Algoritmen en Complexiteit

Leen Torenvliet

30 april 2014

Inhoudsopgave

Voorwoord

Sinds 1983 geef ik jaarlijks het vak Complexiteitstheorie aan de Universiteit van Amsterdam. Soms in combinatie met Automatentheorie, soms in combinatie met Algoritmen, en de laatste tijd ook als rechtstreeks vervolg op het college Datastructuren. Er zijn vele, vele boeken over algoritmen, goede en minder goede. Zo nu en dan vraagt een student mij of er niet een goed boek over algoritmen in het Nederlands geschreven is en mijn antwoord is dan: "Voorzover ik weet niet." Natuurlijk is de "markt" voor Nederlandse boeken over wetenschappelijke onderwerpen sowieso klein en is de neiging om een boek te schrijven in het Nederlands over een onderwerp waarover zoveel goede boeken al beschikbaar zijn niet erg groot. Toch is het idee niet zo bijzonder. Als we naar onze oosterburen kijken dan zien we dat het daar heel gebruikelijk is om colleges in boekvorm te noteren en vervolgens uit te brengen zodat deze boeken door de studenten gebruikt kunnen worden. Nu de druk op studenten om snel door de studie heen te komen ook in ons land steeds groter wordt en bovendien is afgesproken dat in de bachelorfase van de studie het Nederlands de voertaal is, lijkt het geen grote stap om ook het studiemateriaal in het Nederlands aan te bieden. Vandaar deze tekst. Of het een "goed" boek wordt, of is geworden, zoals door de student in kwestie werd gevraagd laat ik graag aan de lezer. Het boek is in grote lijnen de neerslag van een college algoritmiek en bespreekt de onderwerpen die ik graag in zo'n college voorbij laat komen. Ik gebruik de tekst voortdurend tijdens het college en de tekst wordt ook aangepast naar aanleiding van kritische opmerkingen van de studenten.

De tekst bestaat uit drie delen. In het eerste deel bespreken we het onderwerp Algoritmen. Een algoritme is, zoals de lezer wellicht bekend is, een methode om een bepaald probleem door een machine te doen oplossen. Het gaat hier in het bijzonder om problemen die op een stap-voor-stap manier kunnen worden uitgevoerd, waarbij herhaling van een aantal opeenvolgende stappen is toegestaan, maar waarbij toepassing van intelligentie bij het uitvoeren van de stappen of bij het beslissen of een serie stappen herhaald moet worden niet is toegestaan. Een algoritme moet een begin en een eind hebben en de Algoritmiek is de wetenschap die probeert voor soorten van problemen algoritmen te maken die zo snel mogelijk van dat begin naar dat eind te komen. Ook in deze tijd, waarin computers steeds sneller lijken te worden is het sneller maken van algoritmen nog steeds een bedrijvigheid die zinvol is. We zullen daarvan enkele overtuigende voorbeelden voorbij zien komen. Om te kunnen bepalen of de ene algoritme sneller is dan de andere hebben we wiskundige hulpmiddelen nodig die we zullen bespreken voordat we voorbeelden van algoritmische methoden gaan bekijken. Deze methoden bestaan uit een aantal truken die in het verleden nuttig zijn gebleken om problemen van een bepaalde soort van algoritmen te voorzien. We bespreken hier achtereenvolgens de Gulzige Algoritmen, de Verdeel-en-Heersstrategie, en het Dynamisch Programmeren. Met de ontwerpmethoden en analytische methoden in de hand bekijken we verschillende toepassingsgebieden van de algoritmiek, in het bijzonder de algoritmen op grafen en de getaltheoretische algoritmen. Het laatste onderwerp mondt uit in de tegenwoordig in de belangstelling staande public key cryptosystemen en toepassingen daarvan voor de beveiliging van gegevens.

Het tweede deel wordt door studenten doorgaans als moelijker ervaren. We beschouwen de problemen op zich en generaliseren over alle mogelijke algoritmen die voor een probleem kunnen worden bedacht. We proberen hier aan te geven waarom het voor sommige, schijnbaar eenvoudige, problemen schier onmogelijk lijkt een efficiënte algoritme te bedenken, en ontwikkelen technieken die ons in staat stellen zulke problemen te herkennen en te bewijzen dat het op z'n minst onwaarschijnlijk is dat efficiënte algoritmen voor zo'n probleem bestaan. De motivatie hiervoor is uiterst praktisch. Aangezien deze problemen in allerlei vormen in de praktijk vaak voorkomen, is het aannemelijk dat een opdrachtgever een algoritme voor zo'n probleem

zou willen hebben. Het is waarschijnlijk beter om in dat geval een sterk verhaal te hebben en om vooral *van tevoren* te kunnen beslissen dat een efficiënte algoritme voor zo'n probleem niet bestaat. Afhankelijk van hoe erg de ondoenlijkheid van die problemen is, kunnen ze ondergebracht worden in bepaalde klassen en de bestudering van deze klassen is een centraal onderwerp in de complexiteitstheorie. Tenslotte is er nog het onderwerp benadering. Als we hebben vastgesteld dat het geen zin heeft voor een probleem een algoritme te vinden die in alle gevallen efficiënt is, dan kunnen we nog zoeken naar algoritmen die in de "meeste" gevallen efficient zijn. Hierbij kan meeste gevallen betekenen dat de algoritme efficient is voor de meeste inputs van dezelfde lengte, of dat de meeste paden die een algoritme volgt als onderweg een randomgenerator gevolgd wordt, efficient zijn, dwz. dat de verwachte rekentijd klein is.

De eerste twee delen van dit boek bevatten een serie onderwerpen die in twaalf tot vijftien colleges van twee uur gemakkelijk behandeld kunnen worden, en is ongeveer gericht op tweedejaars informaticastudenten, waarbij het deel Algoritmen ongeveer tweederde van de tijd in beslag neemt. In geval minder tijd beschikbaar is, of als bijvoorbeeld de studenten een ruimere wiskundige achtergrond hebben dan hier wordt aangenomen, zou bijvoorbeeld de wiskunde-inleiding kunnen worden overgeslagen en zou een keuze gemaakt kunnen worden uit de onderwerpen uit het gedeelte Algoritmen. Ook zou een gedeelte van het derde deel van het boek behandeld kunnen worden, waarin onderwerpen besproken worden die niet noodzakelijk tot basiskennis algoritmiek gerekend worden.

Nagenoeg alle stellingen, algoritmen en bewijzen kunnen wel ergens teruggevonden worden in de literatuur, vaak in tekstboeken. Slechts wanneer beschikking over de volledige originele tekst—voorzover nog terug te vinden—kan leiden tot dieper inzicht of anderszins vermeldenswaardig is, heb ik de bron vermeld in deze tekst. Voor het overige verwijs ik slechts naar de volgende, naar mijn idee lezenswaardige, boeken en de verwijzingen daarin [Har92, JS04, Wil86, Meh85, BB96, Ski98, GT02, GJ79]. In het studiejaar 2007–2008 heb ik deze tekst voor het eerst als syllabus bij het college gebruikt, in plaats van een boek. Florian Speelman en Jannis Teunissen hebben uit de eerste tekst een aantal fouten gehaald. In het collegejaar 2008-2009 heeft Koos van Strien de tekst nogmaals geheel doorgewerkt, en zijn er opnieuw veranderingen aangebracht. In 2010 heeft Jeroen Zuiddam een aantal nuttige verbeteringen gesuggereerd en heeft Inge Bethke het college over snelle Fouriertransformatie gegeven en uit dat hoofdstuk een aantal fouten gehaald.

Deel I

Algoritmen

Het eerste deel van de tekst behandelt algoritmen en hun complexiteit. We bespreken de belangrijkste complexiteitsmaten en hoe deze maten toe te passen op algoritmen. Vervolgens bekijken we een aantal algoritmische methoden en een aantal voorbeelden van hun toepassingen. Aparte hoofdstukken vormen de toepassingsgebieden grafen en numerieke algoritmen.

Hoofdstuk 1

Voorspel

1.1 Inleiding

Problemen, algoritmen en oplossingen vormen een drieëenheid in de informatica. Elke vraag die men in één of andere taal kan stellen is een probleem, maar zo'n probleem is voor de informaticus niet noodzakelijk interessant. Een probleem wordt pas interessant voor de informaticus als er sprake is van een—doorgaans oneindige—verzameling van soortgelijke vragen die op dezelfde manier naar een oplossing gebracht kunnen worden. In zo'n geval is deze verzameling van problemen te vangen met eenzelfde methode, en we spreken dan al snel van een algoritme om het probleem op te lossen.

Naar een formele invulling van het begrip algoritme is lang gezocht. Algoritmen om problemen op te lossen zijn al zo oud als de wereld. De algoritme die we in paragraaf 4.4 bespreken bijvoorbeeld stamt uit India en is zeker 5000 jaar oud. Toch wordt, zeker in Europa, pas zo'n 100 jaar nagedacht over het begrip algoritme. Als aanleiding voor deze ontwikkeling kunnen we de feestrede noemen die David Hilbert in het jaar 1900 in Parijs voor de verzamelde wiskundige gemenschap hield noemen. In deze rede presenteerde Hilbert 23 open problemen waarvan hij dacht dat het urgent was dat de wiskundigen die in de komende eeuw zouden oplossen.

Een aantal van deze problemen heeft aanleiding gegeven tot revolutionair werk in de wiskunde en de informatica, en een aantal heeft natuurlijk ook minder belangstelling[1] gekregen. Van de interessante problemen noemen we bijvoorbeeld het eerste, en volgens Hilbert ook belangrijkste, probleem van het bepalen van het cardinaalgetal van het continuum, dat aan G. Cantor werd toegeschreven. Cantor had namelijk bewezen dat er meer reële getallen dan natuurlijke getallen bestaan (zie 6.1) maar de vraag of er misschien een verzameling met meer elementen dan de natuurlijke getallen, maar *minder* elementen dan de reële getallen zou kunnen bestaan bleef open. De aanname dat er niet zo'n verzameling bestaat werd bekend onder de naam *continuümhypothese* Later in die eeuw bleek dat deze vraag *onafhankelijk* is van de axioma's van de verzamelingenleer. Gödel bewees in 1938 dat je niet het tegendeel kunt bewijzen, en Cohen bewees in 1963 dat je de hypothese zelf niet kunt bewijzen. Je kunt deze hypothese dus aannemen (of niet) en een consistente verzamelingentheorie overhouden, ongeveer net als het evenwijdige lijnen axioma van de Euclidische meetkunde.

Het tweede probleem dat Hilbert presenteerde was dat van de consistentie van de axioma's van de rekenkunde. Hilbert, als formalist, geloofde stellig dat de hele wiskunde geautomatiseerd zou kunnen worden en dat elke stelling met behulp van mechanische middelen geproduceerd zou kunnen worden door een ijverige wiskundige (computers waren in die tijd onbekend). Deze gedachte dreef onder andere Whitehead en

[1] Als zijn vijftiende probleem presenteerde Hilbert het vinden van een „rigorous foundation of Schubert's enumeration calculus". Schubert was een negentiende eeuwse Duitse wiskundige die een veelheid aan constanten vond verbonden met bepaalde meetkundige objecten. Bijvoorbeeld: het aantal lijnen dat vier gegeven lijnen in drie dimensies snijdt is twee, en het aantal kwadratische oppervlakken dat negen kwadratische oppervlakken in de ruimte raakt is 666.841.048. De Nederlandse wiskundige van der Waerden heeft tussen 1920 en 1930 hier enig werk aan besteed, als onderdeel van het onderzoek in de algebraïsche meetkunde.

Russel tot het produceren van de „Principia Mathematica", een lijvig stuk dat helaas altijd tot vier delen beperkt is gebleven waarin zo'n formalisering van de wiskunde werd ondernomen. In 1931 bewees Gödel de onvolledigheid van de rekenkunde. Elk eindig consistent stelsel van axioma's voor de rekenkunde zal altijd ware stellingen hebben die geen bewijs binnen dit stelsel hebben. Dit probleem inspireerde ook het nadenken over wat nu eigenlijk zo'n mechanisch rekenkundig proces is en leidde in de dertiger jaren ongeveer tegelijkertijd tot een aantal formalismen voor het begrip berekenbaarheid. Turing presenteerde in 1936 zijn Turingmachine [Tur36]. Dit model gebruiken we nog steeds om de uitdrukking te geven aan de complexiteit van problemen (zie Hoofdstuk 5). Ook omstreeks die tijd presenteerde A. Church zijn lambda calculus—hoewel de publicatie van de lambda caluculus pas rond 1940 geschiedde gaf Church er zeker in 1936 al college over in Princeton, en zijn artikel *A note on the Entscheidungsproblem* verscheen in 1936 in het eerste deel van de Journal on Symbolic Logic. Overigens werden beide benaderingen al in minder toegankelijke teksten rond 1920 door Emil Post neergeschreven, die ook rond die tijd een versie van de onvolledigheidsstelling van Gödel in een brief aan Church neerschreef. Post was echter niet tevreden over dit enkele, op zichzelf staande resultaat, en wilde eerst een meer algemene theorie maken.

Wanneer een voorschrift om met een reeks eenvoudige stappen van probleemstelling naar oplossing te komen in het algemeen de naam „algoritme" kreeg is onbekend. Wel leeft het sterke vermoeden dat het woord afkomstig is van het Engelse „algorism" dat de aanduiding is voor het systeem arithmetische berekening in het decimale systeem. Dit woord is weer een verbastering van het Latijnse woord Algoritmi dat de latinizering is van de naam van Abu Ja'afar Abdullah Muhammad Ibnu Mūsā al-Khawārizmī, auteur van het boek Al-ğabr, een verzameling van truuks om met het Hindu getallensysteem om te gaan, vertaald in het latijn als Algoritmi de numero Indorum. Al-ğabr heeft overigens de weg naar onze taal gevonden in de vorm Algebra. Omdat „algoritme" oorspronkelijk dus de naam van een man is, zullen we in deze tekst consequent spreken van „de algoritme", in tegenstelling tot van Dale die het wegens gebrek aan historisch besef heeft over „het algoritme".

Voor problemen zijn er algoritmen die tot een oplossing leiden. Soms zijn dit eenvoudige algoritmen die op de achterkant van een sigarendoos kunnen worden uitgevoerd, soms zijn alle computers in de wereld [CDRH+96] nodig om de berekening uit te voeren. Soms kunnen algoritmen binnen enkele ogenblikken tot resultaat leiden—bijvoorbeeld bij het traceren van subatomaire deeltjes. Soms kan een algoritme jaren nodig hebben voordat het resultaat bereikt wordt—bijvoorbeeld bij het ontcijferen van geheim- of onbekend [Par99] schrift. Ook is er voor hetzelfde probleem een grote, vaak oneindige verzameling algoritmen, waarvan de meeste onbekend zijn. Als er een algoritme voor een probleem gevonden is, is niet altijd met zekerheid te zeggen of er niet een efficiënte, betere algoritme voor hetzelfde probleem bestaat. De complexiteit van een algoritme kunnen we onderzoeken. Een probleem ontleent zijn complexiteit aan alle algoritmen die voor dat probleem bestaan, en dus is de complexiteit van een probleem een lastiger onderwerp. We beginnen daarom met de complexiteit van algoritmen.

1.2 Analyse van Algoritmen

In dit boek zijn we geïnteresseerd in algoritmen waarmee problemen kunnen worden opgelost. Als we een algoritme hebben bedacht, dan willen we graag weten hoe duur het uitvoeren van zo'n algoritme is. Hoeveel tijd en geheugen kost het uitvoeren van zo een algoritme. Voordat we daar iets over kunnen zeggen, moeten we eerst afspreken wat een algoritme is, en hoe we de kosten van een algoritme gaan berekenen.

Een algoritme is een plan waarmee een computer een probleem kan oplossen. Je kunt een algoritme zien als een programma. Het verschil is dat een programma meestal expliciet in één of andere programmeertaal zegt wat een programma moet doen, terwijl een algoritme een veel globalere beschrijving is van de stappen die moeten worden uitgevoerd. In deze tekst beperken we ons wel tot beschrijvingen die inderdaad gaan over de *stappen* die moeten worden uitgevoerd. Elke algoritme die hier gepresenteerd wordt is een recept dat stap voor stap naar een oplossing van het probleem leidt. De algoritmen kunnen nader gespecificeerd worden in één of andere programmeertaal en worden dan programma's, die stap voor stap tot een oplossing leiden. Er zijn in de informatica ook andere manieren om tegen problemen die door computers worden opgelost aan te kijken. Voorbeelden hiervan zijn, onder toepassing van enige simplificatie:

Descriptieve programmering. Je geeft de specificatie van het probleem en het systeem zoekt naar een oplossing. Voorbeelden hiervan zijn logisch programmeren en constraint programming.

Functionele programmering. Je geeft de beschrijving van het probleem in termen van functies en het systeem zoekt naar de juiste waarden.

Parallel programmeren. Er worden verschillende algoritmen (of dezelfde algoritme meerdere keren) tegelijkertijd uitgevoerd, al dan niet met onderlinge communicatie.

Quantum computing. Verschillende stromen van de een algoritme worden in superpositie uitgevoerd. Door interferentie en state collapse wordt de juiste oplossing overgehouden.

Deze alternatieve vormen zullen we hier niet bespreken. Ons gaat het alleen om de klassieke vorm van de algoritme die stap voor stap wordt uitgevoerd, waarbij soms beslissingen worden genomen door het systeem, maar waarbij het systeem nooit over zo'n beslissing hoeft na te denken. Elke stap kan dus *zonder*, al dan niet kunstmatige, intelligentie worden uitgevoerd.

1.2.1 Meten is weten

De eenvoudigste vorm van het onderzoeken hoe duur een bepaalde algoritme is, is natuurlijk het gebruik van de stopwatch. We schrijven een programma voor het oplossen van een bepaald probleem voor een bepaalde computer, en drukken vervolgens voor en na het uitvoeren van het programma de stopwatch in en kijken dan hoeveel tijd er is verstreken. De meeste operating systems hebben hier wel een speciale functie voor en ook in de meeste programmeertalen kan direct naar de systeemklok worden gekeken. Om een indruk te krijgen van de efficientie van een algoritme is deze methode zeer bruikbaar. We kunnen de gebruikte tijd op een aantal vraagstellingen, ook wel instanties van een probleem genoemd, meten en vervolgens uitzetten in een grafiek. Door zo'n grafiek door te trekken, kunnen we vaak ook voorspellingen doen over hoe de algoritme zich zal gedragen op andere, niet geteste instanties. Dat heet extrapolatie.

Een voordeel van deze methode is dat ze eenvoudig uit te voeren is en direct resultaat geeft. Een nadeel is dat je meestal veel instanties moet bekijken om zinnige voorspellingen over het gedrag van de algoritme te doen en dat deze testmethode dus tijdrovend is. Een ander nadeel is dat bij gebleken ongeschiktheid (de algoritme is te traag) helemaal niet duidelijk is wat de oorzaak van die ongeschiktheid is. Deugt de hele aanpak niet, of is een bepaald onderdeel van het programma veel te lang bezig? Tenslotte zijn extrapolaties natuurlijk altijd onzeker.

