \log |\Sigma|))$.

daarna de gerangschikte suffixen door hem in inorder te overlopen. Aangezien beide deeloperaties $O(t)$ zijn, is dat ook zo voor het geheel. Gezien het belang voor grote strings, waarbij suffixbomen te veel plaats innemen, is er actief onderzoek naar methodes met kleinere geheugeninname. Er zijn dan ook een aantal rechtstreekse $O(t)$-methodes gevonden, zie [32, 39].

Er is een belangrijke hulpstructuur die nuttig is bij het gebruik van een suffixtabel en dat is de LGP-tabel, waarin LGP staat voor **L**angste **G**emeenschappelijke **P**refix. Voor een gegeven suffix suff_i is LGP$[i]$ de lengte van het langste gemeenschappelijke prefix van suff_i met zijn opvolger in alfabetische volgorde[2] $\mathit{opvolger}(\mathit{suff}_i)$. In termen van de suffixtabel is, voor gegeven j, $\mathit{opvolger}\left(\mathit{suff}_{\mathrm{SA}[j]}\right) = \mathit{suff}_{\mathrm{SA}[j+1]}$.

Als we de suffixtabel hebben kunnen we vrij gemakkelijk de LGP-tabel opvullen in $O(t)$. We beginnen met het suffix suff_i met $i = 0$. In SA zoeken we zijn opvolger. Dit houdt dus in dat we eerst j bepalen met SA$[j] = 0$, zodat we de opvolger $\mathit{suff}_{\mathrm{SA}[j+1]}$ kennen. We bepalen nu het langste gemeenschappelijke suffix op de gewone manier: startend met $\ell = 0$ verhogen we ℓ tot we $T[i + \ell]$ niet meer overeenkomt met de overeenkomstige waarde van de opvolger. Op dat ogenblik is $\ell =$ LGP$[i]$. Voor de verdere waarden van LGP maken we gebruik van het staartprincipe. Daarmee kunnen we aantonen dat, voor $i < t - 1$, LGP$[i + 1] \geq$ LGP$[i] - 1$. Immers, stel dat LGP$[i] = \ell > 0$. Nu is $staart(\mathit{suff}_i) = \mathit{suff}_{i+1}$. Vermits de opvolger van suff_i een string is groter[3] dan suff_i, is $staart(\mathit{opvolger}(\mathit{suff}_i))$ een string groter dan suff_{i+1} die een prefix van lengte $\ell - 1$ gemeenschappelijk heeft met suff_{i+1}. Hij hoeft zelf niet de opvolger van suff_{i+1} te zijn, maar is minstens even groot en heeft dus hoogstens evenveel letters gemeen met suff_{i+1}. Bijgevolg moeten we, om LGP$[i + 1]$ te vinden, pas beginnen te vergelijken vanaf $\mathit{suff}_{i+1}[\ell - 1]$, wat overeenkomt met $T[i + \ell]$. Bijgevolg hebben we $O(t)$ vergelijkingen. Immers, als we twee karakters vergelijken en ze blijken gelijk te zijn dan verhogen we ℓ. Vermits altijd $i + \ell \leq t$ kan dit dus hoogstens t keer gebeuren. Als de twee karakters verschillen verhogen we daarentegen i, en dat gebeurt $t - 1$ keer. Als we ervoor zorgen dat de verdere behandeling van een suffix (zoeken van zijn opvolger, invullen van LGP) in $O(1)$ gebeurt hebben we dus een $O(t)$-methode.

Een suffixtabel kunnen we in $O(n)$ opbouwen uit een suffixboom. Door de boom diepte-eerst te overlopen komen we zijn bladeren tegen in alfabetische volgorde, zodat we de suffixtabel gewoon kunnen invullen. Omgekeerd kan een suffixboom vrij eenvoudig worden opgebouwd uit een suffixtabel. Hierbij maken we gebruik van de volgende eigenschap:

Is k een expliciete inwendige knoop van de suffixboom van T. Dan is er een suffix suff_i waarbij LGP$[i] = \ell$ *en zodanig dat de padstring voor k het prefix met lengte ℓ is van suff_i. Omgekeerd hoort bij elk suffix suff_i een knoop met padstring $T[i \ldots i +$* LGP$[i] - 1]$.

---

[2] In de praktijk gebruikt men meestal een tabel LCP die gerangschikt is in de volgorde gegeven door SA, zodat LGP[SA$[j]$] = LCP$[j]$. Conceptueel is echter de constructie van LGP eenvoudiger dan deze van LCP.

[3] In deze paragraaf bedoelen we met 'groter' steeds alfabetisch groter. Een string met meer letters dan een andere noemen we langer.

Merk op dat er verschillende suffixen kunnen zijn die op deze manier bij dezelfde knoop horen.

Een patroon (prefix) P opzoeken in een suffixtabel gebeurt met binair zoeken, waarvoor er $O(\lg t)$ *string*vergelijkingen nodig zijn (en dus $O(p \lg t)$ karaktervergelijkingen). Met extra voorzieningen en voorbereidend werk, kan het patroon echter gevonden worden in $O(p + \lg t)$ karaktervergelijkingen:

(1) Binair zoeken gebruikt een linkerindex ℓ, rechterindex r en een middelste index $c = \lfloor(\ell + r)/2\rfloor$ in de suffixtabel A. Stel dat p_ℓ de lengte is van het langste gemeenschappelijk prefix van P en suffix $A[\ell]$, en p_r de lengte van het langste gemeenschappelijk prefix van P en suffix $A[r]$. Noem p_{min} het minimum van p_ℓ en p_r. Dan heeft P een prefix met lengte p_{min} gemeen met alle suffixen tussen ℓ en r (inbegrepen), zodat de volgende stringvergelijking tussen P en het middelste suffix $A[c]$ kan beginnen bij het karakter $P[p_{min}]$, in plaats van vooraan.
Deze lengtes bepalen vergt weinig extra werk, maar vermijdt talrijke nutteloze testen. In de praktijk blijkt dat er gewoonlijk slechts $O(p + \lg t)$ karakters vergeleken worden, hoewel het slechtste geval nog steeds $O(p \lg t)$ blijft.

(2) Om ook in het slechtste geval $O(p + \lg t)$ te halen, is voorbereidend werk (op de tekst) vereist dat $O(t)$ bijkomende gegevens bepaalt. Dit zal ervoor zorgen dat elke iteratie van binair zoeken hooguit één karakter van P test dat vroeger reeds getest werd (een redundante test). Aangezien P uit p karakters bestaat en er $O(\lg t)$ iteraties zijn, wordt het streefdoel gehaald.
Om p_ℓ en p_r te bepalen moet de men de karakters van P testen tot één voorbij het *maximum* van die twee waarden, $P[p_{max}]$, zodat alle karakters van $P[p_{min}]$ tot en met $P[p_{max}]$ dan ook reeds getest zijn. Eén redundante test per herhaling van binair zoeken betekent dat de volgende test op P moet beginnen bij $P[p_{max}]$. Daartoe gebruikt elke iteratie de lengte $p_{\ell,c}$ van het langste gemeenschappelijk prefix van suffix $A[\ell]$ en het middelste suffix $A[c]$, of de lengte $p_{c,r}$ van het langste gemeenschappelijk prefix van suffix $A[c]$ en suffix $A[r]$. Al deze waarden kunnen *op voorhand* bepaald worden omdat de indices gebruikt door binair zoeken enkel afhangen van t (en niet van P). Bovendien zijn er slechts $O(t)$ waarden, zodat de geheugenvereisten slechts met een (kleine) constante factor toenemen. Die waarden kan men bepalen tijdens de constructie van de suffixtabel (ofwel bij het systematisch doorlopen van de suffixboom, ofwel rechtstreeks), maar ze kunnen ook in $O(t)$ afgeleid worden uit de suffixtabel zelf.
Die bijkomende gegevens worden als volgt gebruikt:

- Als $p_\ell = p_r$, dan is $p_{min} = p_{max}$ en is er slechts één redundante test.
- Als $p_\ell > p_r$, dan zijn er drie mogelijkheden:
  - Als $p_{\ell,c} > p_\ell$ dan zijn de eerste p_ℓ karakters van P en $A[c]$ gelijk, en aangezien P rechts van ℓ ligt is het patroonkarakter $P[p_\ell]$ groter dan het karakter van $A[\ell]$ op dezelfde positie, dat echter gelijk is aan het overeenkomstig karakter van $A[c]$. Dus weten we dat P rechts van c ligt, zonder enige test. We maken ℓ gelijk aan c, terwijl p_ℓ niet verandert.

  - Als $p_{\ell,c} < p_\ell$ (bedenk dat steeds $p_{\ell,c} \geq p_r$) dan is het karakter[4] $A[c][p_{\ell,c}]$ groter dan het overeenkomstig karakter van $A[\ell]$ ($A[c]$ ligt immers rechts van $A[\ell]$), dat gelijk is aan het overeenkomstig karakter van P. We weten dus dat P links ligt van c, zonder enige test. We maken nu r gelijk aan c, en p_r wordt $p_{\ell,c}$.
  - Als $p_{\ell,c} = p_\ell$ dan komen de eerste p_ℓ karakters van P en $A[c]$ reeds overeen. We moeten ze echter verder vergelijken vanaf $P[p_\ell]$, waarin $p_\ell = p_{max}$, om te beslissen of P links of rechts van c ligt. Enkel de eerste test is redundant, en als er meer testen gebeuren wordt p_{max} navenant verhoogd. Het laatst geteste karakter van P is dus steeds $P[p_{max}]$.

- Als $p_\ell < p_r$, zijn er drie analoge mogelijkheden, afhankelijk van $p_{c,r}$ en p_r.

Er zijn maximaal p niet-redundante testen op P mogelijk, en elk van de $O(\lg t)$ iteraties van binair zoeken verricht hoogstens één redundante test.

De waarden $p_{\ell,c}$ en $p_{c,r}$ zijn gemakkelijk af te leiden uit de LCP-tabel, vermits

$$p_{a,b} = \min_{a \leq i < b} \text{LCP}[i].$$

voor $b - a = 1$ nemen we dus gewoon LCP$[a]$ terwijl voor $b - a > 1$ we hebben dat, als m het midden is tussen a en b, $p_{a,b} = \min(p_{a,m}, p_{m,b})$, waarbij we gebruik maken van vorig bepaalde waarden.