1.2.2 Voorspellen is beter

Als je toch een beschrijving van de algoritme hebt voordat je begint, kun je misschien eens kijken of je vantevoren een schatting kunt maken van het *aantal* stappen dat die algoritme gaat doen om je probleem op te lossen. Als je dan weet hoeveel tijd een enkele stap duurt, kun je het aantal stappen gewoon met de stapduur vermenigvuldigen om een tijdsduur voor de hele uitvoering te krijgen. Laten we dit eens verduidelijken aan de hand van een overbekend voorbeeld.

Stel ik wil het telefoonnumer van een vriend opzoeken in een telefoonboek. Het telefoonboek bevat ongeveer 25000 namen, gesorteerd op alfabet. Er zijn verschillende manieren om dat te doen. Als ik geen ervaring heb met zoeken, zal ik mogelijk op bladzijde 1 beginnen en dan net zolang de bladzijden omslaan totdat ik de naam van mijn vriend gevonden heb. Deze methode heet in de algoritmiek *linear search* en wordt door vrijwel niemand voor het opzoeken van een naam in een telefoonboek gebruikt (waarom?). Toch zijn er situaties denkbaar waarin linear search de best mogelijke algoritme is.

We zien onmiddelijk dat het aantal namen dat zo de revue passeert zeer afhankelijk is van de naam van onze vriend. Als deze naam met een A begint, zullen we zeer veel minder tijd kwijt zijn dan wanneer deze naam met een Z begint. Erger nog is het als de naam niet in het boek voorkomt (geheim nummer). We zullen dan alle 25000 namen voorbij zien komen om uiteindelijk gefrustreerd te worden. Gemiddeld zullen wel ergens in het midden eindigen, dwz ongeveer 12000 namen moeten onderzoeken. Toch is er maar 1 naam waarvoor we precies 12000 namen moeten bekijken voordat we hem gevonden hebben. De vraag is nu:

wat beschouwen we als kosten van „de algoritme" los van de precieze naam die we zoeken. De algoritme is tenslotte de methode „begin bij bladzijde 1 en zoek in volgorde totdat je de naam vindt" en van die algoritme willen we graag de kosten weten zonder te hoeven zeggen om welke naam het precies gaat.

Verschillende Gevallen van Analyse

Uit het bovenstaande blijkt al dat er een aantal verschillende antwoorden op de vraag „hoeveel kost de algoritme" mogelijk is. Drie antwoorden worden vaak onderscheiden: de *worst case*, de *best case* en de *average case*.

> We kunnen antwoorden: „25000 stappen". In het ergste geval lopen we de hele lijst door. Dit heet *worst case* analyse
>
> We kunnen antwoorden: „1 stap". In het gunstigste geval vinden we de naam onmiddellijk. Dat heet *best case* analyse
>
> We kunnen antwoorden: „12500 stappen". Er zijn net zoveel namen die minder dan 12500 stappen kosten als namen die meer kosten (aangenomen dat alle namen in het telefoonboek staan). Dit heet *average case* analyse

De best case analyse komt niet zo vaak voor, omdat deze voorspelling niet zoveel waarde heeft. De average case analyse komt ook niet zo vaak voor omdat deze vorm van analyseren vaak moelijker is en bovendien ook afhankelijk van wat voor kansverdeling er voor onze invoer geldt. Namen die beginnen met 'A' komen in de praktijk veel vaker voor dan namen die beginnen met 'Q' en het is helemaal niet waar dat alle namen in het telefoonboek voorkomen. We zullen ons dus voornamelijk bezighouden met worst case analyse. In de worst case doet onze aanpak dus net zoveel stappen als er namen in het telefoonboek staan. Als elke vergelijking 0,025 seconden kost, dan betekent dat dus in dit geval de de algoritme 625 seconden doet over het vinden van een willekeurige naam.

Wat is een Stap?

In onze analyse hebben we het over „25.000 stappen", en we bedoelen met één stap één vergelijking van de gezochte naam met een naam in het telefoonboek. In werkelijkheid gebeurt er natuurlijk veel meer. De file met namen moet worden geopend en gesloten, de namen worden niet in één stap vergeleken, maar letter voor letter, er moet een teller bijgehouden worden die zegt of we al aan het einde van het telefoonboek zijn gekomen en nog veel meer. Toch is het zinvol om voor de complexiteit van de algoritme te tellen hoeveel keer twee namen met elkaar worden vergeleken en de rest te verwaarlozen. Dat komt omdat het vergelijken van twee namen in deze algoritme het vaakst voorkomt. Elke andere operatie komt minder vaak voor. Verder is het aantal lettervergelijking dat per naamvergelijking wordt uitgevoerd begrensd door de lengte van de naam die we zoeken. Als we dus alle operaties bij elkaar zouden optellen, dan is er een (constant) getal zodat de totaal aantal operaties begrensd wordt door dit constante getal keer het aantal naamvergelijkingen. Verder is elke individuele operatie te vergelijken met elke andere individuele operatie doordat er een ander constant getal is dat het verschil in tijd nodig om die operaties uit te voeren aangeeft (bijvoorbeeld vermenigvuldigen van twee 8-bits getallen kan 10 keer zo duur zijn als het optellen van twee 8-bits getallen). Tot slot verschillen computers die we willen bekijken nog van elkaar doordat elementaire operaties op de ene computer een (constant) aantal keren sneller kunnen worden uitgevoerd. Als we dus willen weten hoe lang het zoeken in *werkelijke tijd* kan gaan duren, dan kunnen we aan de hand van het aantal naamsvergelijkingen daarvoor een goede bovengrens krijgen door dat aantal met de juiste constanten te vermenigvuldigen. Als we een algoritme willen analyseren zoeken we dus vaak een operatie die minstens net zo vaak wordt uitgevoerd als alle andere. Het kan zijn dat dat voor verschillende delen van de algoritme een andere moet zijn, bijvoorbeeld als we eerst gaan sorteren en dan zoeken, maar dan tellen we de complexiteit van de verschillende delen van de algoritme gewoon bij elkaar op. Van deze operatie, die in Engelse teksten vaak barometer genoemd wordt, maar die we hier bij gebrek aan een betere naam voorlopig maar „spil" zullen noemen, geven we een (boven)schatting voor hoe vaak zij wordt uitgevoerd, en dat is dan een maat voor de kosten van de algoritme.

Samengevat zijn de stappen in de analyse van een algoritme dus:

- Verdeel de algoritme en een aantal samenhangende delen (bijvoorbeeld loops).
- Zoek binnen elk deel een elementaire operatie die het vaakst wordt uitgevoerd, en tel het aantal keren dat die wordt uitgevoerd.
- De (worst-case) complexiteit van zo'n deel is een constante maal het aantal keren dat die elementair operatie wordt uitgevoerd.
- De (worst-case) complexiteit van de algoritme is een constante maal de worst-case complexiteit van het duurste onderdeel.

Verderop in de tekst zullen we een handige notatie, de O-notatie voor deze complexiteit invoeren.

Verschillende Algoritmen, verschillende complexiteit

De analyse van algoritmen levert een vergelijking tussen verschillende methoden die we kunnen gebruiken om voor een probleem de beste algoritme te kiezen. In een geval als het bovenstaande kunnen we gebruik maken van het feit dat de lijst namen gesorteerd is, om een aanzienlijk snellere algoritme te krijgen. Deze methode heet *binary search* en gaat als volgt: Vergelijk de naam die je zoekt met de naam die in het midden van het boek staat, en deel vervolgens het boek in twee delen. Als de naam die je zoekt *groter* is dan de naam die je tegenkomt gebruik je vervolgens het tweede deel en anders het eerste deel. Dit in tweeën delen herhaal je totdat je nog één naam over hebt. Als dat de naam is die je zoekt, dan heb je hem gevonden en anders staat die naam niet in het boek. Deze methode doet in een telefoonboek van 25000 stappen maximaal slechts 15 stappen (de 2-logaritme van 25000, naar boven afgerond) een dramatische verbetering.

Binary search maakt gebruik van een eigenschap van de invoer die het zoeken makkelijk maakt, namelijk dat de invoer alfabetisch gesorteerd is. Deze eigenschap is, zeker voor mensen, noodzakelijk bij een zo grote lijst omdat deze anders onhanteerbaar wordt. De eigenschap kan echter niet van elke lijst vooronderstelt worden en is ook niet altijd nodig. Als de lijst met namen niet groter dan vijf is bijvoorbeeld lijkt het weinig zinvol eerst de lijst te gaan sorteren om het zoeken makkelijker te maken. Lineair zoeken in zo'n lijst kost namelijk vijf vergelijkingen en binair zoeken kost maximaal drie vergelijkingen. Sorteren van zo'n lijst kan, afhankelijk van de gebruikte sorteeralgoritme, wel 20 vergelijkingen kosten. Wanneer is het gewenst de lijst te sorteren?

Mengvormen en een Nieuwe Analyse

Als we een grote ongesorteerde lijst hebben, dan kan het voordelig zijn de lijst eerst te sorteren. Een lijst van 25000 namen sorteren kan met ongeveer 375000 vergelijkingen worden gedaan. Dat is natuurlijk duur, maar het voorwerk dat we gedaan hebben, winnen we terug omdat elke volgende opzoekactie hoogstens 15 vergelijkingen kost in plaats van de gemiddeld 12500 vergelijkingen in de ongesorteerde lijst. Na ongeveer 31 keer een naam opzoeken hebben we de voorinvestering van 375000 vergelijkingen weer terugverdiend. Gemiddeld (average case) is het dus voordeliger de lijst eerst te sorteren en dan de gesorteerde lijst te gebruiken, dan te werken met de ongesorteerde lijst.

Wat als er maar één opzoekactie plaatsvindt? Dan is het onzin om te investeren in het sorteren van de lijst. Dus analyse van het gemiddelde geval is alleen zinvol als we zeker weten dat er veel goedkope acties worden uitgevoerd, die tegen de dure acties kunnen worden weggestreept.

Uitgesmeerde kosten Er is een methode die dat zeker maakt, namelijk: voer altijd *eerst* een heel stel goedkope acties uit voordat een dure actie wordt uitgevoerd. Een voorbeeld uit de praktijk hierbij is het huren van ski's. Het kopen van ski's is relatief duur, maar als je ski's koopt en je gaat vaak skiën haal je dat er op den duur uit. Als je ski's koopt en je gaat niet vaak—omdat je het niet leuk vindt, of omdat er geen sneeuw valt—zijn gekochte ski's relatief duur. De oplossing voor dit probleem is eerst ski's te huren, net zolang totdat je het bedrag hebt uitgegeven dat ski's in de winkel kosten. De volgende keer koop je ze. Op

deze manier hoef je per skigelegenheid nooit méér dan twee keer de optimale prijs te betalen, hoe vaak je ook gaat. In ons geval zouden we ons kunnen voorstellen dat de lijst met namen in het telefoonboek niet altijd heel groot is, maar leeg begint en langzaam groeit. Een opzoekactie zou dan, zo lang de lijst klein genoeg is gewoon lineair zoeken kunnen zijn en alleen als de lijst onhanteerbaar groot wordt, zouden we kunnen besluiten eerst de lijst te sorteren.

Deze aanpak garandeert ons dat we eerst een heleboel goedkope acties hebben uitgevoerd (lineair zoeken in een kleine lijst; nieuwe namen ongesorteerd achter aan de lijst plakken) voordat we een dure operatie (sorteren) uitvoeren om nieuwe zoekacties goedkoper te maken. Als we dan een dure gesorteerde lijst hebben, kunnen nieuwe opzoek operaties snel en nieuwe invoegoperaties kunnen we weer doen met een kleine ongesorteerde lijst, net zo lang tot die lijst weer onhanteerbaar groot wordt. Vervolgens kunnen we de twee lijsten weer samenvoegen tot een nieuwe, grotere, gesorteerde lijst.

De analyse waarbij we de kosten van een dure operatie uitsmeren over goedkopere *eerdere* operaties heet *uitgesmeerde-kostenanalyse* (in Engelse teksten amortized case analysis). Ze wordt veel gebruikt als er datastructuren in het spel zijn die het zoeken vereenvoudigen. Bijvoorbeeld zoekbomen kunnen scheefgroeien door toevoegen van data en sommige zoekacties kunnen daardoor zeer duur worden. Een zoekboom kan gebalanceerd worden, maar dat is een dure operatie en die moet dus nuttig zijn. We beginnen vrijwel altijd met niets (lege zoekboom) en kunnen dan goedkope invoegacties en zoekacties doen net zolang tot de boom te groot en te scheef wordt om zoekacties nog goedkoop te kunnen doen. De volgende stap is dan het balanceren van de boom. Hashtabellen worden op een bepaalde grootte geïnitialiseerd om een aantal records te kunnen opslaan. Als het aantal records groter wordt dan de tabel, moet deze worden uitgebreid, maar een uitbreiding van de tabel met één veld is een dure operatie. Analyse van dit geval leert dat het voordelig is om de tabel, als hij te groot wordt, meteen maar twee keer zo groot te maken, en, symmetrisch, als er records verdwijnen, de tabel te halveren als het aantal records 1/4 van het maximaal mogelijke aantal is (de lezer kan hier gemakkelijk inzien waarom voor 1/4 in plaats van voor 1/2 gekozen is).

Een ander, misschien niet zo realistisch, maar wel inzichtelijk voorbeeld is dat waarbij we een element op een bepaalde plaats in een stapel willen invoegen (waarbij dan de stapel verandert). We moeten eerst een aantal elementen van bovenaf de stapel halen, om vervolgens het nieuwe element op de resterende stapel te zetten. De verwijderde elementen worden niet meer teruggezet. Elke operatie kan ons dwingen om de stapel leeg te maken. Een worst-case analyse zou dus zeggen dat elke operatie op een stapel van grootte n maximaal n stappen kan kosten, maar het afstapelen maakt de maximale kosten voor volgende afstapeloperaties kleiner! Elke operatie houdt een aantal keren afstapelen en een keer opstapelen in, maar in een rij van n van deze operaties, kan het *totaal* van de afstapeloperaties niet meer dan n zijn, omdat elk element maar één keer afgestapeld kan worden, en dat geld voor elke n. Een totaal van n van deze operaties, kan zo niet meer dan $2n$ stappen kosten, dus elke stap kan niet meer dan 2 operaties kosten, als we kosten dure stappen over vorige goedkopere stappen mogen uitsmeren.

De hier gepresenteerde analyse (ook wel „aggregate analysis" genoemd) is een vrij groffe vorm van analyseren: tel alle operaties bij elkaar op en deel door het aantal operaties. Dat is natuurlijk toegestaan, maar wijst niet deze vorm van uitgesmeerde complexiteit precies aan goedkope en dure operaties toe. Hieronder zullen we twee methode van analyse bespreken die dat wel doen, en daardoor nauwkeuriger zijn.

Een voorbeeld van Uitgesmeerde Complexiteit

We kunnen het verschil tussen worst-case analyse en amortized-case analyse zien door naar een binaire teller te kijken. Stel we hebben een binaire teller van k plaatsen en we willen met deze teller tellen tot 2^k. Hoeveel kost dat? Laten we eerst een worst-case analyse doen. Een teller heeft k bits. Als spil voor deze algoritme zullen we de bitflip gebruiken. Als de teller één keer wordt opgehoogd, dan zal dat dus ten hoogste k bitflips kosten. De teller wordt maximaal 2^k keer opgehoogd, dus de totale kosten zullen begrensd blijven door $k \times 2^k$. De worst-case analyse is eenvoudig, maar geeft geen reëel beeld van de werkelijke kosten. Immers niet in *elke* stap worden k bits geflipt. Sterker nog, in de helft van de gevallen wordt maar één bit geflipt. In een kwart van de gevallen worden slechts twee bits geflipt en in 1/8 van de gevallen 3 bits. Als we simpel tellen wat het totaal aantal bitflips in alle 2^k gevallen is, dan komen we tot de volgende som $\frac{2^k}{2} \times 1 + \frac{2^k}{4} \times 2 + \frac{2^k}{8} \times 3 + \ldots + \frac{2^k}{2^k} \times k$. Dit is een bekende reeks. In Sectie 1.4.3 waarin we onze kennis

stap	teller	kosten	saldo	potentiaal	stap	teller	kosten	saldo	potentiaal
1	00001	1	1	1	9	01001	1	2	2
2	00010	2	1	1	10	01010	2	2	2
3	00011	1	2	2	11	01011	1	3	3
4	00100	3	1	1	12	01100	3	1	2
5	00101	1	2	2	13	01101	1	2	3
6	00110	2	2	2	14	01110	2	2	3
7	00111	1	3	3	15	01111	1	4	4
8	01000	4	1	1	16	10000	5	1	1

Figuur 1.1: Uitgesmeerde complexiteit

over het manipuleren van reeksen opfrissen, zullen we zien dat deze reeks voor elke k begrensd is door 2×2^k en dat dus *gemiddeld* het aantal bitflips dat bij één keer ophogen van de teller kan worden afgeschat met 2 in plaats van k flips. Om te bewijzen dat ook de uitgesmeerde complexiteit van de bitflips begrensd wordt door 2 moeten we aantonen dat dat ook het geval is als de teller niet 2^k keer wordt opgehoogd, maar ergens tussen de 1 en de 2^k keer. Hiervoor zijn een tweetal truuks bedacht, de „bankrekening" en de „potentiaal". Omdat deze truuks vaak gebruikt worden, spreekt men van „methoden".

De bankrekening

Een oerhollandse methode om voorbereid te zijn op onverwachte uitgaven is sparen. Zo kunnen we ook de uitgesmeerde complexiteitsanalyse opvatten. In de Engelse teksten wordt deze analyse aangeduid met amortized complexity, wat associaties oproept met hypotheken. In het geval van hypotheken vindt echter altijd eerst een grote lening plaats, waardoor de spaaranalogie misschien een betere is voor deze vorm van analyse. We weten uit de telpartij hierboven dat over een totaal van 2^k verhogingen van de teller niet meer dan 2×2^k bitflips gemaakt worden of, gemiddeld, niet meer dan 2 per verhoging. Stel nu dat we voor elke verhoging 2 operaties in rekening brengen in plaats van het werkelijke aantal operaties. Hebben we dan altijd genoeg „spaartegoed" om voor alle werkelijk uitgevoerde operaties te betalen? Laten we voor een klein geval eens kijken.

Het lijkt erop dat de bankrekening altijd voldoende saldo heeft om de tekorten te dekken als de twee operaties die we steeds in rekening brengen niet genoeg zijn. Hoe bewijzen we dat? Op dit punt doet de lezer er goed aan de tekst even terzijde te leggen en over deze vraag na te denken.

Kijk naar het verloop van het tegoed op de bankrekening. Telkens valt het tegoed terug naar 1, vlak nadat een groot aantal bits geflipt zijn. Voor dat moment loopt het tegoed net genoeg op om voor deze bitflips te kunnen betalen. We zien dat tussen de tijd waarin de meest rechtse i bits op 0 staan en de tijd waarop de meest rechtse i bits op 1 staan, het banktegoed precies i groter wordt dan het daarvoor was.

Hiervan maken we een inductiehypothese: "Als de meest rechtse i bits op 0 staan, dan is het banktegoed i groter geworden vóór dat de meest rechtse i bits op 1 staan."

Begin met $i = 1$. Het kost 1 stap om het meest rechtse bit van 0 in 1 te veranderen, en er worden 2 munten in rekening gebracht. Het banktegoed groeit dus met 1. Neem vervolgens aan dat de hypothese waar is voor één of andere i en kijk naar een tijdstip t_0 waarop de meest rechter $i+1$ bits op 0 staan. Eerst krijgen we een tijdstip waarop de meest rechter i bits op 1 staan en het $i+1$-e bit nog op 0. Wegens onze aanname is nu het banktegoed i groter dan het was. In de volgende stap worden 2 munten op de rekening gezet, i munten worden gespendeerd om de meest rechtse i bits op 0 te zetten, en 1 munt om het $i+1$e bit op 1 te zetten, waardoor het banktegoed precies 1 groter is dan het op t_0 was $(2 + i - (i + 1))$. Als de volgende keer de meest rechter i bits op 1 staan (nu dus op $i + 1$), dan is het banktegoed $i + 1$ groter dan het op tijdstip t_0 was.

De potentiaalmethode

Een nadeel van de bankrekeningmethode is dat je van tevoren een goed idee moet hebben hoeveel je voor een operatie moet sparen om hem te kunnen uitvoeren. Een meer dynamische aanpak is de potentiaalmethode. Deze is afgeleid van de volgende gedachte. Stel ik ben een karretje over een weg aan het duwen. Elke meter ik afleg kost me een bepaalde hoeveelheid boterhammen (energie). Als ik het karretje over een vlakke weg duw dan kost elke meter mij evenveel energie. De enige energie die moet worden toegevoegd, is de energie die door wrijving warmte wordt. Echter, als ik het karretje een heuvel opduw, dan kosten de heuvelop stappen ineens meer energie, omdat het karretje *potentiaal* wint. Al die energie krijg ik volgens de behoudswetten weer terug wanneer het karretje van de heuvel afrolt, omdat de zwaartekracht dan helpt duwen.

Laten we dit eens vertalen naar de kwestie van uitgesmeerde complexiteit. Stel ik heb een functie $\phi(n)$ die de potentiaal van de algoritme na de n-de stap voorstelt. Het verschil $\phi(n) - \phi(n-1)$ is de *verandering* van de potentiaalfunctie tussen stap $n-1$ en stap n. We nemen aan dat $\phi(0) = 0$ en dat $\phi(n) \geq 0$ voor alle n. Laat $t(n)$ het aantal stappen zijn dat in de n-de stap moet worden uitgevoerd en definieer $\hat{t}(n)$ als $t(n)+\phi(n)-\phi(n-1)$. Nu geldt voor N stappen $\sum_{n=1}^{N} \hat{t}(n) = \sum_{n=1}^{N}(t(n)+\phi(n)-\phi(n-1)) = \phi(N)+\sum_{n=1}^{N} t(n)$ Dus $\sum \hat{t}(n)$ is altijd *minstens* zo groot als $\sum t(n)$, dus $\sum \hat{t}(n)$ is een *bovengrens* voor het aantal operaties dat gedaan wordt *voor elke* n. Om nu een goede schatting te krijgen voor de uitgesmeerde complexiteit van het probleem hoeven we slechts een afschatting te maken voor $\hat{t}(n)$ voor een geschikte functie ϕ. Net zoals het bij de bankmethode een probleem was om een geschikte inductiehypothese te vinden is het hier natuurlijk een probleem om een geschikte potentiaalfunctie te vinden. Deze functie moet de eigenschap hebben dat zij aan het begin 0 is, nooit kleiner dan 0 wordt en altijd voldoende groot is om de kosten op te vangen. Ongeveer net zoals wanneer we 's morgens met weinig energie, maar met een volle pot koffie op het werk aankomen. We kunnen goedkope kopjes koffie uit de pot tappen en daarmee energie opdoen. Tegen de tijd dat de pot leeg is, hebben we voldoende energie verzameld om de dure operatie, het zetten van een nieuwe pot, te gaan ondernemen.

In het geval van de binaire teller kiezen we als potentiaalfunctie het aantal bits in de teller dat op 1 staat en berekenen dan de uitgesmeerde kosten van een stap, $\hat{t}$. Deze potentiaalfunctie voldoet aan de eisen. Zij begint met 0, wordt onderweg nooit kleiner dan 0, en als er een dure operatie nodig is (veel bitflips) dan staan er ook veel bits op 1. Laten we kijken of deze functie ook precies aan onze behoeften voldoet. Er zijn 3 gevallen: Als er een even getal in de teller zit, dan wordt het rechterbit geflipt van 0 naar 1 en de potentiaal stijgt met 1, kosten $1+1=2$. Als alle bits in de teller 1 zijn, dan worden alle k bits geflipt, maar de potentiaal valt naar 0, kosten $k-k=0$. In alle andere gevallen worden er i bits van 1 naar 0 gezet, 1 bit wordt van 0 naar 1 gezet, en de potentiaal daalt met $i-1$. De totale kosten zijn dan $1+i-(i-1)=2$.

1.2.3 Sommen

1. De bedoeling van deze som is oefenen met het meten van executieduur van programma's waarbij verschillende algoritmen gebruikt worden. Zoals we hierboven al gezien hebben zijn er verschillende manieren om tegen zoeken en sorteren aan te kijken. We voeren het volgende experiment uit. Elke keer trekken we een random getal tussen de 1 en de n, voegen dat getal in, in de structuur die we gemaakt hebben. Dit doen we n keer. Merk op: een getal kan meerdere keren voorkomen. Vervolgens trekken we m keer een willekeurig getal kleiner dan n en zoeken dat in het array op. De verschillende gevallen zijn de volgende.

   (a) Een ongeordend array. Een nieuw getal wordt steeds achteraan ingevoegd.

   (b) Een geordend array. Elke keer als een nieuw getal wordt toegevoegd zetten we het getal wel achteraan de rij, maar vervolgens wordt de rij gesorteerd met quicksort.

   (c) Een geordend array. Elke keer wordt een nieuw getal ingevoegd met insertion sort.

   (d) Een mengvorm. We houden het array een tijdlang ongeordend, maar als het aantal elementen „te groot" wordt, wordt het hele array opnieuw gesorteerd.

   Implementeer deze gevallen voor verschillende waarden van n en m. Tussen de opdrachten door roepen we de klok (of time afhankelijk van het gebruikte operating system) aan en noteren de tijden die met de

# stappen	tijd	commentaar
1.000	0.0001 sec	
1.000.000	0.001 sec	
10.000.000	0.01 sec	onzichtbaar
100.000.000	0.1 sec	waarneembaar
1.000.000.000	1 sec	
10.000.000.000	10 sec	
100.000.000.000	100 sec	koffie
1.000.000.000.000	16.7 min	praatje
10.000.000.000.000	2uuur 47 min	GRRR
100.000.000.000.000	27 uur	geduld?
1.000.000.000.000.000	11.25 dagen	opgeven?
10.000.000.000.000.000	4 maanden	
100.000.000.000.000.000	3.33 jaar	
1.000.000.000.000.000.000.	33 jaar	
10.000.000.000.000.000.000	3.3 eeuwen	

Figuur 1.2: Tijd versus aantal stappen op een 1GIPS processor

klokfunctie verkregen worden. Maak een tabel. Voor welke n en m gaan de verschillen tellen? Waarom verwachtte je dat?

1.3 Asymptotische Grenzen

Vaak zijn we niet zo geïnteresseerd in de kosten van het zoeken van een naam in één bepaalde lijst, want als zo'n zoekactie gemiddeld 12500 stappen kost, wat zegt dat dan over de gebruikte algoritme? We willen graag weten wat zo'n algoritme kost op *verschillende* instanties van het probleem. Dus, wat als we in het algemeen een lijst van n namen aan de algoritme geven en we vragen of een bepaalde naam in die lijst voorkomt. Voor de twee algoritmen uit de vorige sectie kunnen we zien dat de eerste zoekmethode ongeveer n stappen kost, terwijl de tweede zoekmethode ongeveer $\log n$ stappen kost. Voor steeds groter wordende n zien we dat de eerste zoekmethode dus een stuk duurder is dan de tweede.

Het is meer informatief te weten hoe een algoritme zich gedraagt op steeds groter wordende vragen dan op een enkele instantie van een probleem. Met andere woorden: we willen graag zien hoe het aantal stappen dat een algoritme moet doen om een oplossing te vinden *groeit* als functie van de grootte van de vraagstelling. Als een algoritme over een twee keer zo grote vraag twee keer zo lang doet, vinden we dat in principe geen probleem. Zouden we het wel een probleem vinden als een algoritme over een twee keer zo grote vraag tien keer zo lang doet? Wat zijn functies die een redelijke verhouding geven tussen de grootte van de vraagstelling—de lengte van de invoer—en de grootte van het probleem? Laten we, als voorbeeld, eens kijken naar hoeveel tijd er gemoeid is met het doen van een aantal stappen op een machine die 10^9 bewerkingen per seconde kan uitvoeren (1GIPS). Zie Figuur 1.2.

Als we een machine, al is die nog zo snel, maar voldoende werk geven, wordt de werktijd vanzelf onacceptabel lang. Een tegenwerping zou kunnen zijn dat in de dagelijkse praktijk een berekening die 10^{20} operaties kost niet voorkomt. Dit is echter een misvatting die veel met algoritmiek te maken heeft. Laten we eens naar een voorbeeld kijken waar de keuze van de verkeerde algoritme kan leiden tot excessief processorgebruik, de berekening van Fibonacci getallen, genoemd naar de Italiaanse wiskundige Leonardo Pisano (1170–1250), bijgenaamd Fibonacci.

In zijn boek „Liber Abaci" beschrijft Fibonacci een groot aantal problemen interessant voor koopmannen. Eén van deze problemen is het volgende:

> Een bioloog zet een paar konijnen in een afgesloten ruimte. Hoeveel paren konijnen heeft zij na een jaar, aangenomen dat elk paar elke maand een nieuw paar produceert, dat zelf na de tweede

	n	n^2	n^3	2^n	3^n	$n!$
10	10	100	10^3	1024	59049	3628800
20	20	400	8×10^3	1.04×10^6	3.48×10^9	2.43×10^{18}
40	40	1600	6.4×10^3	1.1×10^{12}	1.2×10^{19}	8.2×10^{47}
80	80	6400	5.12×10^5	1.2×10^{24}	1.4×10^{38}	7×10^{118}
100	100	10^4	10^6	1.2×10^{30}	5.1×10^{47}	9.3×10^{157}
500	500	2.5×10^4	1.25×10^8	3.2×10^{150}	3.6×10^{238}	1.2×10^{1134}
1000	1000	10^6	10^9	1.7×10^{301}	1.3×10^{477}	4×10^{2567}

Figuur 1.3: Functies en aantallen operaties

maand productief wordt?

De oplossing is de overbekende Fibonaccireeks of rij van Fibonacci $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$ (Fibonacci zocht het twaalfde element van deze rij). Hier gaat het ons niet om het vinden van de oplossing, maar om hoeveel stappen het op een computer kost om de oplossing te vinden. Een programmeur kan zich er toe laten verleiden te observeren dat het n-de Fibonaccigetal kan worden gevonden door de twee vorige bij elkaar op te tellen. Zij schrijft een programma als volgt.

```
1: F(n)
2: if n = 1 or n = 2 then
3:     return(1);
4: else
5:     return(F(n − 1) + F(n − 2));
6: end if.
```

Als wij ons afvragen hoeveel rekenstappen zo'n programma kost, dan zien we dat we om het 100e getal uit te rekenen moeten weten wat het 99e en het 98e getal is, en dat we daarvoor moeten berekenen wat het 97e en het 96e getal is enzovoort. In dit programma echter wordt het 98e getal twee keer uitgerekend, zowel in de berekening van het 100e getal als van het 99e getal, en beide berekeningen roepen zelf weer twee andere berekeningen aan. Elke berekening zal de berekening van twee andere getallen vereisen, behalve als $n = 1$ of $n = 2$. De berekening van het 100e Fibonacci getal zal met dit programma ongeveer 2^{100} stappen kosten (we zullen dit verderop nog precies maken), en omdat 2^{10} ongeveer 10^3 is, zijn dit 10^{30} operaties, een aantal dat ons op pagina 21 nog onrealistisch voorkwam. In het geval van de Fibonacci-getallen is het niet de aard van het probleem dat de oplossing ingewikkeld maakt. Immers, als we *onthouden* dat het derde fibonacci getal 2 is in plaats van dat telkens weer uit te rekenen, hebben we „slechts" 100 tussenresultaten nodig, die elk uit de twee voorgaande opgeslagen tussenresultaten kunnen worden berekend, dus er zijn slechts ongeveer 200 stappen nodig om het 100e Fibonacci-getal te berekenen.

Het verschil tussen de beide bovenbeschreven algoritmen is het verschil tussen een algoritme met *exponentiële* (2^n) looptijd versus een algoritme met *lineaire*[2] (n) looptijd. Het beantwoorden van een vraag van grootte n kost in het eerste geval 2^n en in het tweede geval slechts n stappen. Het verschil in aantal stappen voor verschillende waarden van n en een aantal van deze functies wordt in Figuur 1.3 nog eens duidelijk gemaakt.

We zien aan deze tabel dat voor betrekkelijk kleine vragen (1000 bits is natuurlijk helemaal niet zo groot) een algoritme die loopt in tijd $n!$ op onze eerdere 1 gigaflops machine onevenredig veel tijd in beslag neemt. Computers worden echter steeds sneller en het is doorgaans slechts een kwestie van een paar maanden voordat er een computer is die twee keer zo snel is. Dit verschijnsel wordt Moore's law genoemd. Toch zijn er op verschillende manieren grenzen aan te geven aan de snelheid die door machines bereikt kunnen worden. In de eerste plaats is er de lichtsnelheid, en het feit dat om een waarneembaar verschil tussen 0 en 1 te hebben, een electronische schakeling bepaalde minimale afmetingen moet hebben. Hieruit kan worden afgeleid dat een machine die een algoritme uitvoert waar de ene stap de andere moet opvolgen nooit meer dan ongeveer

[2]Strikt genomen kan de vraag: „Wat is het n-de Fibonacci getal?" in ongeveer $\log n$ bits worden gesteld. Het uitvoeren van n stappen betekent dus in de lengte van de invoer nog steeds exponentiële tijd. Meer hierover later.

10^{35} stappen per seconde zal kunnen doen. Voor deze machine en invoeren van 1000 bits, toont de tabel aan dat de tijd die deze machine moet gebruiken om een oplossing te vinden, voor een algoritme die bijvoorbeeld $n!$ stappen doet, onacceptabel groot is.

In de tweede plaats is er een veel eerder bereikte grens, die van de energiedissipatie. Als een machine een oplossing vindt voor de optelling $1+1$, dan is met het vervangen van de vergelijking door de oplossing, 2, de informatie waar deze oplossing vandaan komt verloren. De berekening is *onomkeerbaar*. Dit informatieverlies of toename van de entropie heeft ontwikkeling van warmte tot gevolg. Deze warmte moet ergens naartoe, en wel naar buiten door de oppervlakte van de chip. Een manier om deze energiedissipatie te verminderen is het verlagen van de voedingsspanning van de chip. Hoe lager de voedingsspanning echter, hoe moeilijker het verschil tussen 0 en 1 te zien is en dus hoe eerder er fouten optreden. De huidige voedingsspanning van ongeveer 5V lijkt dicht bij de grens te liggen van wat haalbaar is. Een tegenwoordig in de belangstelling staande aanpak van dit probleem is die van de omkeerbare berekening. In plaats van tussenresultaten weg te gooien kunnen ze ook bewaard blijven zodat van de uitvoer terug kan worden gerekend naar de invoer. Deze aanpak blijkt ook in de praktijk aanleiding te geven tot lagere energiedissipatie.

Hoewel het in de industrie al een tijdje niet meer zo erg hard gaat, vooral vanwege het energiedissipatie-probleem (laptop computers zijn de laatste tijd zelfs weer *langzamer* geworden), heeft het niet veel zin om bij het afschatten van de complexiteit van een probleem te kijken naar factoren als 2, of in het algemeen naar constante factoren. Immers, je moet dan altijd precies zeggen over welke computer het gaat. Er zijn nog andere, meer theoretische redenen om constante factoren in de berekening van de complexiteit van een probleem te verwaarlozen. Onder andere is er de „constant factor speedup" stelling uit de automatentheorie die zegt dat voor een bepaalde algoritme en een bepaalde constante altijd een machine te maken is die die algoritme die constante sneller uitvoert. We gaan hier in deze tekst niet dieper op in. Vaak gaat het in de algoritmiek erom een constante in de exponent aan versnelling te winnen, bijvoorbeeld om een algoritme die in tijd n^3 loopt te verbeteren tot een algoritme die in tijd n^2 loopt. Voor elke constante winst kunnen we in dergelijke gevallen natuurlijk n groot genoeg kiezen om het effect van de constante winst te doen verbleken.

In het algemeen zullen we problemen waarvan de rekentijd begrensd wordt door één of ander polynoom *efficiënt* oplosbare problemen noemen. Dit heeft verschillende redenen.

- In de eerste plaats blijkt dat in de praktijk voorkomende problemen waarvoor polynomiale algoritmen bestaan bijna altijd problemen zijn waarvoor algoritmen met polynomiale grenzen bestaan met een *kleine* exponent (2,3,4 heel soms 5) en deze algoritmen (zie tabel 1.3) gedragen zich op redelijke computers redelijk, dwz vragen van niet al te grote omvang kosten niet al te veel stappen.

- In de tweede plaats hebben polynomen de eigenschap dat ze zich laten samenstellen en dat de samenstelling weer een polynoom is. (Onthoud: een polynoom van een polynoom is een polynoom). Dus hebben we twee bewerkingen die polynomiale tijd begrensd zijn, dan kunnen we, ook omdat in polynomiale tijd niet meer dan polynomiaal veel bits als uitvoer gegenereerd kunnen worden die bewerkingen achter elkaar op de invoer loslaten, waarbij het totaal van de bewerkingen toch efficient blijft.

- In de derde plaats groeien polynomiaal begrensde algoritmen niet al te veel met de groei van de vraagstelling. Dat blijkt al uit de tabel die we hierboven presenteerden, maar ook, als de probleemstelling met een constante groeit (zeg verdubbelt), dan groeit het polynoom ook met een constante (zie: $(2x)^k = 2^k x^k$).

- Polynomiale tijd is een ondergrens voor machinemodel onafhankelijk redeneren. We zullen in het tweede deel zien dat theoretische modellen voor machines die van praktische voorbeelden zijn afgeleid elkaar kunen simuleren in polynomiaal begrensde tijd. Dwz wat de ene machine in n stappen kan uitvoeren kan de andere machine in n^k stappen uitvoeren voor één of andere constante k. Wil je het dus in je redenering niet over een specifieke machine of specifiek machinemodel hebben, dan zul je alle polynomen als grenzen op één hoop moeten gooien.

- Je zou kunnnen beweren dat lineaire tijd (probleemgrootte n kost n stappen) een ondergrens is voor een algoritme, omdat elke algoritme tenminste deze tijd kwijt is voor het lezen van de invoer.