Om voor alle operaties dezelfde performantie te bekomen als bij een suffixboom, moet een suffixtabel voorzien worden van nog meer bijkomende tabellen. Die bevatten eigenlijk dezelfde informatie als de suffixboom, maar hun constructie is behoorlijk ingewikkeld omdat het geheel competitief moet zijn qua tijd en plaats (zie [1]).

12.3 TEKSTZOEKMACHINES

Hiervoor gebruiken we het artikel van Zobel en Moffat [42].

- Op pagina 6 dient de zin 'An example is:', gevolgd door formule (1), vervangen te worden door

  An example is given by tf-idf (*term frequency – inverse document frequency*):

$$\begin{aligned} w_{q,t} &= \ln\left(\tfrac{N}{f_t}\right) \quad W_q = \sqrt{\textstyle\sum_t w_{q,t}^2} \\ w_{d,t} &= f_{d,t} \qquad W_d = \sqrt{\textstyle\sum_t w_{d,t}^2} \end{aligned} \tag{1}$$

$$S_{q,d} = \tfrac{\sum_t w_{q,t} \cdot w_{d,t}}{W_q \cdot W_d}$$

[4] We gebruiken $A[i][k]$ als korte notatie voor $\mathit{suff}_{SA[i]}[k]$. In de tekst is het uiteraard $T[SA[i] + k]$.

- Op pagina 21 in table III, in de kolom voor $b = 16$ is de code voor 4 foutief: deze moet 0 : 0011 zijn.

DEEL 4

HARDNEKKIGE PROBLEMEN

HOOFDSTUK 13
NP

13.1 COMPLEXITEIT: P EN NP

Alle tot hiertoe behandelde problemen hadden een efficiënte oplossing. Meer bepaald, hun uitvoeringstijd werd begrensd door een *veelterm* in het aantal behandelde gegevens n, zoals $O(n^2)$, of in het aantal knopen n en het aantal verbindingen m van een graaf, zoals bijvoorbeeld $O(n^2m)$. Er bestaan echter problemen waarvan men kan aantonen dat ze geen efficiënte oplossing kunnen hebben[1], of zelfs helemaal geen oplossing[2]. Erger nog, er bestaan talrijke praktische problemen waarvoor nog altijd geen efficiënte oplossing gevonden werd, maar waarvan men evenmin kon aantonen dat die niet bestaat. Niemand gelooft dat ook maar een van die problemen efficiënt oplosbaar is, gezien de jarenlange inspanningen in diverse vakgebieden, maar dit zogenaamde P-NP probleem blijft het belangrijkste openstaand theoretisch probleem in de computerwetenschappen.[3] Voor een vrij recente stand van zaken, zie Fortnow [14].

Problemen worden onderverdeeld in *complexiteitsklassen*, naar gelang van hun uitvoeringstijd, of hun geheugenvereisten. (In wat volgt hebben we het enkel over de uitvoeringstijd.) Om technische redenen beperkt men zich tot *beslissingsproblemen*, die 'ja' of 'nee' als resultaat opleveren. Dat is niet echt een beperking, omdat elk probleem wel geformuleerd kan worden als een beslissingsprobleem, en bovendien, als het beslissingsprobleem geen efficiënte oplossing heeft, dan geldt dat zeker voor het originele probleem.

De klasse P bevat alle problemen waarvan de uitvoeringstijd begrensd wordt door een *veelterm* in de grootte van het probleem, bij uitvoering op een 'realistisch' computermodel. (De 'P' van 'Polynoom, Polynomiaal'.) Met 'grootte' wordt het aantal bits bedoeld dat nodig is om de invoergegevens in een computer voor te stellen, waarbij een 'redelijke' voorstelling gebruikt wordt. Een redelijke voorstelling is compact, bevat geen overtollige informatie, en stelt getallen voor in een talstelsel met radix groter dan één. (Een onredelijke voorstelling van een geheel getal met waarde n zou bijvoorbeeld n bits gebruiken.) Een realistisch computermodel heeft een polynomiale bovengrens voor het werk dat in één tijdseenheid kan verricht worden (zoals het vaak gebruikte 'RAM'-model). Al de problemen in P worden als efficiënt oplosbaar beschouwd.

[1] Problemen waarvan de oplossing zelf zo omvangrijk is dat ze onmogelijk efficiënt kan geproduceerd worden, sluiten we hier uit.

[2] Zoals het befaamde 'halting' probleem (Turing, 1936).

[3] Er is zelfs een belangrijke prijs aan verbonden (zie `www.claymath.org`).

Waarom gebruikt men veeltermen om het onderscheid te maken met niet-efficiënt oplosbare problemen? Want $O(n^{100})$ kan bezwaarlijk efficiënt in ce praktijk genoemd worden. Daar zijn drie redenen voor:

(1) Als de uitvoeringstijd niet kan begrensd worden door een veelterm, dan is het probleem zeker niet efficiënt oplosbaar. Bovendien is de graad van de veelterm gewoonlijk heel beperkt (zelden meer dan twee of drie).
(2) Veeltermen vormen de kleinste klasse functies die kunnen gecombineerd worden, en opnieuw veeltermen opleveren. Hun klasse is 'gesloten' onder de operaties optelling, vermenigvuldiging, en samenstelling: een som of product van twee veeltermen is een veelterm, en een veelterm van een veelterm blijft een veelterm. Efficiënte algoritmen voor eenvoudiger problemen kunnen aldus gecombineerd worden tot een efficiënt algoritme voor een meer complex probleem.
(3) De efficiëntiemaat blijft onafhankelijk van het computermodel en van de voorstelling van de invoer die men gebruikt om de uitvoeringstijd te bepalen. Als die tijd voor een bepaald model en voorstelling begrensd wordt door een veelterm, zal dat ook zo zijn bij alle andere (zolang ze 'realistisch' blijven).

Bekijken we nu een hypothetische computer met oneindig veel processoren. Elke processor kan in een tijdsstap een bepaald aantal (laat ons zeggen k) andere processoren aanspreken. Dit levert in t tijdsstappen op dat we t^k processoren aan het werk kunnen hebben (dus exponentieel in de tijd). Deze processoren werken echter niet samen; wel kunnen ze elk hun deel van het probleem oplossen. Dergelijke computer noemen we een niet-deterministische computer.

We hebben nu een beslissingsprobleem (een ja/nee-vraag) van de vorm 'Bestaat er een ...?' Als we deze vraag kunnen oplossen door voor elke kandidaat-oplossing te kijken of ze aan alle voorwaarden voldoet kunnen we deze aan een niet-deterministische computer geven. Als het opsplitsen van de verzameling kandidaten (door delen ervan aan andere processoren te geven) én het controleren van één kandidaat beide in polynomiale tijd kunnen gebeuren, dan kan een niet-deterministische computer het antwoord geven in polynomiale tijd en dan noemen we het probleem niet-deterministisch polynomiaal of kortweg NP[4]

Elk probleem uit P behoort dus zeker tot NP. Maar hoe ongelooflijk irrealistisch deze machinekeuze ook lijkt, toch is men er nog altijd niet in geslaagd om aan te tonen dat ze krachtiger is dan een 'klassieke' deterministische machine: we zijn niet zeker of NP-problemen bevat die niet in P zitten, Dit is het P-versus-NP-probleem. Wel is het zeker dat er problemen zijn die *niet* in NP zitten, en dus zeker niet in P[5].

[4] Strikt genomen is voor NP het voldoende dat een *geschikte* kandidaat in polynomiale tijd kan gecontroleerd worden terwijl ongeschikte kandidaten oneindig veel tijd vragen. Dit houdt verband met het zogenaamde *halting problem.*

[5] Een voorbeeld hiervan vinden we bij bepaalde veralgemeende bordspelen zoals veralgemeend schaken of dammen, waarbij de bordgrootte een parameter is van de veralgemening. Het is bekend dat bij veel van die spelen het de grootte langst mogelijke spel exponentieel toeneemt met de bordgrootte, wat tot exponentiële problemen leidt die niet in NP zitten.

Vaak kan een soort problemen omgezet worden. Dit hebben we al herhaalde malen gedaan: zo hebben we bepaalde koppelingsproblemen omgezet naar stroomnetwerk-problemen. Dergelijke omzetting noemen we een *reductie.* Zonder het verder te vermelden gaan we er altijd van uit dat een reductie kan gebeuren in polynomiale tijd en plaats. Als we een probleem X kunnen reduceren naar een probleem Y dan noemen we Y minstens even zwaar als X. Het is niet moeilijk om in te zien dat er problemen zijn die minstens even zwaar zijn als *elk* NP-probleem. Er is een probleem dat triviaal NP-hard is: gegeven een niet-deterministische computer en een instantie van een NP-probleem, antwoordt dan de computer 'ja' op de vraag? Dergelijke problemen, die de klasse NP als het ware omvatten, noemen we NP-hard.

Wat wel verrassend is: er zijn NP-harde problemen die zeker in NP zitten. Dergelijke problemen noemt men NP-*compleet*. Essentieel bewijst men dat er een NP-compleet probleem bestaat door een NP-probleem (in het originele geval SAT, zie [10]) te nemen en een reductie te geven van het bovenstaande probleem naar dit probleem. Eenmaal dat bekend was dat er een NP-compleet probleem was werd het duidelijk dat heel wat NP-problemen NP-compleet waren: NP-problemen hebben een typische structuur die onmiddellijk aan backtracking doen denken, waardoor het vaak vrij gemakkelijk door reductie van het ene naar het andere probleem over te gaan.

Als er ook maar één NP-compleet probleem efficiënt oplosbaar zou zijn (en dus tot P zou behoren) dan zouden *alle* problemen uit NP ook efficiënt oplosbaar zijn, zodat NP gelijk wordt aan P. Aangezien er veel NP-complete problemen zijn, uit tal van vakgebieden, waarvoor reeds lang tevergeefs efficiënte oplossingen gezocht worden, denkt veel specialisten dat dit onmogelijk is.

Wie geconfronteerd wordt met een bepaald probleem en kan bewijzen dat dit nieuwe probleem NP-compleet is, weet dat men beter niet op zoek gaat naar een efficiënte oplossing, maar neemt dat het probleem natuurlijk niet weg. Er zijn dan een aantal mogelijkheden:

- Als de afmetingen van het probleem *klein* zijn (een gering aantal invoergegevens), kan men toch alle mogelijkheden onderzoeken (via backtracking), en daarbij trachten het werk zoveel mogelijk te beperken (snoeien).
- Misschien bestaan er efficiënte algoritmen om *speciale gevallen* van het probleem op te lossen.
- De begrenzing voor de uitvoeringstijd is voor het slechtste geval. Het is best mogelijk dat de *gemiddelde* uitvoeringstijd goed meevalt.
- Voor verscheidene NP-complete problemen bestaan *benaderende algoritmen*, die de vereisten voor de oplossing wat relaxeren. Vaak is het verband gekend tussen de mate van benadering en de uitvoeringstijd. Benaderingen van NP-complete problemen zijn echter soms zelf NP-compleet.
- Tenslotte kan men efficiënte *heuristische methoden* gebruiken die meestal een goede maar niet noodzakelijk optimale oplossing vinden. Ze garanderen echter niets.

13.2 NP-COMPLETE PROBLEMEN

In deze paragraaf geven we een kort overzicht van enkele belangrijke NP-complete problemen. Een aantal van de problemen zijn gepresenteerd als optimalisatieproblemen. Zo is bijvoorbeeld *Minimum Cover* het probleem van het vinden van de kleinste verzameling die aan bepaalde voorwaarde voldoet; het overeenkomstige NP-probleem is dan of een er een Cover bestaat van een gegeven grootte. Wie een probleem heeft dat onmiddellijk kan omgevormd worden tot een van deze standaardproblemen vindt gemakkelijk bijbehorende informatie over benaderende oplossingen, heuristieken e.d.m. Voor meer informatie verwijzen we naar het klassieke standaardwerk van Garey en Johnson [17], en naar het recentere overzicht van Skiena [37].

Om na te gaan of een probleem NP-compleet is zoekt men het eerst op in dergelijke referentiewerken. Daarbij moet het probleem ontdaan worden van allerlei concrete aspecten, om het tot zijn kern te herleiden. Is het daar niet te vinden, dan kan men trachten om een bekend NP-compleet probleem ertoe (polynomiaal) te reduceren. Goede kandidaten daarvoor zijn 3SAT, Vertex cover, Independent Set, (Integer) Partition, Clique, en Hamiltonian circuit: van de meeste andere problemen uit de lijst is bewezen dat ze NP-compleet zijn door expliciete reducties van een standaardprobleem naar deze problemen.

13.2.1 Het basisprobleem: SAT (en 3SAT)

Gegeven een verzameling logische variabelen $\mathcal{X} = \{x_1, \ldots, x_{|\mathcal{X}|}\}$, en een collectie logische uitspraken $\mathcal{F} = \{F_1, \ldots, F_{|\mathcal{F}|}\}$. Elke uitspraak bestaat uit atomaire uitspraken (atomen): de x_i zelf of de inverse (genoteerd als $\overline{x_i}$), samengevoegd met ofoperaties, zoals:

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

Kan men waarden toekennen aan de variabelen, zodat al deze uitspraken tegelijk waar zijn?

Cook heeft aangetoond dat elk probleem uit NP polynomiaal kan gereduceerd worden tot SAT: het was dus het eerste bekende NP-compleet probleem. Het werd dan gebruikt om aan te tonen dat andere problemen ook NP-compleet zijn.

Het is gemakkelijk in te zien dat we een uitspraak met meer dan drie atomen kunnen herleiden naar een reeks uitspraken met elk drie atomen zodat de uitspraak waar is als en slechts als de 3-uitspraken waar zijn. Dit doen we iteratief door telkens de eerste twee atomen af te splitsen en door een nieuwe variabele toe te voegen. Zo kunnen we de bovenstaande uitspraak opsplitsen in

$$x_2 \vee \overline{x_5} \vee x_n$$
$$\overline{x_n} \vee x_7 \vee x_8$$

waarin x_n de nieuwe variabele is. Dit leidt tot het NP-probleem 3SAT, waarbij elke uitspraak hoogstens drie atomen mag hebben.

13.2.2 Graafproblemen

13.2.2.1 Vertex cover

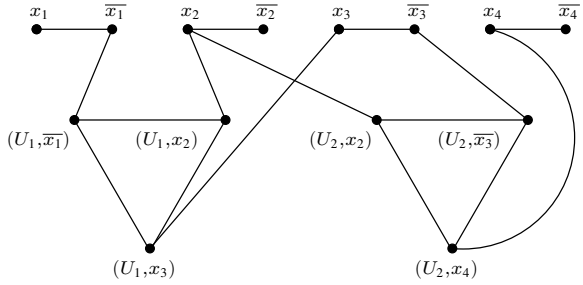
Gegeven een ongerichte graaf. Gevraagd een kleinste groep knopen die minstens één eindknoop van elke verbinding bevat. Het is een relatief gemakkelijk NP-compleet probleem, nauw verwant met 'edge cover': een kleinste groep verbindingen waarin elke knoop voorkomt, al is het edge coverprobleem P. Ook nauw verwant met 'independent set' (zie hieronder): de verzameling knopen min een vertex cover is een independent set.

3SAT can gemakkelijk gereduceerd worden tot vertex cover. Hebben we een instantie van 3SAT dan construeren we een graaf erbij als volgt:

- Voor elke logische variabele x_i maken we twee knopen, een voor x_i en een voor zijn negatie $\overline{x_i}$. Deze verbinden we met elkaar.
- voor elke uitspraak maken we drie knopen, één voor elk atoom. Deze verbinden we met elkaar en met het bijbehorende atoom uit het vorige item.

In Figuur 13.1 zien we de graaf voor het probleem met twee uitspraken

$$\begin{aligned} U_1 &= \overline{x_1} \vee x_2 \vee x_3 \\ U_2 &= x_2 \vee \overline{x_3} \vee x_4. \end{aligned}$$



Figuur 13.1. Een graaf voor een 3SAT-probleem

We zoeken nu een vertex cover voor zulk een graaf. Voor elke logische variabele moeten we al zeker een van de twee knopen x_i of $\overline{x_i}$ opnemen om de verbinding tussen die twee te overdekken. Voor elke uitspraak moeten we al zeker twee van de drie

knopen opnemen om de drie onderlinge verbindingen te overdekken. We hebben dus al minstens $|\mathcal{X}| + 2|\mathcal{F}|$ knopen nodig. Dit aantal is ook voldoende als en slechts als het 3SAT-proleem een oplossing heeft. Immers, als we voor de ware atomen van een 3SAT-oplossing de overeenkomstige knoop in de vertex cover steken, dan is voor elke uispraak minstens één knoop verbonden met een waar atoom en kunnen we de andere twee knopen nemen om de verbindingen binnen de uitspraak te overdekken. Omgekeerd, hebben we een vertex cover van grootte $|\mathcal{X}| + 2|\mathcal{F}|$ dan leidt die duidelijk tot een oplossing van het 3SAT-probleem.

13.2.2.2 Dominating set

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

13.2.2.3 Graph coloring

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

Komt voor bij diverse plannings- en clusteringstoepassingen.

De enige oplossingsmethode is backtracking, die verrassend goed kan werken voor bepaalde random grafen. Ook benaderende oplossingen vinden met een bekende benadering blijft NP-compleet: men moet beroep doen op heuristische methoden.

Eenvoudige gevallen zijn wel efficiënt oplosbaar: tweeledige grafen (met twee kleuren), planaire grafen, en wanneer alle knopen een lage graad hebben.

Edge coloring: Gegeven een ongerichte graaf. Kleur de verbindingen met een minimum aantal kleuren, zodat verbindingen met een gemeenschappelijke eindknoop een verschillende kleur hebben.

Komt voor bij diverse planningstoepassingen.

Het benaderend $O(n^2)$ algoritme van Vizing gebruikt hoogstens één kleur teveel. Een exacte oplossing zoeken is dus vaak niet de moeite waard.

13.2.2.4 Clique

Gegeven een ongerichte graaf. Gevraagd de grootste groep knopen die allemaal met elkaar verbonden zijn.

Wordt gebruikt om clusters van verwante objecten op te sporen.

De enige oplossingsmethode is backtracking. Er zijn geen efficiënte benaderende algoritmen bekend: het is een van de lastigste problemen. Heuristische methoden zoals simulated annealing werken vaak redelijk goed.

Eenvoudige gevallen zijn wel efficiënt op te lossen: een maximale clique zoeken (die niet meer kan uitgebreid worden), grote dichte deelgrafen zoeken (dus niet noodzakelijk compleet), en cliques in planaire grafen (die bevatten geen clique groter dan vier).

Het is vrij gemakkelijk om SAT te herleiden tot clique. Voor elk voorkomen van een atoom in een uitspraak maken we een bijbehorende knoop (F_i, x_j) dan wel $(F_i, \overline{x_j})$ en leggen volgende verbindingen tussen knopen horende bij *verschillende* uitspraken (dus $i \neq j$) als de atomen elkaar niet uitsluiten:

- (F_i, x_k) met (F_j, x_ℓ) voor alle k en ℓ.
- $(F_i, \overline{x_k})$ met $(F_j, \overline{x_\ell})$ voor alle k en ℓ.
- (F_i, x_k) met $(F_j, \overline{x_\ell})$ als en slechts als $k \neq \ell$.

Een clique kan hoogstens $|\mathcal{F}|$ elementen bevatten, want knopen horende bij dezelfde uitspraak zijn niet verbonden. Een clique met exact $|\mathcal{F}|$ elementen leidt duidelijk tot een oplossing van SAT (maak alle atomen horende bij leden van de clique waar; de waarde van andere logische variabelen mag willekeurig gekozen worden). Niet elke oplossing van SAT leidt noodzakelijk rechtstreeks tot een clique, omdat dit te veel knopen kan opleveren. Echter, binnen deze verzameling knopen zit een clique van grootte $|\mathcal{F}|$.