Dit laatste is niet zo'n heel sterk argument omdat bijvoorbeeld zoeken in een gesorteerde rij niet de gehele invoer leest, anders zou een $\log n$ algoritme niet mogelijk zijn. Ook andere algoritmen kunnen volstaan met het slechts gedeeltelijk lezen van de invoer. We zien echter dat er genoeg argumenten bestaan om algoritmen waarvan de looptijd begrensd wordt door een polynoom in de lengte van de invoer samen te nemen en deze groep de „efficiënte" algoritmen te noemen.

1.3.1 Notatie voor de Complexiteit van Algoritmen

Nadat we beargumenteerd hebben dat voor de afschatting van de complexiteit van algoritmen constanten niet interessant zijn en dat het er bij de afschatting van de complexiteit van algoritmen vooral om gaat hoe deze algoritmen zich gedragen op steeds groter wordende invoeren, zullen we hier een paar symbolen invoeren waarmee we het gedrag van algoritmen op invoeren van lengte n kunnen noteren. Deze notaties hebben de eigenschap dat ze eenvoudige makkelijk te vergelijken functies geven die een goede afschatting zijn van de werkelijke complexiteit (altijd binnen een constante factor van de werkelijke complexiteit). Bovendien laten deze afschattingen zich samenstellen, dwz als je een stel afschattingen hebt voor gedeelten van de algoritme, dan kun je daaruit een afschatting voor de gehele algoritme verkrijgen. De afschattingen worden gedefinieerd als *klassen* van functies. De werkelijke complexiteit van een algoritme is dus altijd een element van de afschatting van een functie. Als we bijvoorbeeld kijken naar één van de meest gebruikte afschattingen O hieronder dan is de goede uitspraak: „de complexiteit van deze algoritme is *in* $O(n)$", genoteerd als $\in O(n)$, om een lineaire bovengrens aan te geven, hoewel in veel boeken gezegd wordt „...is *van* orde n", genoteerd als $= O(n)$. Deze taalvervlakking is van dezelfde orde als het gebruik van „het" algoritme in plaats van het correcte „de" algoritme. Helaas is dit niet tegen te gaan.

Bovengrenzen

De volgende bovenafschattingen voor functies zijn in gebruik

$O(f) = \{g : (\exists c, n_0 \in \mathbb{N})(\forall n > n_0 \in \mathbb{N})[g(n) \leq cf(n)]\}$ In de klasse $O(f)$ zitten zo alle functies die op den duur (dat wil zeggen voor voldoende grote n kleiner worden dan één of andere constante c keer $f(n)$. Voorbeeld $3n \in O(n^2)$ omdat vanaf $n = 3$ geldt dat $3n \leq n^2$, maar ook is $3n$ al kleiner dan of gelijk aan $3n^2$ vanaf $n = 1$.

$o(f) = \{g : \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0\}$. In $o(f)$ zitten alle functies die *echt* kleiner worden dan f. Vaak zie je dat $g \in O(f)$ en $g \in o(f)$ samengaan, maar bijvoorbeeld $\sin n \in O(1)$ maar niet in $o(1)$, terwijl bijvoorbeeld wel $\sin n \in O(n)$.

Ondergrenzen

De volgende onderafschattingen voor functies zijn in gebruik

$\Omega(f) = \{g : (\exists \epsilon > 0, \exists^\infty n_i) \left[\frac{g(n_i)}{f(n_i)} > \epsilon\right]\}$. Opmerkingen:

1. De notatie $\exists^\infty$ is niet zo heel bekend. Ze betekent "er zijn oneindig veel"
2. Deze definitie maakt van Ω de tegenhanger van o bij ondergrenzen. In een groot aantal teksten figureert Ω ook als de tegenhanger van O, en is zij gedefineerd als: $\Omega(f) = \{g : (\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N})(\forall n > n_0 \in \mathbb{N})[g(n) \geq cf(n)]\}$. Wij zullen hier aan de eerste definitie vasthouden om historische redenen, hoewel het in de praktijk vaak niet zoveel uitmaakt welke van de twee definities je gebruikt. Ook maakt het in de praktijk niet zoveel uit of we nu eisen dat $c \in \mathbb{N}$ of $c \in \mathbb{R}^+$. In het laatste geval geldt weliswaar niet dat $n^2 \in \Omega\frac{n^2}{2}$ en in het eerste geval wel, maar dergelijke gevallen doen zich in de praktijk nauwelijks voor.

$\omega(f) = \{g : \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0\}$

Dubbele grenzen

Naast boven- en ondergrenzen zijn er ook nog twee notaties voor simultane onder en bovengrenzen.

$$\theta(f) = \{g : (\exists c_1, c_2 \in \mathbb{R}, n_0 \in \mathbb{N})(\forall n > n_0 \in \mathbb{N})[c_1 f(n) \leq g(n) \leq c_2 f(n)]\}$$

$$\sim (f) = \{g : \lim_{n\to\infty} \tfrac{g(n)}{f(n)} = 1\}$$

Al deze notaties hebben de eigenschap dat ze simpele betekenisvolle grenzen kunnen geven voor ingewikkelde algoritmen. Een algoritme bestaat namelijk vaak uit veel gedeelten die allemaal hun eigen complexiteit hebben. Het programma kan bestaan uit verschillende, vaak geneste loops, procedure en function calls etc., allemaal met eigen grenzen aan de rekentijd. Deze grenzen kunnen we apart bepalen en vervolgens optellen. Het grote voordeel van de notatie die we hier hebben afgesproken is dat het optellen van al die onderdelen niet leidt tot een grote, lelijke, onoverzichtelijke uitdrukking. Er geldt bijvoorbeeld de optelregel voor O die zegt dat $O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$. Als dus bijvoorbeeld een programma bestaat uit een loop die $O(n^2)$ keer wordt uitgevoerd, gevolgd door een loop van complexiteit $O(n^3)$, dan is de totale complexiteit van het programma $O(n^3)$. Ook zien we hier het grote voordeel van het weglaten van de constanten. In de verschillende loops worden een aantal operaties ongeveer even vaak uitgevoerd. Hebben we bijvoorbeeld een loop als **for** $i = 1$ **to** n **do** $p = p + 1$, dan wordt de variabele p precies n keer verhoogd, maar *ook* de variabele i. Toch kunnen we de kosten van deze loop op $O(n)$ operaties stellen omdat constante factoren bij de bepaling van onze grenzen niet meetellen. Voor geneste for loops kunnen we zo de complexiteit bepalen door een operatie te vinden die in zo'n for loop het vaakst wordt uitgevoerd. In een programmafragment als:

for $i = 1$ to n **do**
  if $a = b$ **then**
    for $j = 1$ to m **do** $x = x + 1$;
    end for
  end if
end for

kunnen we een aantal operaties onderscheiden. De variabele i wordt n keer opgehoogd. De test $a = b$ wordt n keer uitgevoerd. De variabele j wordt (hoogstens) $n \times m$ keer opgehoogd en ook de variabele x wordt hoogstens $n \times m$ keer opgehoogd. We zien hier echter dat de optelling $x = x + 1$ *minstens* net zo vaak wordt uitgevoerd als elke andere operatie in deze geneste loop en daarom kunnen we de kosten van deze loop van boven afschatten met een constante keer het aantal keren dat $x = x + 1$ wordt uitgevoerd. De operatie $x = x + 1$ wordt in Engelse teksten vaak *barometer* genoemd. Dat lijkt in deze tekst minder zinvol, maar omdat de complexiteit van dit stukje programma als het ware hangt aan de operatie $x = x+1$ zullen we deze operatie hier *spil* noemen (zie ook pagina 16). Als we de complexiteit van een loop willen bepalen gaan we dus op zoek naar een spil op het diepste niveau en bepalen hoe vaak deze wordt uitgevoerd. De complexiteit van een nesting van loops is vaak het product van de complexiteiten van die loops en de complexiteit van een stel opeenvolgende loops kunnen we afschatten met de maximale complexiteit van die loops.

1.3.2 Sommen

1. Bedenk enkele voorbeelden van algoritmen die niet de gehele invoer hoeven te lezen voordat ze stoppen en een antwoord geven.

2. Onderzoek de uitgesmeerde complexiteit van zoekbomen en hashtabellen. Waarom halveren we de hashtabel pas wanneer deze voor 1/4e gevuld is en niet wanneer deze half gevuld is?

3. Laat zien dat $\log^3 n \in O(n^{1/3})$.

4. Geef een voorbeeld van een functie $f(n)$ zodat $f(n) \notin O(n)$ *en* $f(n) \notin \Omega(n)$.

5. Een polynoom van graad n is een functie $a_0 + a_1 x + \ldots + a_n x^n$. Geef een algoritme om de waarde van zo'n polynoom in een gegeven punt uit te rekenen en bepaal de complexiteit. Bepaal ook de complexiteit van het Horner product, dat is de berekening $p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \ldots + xa_n))\ldots)$.

6. Conditionele asymptotische notatie. Niet altijd is een asymptotische grens te geven voor elke waarde van n. Het kan zijn dat een functie bijvoorbeeld $O(n^2)$ is alleen maar als n een macht van 2 is. Voor bepaalde veel voorkomende functies kunnen we zo'n *conditionele* grens vertalen in een grens die voor alle n geldt. We zeggen dat een functie f *uiteindelijk niet dalend is* als er een n_0 bestaat zo dat $f(n) \leq f(n+1)$ voor alle $n \geq n_0$. Verder heet een functie b-glad als $f(bn) \in O(f(n))$, en glad als zij b-glad is voor elke $b \geq 2$. Als t uiteindelijk niet dalend is, en f is b-glad dan geldt. $t(n) \in \theta\{f(n) : n$ is een macht van $b\} \Rightarrow t(n) \in \theta(f(n))$. Bewijs dit.

7. Bewijs dat $O(f+g) = O(\max\{f,g\})$.

8. De symbolen O,o,θ,Ω, en $\sim$ kunnen opgevat worden als relaties tussen functies (bijvoorbeeld $R(f,g) \leftrightarrow f \in O(g)$. Welke van de aldus gedefiniëerde relaties zijn reflexief ($R(f,f)$), transitief ($R(f,g) \wedge R(g,h) \rightarrow R(f,h)$) en/of symmetrisch ($R(f,g) \leftrightarrow R(g,f)$)?

9. Geef aan welke relaties $f \in \bullet(g)$ gelden in onderstaande gevallen voor $\bullet \in \{O, o, \sim, \theta, \Omega\}$

   (a) $f(x) = (x^2+3x+1)^3$; $g(x) = x^6$.
   (b) $f(x) = 2^x$; $g(x) = 3^x$.
   (c) $f(x) = x+4$; $g(x) = x^2-3$.
   (d) $f(x) = \sum_{j \leq x} \frac{1}{j^2}$; $g(x) = 1$.

10. Onderzoek de volgende uitspraken op waarheid als $f(n) \in O(g(n)$ en als $f(n) \in \Omega(g(n))$.

    (a) $f(n) + g(n) \in O(g(n))$.
    (b) $f(n) \times g(n) \in O(g(n)$.
    (c) $f(n) - g(n) \in O(f(n))$.
    (d) $\frac{f(n)}{g(n)} \in \theta(f(n))$.

1.4 Wiskundige Hulpmiddelen

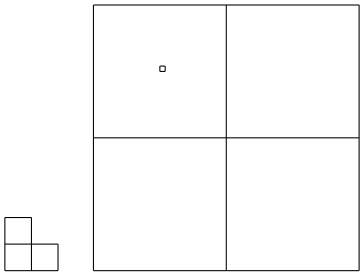
Belangrijk bij de Analyse van Algoritmen en bij het bepalen van de Complexiteit van Problemen is dat we kunnen tellen. In het bijzonder moeten we boven en ondergrenzen kunnen bepalen. Omdat het aantal operaties, of geheugenplaatsen dat gebruikt wordt vaak als functie van de invoer gerepresenteerd wordt, en dan bovendien lang niet altijd in gesloten vorm maar veel meer als samenraapsel van een aantal telpartijen en observaties, hebben we wat elementaire wiskunde nodig om een „schone" afschatting van de complexiteit te krijgen. In deze sectie zullen we een aantal technieken die hierbij kunnen helpen de revue laten passeren. Dit is mogelijk een herhaling van eenvoudige wiskunde die de lezer al eens eerder heeft gezien. Opfrissing van deze kennis is echter altijd nuttig.

1.4.1 Inductie en Recursie

Inductie

Bij een van zijn vele bezoeken aan Amsterdam vertelde Carl Smith eens over zijn „favorite induction proof". Het is een bewijs met inductie dat je nagenoeg zonder wiskundige hulpmiddelen kunt geven en dat door iedereen onmiddellijk begrepen wordt. Wellicht heeft de lezer dit probleem al eens eerder gezien. Het gaat om het volleggen van een bord, bestaande uit velden, met stukken van de vorm van Figuur 1.4.1. Zo'n L-vormige figuur beslaat precies 3 velden van het bord. De bewering is dat voor elk vierkant bestaande uit $2^k \times 2^k$ velden de stukken zo in het vierkant gelegd kunnen worden dat slechts één veld onbezet blijft, en dat veld kan bovendien willekeurig vantevoren worden gekozen.

Het inductiebewijs is als volgt. Voor een 2×2-vierkant is het simpel. Door rotatie kan elk van de vier velden onbedekt gelaten worden. Stel dat het probleem kan worden opgelost voor $k-1$. Bekijk een $2^k \times 2^k$-vierkant. Kies een veld dat onbedekt moet blijven. Dit veld ligt in één van de vier $2^{k-1} \times 2^{k-1}$ vierkanten



Figuur 1.4: Met volledige inductie...

die volgens de inductiehypothese kunnen worden volgelegd. De andere drie worden zo volgelegd dat de open velden de drie velden in het midden zijn waar ze alledrie bij elkaar komen en op die velden leggen we precies één stuk.

Recursie

Een recursieve aanpak van programmeren, zoals we in Sectie 2.2 zullen bekijken, verdeelt een algoritmisch probleem in kleinere, vaak identieke stukken. Op deze stukken kan dan weer dezelfde algoritme worden gebruikt, net zo lang tot de stukken zo klein geworden zijn dat de oplossing triviaal is. Recursieve aanpak van problemen en inductieve bewijzen gaan vaak hand in hand. Hoe zouden we namelijk een oplossing vinden voor het probleem dat hierboven geschetst is? We weten dat het probleem eenvoudig is voor 2×2-vierkanten, en als we een groter vierkant krijgen met daarin een veld, dan kunnen we het probleem in vier kleinere problemen opdelen, elk met hun eigen veld dat vrij gelaten moet worden om, als we voor deze vier kleinere problemen een oplossing hebben gevonden, met toevoeging van één stuk een oplossing voor het hele probleem te genereren.

1.4.2 Recurrente betrekkingen

De methode om een afschatting te krijgen voor de complexiteit van een probleem waarvoor we een recursieve algoritme hebben bedacht is het oplossen van de recurrente betrekking. Stel dat we zouden willen weten hoe moelijk het is de inductieopgave uit 1.4.1 te vinden. We zien dat we om een probleem van afmetingen n op te lossen, we 4 problemen van afmeting $n/2$ moeten oplossen, ofwel $T(n) = 4 \times T(n/2)$. Nu kunnen we het volgende zien.

$$\begin{aligned} T(n) &= 4 \times T(n/2) \\ &= 16 \times T(n/4) \\ &= 4^3 \times T(n/2^3) \\ &\vdots \quad \vdots \\ &= 4^{\log n} \times T(n/2^{\log n}) \\ &= 4^{\log n} \\ &= (2^2)^{\log n} \\ &= (2^{\log n})^2 \\ &= n^2 \end{aligned}$$

We hebben hier aangenomen dat de laatste T in deze reeks $T(n/2^{\log n}) = T(1)$ een triviale waarde (bijvoorbeeld 1) aanneemt. Aangezien de complexiteit voor constante afmetingen in ieder geval constant is, en we constanten in de O-notatie toch verwaarlozen is dit in ons geval geen verkeerde aanname. De recurrente betrekking laat zich dan als het ware uitrollen totdat we een gesloten uitdrukking voor de complexiteit van het probleem overhouden.

Dat is niet altijd het geval. Soms is een recurrente betrekking algemener. In het geval van de recursieve aanpak van de Fibonacci-getallen zagen we dat $F(n) = F(n-1) + F(n-2)$. Hier zijn al twee startwaarden nodig om van de grond te komen. In dit speciale geval hebben we $F(0) = F(1) = 1$, dus kunnen we die gebruiken om $F(2), F(3), \ldots$ uit te rekenen. Echter, de vergelijking laat zich dan al niet meer gemakkelijk uitrollen, maar meer uitbomen. Van boven naar beneden geeft $F(n)$ steeds twee nieuwe waarden, dus uiteindelijk exponentieel veel waarden om uit te rekenen.

Een andere eenvoud bevorderende eigenschap aan ons inductievoorbeeld is dat de factor die voor de $T(n/2)$ staat steeds dezelfde is. Dat kan natuurlijk ook afhankelijk van de probleemgrootte zijn. Een recurrente betrekking kan bijvoorbeeld van de vorm $x_{n+1} = b_{n+1}x_n$ zijn. In dit geval, kunnen we de vergelijking ook uitrollen als $x_{n+1} = b_{n+1}x_n = b_{n+1}b_n x_{n-1} = \ldots = b_{n+1}b_n b_{n-1}\cdots b_1 \times x_0$. Dat is kennelijk nog niet moelijk genoeg. Laten we dus een extra moeilijkheid toevoegen en kijken naar de vergelijking

$$x_{n+1} = b_{n+1}x_n + c_{n+1}.$$

Het uitrollen van de vergelijking geeft dan achtereenvolgens

$$\begin{aligned} x_{n+1} &= b_{n+1}x_n + c_{n+1} \\ &= b_{n+1}(b_n x_{n-1} + c_n) + c_{n+1} \\ &= b_{n+1}(b_n(b_{n-1}x_{n-2} + c_{n-1}) + c_n) + c_{n+1}. \end{aligned}$$

We raken hierbij al snel het overzicht kwijt.

Een betere aanpak voor dit soort vergelijkingen is de volgende. Eerst definieren we een nieuwe variabele y_n door $x_n = b_1 b_2 \cdots b_n y_n$. Vul in $b_1 b_2 \cdots b_{n+1} y_{n+1} = b_{n+1} b_1 b_2 \cdots b_n y_n + c_{n+1}$. We kunnen nu door de coefficiënt van y_n delen en krijgen $y_{n+1} = y_n + d_{n+1}$, waarbij $d_{n+1} = c_{n+1}/(b_1 \cdots b_{n+1})$. Deze vergelijking laat zich weer uitrollen tot de oplossing $y_n = y_0 + \sum_{j=1}^{n} d_j$ en, door terugsubstitutie $x_n = (b_1 \cdots b_n)[x_0 + \sum_{j=1}^{n} d_j]$. Deze substitutie van variabelen blijkt een winnende strategie te zijn.