13.2.2.5 Independent set

Gegeven een ongerichte graaf. Gevraagd de grootste groep knopen zonder onderlinge verbindingen.

Komt voor wanneer men conflicten tussen elementen wil vermijden, zoals in codeertheorie, of bij planningsproblemen.

Is nauw verwant met kleuren van knopen (verdeelt een graaf in een klein aantal groepen van knopen die onafhankelijk zijn), en identiek aan zoeken van een clique in de complementaire graaf (heeft alle verbindingen die de originele niet heeft, en omgekeerd). Is in zekere zin duaal met het maximaal koppelingsprobleem in grafen (de grootste groep verbindingen zonder gemeenschappelijke knopen), en kan soms zo geformuleerd (en efficiënt opgelost) worden.

Zoals bij clique, is een maximale independent set een eenvoudiger probleem.

13.2.2.6 Hamiltonian circuit/path

Gegeven een al dan niet gerichte graaf. Bestaat er een circuit dat elke knoop eenmaal bevat?

Nauw verwante problemen zijn zoeken naar de langste weg of het langste circuit, beide ook NP-compleet, zelfs voor beperkte klassen van ongewogen grafen.

De enige oplossingsmethode is backtracking. (Test eerst of de graaf dubbel lijnsamenhangend is.)

Eenvoudiger gevallen zijn gemakkelijker op te lossen: een circuit dat elke verbinding eenmaal bevat (een Eulercircuit), knopen meermaals bezoeken maar zo weinig mogelijk, dichte grafen waarin snel een circuit kan gevonden worden.

Ook zoeken van een Hamiltonweg (bevat ook alle knopen, maar is niet gesloten) in een al dan niet gerichte graaf is NP-compleet. (Het is echter efficiënt oplosbaar voor een gerichte graaf zonder lussen.)

13.2.3 Problemen bij verzamelingen

13.2.3.1 Minimum cover

Gegeven een verzameling $\mathcal{S}$ en een collectie deelverzamelingen $\mathcal{C}$ van $\mathcal{S}$. Gevraagd de kleinste deelverzameling $\mathcal{C}'$ van $\mathcal{C}$ zodat elk element van $\mathcal{S}$ tot minstens een van de deelverzamelingen uit $\mathcal{C}'$ behoort.

Een aantal andere problemen kan in deze vorm gegoten worden:

- **Vertex cover.** $\mathcal{S}$ is de verzameling verbindingen; $\mathcal{C}$ is de verzameling knopen, waarbij elke knoop wordt opgevat de verzameling van de verbindingen die hij heeft.
- **Edge cover.**
- **Dominating set.** $\mathcal{S}$ is de verzameling knopen. Voor elke knoop is er een element van $\mathcal{C}$ bestaande uit de knoop met al zijn buren.

Minimum cover kan efficiënt opgelost worden wanneer de deelverzamelingen in $\mathcal{C}$ niet meer dan twee elementen bevatten. Als elke $C \in \mathcal{C}$ exact twee elementen bevat is het duidelijk equivalent met edge cover; als er $C \in \mathcal{C}$ zijn met maar één element kan men deze (voorlopig) schrappen, het edge coverprobleem oplossen en eventueel nodige singletons terug toevoegen.

Het is gemakkelijk om SAT te herleiden naar minimum cover. Neem $S = \mathcal{F} \cup \mathcal{X}$. Voor

elke logische variabele x_i zijn er twee elementen van $\mathcal{C}$

$$\begin{aligned} C_{x_i} &= \{x_i\} \cup \{F \in \mathcal{F} : x_i \in F\} \\ C_{\overline{x_i}} &= \{x_i\} \cup \{F \in \mathcal{F} : \overline{x_i} \in F\} \end{aligned}$$

Het is duidelijk dat het SAT-probleem een oplossing heeft als en slechts als er een minimum cover is van grootte $|\mathcal{X}|$.

Het gebeurt vrij vaak dat $\mathcal{S}$ uiteenvalt in twee verzamelingen $\mathcal{S} = \mathcal{F} \cup \mathcal{X}$ waarbij elk element van $\mathcal{C}$ juist één element van $\mathcal{X}$ bevat en we een cover zoeken met grootte $|\mathcal{X}|$. We kunnen dan $\mathcal{X}$ opvatten als een lijst van variabelen x_i die een waarde moeten krijgen, waarbij een element $C \in \mathcal{C}$ overeenkomt met een keuze van een waarde voor die x_i waarvoor $x_i \in C$. Bij SAT is elke x_i een logische variabele met twee mogelijke waarden.

Een voorbeeld hiervan wordt gegeven door betegelingsproblemen (*tiling puzzles*). Hierbij hebben we een speelbord $\mathcal{F}$ met vakjes, vaak een deel van een oneindig raster zoals een schaakbord, en een aantal tegels van verschillende of gelijke vorm. Bedoeling is om het bord te bedekken met de tegels. Met elke $C \in \mathcal{C}$ komt dan een toegelaten positie van een tegel op het bord overeen: C bevat zowel de tegel als de bedekte vakjes.

Een minimum coverprobleem kan beschreven worden door een 0/1-matrix met $|\mathcal{S}|$ rijen en $|\mathcal{C}|$ kolommen. Als een probleem klein genoeg is kan het opgelost worden met backtracking. Gebruik van de *dancing links*-implementatie voor de matrix zorgt voor voor een efficiënte structuur voor de matrix.

13.2.3.2 Subset sum

Gegeven een verzameling elementen, elk met een positieve gehele grootte, en een positief geheel getal k. Bestaat er een deelverzameling van die elementen, zodat de som van hun grootten gelijk is aan k?

Oplosbaar in pseudopolynomiale tijd, met dynamisch programmeren.

13.2.3.3 Partition

Gegeven een zak (een verzameling waarbij meerdere keren hetzelfde element kan voorkomen) van positieve gehele getallen. Kan die opgesplitst worden in twee deelverzamelingen, zodat de som van de grootten van de ene gelijk is aan de som van de grootten van de andere?

Ook als we vereisen dat beide deelverzamelingen even groot zijn, blijft het probleem NP-compleet. Het is evenwel oplosbaar in pseudopolynomiale tijd, met dynamisch programmeren.

13.2.4 Netwerken

13.2.4.1 TSP

Gegeven een aantal steden, en de afstand tussen elk paar steden. Gevraagd het kortste circuit dat elke stad eenmaal bevat.

Dit is veruit het meest bekende en ook meest bestudeerde NP-compleet probleem (zie bijvoorbeeld [3]). Hoewel transporttoepassingen voor de hand liggen, is de belangrijkste toepassing het minimaliseren van de bewegingen van machines.

Bijna elke versie van dit probleem is NP-compleet. Heuristische methoden zijn aangewezen, en het resultaat is vaak zeer goed (enkele procenten onder het optimum).

Wanneer de steden punten in het vlak zijn en de Euclidische[6] afstand (L_2) gebruikt wordt heeft men het geometric TSP probleem. Dat is eveneens NP-compleet, ook met andere afstandsdefinities die aan de driehoekseigenschap voldoen (zoals L_1 en L_∞).

13.2.4.2 Longest path

Gegeven een gerichte of ongerichte gewogen graaf, en twee knopen s en t. Gevraagd de langste simpele weg (zonder lussen) van s naar t.

Het probleem blijft NP-compleet, zelfs als alle gewichten gelijk zijn aan één. Het is efficiënt oplosbaar voor gerichte lusloze grafen (dags).

Zoals bekend is het probleem van de *kortste* simpele weg in een graaf efficiënt oplosbaar, tenzij de gewichten negatief mogen zijn, en er negatieve lussen aanwezig zijn.

13.2.5 Gegevensopslag

13.2.5.1 Bin packing

Gegeven een verzameling van n objecten met afmetingen $s_1, \ldots, s_n$, en een verzameling van m bakken met capaciteiten $c_1, \ldots, c_m$. Gevraagd alle objecten op te slaan in zo weinig mogelijk bakken. Er kunnen beperkingen opgelegd worden aan oriëntatie en plaatsing van de objecten.

Komt voor bij verpakkings- en fabricatieproblemen.

Zelfs de eenvoudige versies zijn NP-compleet. Gelukkig zijn er relatief eenvoudige heuristische methoden die meestal goed werken. Voor het eendimensionaal geval is

[6] Voor twee punten p_1 en p_2 met coördinaten (x_1, y_1) en (x_2, y_2) definieert men de Minkowski metriek $L_m(p_1, p_2) = \sqrt[m]{|x_1 - x_2|^m + |y_1 - y_2|^m}$. L_2 is de Euclidische afstand, L_1 de 'Manhattan distance', en $L_\infty(p_1, p_2) = \max(|x_1 - x_2|, |y_1 - y_2|)$.

dat 'first-fit, decreasing', dat $O(n \lg n)$ is en nooit meer dan 22% bakken (met dezelfde grootte) teveel gebruikt. Een variant garandeert zelfs minder dan 18% teveel.

13.2.5.2 Knapsack

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

Komt voor bij financiële beperkingen: zoveel mogelijk waar(de) verkrijgen voor een beperkt budget.

De meest voorkomende versie is het '$0/1$ knapsack problem', waarbij objecten niet mogen opgedeeld worden. Als dat wel mag, dan bestaat er een eenvoudig inhalig algoritme dat de optimale oplossing vindt. Als de afmetingen relatief kleine gehele getallen zijn, dan kan de optimale oplossing gevonden worden met dynamisch programmeren. Nog eenvoudiger wordt het wanneer elk object dezelfde waarde heeft, of dezelfde kost. Wanneer de 'prijs per meter' voor elk object gelijk is, kan men de waarde verwaarlozen en trachten de knapzak zo goed mogelijk op te vullen, maar ook dat probleem is NP-compleet.