Voorbeeld 1.4.1: Kijk naar de vergelijking $x_{n+1} = 3x_n + n$, met $x_0 = 0$. Substitueer $x_n = 3^n y_n$ en vind dat $y_{n+1} = y_n + n/3^{n+1}$ ofwel $y_n = \sum_{j=1}^{n-1} j/3^{j+1}$. Terugsubsitutie van $y_n = x^n/3^n$ geeft

$$x_n = 3^n \times \sum_{j=1}^{n-1} j/3^{j+1}.$$

Dit is al behoorlijk precies, maar met de technieken uit de sectie 1.4.3 kunnen we $\sum_{j=1}^{n-1} j/3^{j+1}$ nog nauwkeuriger bepalen, zoals verderop blijkt. □

Recurrente betrekkingen als tot nu toe behandeld, noemen we eerstegraads recurrente betrekkingen. Er is sprake van twee variabelen, x_n en x_{n-1}, en nog wat andere termen die niet van de recurrentie afhangen. Deze afhankelijkheid lijkt dus een beetje op de manier waarop de coordinaten van een eerstegraadsfunctie (lijn) van elkaar afhangen. Het is echter ook in de algoritmiek mogelijk dat de afhankelijkheid complexer is. In de recursieve implementatie van het n-de Fibonacci getal is er afhankelijkheid van x_n van zowel x_{n-1} als x_{n-2} ($F(n) = F(n-1) + F(n-2)$). Ook deze recurrente betrekking kunnen we „uitrollen" totdat alleen $F(0)$ en $F(1)$ nog in de vergelijking staan, maar er is een slimmere manier. Als we een vergelijking hebben als $x_n = x_{n-1} + x_{n-2}$, dan kunnen we eens een oplossing proberen van de vorm $x_n = \alpha^n$ voor constante α. Invullen van dit probeersel geeft $\alpha^n = \alpha^{n-1} + \alpha^{n-2}$ of (delen door α^{n-2}) $\alpha^2 = \alpha + 1$. De kwadratische vergelijking $\alpha^2 - \alpha - 1 = 0$ laat zich direct oplossen en geeft $\alpha = \frac{1 \pm \sqrt{5}}{2}$.

In het algemeen zijn tweede orde recurrente betrekkingen van de vorm $x_n = ax_{n-1} + bx_{n-2} + f(n)$. Als de vergelijking $\alpha^2 = a\alpha + b$ een oplossing heeft, dan heeft de recurrente betrekking $x_n = ax_{n-1} + bx_{n-2}$ een oplossing in gesloten vorm $x_n = \alpha^n$, waarbij α één van de wortels van de vergelijking is. Bij twee gelijke wortels vinden we een oplossing van de vorm $\alpha^n(c_1 + c_2 n)$ analoog aan de oplossing van differentiaalvergelijkingen. Het punt is dat de oplossingen van deze tweedegraadsvergelijkingen slechts begrensd lijken te worden door exponentiële functies. Zeker is α^n een ondergrens, maar als $f(n)$ ook door α^n begrensd wordt, dan is α^n ook een bovengrens voor de groei van x_n. We vatten dit samen in de volgende stelling.

Stelling 1.4.1 *Laat* $\{x_n\}_n$ *voldoen aan* $x_n \leq b_1 x_{n-1} + \ldots + b_k x_{n-k} + f(n)$ *met* $(\forall i)[b_i \geq 0]$, $\sum b_i > 1$ *en laat* $c > 1$ *zo dat* $c^k = b_1 c^{k-1} + \ldots + b_k$. *Als* $f(n) \in o(c^n)$, *dan geldt* $x_n \in O((c+1)^n)$.

Het bewijs van deze stelling gaat als volgt. Eerst merken we op dat als $t = (c+1)^k - b_1(c+1)^{k-1} - \ldots - b_k$, dan is $t > 0$. Immers, $c^k = b_1 c^{k-1} + \ldots + b_k$, en als een k-de graads functie eenmaal groter is geworden dan een $k-1$-ste graads functie in voor positieve argumenten, dan blijft dat zo.

Definieer

$$K = \max\{|x_0|, |x_1|/(c+1), \ldots |x_k|/(c+1)^k, \max\{f(n)/t(c+1)^{n-k} : n \geq k\}\}$$

Dan is K eindig en bovendien geldt $|x_j| \leq K(c+1)^j$ voor $j \leq n-1$. We beweren dat $|x_n| \leq K(c+1)^n$ voor alle n. Stel dat de bewering waar is tot en met $n-1$, dan

$$\begin{array}{rcc} |x_n| & \leq & b_1|x_{n-1}| + \ldots + b_k|x_{n-k}| + f(n) \\ & \leq & b_1 K(c+1)^{n-1} + \ldots + b_k K(c+1)^{n-k} + f(n) \\ & = & K(c+1)^{n-k}(b_1(c+1)^{k-1} + \ldots + b_k) + f(n) \\ & = & K(c+1)^{n-k}((c+1)^k - t) + f(n) \\ & = & K(c+1)^n - (tK(c+1)^{n-k} - f(n)) \\ & \leq & K(c+1)^n. \end{array}$$

Een Master Theorem

Hoewel goed om te hebben, leert Stelling 1.4.1 ons vooral dat algoritmen waarvan de tijd gegeven wordt door een recurrente betrekking met meerdere termen die als index $n-c$ hebben, met c een constante, geen efficiënte algoritmen zijn. In het bereik van de efficiënte algoritmen komen recurrente betrekkingen als $T(n) = T(n/2) + f(n)$ veel vaker voor, zoals we bijvoorbeeld bij zoeken en sorteren verderop zullen zien. Een recursie als de bovenstaande kunnen we natuurlijk uitrollen en een gesloten uitdrukking voor $T(n)$ in $f(n)$ krijgen, maar er is een meer generieke aanpak. In een tekstboek als bijvoorbeeld dat van Cormen, Leiserson en Rivest [CLR90] wordt deze aanpak „The Master Theorem" genoemd. Op veel andere plaatsen wordt deze aanpak de Akra-Bazzi methode genoemd, naar de auteurs van [AB98]. Beide methoden hebben veel gemeen, al dekt de Akra-Bazzi methode meer gevallen. Het gaat om recurrente betrekkingen van de vorm

$$u_n = \sum_{i=1}^{k} a_i u_{\lfloor \frac{n}{b_i} \rfloor} + g(n).$$

De wiskundig geïnteresseerde lezer wordt aangemoedigd om het artikel [AB98] te lezen, maar voor de analyse van algoritmen kan vaak worden volstaan met een sterk vereenvoudigde aanpak zoals in [CLR90].

Vaak is er in een recurrente betrekking die uit een complexiteitsanalyse volgt namelijk slechts sprake van één enkele veranderlijke, d.w.z. de recurrente betrekking is van de vorm.

$$T(n) = \begin{cases} c & \text{als } n < d \\ aT(n/b) + f(n) & \text{als } n \geq d \end{cases}$$

We kunnen dan volstaan met de vereenvoudiging:

1. Als $f(n) \in O(n^{\log_b a - \epsilon})$ dan $T(n) \in \theta(n^{\log_b a})$.
2. Als $f(n) \in \theta(n^{\log_b a} \log^k n)$ dan $T(n) \in \theta(n^{\log_b a} \log^{k+1} n)$
3. Als $f(n) \in \Omega(n^{\log_b a + \epsilon})$ en $(\exists \delta < 1)[af(n/b) \leq \delta f(n)]$ dan $T(n) \in \theta(f(n))$.

Deze vereenvoudiging van de stelling is bij veel complexiteitsanalyse van algoritmen een machtig wapen.

Voorbeeld 1.4.2:

1. Bij de algoritme voor mergesort, wordt een array verdeeld in twee arrays van de halve grootte, die vervolgens gesorteerd worden en weer samengevoegd tot een gesorteerd array van de oorspronkelijke grootte. Dat samenvoegen kost tijd proportioneel aan de lengte van het array. De recurrente betrekking die daarbij hoort is
$$T(n) = 2T(n/2) + O(n).$$
Met bovenstaande stelling zien we dat $a = b = 2$ dus $n^{\log_b a} = n$, dus $f(n) \in O(n^{\log_b a})$, maar $f(n) \notin O(n^{\log_b a - \epsilon})$, ofwel we zitten in geval 2 met $k = 0$. Gevolg: $T(n) \in \theta(n \log n)$.

2. Gewone matrixoptelling kunnen we recursief doen door vier matrices van de halve grootte bij elkaar op te tellen. De recurrente betrekking die daarbij hoort is
$$T(n) = 4T(n/2) + O(1).$$
We zitten hiermee in geval 1 van de stelling, en dus $T(n) \in \theta(n^2)$, want $\log_2 4 = 2$.

3. Gewone matrixvermenigvuldiging kan gebeuren door 8 matrices van de halve grootte met elkaar te vermenigvuldigen, en dan de resultaten van de vermenigvuldiging twee aan twee bij elkaar op te tellen. Dat zijn vier optellingen. We hebben gezien dat matrixoptelling $\theta(n^2)$ bewerkingen kost, dus is de recurrente betrekking die hierbij hoort
$$T(n) = 8T(n/2) + O(n^2).$$
We zien hier $a = 8, b = 2$, dus $log_b a = 3$, en $n^2 \in O(n^3)$ dus $T(n) \in \theta(n^3)$.

4. Stel we hebben een rij van n getallen waaruit we een deel van $\log n$ getallen willen selecteren die van klein naar groot gesorteerd is. Dat kunnen we doen door bijvoorbeeld $\log n$ keer het minimum van de rij te nemen, maar aangezien het vinden van het minimum van een rij $O(n)$ stappen kost, zijn we dan $n+n-1+n-2+\ldots+n-\log n$ stappen bezig en dat is $\theta(n \log n)$. Omdat we niet zoveel getallen nodig hebben, kunnen we *ook* als we het minimum van een rij hebben gevonden, vervolgens het minimum zoeken van de *helft* van de overgebleven rij. Dan zijn de kosten
$$T(n) = n + T(n/2).$$
Volgens bovenstaande stelling is dan $a = 1, b = 2 \rightarrow \log_b a = 0$, dus komen we in geval 3, waaruit volgt dat $T(n) \in \theta(n)$.

□

1.4.3 Reeksen

Het optellen van complexiteit van verschillende stukjes van een programma is een belangrijk onderdeel van de bepaling van de complexiteit van problemen. Echter, de optelling is vaak afhankelijk van een variabele. De tweede keer dat een stukje programma wordt uitgevoerd is vaak de complexiteit van dat stukje programma anders. Een voorbeeld is het samenstellen van twee gesorteerde rijen tot een nieuwe gesorteerde rij in de algoritme *mergesort*(zie 2.2). Deze algoritme laat zich uitleggen als volgt. Een rij die bestaat uit één getal is altijd gesorteerd. Verder kunnen we twee gesorteerde rijen van lengte n samenstellen tot een gesorteerde rij van lengte $2n$ door de twee eerste elementen met elkaar te vergelijken en daarvan de kleinste weg te schrijven, net zo lang tot de beide rijen leeg zijn. Zo krijg je bijvoorbeeld uit de rijen $1, 2, 4, 7$ en $3, 5, 6, 8$ de rij $1, 2, 3, 4, 5, 6, 7, 8$. Elke keer als je dit stukje programma uitvoert, is de lengte van de resulterende rij twee keer zo groot als de vorige keer. De hele executie van mergesort voegt $n/2$ rijen van lengte 1 samen tot $n/4$ rijen van lengte 2 die worden samengevoegd tot $n/8$ rijen van lengte 4 eindigend in het samenvoegen van 2 rijen van lengte $n/2$ (even uitgaande van het gunstige geval dat n een macht van 2 is). Om te bepalen wat de totale complexiteit van het steeds samenvoegen van twee rijen is, moet je dus die getallen bij elkaar

kunnen optellen, dus bijvoorbeeld kunnen uitrekenen wat de som $\sum_{i=1}^{\log n} 2^i$ is. In deze sectie zullen we een aantal truuks voor het sommeren van reeksen die de lezer waarschijnlijk eerder gezien heeft, maar mogelijk weer vergeten is, de revue laten passeren. We beginnen met een eenvoudige.

Bekijk de reeks

$$\sum_{i=0}^{n-1} x^i = 1 + x + x^2 + \ldots + x^{n-1} = (1 - x^n)/(1 - x) \text{ voor } x \neq 1.$$

Dat dit waar is, is in te zien door beide kanten met $1 - x$ te vermenigvuldigen. Met deze observatie kunnen we een antwoord krijgen op de vraag hoe groot $\sum_{i=0}^{\log n} 2^i$ is, zoals we hierboven zochten, want het is gewoon deze reeks met $x = 2$ en $\log n$ gesubstitueerd voor n, er komt dus uit $(1 - 2^{\log n})/(1 - 2) = n$. Omdat we deze reeks echter voor algemene x hebben opgelost kunnen we ook bepalen wat $\sum_{i=0}^{n} 3^i$ voor het geval dat we een algortime hebben waarbij volgende slagen steeds drie keer zo lang zijn.

Een reeks als deze laat zich gebruiken voor nog andere toepassingen. Stel eens dat we niet $1 + x + x^2 + \ldots + x^{n-1}$ moeten uitrekenen, maar $1 + 2x + 3x^2 + \ldots + (n-1)x^{n-2}$. Deze reeks is de *afgeleide* van de reeks $1 + x + x^2 + \ldots + x^{n-1}$ en dus is de som ook de afgeleide van $(1 - x^n)/(1 - x)$ dus

$$\frac{1 - nx^{n-1} + (n-1)x^n}{(1-x)^2}.$$

Voor machtreeksen geldt dat als we de som van de machtreeks kennen, zeg bijvoorbeeld $\sum a_n x^n = f(x)$, dan kennen we ook $\sum n a_n x^{n-1}$, want dat is de afgeleide van $f(x)$, maar dan natuurlijk ook $\sum n a_n x^n$ want dat is weer $xf'(x)$. Dus als de $n$de coëfficient met n vermenigvuldigd wordt, dan verandert de som van $f(x)$ naar $x\frac{d}{dx}f(x)$.

Dit spel—neem de afgeleide en vermenigvuldig het resultaat weer terug met x—kunnen we herhalen. Immers als $xf'(x) = \sum n a_n x^n$ dan is $xf''(x) = \sum n^2 a_n x^{n-1}$ en is $x^2 f''(x) = \sum n^2 a_n x^n$. Met andere woorden, het vermenigvuldigen van de n-de coëfficient met n^2 verandert de som van de reeks van f naar $(x\frac{d}{dx})^2 f(x)$. Algemener: als we de n-de coëfficient van een machtreeks met n^p vermenigvuldigen, dan verandert de som van de reeks van f naar $(x\frac{d}{dx})^p f(x)$. Vanwege het feit dat we eindige sommen kunnen herordenen in elke volgorde die we willen, geldt dit voor algemene polynomen. Als we bijvoorbeeld de coëfficient van x^n vermenigvuldigen met $5n^3 + n^2 + 4$, dan verandert de som van $f(x)$ naar $(5(x\frac{d}{dx})^3 + (x\frac{d}{dx})^2 + 4)f(x)$. De algemene regel voor een willekeurig polynoom P is als volgt.

$$\sum_j P(j) a_j x^j = P\left(x\frac{d}{dx}\right) \sum_j a_j x_j$$

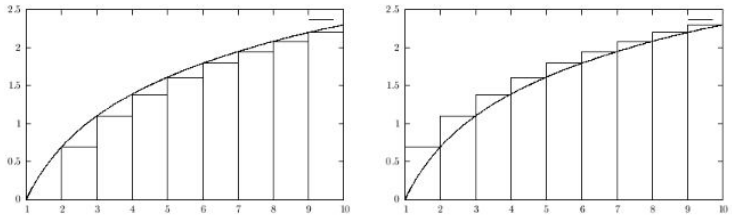
In plaats van x^i als sommand kunnen we ook te maken krijgen met i^x. Het bekendste voorbeeld is wel $1 + 2 + 3 + 4 + \ldots + n = n(n+1)/2$ dat veelvuldig gebruikt wordt om bewijzen met inductie te illustreren. Niet iedereen kent echter de veralgemenisering van dit voorbeeld: $\sum_{i=0}^{n} i^3 = (n(n+1))^2/4$. Dit geldt veel algemener. Stel dat we (zie som 5) de beschikking hebben over de volgende stelling.

Stelling 1.4.2 *Als p een polynoom in i van graad d is, dan is $\sum_{i=0}^{n} p(i)$ een polynoom in n van graad $d+1$.*

Nu kunnen we de bewering $\sum_{i=0}^{n} i^3 = (n(n+1))^2/4$ bewijzen door te controleren dat $n = 0, n = 1, n = 2, n = 3$ en $n = 4$ aan beide kanten respectievelijk 0, 1, 9, 36 en 100 opleveren omdat 5 punten een polynoom van de graad 4 volledig vastleggen, maar dat niet alleen. We kunnen nu voor elk polynoom p achter de waarde van $\sum_{i=0}^{n} p(i)$ komen door een paar lineaire vergelijkingen op te lossen die de coefficiënten van het resulterende polynoom bepalen.

Voorbeeld 1.4.3: Bereken

$$\sum_{i=0}^{n} 2i^2 + 4.$$



Figuur 1.5: De integraal en de som van $\log x$.

We weten door Stelling 1.4.2 dat hier een polynoom in n van graad 3 uit moet komen. De algemene vorm is an^3+bn^2+cn+d. Vier punten van het polynoom zijn voldoende om het uniek te bepalen. Voor $n = 0, 1, 2, 3$ is de waarde van de som respectievelijk 4, 6, 12 en 22. We lossen op:

$$\begin{aligned} d &= 4 \\ a+b+c+d &= 10 \\ 8a+4b+2c+d &= 22 \\ 27a+9b+3c+d &= 44 \end{aligned}$$

Zodat we als som voor deze reeks vinden

$$\frac{2}{3}n^3 + n^2 + \frac{13}{3}n + 4.$$

□

Ten slotte noemen we nog een aantal voorbeelden van sommen die vaak voorkomen en goed zijn om te kennen.

$$\begin{aligned} e^x &= \textstyle\sum_{m=0}^{\infty} x^m/m! \\ \sin x &= \textstyle\sum_{r=0}^{\infty}(-1)^r x^{2r+1}/(2r+1)! \\ \cos x &= \textstyle\sum_{r=0}^{\infty}(-1)^r x^{2r}/(2r)! \\ \log \tfrac{1}{1-x} &= \textstyle\sum_{j+1}^{\infty} x^j/j \end{aligned}$$

1.4.4 Reeksen en Integralen

Integralen zijn in de wiskunde de infinitesimale vorm van sommeren. Een integraal geeft, zeer informeel gesproken, een uitdrukking van de oppervlakte van een gebied onder een functie doordat de som genomen wordt van oneindig veel rechthoeken van oneindig kleine dikte. We kunnen deze eigenschap, die in de analyse „het integraalkenmerk van d'Alembert" genoemd wordt, gebruiken om sommen af te schatten. Kijk bijvoorbeeld eens naar de functie $f(x) = \log x$. Figuur 1.5 laat twee plots voor deze functie zien, waarbij de waarden tussen 1 en 10 op twee manieren als een staafdiagram zijn weergegeven. De som van de functiewaarden (vermenigvuldigd met 1 maar dat kun je niet zien) geeft op twee manieren een schatting van de waarde van de oppervlakte onder de functie, ofwel de oppervlakte van de functie geeft op twee manieren een schatting van de som van de functiewaarden. Nemen we de functiewaarden van 1 t.e.m. 9 dan zien we dat de som een onderschatting is van de integraal—ofwel de integraal is groter dan deze som, terwijl de functiewaarden van 2 t.e.m. 10 een bovenschatting zijn van de integraal—ofwel de integraal is kleiner dan de som. Deze eigenschap kunnen we uiteraard gebruiken om de afschatting van een reeks te krijgen, wanneer we de integraal wel kennen maar de som niet. In het geval van $\log x$ is de integraal $xlogx - x$, dus als we een afschatting willen hebben van bijvoorbeeld $\sum_{i=1}^{n} \log i$ dan weten we dat $\theta(n \log n)$ een geschikte afschatting is.

1.4.5 Sommen

1. Rondom een cirkel staan $2n$ nullen en éénen, n van elke soort. Bewijs met inductie dat het altijd mogelijk is een punt te vinden zó dat als je een volledige cirkel met de klok meedraait, je op elk punt van de draai minstens evenveel nullen als éénen gezien hebt.

2. Los de volgende recurrente betrekkingen op.

   (a)
   $$t_n = \begin{cases} n & \text{als } n = 0 \text{ of } n = 1 \\ 5t_{n-1} - 6t_{n-2} & \text{anders} \end{cases}$$

   (b)
   $$t_n = \begin{cases} 9n^2 - 15n + 106 & \text{als } n = 0 \text{ of } n = 1 \text{ of } n = 2 \\ t_{n-1} + 2t_{n-2} - 2t_{n-3} & \text{anders} \end{cases}$$

   (c)
   $$t_n = \begin{cases} n & \text{als } n = 0 \text{ of } n = 1 \text{ of } n = 2 \\ t_{n-1} + t_{n-3} - t_{n-4} & \text{anders} \end{cases}$$

   (d)
   $$t_n = \begin{cases} n+1 & \text{als } n = 0 \text{ of } n = 1 \\ 3t_{n-1} - 2t_{n-2} + 3 \times 2^{n-2} & \text{anders} \end{cases}$$

   (e)
   $$t_n = \begin{cases} 0 & \text{als } n = 0 \\ 1/(4 - tn - 1) & \text{anders} \end{cases}$$

3. Laat S een verzameling van n lijnen zijn waarvan geen twee parallel zijn en geen drie in hetzelfde punt snijden. Bewijs met inductie dat de lijnen van S in $\theta(n)$ punten snijden.

4. Bereken

   (a) $\sum_{i=0}^{n-1} i \times n^i$
   (b) $\sum_{i=0}^{n} i^3 + 2i^2 + 4$
   (c) $\sum_{i=0}^{n} (i^3 + 2i^2 + i)x^i$

5. Bewijs met volledige inductie dat als p een polynoom in i van graad d is, dan is $\sum_{i=0}^{n} p(i)$ een polynoom in n van graad $d + 1$ (Stelling 1.4.2).

6. Geef schattingen voor $\sum_{i=1}^{n} \lceil logi \rceil$ en $\sum_{i=1}^{n} \lceil log(n/i) \rceil$.

Hoofdstuk 2

Technieken

In dit hoofdstuk bekijken we een aantal veelgebruikte technieken om te komen tot efficiënte algoritmen voor problemen. We maken een indeling naar de aard van de algoritme die voor het probleem wordt besproken. Sommige van de algoritmen in dit hoofdstuk zouden met evenveel recht in het volgende hoofdstuk, Grafenalgoritmen, kunnen worden geplaatst, omdat ze specifiek voor grafenproblemen zijn ontworpen. De keuze om ze in dit hoofdstuk te behandelen is arbitrair.

2.1 Gulzige Algoritmen

De eerste soort algoritmen die we zullen bekijken zijn optimalisatie algoritmen. Bij optimalisatiealgoritmen is altijd sprake van verschillende mogelijke uitkomsten waarvan er één of meer als beste kan worden aangemerkt. Bijvoorbeeld:

1. Er zijn mogelijke paden tussen A en B, en het pad dat in lengte het kortst is vinden wij het beste,

2. uit hop en water kunnen we verschillende soorten en hoeveelheden bier maken en de verdeling die het meeste geld oplevert vinden wij het beste,

3. een verzameling bussen kan langs een verzameling steden rijden en het schema waarbij de totale reislengte van alle passagiers zo klein mogelijk gehouden wordt, vinden wij het best.

Een aanpak die altijd werkt voor optimalisatiealgoritmen is het bekijken van alle mogelijke oplossingen en daaruit de oplossing(en) te kiezen die het beste resultaat geven. Deze uitputtende methode (exhaustive search) kan echter soms heel duur zijn. Als we het grootste getal willen vinden dat in n bits geschreven kan worden, kunnen we alle getallen van n bits onder elkaar schrijven en de grootste noteren, dat kost 2^n stappen. Er zijn snellere methoden denkbaar, en één van die methoden is de gulzige methode (of Greedy Method).

We noemen een optimalisatiealgoritme "Gulzig" of „Greedy" als ze de volgende karakteristieken heeft.

- Een optimale oplossing wordt stap voor stap opgebouwd.

- Een éénmaal genomen stap wordt nooit ongedaan gemaakt.

- In iedere stap wordt een zo groot of zo goed mogelijk tussenresultaat behaald.

Gulzige methoden zijn vaak niet moeilijk om te bedenken. Het is de formule „grote stappen gauw thuis" en dus lijkt het op wat een mens als eerste geneigd is om te doen als de situatie onoverzichtelijk is. Een gulzige methode is ook bijzonder efficiënt omdat je nooit een verkeerd pad inslaat. Veel moeilijker is het om te bewijzen dat de gulzige methode ook tot de optimale oplossing leidt, omdat er talloze problemen zijn waarvoor helemaal geen gulzige algoritmen (kunnen) bestaan. Vaak leidt de weg die het gunstigst lijkt

verderop tot een uitzichtloze situatie. We zullen een aantal voorbeelden van problemen beschrijven waarin gulzige algoritmen tot een optimale oplossing leiden.

Laten we eerst eens ons triviale getallenvoorbeeld bekijken: we weten dat bij éénbits getallen een 1 groter is dan een 0. Als we dus een keuze moeten maken om een 1 of een 0 in te vullen en we willen een zo groot mogelijk getal hebben, dan kiezen we voor de 1. In plaats van alle 2^n getallen op te schrijven kunnen we ook een groot n bits getal maken door bij ieder bit de best mogelijke keuze te doen. Voor het eerste bit weten we dat een 1 groter is dan een 0 en we kiezen dus 1, voor het tweede bit geldt hetzelfde, voor het derde bit ook enzovoort. Als grootste n-bits getal krijgen we met deze keuzen vanzelf $11\ldots1$.

Dit lijkt allemaal eenvoudiger dan het is. Niet in alle gevallen kunnen we een globaal optimum bereiken door steeds locaal een optimale stap te kiezen. Als we bijvoorbeeld van Amsterdam naar Saarbruecken (Dld) willen rijden met de auto kunnen we zowel bij Arnhem als bij Maastricht de grens over. Arnhem is aanmerkelijk dichterbij Amsterdam dan Maastricht, toch is de keuze voor Arnhem de verkeerde als we zo snel mogelijk bij ons doel willen zijn. In dit geval is dus de keuze van het dichtstbijzijnde tussenstation niet goed genoeg om ook de kortste weg te vinden. In 2.1.3 zullen we nog uitgebreid bespreken wat wel een goede manier is om de kortste weg van A naar B te vinden.

Om van een probleem aan te tonen dat er een gulzige algoritme voor bestaat zullen we een eigenschap van het probleem moeten vinden die het mogelijk maakt gedeeltelijke oplossingen steeds uit te breiden totdat we een optimale oplossing voor het hele probleem gevonden hebben. Hiervan zullen we dan moeten bewijzen dat de keuze die we maken bij het uitbreiden van de deeloplossing steeds leidt tot een optimale oplossing. We bekijken een paar voorbeelden.

2.1.1 Geld Wisselen

We beginnen met een algoritme die elke cassiëre van de lokale supermarkt beheerst. Het probleem is het teruggeven van wisselgeld, waarbij het aantal bankbiljetten en muntstukken dat moet worden teruggegeven wordt geminimaliseerd. In ons muntsysteem is het allereerst zo dat *elk* bedrag dat moet worden teruggegeven kan worden gemaakt, omdat in ons muntsysteem de eenheid zit. In het geval van de euro is de eenheid de eurocent. Om verschillende reden is die eurocent uit het systeem gehaald, maar alle af te rekenen bedragen worden vervolgens afgerond op vijf cent, zodat de vijf cent munt de rol van de eenheid gaat vervullen.

Als we de bedragen die moeten worden teruggegeven vormen uit *uitsluitend* vijf eurocent munten, dan is het onmiddelijk duidelijk dat het aantal munten dat wordt teruggegeven niet minimaal is. Een oplossing zou kunnen zijn, om eerst het terug te geven bedrag uit vijf cent munten te maken, en vervolgens die munten "in te wisselen" voor grotere, net zo lang tot dat niet meer kan. Dit zou echter tot problemen kunnen leiden aangezien eerder genomen beslissingen, latere beslissingen zouden kunnen beïnvloeden. Bijvoorbeeld twee vijf cent stukken en één tien cent stuk maken een twintig cent stuk, maar dat twintig cent stuk kan ook uit twee tien cent stukken gemaakt worden. De oplossing die door cassiëres gehanteerd wordt (en die ook leidt tot een optimale oplossing) is die waarbij telkens de grootste munteenheid die in het gat past dat gevuld moet worden daarin gestopt wordt, totdat er geen gat meer over is. Deze algoritme werkt helaas niet in elk muntsysteem. Het muntsysteem moet ervoor ontworpen zijn om deze eenvoudige manier van optimaliseren mogelijk te maken.

2.1.2 Knapsack

Het probleem is het vullen van een rugzak met objecten die allemaal een eigen gewicht en een eigen waarde hebben. Als men bijvoorbeeld gaat kamperen, dan is de tent van waarde (afhankelijk van waar men naartoe gaat uiteraard) maar het krat bier ook. Het gewicht van beide objecten verschilt. Doorgaans zijn zwaardere objecten waardevoller, maar niet altijd. De creditcard is zeer licht van gewicht, maar bij sommige bestemmingen onmisbaar. Als het totaal gewicht begrensd is door één of andere waarde b, wat is dan de juiste combinatie van mee te nemen objecten, zo dat de totale waarde gemaximaliseerd wordt?

We beginnen eenvoudig en nemen aan dat we niet altijd hele objecten mee hoeven nemen. Koffie bijvoorbeeld laat zich tot ongeveer elk deel verdelen en van een voorraad koffie kunnen we dus besluiten een aantal gram mee te nemen. Hetzelfde geldt voor water, geld, etc. Om ons probleem eenvoudig te maken nemen we

dit aan voor alle mee te nemen objecten. Elk object heeft dus een bepaald gewicht en een bepaalde waarde. Een gulzige algoritme kan dus op deze gewichten en waarden in elke stap selecteren en net zo lang kiezen tot de grens is bereikt. Volgens onze aanname kunnen we van elk object een deel meenemen, dus de grens die gesteld is kan altijd precies worden bereikt. Er zijn twee voor de hand liggende keuzen.

1. We kunnen zoveel mogelijk objecten van zo klein mogelijk gewicht meenemen. Daarmee kunnen we veel objecten in de rugzak steken en dus de totale waarde van de rugzak groot maken.

2. We kunnen in elke stap een object met een zo groot mogelijke waarde kiezen. Dan vullen we de rugzak met zeer waardevolle objecten en krijgen we ook een totale waarde die groot is.

Beide aanpakken werken niet. Om dat aan te tonen bekijken we twee kleine voorbeelden.

Voorbeeld 1 geeft aan dat het kiezen van een zo klein mogelijk gewicht niet goed is. Neem als voorbeeld een gewichtsgrens 5 en 3 objecten met gewichten 2,2,5 en waarden 2, 2, 10. Kiezen voor het kleinste gewicht eerst zal leiden tot het meenemen van objecten 1, 2 en 1/5e van object 3 voor een totale waarde van $2 + 2 + 2 = 6$, terwijl het meenemen van alleen object 3 een waarde 10 oplevert.

Voorbeeld 2 geeft aan dat het kiezen van het object met de grootste waarde niet altijd goed is. Als we weer een gewichtsgrens 5 hebben en nu 3 objecten met waarde 5,4,3 en gewicht 5,3,2 dan krijgt de rugzak een totale waarde 5 als we alleen object 1 meenemen, maar de rugzak krijgt waarde 7 als we objecten 2 en 3 meenemen. De oplossing is hier dat we de objecten moeten rangschikken naar oplopende waarde per gewichtseenheid en dan eerst het object kiezen dat de grootste waarde per gewichtseenheid heeft. In het eerste geval leidt dat tot een ordening: 1, 1, 2, wat inhoudt dat we eerst zullen kiezen voor zoveel mogelijk meenemen van object 3 en in het tweede geval leidt dat tot een ordening 1, 4/3, 3/2, waardoor we eerst voor object 3, en daarna voor object 2 zullen kiezen, waarna de rugzak vol is. Deze strategie leidt ook in het algemeen tot een optimale oplossing. Dit zullen we bewijzen in Opgave 3.

2.1.3 Grafen: kortste paden

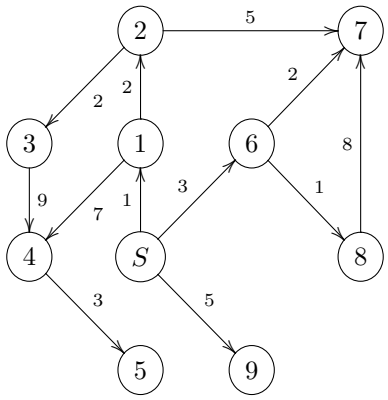
Het probleem is het vinden van het kortste pad in een gewogen graaf. Gegeven is dus een graaf $G = (V, E)$ met $E \subseteq V \times V$, een gewichtsfunctie $f : E \mapsto R+$ (alleen positieve gewichten) en een speciaal startpunt S in de graaf, en gevraagd is voor elk punt A in de graaf de lengte van een pad van S naar A te vinden zo dat de som van de gewichten van de kanten in dat pad minimaal is. Ook voor dit probleem bestaat een gulzige algoritme, bedacht door E.W. Dijkstra in 1959 die in de praktijk nog steeds gebruikt wordt in bijvoorbeeld routeplanners. Een voor de hand liggende gulzige strategie zou zijn om eerst de kant met het kleinste gewicht te kiezen en van daaruit verder te werken, maar een zeer klein tegenvoorbeeld laat zien dat dat niet tot de optimale oplossing kan leiden (zie opgave 6). De aanpak die Dijkstra koos is de volgende.

Aan het begin van de algoritme is er precies één knoop waarvan we met zekerheid kunnen zeggen wat de afstand tot S is, namelijk S zelf. Omdat alle afstanden in de graaf groter dan 0 zijn, kunnen we geen korter pad naar S vinden dan het pad zonder kanten. De afstand van S tot S is dus met zekerheid 0. Dit wetende, kunnen we van één andere knoop de afstand tot S bepalen. S heeft namelijk een aantal buren $v_1, v_2, \ldots, v_k$, die allemaal door een kant met gewicht groter dan 0 met S verbonden zijn. Elk pad naar een willekeurige knoop in de graaf (dus zeker naar $v_1, \ldots, v_k$) gaat door één van deze knopen. Als dus v_m een buur van S is waarvan het gewicht van de verbindende kant minimaal is, dan kan er geen pad in de graaf zijn dat korter is dan het gewicht van deze kant. Kortom v_m vormt samen met S een groep knopen waarvan we *de* afstand tot S in de graaf weten. Noem deze groep knopen W. De algoritme breidt nu stap voor stap de verzameling W uit, totdat alle knopen in de graaf in W zitten.

Op elk moment geldt dat we, kijkend naar de buren van knopen in W, zien dat we één van deze knopen ook het kortste pad in de graaf weten. Dat is namelijk van de knoop, v, waarvan de afstand tot S, *uitsluitend* via knopen in W, minimaal is. Immers, als het korste pad van S naar v niet uitsluitend via knopen in W zou lopen, dan zou het via een andere knoop w *buiten* W lopen. Laat w zonder beperking der algemeenheid de *eerste* knoop zijn die op het korste pad naar v ligt en niet in W zit. Dan loopt het pad van S naar w *uitsluitend* door knopen in W, en is *korter* dan het pad dat uitsluitend door knopen in W naar v loopt, want

$v \neq w$. Dat is in tegenspraak met de manier waarop v gekozen is.

Voorbeeld 2.1.1: Beschouw de volgende graaf.



De algoritme van Dijkstra vindt achtereenvolgens de volgende afstanden

S	1	2	3	4	5	6	7	8	9	W
0	1	∞	∞	∞	∞	3	∞	∞	5	$\{S\}$
0	1	3	∞	8	∞	3	∞	∞	5	$\{S, 1\}$
0	1	3	∞	8	∞	3	5	4	5	$\{S, 1, 6\}$
0	1	3	5	8	∞	3	5	4	5	$\{S, 1, 6, 2\}$
0	1	3	5	8	∞	3	5	4	5	$\{S, 1, 6, 2, 8\}$
0	1	3	5	8	∞	3	5	4	5	$\{S, 1, 6, 2, 8, 3\}$
0	1	3	5	8	∞	3	5	4	5	$\{S, 1, 6, 2, 8, 3, 7\}$
0	1	3	5	8	∞	3	5	4	5	$\{S, 1, 6, 2, 8, 3, 7, 9\}$
0	1	3	5	8	11	3	5	4	5	$\{S, 1, 6, 2, 8, 3, 7, 9, 4\}$
0	1	3	5	8	11	3	5	4	5	$\{S, 1, 6, 2, 8, 3, 7, 9, 4, 5\}$

□

2.1.4 Grafen: opspannende bomen

Een opspannende boom in een gewogen graaf is een deelverzameling van de kanten die samen een boom vormen waarin alle knopen zijn opgenomen. Deze boom is niet noodzakelijk geworteld. De meest voor de hand liggende toepassing van een opspannende boom is die van de netwerk (bijvoorbeeld telefoon) verbinding. Tussen een aantal punten van het vlak zijn verbindingen mogelijk, ieder met hun eigen kosten. Het vinden van een opspannende boom is dan het vinden van een deelverzameling van de kanten zo dat elk van de punten door middel van een pad met elk ander punt verbonden is, maar niet door twee paden. De kunst van de optimalisatie is hier natuurlijk het vinden van een opspannende boom waarvan het totale gewicht minimaal is, de zogenoemde *mincost spanning tree*. Er zijn twee bekende gulzige algoritmen voor het vinden van de opspannende boom, vernoemd naar de uitvinders van deze algoritmen. Beide algoritmen starten met een lege verzameling kanten en breiden deze verzameling uit totdat een opspannende boom van minimale kosten is gebouwd.