HOOFDSTUK 14
METAHEURISTIEKEN

Heuristieken zijn vuistregels bij het zoeken naar een oplossing van een probleem. In tegenstelling tot wat de naam doet vermoeden (die komt van het Griekse $\beta\rho\acute{\iota}\sigma\kappa\omega$: ik vind) garanderen ze niet altijd dat een oplossing gevonden wordt, maar ze versnellen meestal wel de zoektocht ernaar. Er zijn veel optimalisatieproblemen van groot praktisch belang, voornamelijk in de logistiek, waarvoor het zoeken van de beste oplossing te veel rekentijd vergt om doenbaar te zijn. In principe kan men wel alle mogelijkheden afgaan en de beste oplossing daaruit kiezen, maar dit is niet praktisch haalbaar. Sommige van de NP-problemen uit het vorige hoofdstuk hebben een naam die direct naar zo'n praktische situatie verwijst, zoals bin packing en TSP. Toen computers voor het eerst op grotere schaal ter beschikking kwamen is men begonnen voor afzonderlijke problemen benaderende algoritmen op te stellen die binnen een redelijke tijd een redelijk individu gaven. Sommige daarvan zijn alleen toepasbaar op het specifieke probleem, zoals *first-fit, decreasing* bij bin packing, maar het bleek dat een aantal methodes meer algemeen toepasbaar waren. Hieruit is het veld van de *metaheuristieken* ontstaan, waar men deze algemene methodes systematisch ging bestuderen.

14.1 COMBINATORISCHE OPTIMISATIE

Om een algemeen kader te hebben moeten we eerst een abstracte karakterisering hebben van het soort problemen dat we willen oplossen. Optimisatie houdt in dat we uit een gegeven verzameling $\mathcal{S}$ een individu halen dat een 'beste' element is uit die verzameling. Hierbij is $\mathcal{S}$ een eindige verzameling van strings over een eindig alfabet. Meestal wordt $\mathcal{S}$ niet gekarakteriseerd door opsomming, maar wel door een aantal voorwaarden te geven waaraan strings in $\mathcal{S}$ moeten voldoen. Het beste individu van $\mathcal{S}$ wordt bepaald door een *evaluatiefunctie* f. We gebruiken de conventie dat de beste waarde de kleinste is.

Een klein voorbeeld zal veel duidelijk maken. We nemen TSP. Het alfabet is hier de verzameling verbindigen. De verzameling $\mathcal{S}$ bestaat uit een reeks verbindingen zó dat elke verbinding vertrekt uit het eindpunt van de vorige (de eerste verbinding vertrekt uit het eindpunt van de laatste) en zó dat alle steden bezocht worden. Het beste individu is die met de kortste lengte.

Vrij vaak zijn alle strings in $\mathcal{S}$ even lang. In dat geval kunnen we elementen van $\mathcal{S}$ ook beschrijven door een aantal variabelen in te vullen. Elke letter van een $s \in \mathcal{S}$ geeft dan

de waarde aan van een variabele; een voorwaarde is natuurlijk dat die letter behoort tot het domein van de overeenkomstige variabele. Dat is bijvoorbeeld het geval bij het lessenroosterprobleem. Hierbij zijn er een aantal lessen, een aantal docenten, een aantal klaslokalen, een aantal mogelijke tijdsslots en een aantal groepen studenten. Bij elke les horen een of meerdere docenten, een of meerdere klasgroepen enzovoorts. Een lessenrooster is goed als het weinig negatieve elementen heeft (springuren, onevenwichtige verdelingen voor groepen docenten en groepen studenten enzovoorts. Elke string van $\mathcal{S}$ is even lang en beschrijft een lessenrooster zonder conflicten door aan elke les een lokaal en een tijdsslot toe te kennen. $\mathcal{S}$ heeft dus twee letters voor elke les: de eerste duidt de tijdsslot aan en de tweede het lokaal.

Onze beschrijving van $\mathcal{S}$ maakt duidelijk dat we $\mathcal{S}$ in principe kunnen aflopen met backtracking. We kunnen de hier beschreven methodes dus samenvatten door te stellen dat ze heuristieken zijn om een beste individu te benaderen voor een probleem dat te groot is voor backtracking. Bij alle metaheuristieken wordt er daarvoor een steekproef genomen uit de totale verzameling $\mathcal{S}$. Voor elk individu wordt de f-waarde berekend, waarbij het beste individu dat we ooit ontmoet hebben bewaard wordt. De methode gaat verder tot een bepaalde stopvoorwaarde bereikt is. Vaak wordt er gestopt omdat de tijd op is of omdat de gevonden oplossing zo goed is dat er niet veel meer verbetering verwacht wordt[1]. De verschillende metaheuristieken verschillen in de manier waarop individuen worden uitgekozen om te bekijken.

Er is een zeer groot aantal metaheuristieken. Sommige daarvan zijn varianten op een andere; hier en daar gebruikt een metaheuristiek geheel eigen ideeën en methodes. Niet elke metaheuristiek zal even geschikt zijn voor een gegeven probleem zodat de keuze moeilijk is. Bovendien zal het nodig zijn om het probleem in te passen in de gekozen methode en anderzijds kan het nuttig zijn de deze methode aan te passen aan het probleem. Een klein voorbeeld: bij veel problemen kan het zinvol zijn een gevonden individu aan te passen met kleine verbeteringen. Bijna elke metaheuristiek leent zich ertoe om dergelijke lokale optimisaties door te voeren, ook als dat, zoals bijvoorbeeld bij genetische algoritmes het geval is, niet expliciet ingebakken is. Het is dan in elk geval zinvol om een versie met en een versie zonder lokale optimisatie met elkaar te vergelijken.

De meeste uiteenzettingen over metaheuristieken gaan uit van een indeling van metaheuristieken aan de hand van bepaalde eigenschappen. Dat is niet zeer praktisch voor wie met een bepaald probleem geconfronteerd wordt. In deze cursusnota's is ervoor gekozen om zeer veel aandacht te besteden aan de eigenschappen van op te lossen problemen en om te kijken welke metaheuristieken het beste aansluiten bij deze eigenschappen.

[1] In de rest van dit hoofdstuk vermelden we het bewaren van het beste al gevonden individu niet meer.

14.2 VOORONDERSTELLINGEN

De methodieken die we gaan bekijken werken niet voor alle mogelijke problemen. Er zijn voorwaarden die moeten voldaan zijn om ze zinvol te maken.

Eén ervan is dat het zinvol moet zijn om op zoek te gaan naar betere individuen in de buurt van een gegeven individu. Om het belang daarvan duidelijk te maken vergelijken we even hoe we aan individuen komen bij backtracking met deze bij de meeste metaheuristieken. Backtracking maakt gebruik van kandidaat-deeloplossingen. Deze worden uitgebreid tot langere strings om zo individuen van $\mathcal{S}$ te bekomen. Bij de meeste heuristieken is dat *niet* het geval. Omdat het niet mogelijk is om de gehele verzameling $\mathcal{S}$ te bekijken moet men een selectie maken. Men gaat steeds nieuwe individuen van $\mathcal{S}$ bekijken, meestal tot de tijd op is. Van al de bekeken individuen houdt men het beste over. Bij het construeren van elementen zal men bijna steeds een zekere mate van randomisatie gebruiken. Het kan zijn dat we een individu van de grond af opbouwen (dat is zeker zo bij het eerste individu dat men bekijkt, natuurlijk), maar ook dat we pogen individuen aan te maken die op een of andere gelijkenis vertonen met al bekeken individuen.

Soms is het niet zinvol om, bij het opbouwen nieuwe individuen van $\mathcal{S}$, uit te gaan van de reeds gevonden individuen. Bij sommige problemen is dat niet het geval en dan blijft er niet veel meer over dan het nemen van een random steekproef zoals gegeven in pseudocode 14.1, waarin f de evaluatiefunctie voorstelt. Voor we echter gaan kijken

```
S RandomZoekBeste(){
    S beste=maakRandom();
    while (nog tijd){
        S nieuw=maakRandom();
        if (f(nieuw) < f(beste))
            beste=nieuw;
    };
    return beste;
};
```

Code 14.1. Random zoeken van beste individu

hoe we nuttige informatie kunnen halen uit al bezochte individuen bespreken we eerst de autonome aanmaak van nieuwe elementen.

Als men een nieuw element van de grond opbouwt zijn er twee belangrijke mogelijkheden:

(1) Constructie: het individu wordt opgebouwd uit componenten. Deze componenten komen overeen met de letters van de strings in $\mathcal{S}$. In tegenstelling tot backtracking

gaat men hier niet alle uitbreidingsmogelijkheden bekijken maar kiest men er één uit. Dit gebeurt at random, al kan er een heuristiek zijn die de keuze beïnvloedt. Een voorbeeld: bij een Covering Setprobleem voegen we steeds deelverzamelingen toe tot we een cover hebben. Dit is de meestgebruikte methode. Sommige metaheuristieken eisen dat deze methode gebruikt wordt.

(2) Individuen worden rechtstreeks aangemaakt.

In sommige gevallen zijn beide mogelijk: bij TSP kan men een individu beschouwen als een pad in een graaf. Men begint dan van een leeg pad en voegt verbindingen toe. Maar men kan ook een individu beschouwen als een permutatie van de verzameling winkels: zulk een permutatie kan volledig random worden aangemaakt.