De algoritme van Prim

De opzet van de algoritme is simpel. Begin met de lege verzameling. Voeg een kant van minimaal gewicht toe, dan hebben we een boom met 2 knopen. Als we een boom met k knopen hebben, kunnen we een boom met $k + 1$ knopen krijgen door de minimale kant toe te voegen die deze boom verbindt met een knoop die nog niet in de boom zit. Zo breiden we de boom uit totdat alle knopen in de boom zijn opgenomen. Deze boom is ook van minimale kosten, maar dat zullen we voor beide algoritmen tegelijkertijd bewijzen.

De algoritme van Kruskal

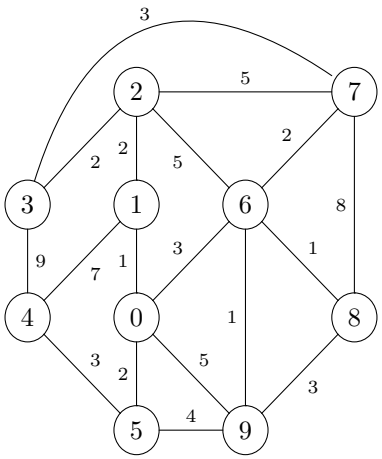
Ook hier beginnen we met een lege graaf en voegen we kanten toe, te beginnen met de kant van minimaal gewicht. Orden de kanten naar gewicht en voeg een volgende kant (in de ordening) aan de verzameling kanten toe als deze geen cykel introduceert. Als de graaf waarmee je begint verbonden is, dan worden zo uiteindelijk alle knopen in de graaf met elkaar verbonden, en heb je een opspannende boom. In tegenstelling tot Prim's algoritme maken we ons niet druk om het bijhouden van een gedeelte van een opspannende boom, maar laten we een bos ontstaan dat langzaamaan tot een enkele boom aan elkaar groeit.

Correctheid

Dat deze algoritmen correct zijn volgt uit het feit dat ze beide de volgende invariant respecteren. *op elk tijdstip is de opgenomen verzameling kanten deelverzameling van de verzameling kanten van een minimale opspannende boom*. Merk op, er kan meer dan één minimale opspannende boom zijn. We zullen laten zien dat de invariant behouden blijft m.b.v. een inductief bewijs. Uiteraard is voor beide algoritmen de invariant gerespecteerd aan het begin van de uitvoering. Immers de verzameling kanten is leeg, en de lege verzameling is deelverzameling van elke verzameling. Stel nu eens dat er ergens in de algoritme een punt t is waar de invariant niet meer gerespecteerd wordt, en neem aan dat we de eerste keer bekijken dat een kant e wordt opgenomen die niet element is van *enige* minimale opspannende boom. De verzameling van kanten die tot tijdstip $t-1$ was opgenomen, E_{t-1}, was nog *wel* deelverzameling van een minimale opspannende boom, zeg T_{t-1}. Bekijk de verzameling $T = T_{t-1} \cup \{e\}$. Omdat T_{t-1} een boom is en $e \notin T_{t-1}$ heeft T een cykel.

Laat V_{t-1} de verzameling eindpunten van E_{t-1} zijn. Er is minstens één punt op de cykel dat niet in V_{t-1} zit. Immers e heeft maar één eindpunt in in V_t (Prim) of de opname van e in stap t introduceert geen cykel (Kruskal). Er is dus een kant e' in T die V_{t-1} verbindt met de rest van de boom. Het gewicht van deze kant is minstens zo groot als het gewicht van e, anders was e' in stap t gekozen. Dit betekent echter dat $T - \{e'\}$ een spanning tree is, waarvan de kosten hoogstens zo groot zijn als die van $T - \{e\}$ (en die dus ook een mincost spanning tree is) waarvan e een kant is.

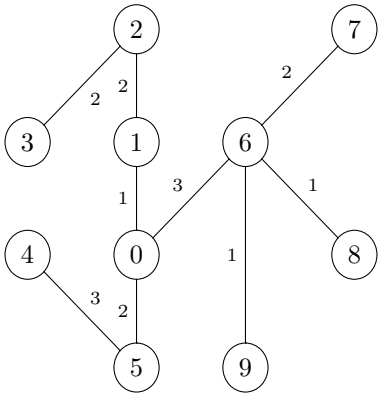
Voorbeeld 2.1.2: We gebruiken als voorbeeld de volgende graaf.



Met de beide besproken algoritmen nemen we dan achtereenvolgens de volgende knopen en kanten op.

Prim			Kruskal		
knoop	kant	gewicht	knoop	kant	gewicht
0	-	0	0	-	0
1	(0,1)	1	1	(0,1)	1
2	(1,2)	3	6,8	(6,8)	2
5	(0,5)	5	9	(9,6)	3
3	(2,3)	7	5	(5,0)	5
6	(0,6)	10	2	(1,2)	7
8	(6,8)	11	3	(2,3)	9
9	(6,9)	12	7	(6,7)	11
7	(6,7)	14	-	(0,6)	14
4	(5,4)	17	4	(4,5)	17

Het toeval wil dat de algoritmen allebei dezelfde opspannende boom opleveren, namelijk de volgende.



□

2.1.5 Scheduling

Bij het postkantoor staan mensen in een rij. Iedereen wil natuurlijk zo snel mogelijk uit de rij vandaan en daarom zou iedereen het eerst geholpen willen worden. Voor de beambte achter het loket zal het om het even zijn wie eerst geholpen moet worden, want de totaal benodigde tijd om iedereen te helpen is niet afhankelijk van in welke volgorde de personen in de rij moeten worden geholpen. Toch maakt de volgorde iets uit omdat de verschillende handelingen een verschillende hoeveelheid tijd kosten, en als de postzegels eerst verkocht worden dan moet de klant die het aangetekende stuk wil verzenden daar weliswaar op wachten, maar als het aangetekende stuk verzonden moet worden, moet de klant die de postzegels wil kopen dáár op wachten. Kortom, behalve de nuttig bestede tijd die kan worden afgemeten aan de werktijd van de beambte—deze is naar wij aannemen *altijd* nuttig bezig—is er een hoeveelheid onnuttig bestede tijd, namelijk die de klanten op elkaar staan te wachten. Ons eerste scheduling probleem is het minimaliseren van deze tijd.

Voorbeeld 2.1.3: Laat drie taken met hun tijdsduur gegeven zijn $t_1 = 5$, $t_2 = 10$ en $t_3 = 3$. Er zijn zes mogelijke volgordes waarin deze taken kunnen worden uitgevoerd.

Volgorde	Tijd
123	5+(5+10)+(5+10+3)=38
132	5+(5+3)+(5+3+10)=31
213	10+(10+5)+(10+5+3)=43
231	10+(10+3)+(10+3+5)=41
312	3+(3+5)+(3+5+10)=29
321	3+(3+10)+(3+10+5)=34

□

In de eerste regel van dit voorbeeld is klant 1 het eerst aan de beurt, haar totale tijd in de rij is dus 5, daarna komt klant 2 die eerst op klant 1 heeft moeten wachten zodat haar totale tijd in de rij 15 is. Tenslotte komt klant 3 die eerst op klant 1 en klant 2 heeft moeten wachten zodat haar totale tijd in de rij 18 is. In dit voorbeeld valt op dat de volgorde van behandelen waarbij de klant die het *minste* tijd vraagt het eerst behandeld wordt, de minste totale wachttijd oplevert. Mogelijk is dat ook de basis voor onze gulzige algoritme.

We bewijzen dit als volgt. Laat $P = p_1, p_2 \ldots, p_n$ een permutatie zijn van $1, \ldots, n$ en laat $s_i = t_{p_i}$—de tijd die taak p_i nodig heeft—zijn. Als de taken worden uitgevoerd in volgorde P dan is de totale tijd die nodig is om alle taken uit te voeren

$$\begin{array}{rcl} T(P) & = & s_1 + (s_1 + s_2) + (s_1 + s_2 + s_3) + \ldots \\ & = & ns_1 + (n-1)s_2 + (n-2)s_3 + \ldots \\ & = & \sum_{k=1}^{n}(n-k+1)s_k \end{array}$$

Stel dat P niet de taken in volgorde van benodigde tijd uitvoert, dan zijn er getallen a en b met $a < b$ en $s_a > s_b$. Als we *alleen* a en b verwisselen, dan krijgen we een nieuwe volgorde P' met totale tijd

$$T(P') = (n-a+1)s_b + (n-b+1)s_a + \sum_{\substack{k=1 \\ k \neq a,b}}^{n} (n-k+1)s_k.$$

We berekenen

$$\begin{array}{rcl} T(P) - T(P') & = & (n-a+1)(s_a - s_b) + (n-b+1)(s_b - s_a) \\ & = & (b-a)(s_a - s_b) > 0 \end{array}$$

Dus elk schema waarin een twee taken worden uitgevoerd die niet in oplopende volgorde van de tijd staan die die taken nodig hebben is suboptimaal.

Deadlines

Allerlei verfijningen kunnen aan scheduling worden opgehangen. Eén van de meest gebruikelijke is dat taken niet onbegrensd lang op elkaar kunnen wachten. Aan elke taak kunnen we een "deadline" hangen, dwz een tijdstip waarop de taak moet worden uitgevoerd om nog er nog profijt van te kunnen hebben. We maken de vereenvoudiging dat elke taak nu slechts één eenheid tijd kost. Laat d_i de deadline voor taak i zijn, die als zij vóór de deadline wordt uitgevoerd profijt g_i oplevert. Laten we eens naar de volgende vier taken kijken.

i	1	2	3	4
g_i	50	10	15	30
d_i	2	1	2	1

De deadlines in dit voorbeeld zijn zo strak gesteld dat er maar een paar mogelijke manieren zijn om de taken uit te voeren. De overgebleven volgordes zien er als volgt uit.

Volgorde	Opbrengst	Volgorde	Opbrengst
1	50	2,1	60
2	10	2,3	25
3	15	3,1	65
4	30	4,1	80
1,3	65	4,3	45

We noemen een deelverzameling van de taken *uitvoerbaar* als er een volgorde is waarin elk van de taken in deze deelverzameling kan worden uitgevoerd voor zijn deadline. Een voor de hand liggende gulzige algoritme om ons probleem op te lossen is nu—beginnend met de lege verzameling—telkens de taak met de grootste

opbrenst aan de verzameling toe te voegen zodat de samengestelde verzameling uitvoerbaar blijft, totdat dit niet meer mogelijk is. In het voorbeeld zouden we eerst taak 1 kiezen en daar vervolgens taak 4 aan toe kunnen voegen omdat de volgorde 4,1 uitvoerbaar is. De volgende taak zou taak 3 zijn, maar $\{1, 3, 4\}$ is niet uitvoerbaar en ook $\{1, 2, 4\}$ is niet uitvoerbaar, dus deze algoritme eindigt met $\{1, 4\}$. Vindt deze algoritme altijd een optimaal schema?

Eerst merken we op dat als we een verzameling taken hebben dat uitvoerbaar is en we willen er een nieuwe taak aan toevoegen, dat we dan niet alle permutaties hoeven na te gaan om te zien of de nieuwe verzameling uitvoerbaar is. Dit wegens de volgende eigenschap.

> Als J een verzameling van k taken is, genummerd $d_1 \leq \ldots \leq d_k$ dan is J uitvoerbaar als en alleen als de volgorde $1, 2, \ldots, k$ uitvoerbaar is.

Het is duidelijk dat de verzameling uitvoerbaar is als de volgorde $1, \ldots, k$ uitvoerbaar is. Omgekeerd als $1, 2, \ldots, k$ niet uitvoerbaar is, dan wordt in die volgorde tenminste één taak na zijn deadline gepland. Als r zo'n taak is dan is dus $d_r \leq r - 1$. Aangezien de taken in de volgorde van niet-dalende deadlines zijn gerangschikt betekent dat dat tenminste r taken een deadline kleiner dan of gelijk aan $r - 1$ hebben. Hoe je die ook rangschikt, er komt er dan altijd één te laat.

> De gulzige algoritme hierboven geschetst vindt altijd de optimale volgorde.

Stel dat deze algoritme één of andere volgorde I kiest die niet optimaal is. Er is een andere volgorde J die *wel* optimaal is. Zet de volgordes S_I en S_J onder elkaar, mogelijk met gaten. We herschikken de taken in S_I en S_J, ook mogelijk met gaten, zodat uitvoerbare volgordes S'_I en S'_J ontstaan, waarin elke taak die S'_I en S'_J voorkomt in S'_J recht onder dezelfde taak in S'_I staat. Dat dit kan is als volgt in te zien. Stel dat taak a in S_I en S_J voorkomt. op plaasten t_I en t_J respectievelijk. Als $t_I = t_J$ is er niets te doen, stel daarom dat $t_I < t_J$. De deadline voor a is niet eerder dan t_J. We kunnen nu a in S_I verplaatsen naar t_J. Als er in S_I een gat zit op plaats t_J, dan is er niets meer te doen. Als op plaats t_J een taak b staat, dan verwisselen we a en b. De taak b wordt dan naar voren gehaald in de volgorde, en dat kan zeker niet tot onuitvoerbaarheid leiden.

Nu we aldus alle gemeenschappelijke taken op dezelfde plaats hebben gezet zijn dus de overige taken (inclusief gaten) die tegenover elkaar staan verschillend. Stel er is een tijdstip t waarop in S'_I een taak a staat en in S'_J iets anders.

1. Als op tijdstip $t = t_1$ een gat in S'_J zit, dan zit a nergens in S'_J en kan a worden toegevoegd waardoor de opbrengst in J' stijgt. Dit is in tegenspraak met de vooronderstelde optimaliteit van S_J.

2. Als a een gat is op $t = t_2$ en ertegenover staat een taak b, dan is de verzameling $I \cup \{b\}$ een uitvoerbare verzameling en heeft de gulzige algoritme geen geldige reden om te stoppen na de vorming van I.

3. De overgebeleven mogelijkheid is dat tegenover een werkelijke taak a' op $t = t_3$ een werkelijke taak b' staat. Dan staat a' niet in J en b' niet in I. Er zijn dan drie mogelijkheden.

   (a) $g_{a'} > g_{b'}$, dan kan b' vervangen worden door a' en de opbrengst van J vergroot worden. Dit is in tegenspraak met de optimaliteit van J.

   (b) $g_{a'} < g_{b'}$, maar dan is $I - \{a'\} \cup \{b'\}$ uitvoerbaar *en* $g_{a'} < g_{b'}$ dus als de gulzige algoritme a' kiest, dan kiest zij b'. Tegenspraak.

   (c) $g_{a'} = g_{b'}$.

		t_1		t_2		t_3	
S'_I	$\ldots$	a	$\ldots$		$\ldots$	a'	$\ldots$
S'_J	$\ldots$		$\ldots$	b	$\ldots$	b'	$\ldots$

De enige manier waarop I en J dus kunnen verschillen is als ze verschillende taken hebben met dezelfde opbrengst, maar dan zijn hun totale opbrengsten natuurlijk ook gelijk.

Meerdere Verwerkingseenheden

Tot hier hebben we het gehad over taken die moeten worden uitgevoerd door één verwerkingseenheid. We kunnen ons echter een postkantoor voorstellen waar meerdere beambten aan een balie staan. Iedere beambte kan in principe elke klant helpen, en de klant wordt geholpen door de eerstvolgende vrijkomende beambte. Gebruikelijk is in dit systeem dat de klanten worden afgehandeld in volgorde van binnenkomst, maar is dit ook optimaal?

Ons model is een verzameling van machines die allemaal elke taak van een verzameling taken uit $\{1, \ldots, n\}$ kunnen uitvoeren. De eenvoudigste vorm van het probleem is die waarbij elke uit te voeren taak een starttijd s_i en een eindtijd t_i heeft. Het optimaliseringsprobleem is het gebruiken van een minimaal aantal machines om alle taken uit te voeren. Het is duidelijk dat twee taken niet op dezelfde machine kunnen worden uitgevoerd, als de tijd waarin ze worden uitgevoerd overlapt. De voor de hand liggende strategie is dus de taken net zolang aan een gegeven verzameling machines toe te wijzen totdat het niet meer mogelijk is om een nieuwe taak aan één van de machines uit de verzameling toe te wijzen. Dan schakelen we een nieuwe machine in. Om deze algoritme geordend uit te voeren, ordenen we eerst de taken naar starttijd en voegen de taken in die volgorde in. Deze strategie blijkt in dit geval ook optimaal. Dit kunnen we als volgt bewijzen.

Stel onze algoritme vindt voor de taken $\{1, \ldots, n\}$, waarbij de starttijden oplopend, dat er k machines nodig zijn. We moeten laten zien dat het niet met minder kan. Laat taak i de taak zijn waarvoor de k-de machine erbij is geroepen. Taak i kon niet worden uitgevoerd op machines $1, \ldots, k-1$ dus moet er aan elk van deze machines een taak toegewezen T_j zijn die een conflict heeft met taak i. De starttijd van elke T_j moet liggen voor s_i, want T_j was eerder ingedeeld en de lijst met taken is gesorteerd. De eindtijd van elke T_j moet liggen ná s_i, anders was er niet met elke machine een conflict. We zien dat deze $k-1$ taken niet alleen een conflict hebben met de i-de taak, maar *ook met elkaar*. Immers het tijdstip s_i komt in al deze taken voor. In totaal hebben we dus een verzameling van k taken die paarsgewijs een conflict hebben. Zo'n verzameling taken kan niet op $k-1$ machines worden uitgevoerd.

Op het thema scheduling bestaan enorm veel variaties. Jobs kunnen eenheid tijd nemen of een tijdsinterval. Er zijn situaties waarin elk van de jobs op elke machine kunnen worden uitgevoerd, en er zijn scenario's waarin elke job zijn eigen soort machine heeft. Een job kan op één machine worden uitgevoerd en is dan klaar of een job kan langs meerdere machines moeten. Het kan zo zijn dat een job die eenmaal op een machine is gestart ook afgemaakt moet worden of een job kan onderbroken worden en later weer opgepakt etc. etc. Voor lang niet alle van deze problemen bestaan gulzige algoritmen. Veel van deze problemen zijn zo moeilijk dat er alleen maar exponentiële tijd begrensde algoritmen voor bestaan. Verder kunnen de problemen ook nog op andere manieren als optimaliseringsproblemen worden voorgesteld dan het minimaliseren van het aantal gebruikte machines. Je kunt ook bijvoorbeeld de totale overschrijding van de deadlines minimaliseren of de totaal gebruikte tijd op alle machines. Kortom de variaties zijn eindeloos en dit onderwerp is een actief onderwerp van research.

2.1.6 Sommen

1. Laat zien dat het systeem van euromunten de eigenschap heeft dat de gulzige algoritme voor het teruggeven van een bepaald bedrag altijd het minimum aantal munten vindt. Aanwijzing. Gebruik een Lemma dat zegt dat er voor elke muntwaarde, behalve de grootste, een maximum aantal munten is dat in de optimale oplossing kan zitten.

2. Bedenk een systeem van muntwaarden waarbij de gulzige algoritme uit deze sectie niet tot een optimale oplossing leidt.