Methodes zoals Simulated Annealing gebruiken maar één startindividu, maar veel andere methodes hebben verschillende startindividuen nodig en dan is het nuttig om een zekere randomisatie te hebben. Dat kan trouwens ook bij Simulated Annealing nuttig zijn: de procedure herhaalde malen starten met verschillend startindividu geeft een idee of de gekozen parameters stabiel leiden tot een goed resultaat. Bij constructiemethodes hebben we telkens een deeloplossing. Misschien kunnen sommige componenten niet meer zinvol toegevoegd worden (bijvoorbeeld bij TSP: het heeft geen zin een al bezochte winkel nogmaals te bezoeken). Als er geen heuristiek is maakt men een *ongewogen* keuze: alle overblijvende mogelijkheden zijn even waarschijnlijk. Als er een heuristiek is die aangeeft dat bepaalde componenten waarschijnlijk tot een goed individu leiden neemt deze meestal de vorm aan van een kwaliteitsfunctie: hoe groter de functiewaarde, hoe beter de kwaliteit. Zo gebruikt men bij Minimum Cover wel de heuristiek om, als men reeds een aantal deelverzamelingen heeft gekozen, de volgende deelverzameling te kiezen die het grootste aantal onbedekte punten bedekt. Er zijn dan twee mogelijkheden:

(1) *Shortlisting.* Men neemt de beste kandidaten (de shortlist) en kiest daaruit met gelijke waarschijnlijkheid.
(2) *Gewogen keuze.* Elke mogelijke keuze krijgt een gewicht w_i. De kans $p(i)$ dat i gekozen wordt is dan evenredig met w_i,

$$p(i) = \frac{w_i}{\sum_j w_j}.$$

Voor het kiezen van een nieuw individu uit $\mathcal{S}$, uitgaande van andere individuen die men al bekeken heeft, zijn er fundamenteel er drie methodes:

(1) Kleine wijzigingen aanbrengen in één reeds bekeken individu s van $\mathcal{S}$.
(2) Kruisen van twee al bekeken individuen. Dit gebeurt vaak als we elk individu zinvol kunnen indelen in 'blokken'. Is het probleem bijvoorbeeld het vinden van efficiënte programmeercode en zijn de individuen implementaties van een bepaald klasse (in de zin van objectgerichte programmeertaal) dan kan het kind voor elke lidfunctie de implementatie van een van de twee ouders overnemen.
(3) Het vermengen van grote hoeveelheden al bekeken individuen. Dit is een weinig

gebruikte methode, die bijvoorbeeld bij ACO (Ant Colony Optimization) gebruikt wordt.

Deze vier methodes leveren de belangrijkste indeling voor de verschillende optimisatieheuristieken. Om een oplossingsmethode te kiezen voor een bepaald probleem moeten we eerst bepalen welke van de vier methodes zinvol kunnen zijn voor dat probleem.

14.3 LOKAAL VERSUS GLOBAAL ZOEKEN

Zoals al gezegd: een algemene veronderstelling bij het gebruik van metaheuristieken is dat het zinvol is om, bij het opbouwen nieuwe individuen van $\mathcal{S}$, uit te gaan van de reeds gevonden individuen. Eén van de belangrijkste factoren daarbij is of het zin heeft te proberen een individu van $\mathcal{S}$ te verbeteren door het aanbrengen van kleine veranderingen.

Wat een kleine wijziging is hangt ook weer af van het probleem: men kan bijvoorbeeld één letter van s veranderen, maar soms is het meer zinvol twee letters te verwisselen (bij het uurroosterprobleem verwisselt men bijvoorbeeld de tijdsslots of de lokalen van twee lessen). Deze methode leidt tot het begrip omgeving: voor elke $s \in \mathcal{S}$ is $\mathcal{N}(s)$ de omgeving van s die bestaat uit s zelf en alle strings uit $\mathcal{S}$ die door één kleine wijziging uit s kunnen bekomen worden. Vaak is het belangrijk ervoor te zorgen dat we vanuit een willekeurige $s \in \mathcal{S}$ naar een willekeurige andere $s' \in \mathcal{S}$ kunnen komen door een aaneenschakeling van kleine wijzigingen. Als bijvoorbeeld niet alle strings in $\mathcal{S}$ even lang zijn moeten er kleine wijzigingen zijn die de string langer of korter maken. Men moet ervoor zorgen dat kleine wijzigingen weer een geldig individu uit $\mathcal{S}$ opleveren. Dat is niet altijd even gemakkelijk. Zo kan bijvoorbeeld het omwisselen van twee lesuren zorgen voor een conflict waardoor het nieuwe individu onaanvaardbaar is. Soms 'repareert' men dat nieuwe individu. Men kan het ook weggooien als het niet tot $\mathcal{S}$ behoort. Een andere oplossing is om strings die niet tot $\mathcal{S}$ behoren toe te laten, maar ze een heel grote evaluatiewaarde te geven, zodat ze niet geaccepteerd worden als eindoplossing. Bij het lessenroosterprobleem geeft men bijvoorbeeld één strafpunt voor een springuur en 1000 strafpunten voor een conflict waarbij een docent twee lessen tegelijk moet geven.

Over het algemeen is het zo dat een kleine wijziging van een goed individu een zeer slecht individu kan maken, maar vaak is het zo dat in de omgeving van een goed individu er ook andere goed individuen zitten. Het wordt pas echt interessant als het zinvol is om lokale verbeteringen iteratief toe te passen, m.a.w. als het vaak voorkomt dat, uitgaande van een individu $s_0 \in \mathcal{S}$, we een rij van individuen $s_0, s_1, \ldots, s_n \in \mathcal{S}$ kunnen construeren zodanig dat voor alle $i = 0, \ldots n-1$, $s - i + 1 \in \mathcal{N}(s_i)$ en $f(s_{i+1}) < f(s_i)$. Dergelijke rij kan lang of kort zijn, maar eindigt zeker met een *lokaal minimum*, dit is een individu s_n zodat $\forall s \in \mathcal{N}(s_n) : f(s) \geq f(s_n)$.

Het vinden van zo'n rij staat bekend als *lokale optimisatie*. Er zijn verschillende manie-

ren om zo'n rij op te bouwen. In een aantal gevallen is het gemakkelijk om, gegeven s, een[2] element in $\mathcal{N}(s)$ te vinden met de beste f-waarde. Dat kan zijn omdat $\mathcal{N}(s)$ weinig individuen bevat, of omdat kennis van het probleem toelaat de beste verbetering onmiddellijk te vinden. Dergelijke methodes staan bekend als *steepest hill climbing* omdat men meestal een evaluatiefunctie neemt die zo groot mogelijk moet worden. Met onze conventie voor f zouden we moeten spreken van *steepest slope descending*.

In andere gevallen gebruikt men randomisatie: men neemt random een kleine verandering en kijkt of deze een verbetering oplevert. Zo ja, dan gaat men over naar het nieuwe individu. Zo nee, dan probeert men opnieuw. Indien men na een bepaald vooropgegeven aantal pogingen geen verbetering vindt dan veronderstelt men in een lokaal minimum te zitten.

Bij problemen waar er maar één lokaal minimum is is dit ook het beste individu. Nu is het zo dat er vele lokale minima in $\mathcal{S}$ kunnen zijn en dat er geen enkele garantie is dat al die lokale minima dezelfde f-waarde hebben: bij veel problemen zijn er lokale minima met een zeer slechte f-waarde. Bovendien kan het construeren van zo'n rij zeer veel tijd in beslag nemen, vooral als de startwaarde s_0 zeer slecht is.

Vandaar dat metaheuristieken naast lokale optimisatie ook een mechanisme moeten bevatten dat toelaat om te ontsnappen aan een lokaal minimum. Dit mechanisme staat bekend als *exploratie*. Hier zijn er grosso modo twee mogelijkheden:

(1) Exploratie door helemaal opnieuw te beginnen. Hierbij worden individuen van $\mathcal{S}$ random opgebouwd, zoals bij random zoeken.
(2) Exploratie door grotere wijzigingen aan te brengen aan gekende individuen.

Met de twee basisideeën, exploratie en lokale optimisatie, kunnen een aantal metaheuristieken beschreven worden. Bij de meeste methodieken, zoals Simulated Annealing en Tabu Search, zijn lokale optimisatie en exploratie wel aanwezig, maar ze zijn niet strikt gescheiden.

14.4 METHODES ZONDER RECOMBINATIE

Aan het gewone random zoeken kunnen twee elementen toegevoegd worden die de efficiëntie kunnen opvoeren.

(1) Lokaal zoeken verbetert vaak random aangemaakte individuen.
(2) Voor sommige problemen zijn er heuristieken die constructie van individuen kunnen sturen.

Een paar voorbeelden:

[2] *Een* element en niet *het* element: er is geen garantie op uniciteit.

```
S SimulatedAnnealing(){
    S s=maakRandom();
    Temperatuur T=T0();
    while (!stopconditie){
        S s'=kiesRandomIn(N(s));
        if (f(s') < f(s)
                || randomKans(T,f(s'),f(s)))
            s=s';
        T=update(T);
    };
    return s;
};
```

Code 14.2. Simulated Annealing

- **ILS,** (*Iterated Local Search). Zoekt een individu door een vorig individu sterk te wijzigen en past op het resultaat lokale optimisatie toe.*
- **GRASP** *(*Greedy Randomised Adaptive Search Procedure). Dit is geschikt voor problemen waarbij er een inhalige (*greedy) heuristiek bestaat om een individu op te bouwen. Deze wordt gerandomiseerd, waarbij GRASP shortlisting gebuikt. Dit kan uiteraard worden aangevuld worden met lokale optimisatie.*

14.4.1 Simulated Annealing

Bij lokale optimisatie hebben we gezien dat deze kan gebeuren door telkens een random s' in de omgeving $\mathcal{N}(s)$ te nemen en het nieuwe individu te bewaren als het beter is dan het oude. Bij Simulated Annealing is er een, op het eerste zicht kleine, wijziging. Er is een zekere kans dat s' gekozen wordt ook als s' slechter is dan s. Deze kans is afhankelijk van de evaluatiewaarden van s en s', maar ook van een zogeheten temperatuur T. In het algemeen hangt de kans enkel af van het verschil $f(s) - f(s')$ en T, en wel op zo'n manier dat de kans groter is naarmate $f(s') - f(s)$ kleiner is, maar ook groter naarmate T lager is. Een vaak gekozen functie is

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

Naar analogie met de Boltzmanndistributie. De begintemperatuur wordt vrij hoog gekozen, zodat exploratie bevorderd wordt. Het klassieke patroon is om T te laten dalen tot (bijna) nul: op deze manier benadert de eindfase van Simulated Annealing lokaal zoeken. Dit dalen moet traag gebeuren om de methode de kans te geven volfoende te exploreren en lokaal te zoeken. Men kan bewijzen dat het algoritme zeer waarschijnlijk

```
S TabuSearch(){
    S s=maakRandom();
    lijst<S> Taboe;
    while (!stopconditie){
        Taboe.voegtoe(s);
        s=besteNietTaboe(N(s),Taboe);
    };
    return s;
};
```

Code 14.3. Tabu Search, basisversie

een globaal optimum vindt als er een $\Gamma > 0$ bestaat waarvoor

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

Het blijkt echter dat Simulated Annealing er baat bij heeft om een meer oscillerend temoeratuurverloop te gebruiken zodat fases van lokaal zoeken en exploreren elkaar afwisselen. Omdat de temperatuur nooit nul wordt zal, als het lokaal zoeken niet succesvol is, dit vrij snel worden afgebroken.

14.4.2 Tabu Search

Een alternatieve aanpak vinden we bij Tabu Search. In zijn eenvoudigste vorm houdt Tabu Search een taboelijst bij: een lijst van al gecontroleerde individuen die niet opnieuw mogen bezocht worden. In pseudocode 14.3 geeft `kiesBesteNietTaboe (N(s), Taboe)` het beste element in $\mathcal{N}(s)$ terug dat niet in de taboelijst zit.

De gehele geschiedenis bijouden is niet echt efficiënt. In de praktijk wordt dus het aantal elementen in de taboelijst beperkt. Als de lijst vol is wordt het oudste element in de lijst verwijderd voor er een nieuw wordt toegevoegd.

Het gebruik van de taboelijst moet ervoor zorgen dat de methode voldoende exploreert: met een taboelijst met grootte 0 wordt Tabu Search gewoon lokale verbetering. De taboelijst moet echter zo groot zijn dat men ver genoeg uit de buurt van een lokaal optimum kan komen om het *basin of attraction* te verlaten. Dit is meestal niet haalbaar, zodat men vaak taboelijsten gebruikt die geen individuen bevatten maar wel eigenschappen. Een voorbeeld daarvan kan zijn om geen *individuen* bij te houden in de lijst maar wel *componenten*. Indien we veronderstellen dat $\mathcal{N}(s)$ de verzameling is van individuen die we bekomen door uit s exact één component te verwijderen uit s en deze te vervangen door een andere dan kunnen we pseudocode 14.4 gebruiken. In dit geval geeft `kiesBesteNietTaboe (N(s), Taboe)` het beste element in $\mathcal{N}(s)$ terug dat niet in de taboelijst zit en verschilt van s.

```
S TabuSearch(){
    S s=maakRandom();
    lijst<Component> Taboe;
    while (!stopconditie){
        s=kiesBesteNietTaboe(N(s),Taboe);
        Taboe.voegToe(gewijzigdeComponent);
    };
    return s;
};
```

Code 14.4. Tabu Search, componentversie

Ook hier zijn er varianten denkbaar. Zo kan men bijvoorbeeld de taboelijst leeg maken als de f-waarde daalt (wat betekent dat men lokaal zoeken gebruikt tot men een lokaal optimum gevonden heeft), alhoewel men dan voorzichtig moet zijn om niet herhaaldelijk in hetzelfde lokale minimum terecht te komen.

14.5 GENETISCHE ALGORITMEN

14.5.1 Kruising

Metaheuristieken die kruising gebruiken worden meestal aangeduid als *genetische algoritmen*[3]. Bij kruising gaan we twee (hoogst zelden meer dan twee) individuen vermengen. Dit is alleen zinvol als er reden is om aan te nemen dat zo'n vermenging kan leiden tot individuen die beter zijn dan de ouders. Dit betekent niet dat *elke* vermenging leidt tot verbetering: in de meeste toepassingen gebeurt vermenging vrij blindelings, waardoor veel, zoniet de meeste, kruisingen leiden tot een slecht resultaat. Onderliggend is meestal de notie dat er combinaties zijn van componenten die 'goed' kunnen zijn, ook al heeft het individu een slechte f-waarde. De bedoeling is dan dat kruising met enige kans op succes dergelijke combinaties samenneemt.

Het is duidelijk dat de manier waarop de kruising gebeurt zeer veel invloed heeft op de efficiëntie van de methode. Fundamenteel zijn er twee opties:

(1) **Gerichte kruising.** Als men een idee heeft van de algemene structuur van individuen dan kan men deze structuur gebruiken om zinvolle combinaties van com-

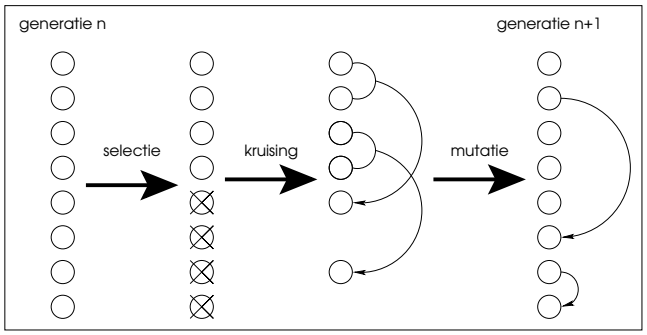
[3] Genetisch is een populair begrip in de context van metaheuristieken en het woord wordt dan ook voor zeer diverse technieken gebruikt.

ponenten te vinden. Wil men bijvoorbeeld programmacode maken, dan zijn is elke operatie van een klasse duidelijk een geheel. Men kan dan proberen om zo te kruisen dat dergelijke gehelen samen blijven[4].

(2) **Blinde kruising.** Als alternatief kan men een blinde vorm van kruising gebruiken. De meestgebruikte is die waarbij een individu wordt voorgesteld als een bitreeks van bepaalde lengte. Een manier om toch nog enige structuur te behouden is om ervoor te zorgen dat verwante bits dicht bij mekaar staan. Kruising van twee bitstrings gebeurt dan door de suffixen van de twee strings om te wisselen.

In dit verband wordt soms gesproken over het onderscheid tussen *genotype* en *fenotype*. Het fenotype is daarbij de individu als dusdanig (het lessenrooster, het computerprogramma, ...); het genotype is de voorstelling van het genotype waarop de kruising en eventueel de mutatie wordt toegepast. Vermits bij kruising twee individuen (de ouders) nodig zijn is moeten we verschillende individuen samen bijhouden. Men spreekt van een *populatie* van individuen. Meestal is die populatie van een vaste grootte en probeert men de populatie zo te wijzigen dat de kans op zeer goede individuen toeneemt. Merk op dat ons doel één zo goed mogelijk individu is. Vaak zullen de methodes die we toepassen de gemiddelde fitheid van de populatie verbeteren, maar dat is niet essentieel: een populatie met zeer goede en zeer slechte individuen is vaak interessanter dan een met vele matig goede individuen.

Bij genetische algoritmen spreekt men van generaties. Gegeven een populatie gaat men over naar een volgende generatie volgens het schema gegeven in Figuur 14.1. De overgang verloopt in twee stappen:



Figuur 14.1. Overgang naar een volgende generatie.

1. In de eerste stap wordt de populatie uitgedund door een aantal individuen te verwijderen. Alleen de beste individuen blijven over. In de woorden van Darwin heet dit *the survival of the fittest*.

[4] Dit valt dan weer onder *genetisch programmeren*.

2. Gebaseerd op de overblijvende individuen wordt populatie terug aangevuld tot haar oorspronkelijke grootte door kruising en mutatie.

Mutaties zijn kleine wijzigingen. Dit komt overeen met het kiezen van een individu uit $\mathcal{N}(s)$. Voor beide stappen kan men een aantal variaties bedenken. Zo kan men bij de tweede stap bepalen hoeveel individuen in de nieuwe generatie rechtstreeks uit de vorige komen, hoeveel er gemuteerd zijn en hoeveel er kruisingen zijn.

Omdat kruising en de efficiëntie ervan heel erg probleemafhankelijk zijn bestaan er heel wat variatie op. We vermelden twee belangrijke verfijningen:

- Als componenten van individuen met mekaar interageren dan kan het voordelig zijn om de kans te vergroten dat interagerende componenten bij elkaar blijven. Dit kan ingebouwd worden bij gerichte kruising, maar men kan ook een maat voor gelijkaardigheid invoeren. Daarbij gaat men dan alleen kruisingen toelaten tussen gelijkaardige individuen.
- Kruising moet niet alleen zorgen voor betere individuen, maar ook voor diversificatie en zo voor exploratie, iets wat door de voorgaande techniek beperkt wordt. Om toch meer exploratie te krijgen gaat men soms gebruik maken van *niching*, waarbij de fitheid van individuen verminderd wordt naarmate er meer gelijkaardige individuen in de populatie zitten.

14.6 VERMENGING

Bij vermengingsmethodes gaat men niet meer twee individuen kruisen. Wel gaat men uit van de globale eigenschappen van een populatie om de volgende te nemen. Hierbij kan men werken op twee niveau's:

(1) **Componentniveau.** Hierbij gaat men ervan uit dat het succes van een individu afhangt van de individuele componenten die ze bevat.
(2) **Combinatieniveau.** Hierbij gaat men ervan uit dat het succes van een individu afhangt van combinaties van componenten die ze bevat.

14.6.1 Recombinatie op componentniveau

Er zijn twee belangrijke, bijna identieke, metaheuristieken die gebruik maken van van dit soort recombinatie: *Gene Pool Recombination (GPR) en* Ant Colony Optimisation (ACO). Zoals een genetische algoritme werken beide met een populatie die telkens vervangen wordt door een nieuwe generatie. Bij beide gaat men een fitheid toekennen niet alleen aan individuen maar ook aan componenten. Bij het opbouwen van een nieuwe populatie gaat men ongeveer te werk zoals bij een gerandomiseerd inhalige constructie waarbij de fitheid w_i van een component i dient als gewicht voor een gewogen keuze.

```
populatie.maakLeeg();
while (populatie.size() < n){
    S s;//lege begindeeloplossing
    while (! s.isOplossing() en aanvulling mogelijk){
        component k=gewogenRandomKeuze
                (s.verzamelingMogelijkeAanvullingen());
        s.vulAanMet(k);
    };
    if (s.isOplossing())
        populatie.voegToe(s);
};
```

Code 14.5. Gewogen random constructie van een populatie.

De gewogen random keuze kiest uit een verzameling V van componenten een element waarbij de waarschijnlijkheid om een component i te kiezen evenredig is met w_i,

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

Merk op dat de verzameling waaruit gekozen afhankelijk is van s, maar dat de gewichten w_i onafhankelijk van s zijn.

Het verschil tussen GPR en ACO zit in de keuze van de gewichten w_i:

- Bij GPR wordt alleen rekening gehouden met de laatste generatie. Eerst wordt deze uigedund met natuurlijke selectie. Daarna krijgt elke component een gewicht evenredig met het aantal individuen in de overblijvende populatie die de component bevatten.
- Bij ACO speelt het oude gewicht wel een rol. Bovendien worden slechte individuen uit de laatste generatie niet uitgesloten. Wel krijgt elke s een kwaliteitswaarde $F(s)$. Deze waarde is groter naarmate $f(s)$ kleiner is. Het nieuwe gewicht voor een component i is dan

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

  Hierbij is ρ een vergeetfactor.

14.6.2 Recombinatie op combinatieniveau

Deze vorm van recombinatie vereist dat het probleem zich ertoe leent. Daarvoor zijn er twee eigenschappen belangrijk:

(1) Opeenvolgende letters in een individu moeten een sterk onderling verband vertonen.
(2) Een deelstring van een goed individu gebruiken in een ander individu verhoogt de kans dat het andere individu ook goed is.

In principe kan men de voorgaande methodes gebruiken door opeenvolgingen van letters nu als componenten te beschouwen. Het is echter meer gebruikelijk om een graaf te gebruiken als voorstelling. In deze vorm is er ook meer overeenkomst met het gedrag van mieren.

De componenten van onze individuen worden daarbij de knopen van een ongerichte gewogen graaf. Indien twee componenten samen kunnen voorkomen in een individu is er een verbinding (hoe we het gewicht van de verbinding bepalen bekijken we later).

We hebben nu een aantal mieren die proberen individuen uit $\mathcal{S}$ te construeren. Zo'n individu wordt voorgesteld als een pad met een bepaald vertrekpunt. Elke mier vertrekt vanuit dit punt en construeert een weg als volgt:

(1) Op een bepaald punt aangekomen bepaalt de mier of het pad dat ze al heeft afgelegd een individu definieert. Zo ja, dan stopt ze.
(2) Zo nee bepaalt ze welke volgende componenten nog in aanmerking komen. Uiteraard zijn al die componenten verbonden met de knoop waar de mier zich op dat ogenblik bevindt, maar niet alle verbindingen vanuit de knoop leiden zeker naar een nog toegestane component.
(3) Als er geen mogelijkheden zijn stopt de mier.
(4) Anders kiest ze één van de mogelijkheden. Ze maakt daarbij een gewogen random keuze met als gewichten de gewichten van de verbindingen.

Het is gebruikelijk om in deze context te spreken van *feromonen* in plaats van gewichten, omdat echte mieren feromoonsporen volgen.

Nadat elke mier geprobeerd heeft om een individu te construeren worden de hoeveelheden feromonen aangepast. Eerst wordt het feromoon bij elke verbinding vermenigvuldigd met een factor $1 - \rho$. Hier is ρ de vergeetfactor zoals we die al kennen; in de context van mieren spreekt men van het verdampen van feromonen. Daarna voegt elke mier die een individu s heeft gevonden een hoeveelheid feromonen $F(s)$ toe aan elke verbinding die ze gebruikt heeft. Ook hier is $F(s)$ de kwaliteitswaarde van het individu s. In formule wordt dit

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

$\forall i, j$, waar A_{ij} de verzameling individuen is gevonden door mieren die verbinding v_{ij} gebruikt hebben.

Bij het begin van het algoritmen kan men alle waarden τ_{ij} gelijk nemen. Wanneer men echter een heuristiek heeft kan men andere beginwaarden nemen, zodat bepaalde

verbindingen initieel bevorderd worden. In sommige gevallen wil men dat deze initiële waarden niet geheel vergeten worden: zoals gewoonlijk is dit een van de verfijningen van de methodiek die al dan niet kan werken.

BIBLIOGRAFIE

[1] ABOUELHODA M.I., KURTZ S., en OHLEBUSCH E. Replacing Suffix Trees with Enhanced Suffix Arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[2] AHUJA R.K., MAGNANTI T.L., en ORLIN J.B. *Network Flows: Theory, Algorithms and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[3] APPLEGATE D.L., BIXBY R.E., CHVÁTAL V., en COOK W.J. *The Traveling Salesman Problem - A Computational Study*. Princeton University Press, Princeton, NJ, 2006.

[4] ATALLAH M.J. *Algorithms and Theory of Computation Handbook*. CRC Press LLC, Boca Raton, FL, 1999.

[5] BENTLEY J.L. en FRIEDMAN J.H. Data Structures for Range Searching. *ACM Computing Surveys*, 11(4):398–409, December 1979.

[6] BENTLEY J.L. en SEDGEWICK R. Fast Algorithms for Sorting and Searching Strings. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, Januari 1997.

[7] BLUM C. en ROLI A. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. *ACM Computing Surveys*, 35(3):268–308, September 2003.

[8] BÖHM C., BERCHTOLD S., en KEIM D.A. Searching in High-Dimensional Spaces–Index Structures for Improving the Performance of Multimedia Databases. *ACM Computing Surveys*, 33(3):322–373, September 2001.

[9] CHUANG S.-T., GOEL A., MCKEOWN N. en PRABHAKAR B. Matching Output Queueing with a Combined Input Output Queued Switch. *IEEE J. on Selected Areas in Communications*, 17(6):1030–1039, Juni 1999.

[10] Cook, Stephen A., The Complexity of Theorem-proving Procedures, In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 1971:151–158.

[11] CORMEN T.H., LEISERSON C.E., RIVEST R.L., en STEIN C. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3rd edition, 2009.

[12] CROCHEMORE M. en RYTTER W. *Jewels of Stringology. Text Algorithms.* World Scientific, Singapore, 2003.

[13] ENBODY R.J. en DU H.C. Dynamic Hashing Schemes. *ACM Computing Surveys*, 20(2):85–113, Juni 1988.

[14] FORTNOW L. The Status of the P versus NP Problem. *Communications of the ACM*, 52(9):79–86, September 2009.

[15] GAEDE V. en GÜNTHER O. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, Juni 1998.

[16] GALIL Z. Efficient Algorithms for Finding Maximum Matching in Graphs. *ACM Computing Surveys*, 18(1):23–38, Maart 1986.

[17] GAREY M.R. en JOHNSON D.S. *Computers and Intractability. A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., New York, 1979.

[18] GOODRICH M.T. en TAMASSIA R. *Algorithm Design*. Wiley, Hoboken, NJ, 2002.

[19] GRAEFE G. B-Tree Indexes, Interpolation Search, and Skew. In *Proceedings of the Second International Workshop on Data Management on New hardware (DaMoN 2006)*. ACM, Juni 2006.

[20] GUSFIELD D. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, Cambridge, MA, 1997.

[21] HIRSCHBERG D.S. A Linear Space Algorithm for Computing Maximal Common Subsequences. *Communications of the ACM*, 18(6):341–343, Juni 1975.

[22] HOPCROFT J.E., MOTWANI R., en ULLMAN J.D. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA, 2006.

[23] JUNGNICKEL D. *Graphs, Networks and Algorithms*. Spinger-Verlag, Berlin, 1999.

[24] KAM A.C. en SIU K-Y. Linear-Complexity Algorithms for QoS Support in Input-Queued Switches with No Speedup. *IEEE J. on Selected Areas in Communications*, 17(6):1040–1056, Juni 1999.

[25] KLEINBERG J. en TARDOS É. *Algorithm Design*. Addison-Wesley, Reading, MA, 2006.

[26] KNUTH D.E. *The Art of Computer Programming, Vol 3, Sorting and Searching*. Addison-Wesley, Reading, MA, 2nd edition, 1998.

[27] LEVITIN A. *Introduction to the Design and Analysis of Algorithms*. Addison-Wesley, Reading, MA, 2nd edition, 2007.

[28] LUKE S. *Essentials of Metaheuristics* Lulu, 2013.

[29] MARTÍNEZ C. en ROURA S. Randomized Binary Search Trees. *Journal of the ACM*, 45(2):288–323, Maart 1998.

[30] MORET B.M.E. en SHAPIRO H.D. An Empirical Assessment of Algorithms for Constructing a Minimum Spanning Tree. In *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, 1994.

[31] PETTIE S. Towards a Final Analysis of Pairing Heaps. In *Proceedings of the 46th Annual Symposium on Foundations of Computer Science (FOCS'05)*. IEEE, 2005.

[32] PUGLISI S.J., SMYTH W.F., en TURPIN A.H. A Taxonomy of Suffix Array Construction Algorithms. *ACM Computing Surveys*, 39(2):1–31, Juni 2007.

[33] SAMET H. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2):187–260, Juni 1984.

[34] SCHRIJVER A. *Combinatorial Optimization. Polyhedra and Efficiency*. Springer-Verlag, Berlin, 2003.

[35] SEDGEWICK R. *Algorithms in C++, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching, Part 5: Graph Algorithms*. Addison-Wesley, Reading, MA, 3rd edition, 1998.

[36] SEIDEL R. en ARAGON C.R. Randomized Search Trees. *Algorithmica*, 16:464–497, 1996.

[37] SKIENA S.S. *The Algorithm Design Manual*. Springer-Verlag, New York, 2nd edition, 2008.

[38] STASKO J.T. en VITTER J.S. Pairing Heaps: Experiments and Analysis. *Communications of the ACM*, 30(3):234–249, 1987.

[39] TIMOSHEVSKAYA, N. en Feng, W. SAIS-OPT: On the Characterization and Optimization of the SA-IS Algorithm for Suffix Array Construction. *4th IEEE International Conference on Computational Advanced in Bio and Medical Sciences, Miami Beach, Florida, June, 2014*, pages 1–6, 2014.

[40] VITTER J.S. External Memory Algorithms and Data Structures: Dealing with Massive Data. *ACM Computing Surveys*, 33(2):209–271, Juni 2001.

[41] WEISS M.A. *Data Structures and Algorithm Analysis in C++*. Addison-Wesley, Reading, MA, 3rd edition, 2006.

[42] ZOBEL J. en MOFFAT A. Inverted Files for Text Search Engines. *ACM Computing Surveys*, 38(2):1–56, Juli 2006.