3. Toon aan dat de gulzige algoritme voor knapsack waarbij eerst zoveel mogelijk van het object met de grootste waarde/gewicht ratio altijd leidt tot een optimale oplossing.

4. Laat $P_1, \ldots, P_n$ een verzameling programmas zijn die op een schijf worden opgeslagen. Programma P_i heeft s_i megabyte ruimte nodig. De capaciteit van de schijf is D megabyte, waar $D < \sum s_i$.

(a) Kunnen we een gulzige algoritme gebruiken om zoveel mogelijk programma's op de schijf op te slaan?

(b) Kunnen we een gulzige algoritme gebruiken om zoveel mogelijk megabyten van de schijf te gebruiken?

5. Gegeven zijn de lijsten L_1, L_2, L_3 en L_4 met respectievelijke lengten 10, 20, 30, 40.

   (a) Neem aan dat de kosten van het samenvoegen van twee lijsten evenredig is met de lengte van de resulterende lijst, en dat we niet meer dan twee lijsten in één keer kunnen samenvoegen. Wat is de optimale volgorde om $L_1, \ldots, L_4$ tot één lijst L samen te voegen?

   (b) Beschrijf een greedy algoritme voor het algemene geval: input: lijsten $L_1, \ldots, L_n$ van verschillende lengte; output: één lijst L

   (c) Bewijs dat deze algoritme optimaal is; dwz. dat het gebruikte aantal bewerkingen minimaal is.

6. Geef een voorbeeld van een graaf waaruit blijkt dat bij het zoeken naar het kortste pad in de graaf het altijd kiezen van de kant met het kleinste gewicht niet tot een optimale oplossing leidt.

7. Als een graaf negatieve kanten heeft, is het vinden van een opspannende verzameling van minimale kosten nog steeds zinvol. De opspannende structuur hoeft dan niet noodzakelijk meer een boom te zijn. Hoe kunnen we onze algoritmen aanpassen zodat ze het gewenste resultaat opleveren?

8. Een graaf hoeft niet noodzakelijke één minimale opspannende boom te hebben. Wanneer is dit wel het geval? Geef een bewijs.

9. Een *Dijkstra boom* is een boom die bestaat uit een punt v en verder alle kortste paden van v naar de knopen in de graaf. (Waarom is dat een boom?). Laat zien dat voor een gegeven graaf een Dijkstra boom niet altijd een minimale opspannende boom is, maar dat een Dijkstra boom wel altijd minstens één kant gemeen heeft met een minimale opspannende boom.

10. In een bepaald gebied bevinden zich n draadloze stations. Elk station heeft zijn eigen communicatiesnelheid. Als een snel station met een langzaam station praat, dan moet de snelheid van de communicatie aangepast worden aan het langzame station. Men wil een topologie schetsen waarbij vanaf één punt elk station kan worden bereikt (en dus elk station met minstens één ander station praat) en waarbij de gemiddelde snelheid zo hoog mogelijk is. Bedenk een greedy algoritme voor dit probleem. U mag hierbij het vertragende effect van het praten van één station met meerdere andere verwaarlozen (en dus aannemen dat een snel station met meerdere snelheden kan communiceren).

Programmeren

1. Een verladingsbedrijf onderhoudt diensten tussen Rotterdam en Schaffhausen, beide gelegen aan de Rijn. Het bedrijf heeft 10 schepen met verschillende capaciteiten van twee t.e.m. tien ton. Een schip dat twee keer zo zwaar is doet anderhalf keer zo lang over de reis. Van Amsterdam naar Shaffhausen vervoert het bedrijf tarwe, rijst en mais, en in omgekeerde richting rogge, grind en zand. De winst per 100 kilo is respectievelijk 1, 3 en 4 euro voor tarwe rijst en mais, en 2, 5 en 0.5 euro rogge, grind en zand, en we houden er in de simulatie geen rekening mee dat schepen moeten worden schoongemaakt voordat ze de terugweg met een nieuwe lading kunnen aanvaarden. Experimenteer met verschillende verladingsstrategieen (hoge winst in grote schepen, snelle schepen etc.) en bereken de winst. Tekent er zich een gulzige algoritme af? U hoeft hier geen bewijzen te geven.

2.2 Verdeel en Heers

Verdeel en heers is een adagium dat teruggaat tot de oude Romeinen (divide et impera). De betekenis van dat adagium is tot op zekere hoogte vergelijkbaar met de betekenis die we er in de hedendaagse algoritmiek

aan hechten. De Romeinen waren erop gericht de tegenstanders uit elkaar te spelen, zodat ze met elkaar veel meer dan met de Romeinen oorlog zouden voeren waardoor de Romeinen hun macht konden handhaven. In de algoritmiek willen we een enkel groot probleem graag uit elkaar trekken in kleinere deelproblemen die gemakkelijk te hanteren zijn. De oplossingen voor de kleine deelproblemen *samen* vormen dan een oplossing voor het grote probleem. Als voorbeeld gebruiken we het voorbeeld van de volledige inductie gegeven in Sectie 1.4 op pagina 26. Het probleem is het volleggen van een vierkant met L vormige figuren, en het probleem is simpel als het vierkant afmetingen 2×2 heeft. Een verdeel-en-heersalgoritme dat dit probleem oplost voor grotere vierkanten gaat (dus) als in Figuur 2.1.

```
1: Square(n, i, j)
2: if n = 2 then
3:     Leg de enige L zo dat de opening op i,j komt.
4: else
5:     do 4 times
6:     set(i', j');
7:     call Square(n/2,i',j');
8:     end do 4 times
9:     put results together;
10:    return(result);
11: end if
```

Figuur 2.1: Recursieve algoritme voor de L-overdekking

Inductie, verdeel en heers, en recurrente betrekkingen gaan in de algoritmiek hand in hand. De inductie levert het bewijs van correctheid, de verdeel-en-heersstrategie levert de oplossing voor het probleem, en de bijbehorende recurrente betrekking levert de complexiteit van de algoritme. Niet altijd is het efficiënt om een verdeel-en-heersstrategie toe te passen, ook al ligt die voor de hand. In Hoofdstuk 1 hebben we daar (pagina 21) al een afschrikwekkend voorbeeld van gezien. Soms echter is verdeel en heers de enige manier om een probleem te behappen en soms ook levert het wel degelijk een efficiëntere algoritme dan de voor de hand liggende. Een beroemd geworden voorbeeld hiervan is de matrixvermenigvuldigingsalgoritme van V. Strassen [Str69].

2.2.1 Matrixvermenigvuldiging

Stel we hebben twee $n \times n$ matrices A en B die we met elkaar willen vermenigvuldigen. De algoritme om dat te doen is dat we van matrix A steeds een rij vector nemen en van matrix B een kolomvector en van deze twee vectoren het inproduct bepalen. Zo geeft het inproduct van de $i$de rij van A en de $j$de kolom van B het element i, j van de productmatrix C. De complexiteit van deze algoritme is n^3. Immers, we moeten n^2 inproducten bepalen en elk inproduct kost n vermenigvuldigingen. Er is een andere methode, gebaseerd op het verdeel-en-heersprincipe die een efficientere algoritme oplevert.

Wanneer we voor het gemak aannemen dat n een 2-macht is, $n = 2^k$ dan zien we dat we matrix A in vier deelmatrices kunnen verdelen $A_1, \ldots, A_4$ en ook matrix B in $B_1, \ldots, B_4$, met de eigenschap dat we de productmatrix C ook kunnen bepalen door 8 matrixvermenivuldigingen(zie Figuur 2.2). Immers, $C_1 = A_1 \times B_1 + A_2 \times B_3$, $C_2 = A_1 \times B_2 + A_2 \times B_4$, $C_3 = A_3 \times B_1 + A_4 \times B_3$ en $C_4 = A_3 \times B_2 + A_4 \times B_4$. Bovendien is bij deze vermenigvuldiging de aanname van commutativiteit niet nodig, zodat ze inderdaad voor matrixvermenigvuldiging geldig is.

$$\begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix} = \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix}$$

Figuur 2.2: Onderverdeling bij matrixvermenigvuldiging

We kunnen dus in plaats van de standaardalgoritme voor matrixvermengivuldiging een verdeel-en-heersstrategie implementeren die steeds kleinere (vierkante) matrices met elkaar vermenigvuldigt, waarbij het triviale geval de vermenigvuldiging van één bij één matrices is. De recurrente betrekking die bij deze algoritme hoort is $T(n) = 8T(n/2)$. Wanneer we deze betrekking oplossen zien we $T(n) \in O(n^3)$ zodat de verdeel-en-heersstrategie hier wel een andere, maar niet meteen een betere algoritme oplevert. Het probleem zit in het aantal recursieve aanroepen van de matrixvermenigvuldigingsalgoritme dat gedaan wordt. Als we op dit aantal zouden kunnen bezuinigen, dan zou de algoritme sneller worden. Dit is precies de observatie die V. Strassen [Str69] heeft gedaan. In plaats van met 8 kunnen we, door slim eerder verkregen tussenresultaten te gebruiken, toe met zeven vermenigvuldigingen, waardoor de complexiteit van de algoritme daalt naar $n^{\log 7}$, een beduidend efficiëntere algoritme.

De zeven producten zijn de volgende.

1. $M_1 = (A_3 + A_4 - A_1)(B_3 - B_2 + B_1)$
2. $M_2 = A_1 B_1$
3. $M_3 = A_2 B_3$
4. $M_4 = (A_1 - A_3)(B_4 - B_2)$
5. $M_5 = (A_3 + A_4)(B_2 - B_1)$
6. $M_6 = (A_2 - A_3 + A_1 - A_4)B_4$
7. $M_7 = A_4(B_1 + B_4 - B_2 - B_3)$

Waarna de productmatrix wordt

$$C = \begin{pmatrix} M_2 + M_3 & M_1 + M_2 + M_5 + M_6 \\ M_1 + M_2 + M_4 - M_7 & M_1 + M_2 + M_4 + M_5 \end{pmatrix}.$$

2.2.2 Zoeken en sorteren

Een zeer bekend voorbeeld van een probleemgebied waar de verdeel-en-heersstrategie succesvol is, is het zoeken en sorteren. In het voorwoord van "The Art of Computer Programming, Vol. 3" [Knu73] zegt Donald E. Knuth: "...Indeed, I believe that virtually *every* important aspect of programming arises somewhere in the context of sorting or searching!". Het hele deel 3 is dan ook aan zoeken en sorteren gewijd. Intussen, meer dan 30 jaar later, zijn er meer toepassingen van programmeertechnieken gekomen, maar toch is zoeken en sorteren nog een zeer belangrijk onderwerp in de informatica. Moderne ontwikkelingen als het internet hebben zoektechnieken alleen nog maar belangrijker gemaakt. Wij zullen hier niet honderden pagina's aan zoeken en sorteren wijden, maar we zullen wel in het kader van de verdeel-en-heerstechniek een paar van de bekendste algoritmen bespreken.

zoeken: binary search

We zullen steeds doen alsof de invoer voor zoek en sorteeralgoritmen bestaat uit een rij getallen. De complexiteit van de algoritme zullen we dan uitdrukken in het *aantal* getallen. De lengte van de invoer hangt hiermee wel samen maar is hieraan niet gelijk. Als er n getallen op de invoer staan die in m bits kunnen worden uitgedrukt is de lengte van de invoer natuurlijk nm en niet n. Toch zullen we de probleemgrootte hier voor het gemak met n aanduiden. Bij het zoeken in een rij getallen naar een gegeven getal maakt het een dramatisch verschil of de gegeven rij gesorteerd is of niet. In een ongesorteerde rij is het enige dat we kunnen doen vooraan de rij beginnen en deze van voor naar achter onderzoeken totdat we het getal gevonden hebben of niet. Worst case kost dus het doorzoeken van n getallen $\theta(n)$. Dat deze ondergrens geldt is te bewijzen met een zogenoemd "adversary argument". Voor elke methode die in één of andere volgorde de rij doorzoekt, is er een rij die het gezochte getal als laatste in deze volgorde heeft staan. Voordat alle getallen

zijn bekeken kun je dus onmogelijk zeggen of het gezochte getal in de rij staat of niet. In het bijzonder als dat getal *niet* in de rij staat.

De situatie wordt heel anders wanneer de rij getallen gesorteerd is. We kunnen nu gebruik maken van een vergelijking van de getallen en bijvoorbeeld concluderen dat als een het gezochte getal groter is dan dan een getrokken getal uit de gesorteerde rij, het getal ook groter zal zijn dan alle getallen die in de rij daarvoor komen. Dus die getallen hoeven dan niet meer bekeken te worden. Deze observatie geeft aanleiding tot de zoekalgoritme die bekend staat als *binary search*.

```
1: input: een rij getallen row[1 ... n]; een getal k
2: min=1, max=n
3: while min < max do
4:     middle=⌊min + max /2⌋;
5:     if row[middle] == k then
6:         return(middle);
7:     end if
8:     if row[middle] > k then
9:         max=middle;
10:    end if
11:    if row[middle] < k then
12:        min=middle;
13:    end if
14: end while
15: return(not found);
```

De recurrente betrekking die bij deze zoekmethode hoort is $T(n) = T(n/2) + c$, waaruit volgt dat de tijdcomplexiteit van deze algoritme $O(\log n)$ is.

sorteren: quicksort

Net als bij zoeken is de divide en conquer methode ook bij sorteren zeer in trek. Een in de praktijk veelgebruikte zoekmethode is quicksort. Deze methode heeft een worst-case complexiteit van $O(n^2)$, dus behoort de methode eigenlijk tot de "dommere" sorteeralgoritmen. Niettemin is ze zeer populair omdat ze eenvoudig te implementeren is, en in de praktijk vrijwel altijd veel goedkoper is dan $O(n^2)$. De average case complexiteit van deze sorteermethode is $O(n \log n)$, zoals we hieronder zullen zien. Eerst bespreken we de methode. Uitgangspunt is dat een rij getallen die bestaat uit 1 element gesorteerd is. Een rij van n getallen kunnen we in twee rijen verdelen door een element uit de rij te kiezen, vervolgens alles dat kleiner dan dat element is in de eerste rij op te slaan en alles dat groter dan of gelijk is aan dat element in de tweede rij op te slaan. Als we vervolgens de aldus geconstrueerde rijen sorteren en aan elkaar plakken, dan is de hele rij gesorteerd. Het element dat we kiezen om de twee deelrijen te maken zullen we de *as* noemen. In algoritmische vorm:

```
1: function Sort( row[1, ..., n])
2: if n > 1 then
3:     Choose 1 ≤ k < n;
4:     row1={row[i] : row[i] < row[k]};
5:     row2={row[i] : row[i] ≥ row[k] ∧ i ≠ k};
6:     Sort(row1);
7:     Sort(row2);
8:     return(concatenate(row1,row[k],row2);
9: else return(row);
10: end if
```

De kosten van deze algoritme zijn zeer afhankelijk van een goede keuze van de as, row[k]. We zien dat het mogelijk is deze telkens zo ongelukkig te kiezen dat de bovengrens op de complexiteit kwadratisch wordt. Als de as telkens zo gekozen wordt, dat alle overige elementen ofwel in row1 ofwel in row2 terchtkomen, dan is $T(n) = T(n-1) + n - 1$ en deze recurrente betrekking heeft de afschatting $T(n) = \theta(n^2)$. Alleen als de

beide rijen een constante *fractie* van de oorspronkelijke rij zijn, dan wordt een betere grens bereikt. Immers $T(n) = T(n/c) + T(n - n/c) + O(n)$ heeft een $O(n \log n)$ afschatting. We zullen bewijzen dat dit in het gemiddelde geval, bij een uniforme distributie over alle mogelijke ongesorteerde (en een gesorteerde) rijen geldt.

Er zijn $n!$ mogelijke volgorden waarin de getallen (laten we voor het gemak $1, \ldots, n$ aannemen) kunnen staan. Laat R_i de rij in volgorde i voor $i \leq n!$ zijn, $T(R_i)$ is dan de tijd die het kost om R_i met quiksort te sorteren. De gemiddelde tijd $\overline{T}$ is dan $\frac{\sum_{i=1}^{n!} T(R_i)}{n!}$. Als bekend is op welke plaats de as in R_i thuishoort, kunnen we de kosten van het sorteren van R_i bepalen. Dat is namelijk $T(R_{i1}) + T(R_{i2}) + n - 1$. Als R_i uniform verdeeld is, dan is elke plaats voor de as in de gesorteerde rij even waarschijnlijk, en is dus $T(R_i) = \sum_{j=0}^{n} T(R_i[1, \ldots j-1]) + T(R_i[j+1, \ldots n]) + n - 1$. Merk op dat het geval dat de hele rij recursief wordt doorgegeven in de procedure hier ook wordt meegenomen. Aangezien R_i uniform verdeeld is, zijn de beide deelrijen ook uniform verdeeld, dat will zeggen voor elke plaats i heeft elk element dezelfde kans om op plaats i te staan. We kunnen dus voor een willekeurige rij R_i ook stellen $T(n) = n - 1 + \frac{\sum_{i=1} T(i-1) + T(n-i)}{n}$. Nu merken we op dat $\sum_{i=1}^{n} T(i-1) = \sum_{i=1}^{n} T(n-i)$.

We herschrijven de uitdrukking voor $T(n)$ tot:

$$T(n) = n - 1 + 2/n \left(\sum_{i=1}^{n} T(i-1) \right),$$

met $T(0) = T(1) = 0$.

Vermenigvuldigen met n levert

$$nT(n) = n(n-1) + 2 \sum_{i=1}^{n} T(i-1).$$

Dus ook dat

$$(n-1)T(n-1) = (n-1)(n-2) + 2 \sum_{i=1}^{n-1} T(i-1).$$

Trekken we deze twee van elkaar af, dan zien we dat

$$nT(n) - (n-1)T(n-1) = n(n-1) - (n-1)(n-2) + 2T(n-1)$$

Ofwel

$$T(n) = (1 + \frac{1}{n})T(n-1) + (2 - \frac{2}{n})$$

Substitueer $T(n) = (n+1)y_n$ en we krijgen

$$y_n = y_{n-1} + \frac{2(n-1)}{n(n+1)}$$

Uitrollen van deze recurrentie geeft

$$\begin{array}{rcl} y_n & = & 2 \sum_{j=1}^{n} \frac{j-1}{j(j+1)} \\ & = & 2 \sum_{j=1}^{n} \{ \frac{2}{j+1} - \frac{1}{j} \} \\ & = & 2 \sum_{j=1}^{n} \frac{1}{j} - \frac{4n}{n+1} \end{array}$$

Dus is $T(n) = 2(n+1) \sum_{j=1}^{n} \frac{1}{j} - 4n$, waaruit volgt dat $T(n) \in O(n \log n)$.

sorteren: mergesort

De toepassing van verdeel en heers die we als laatste voorbeeld bespreken heeft een $O(n \log n)$ bovengrens, en kan dus, gegeven de volgende sectie worden geklassificeerd als de meest efficiënte sorteermethode. Dat deze methode in de praktijk toch niet het meest wordt toegepast is gelegen in het feit dat quicksort toch in de meeste gevallen een $O(n \log n)$ performance haalt en bovendien zeer eenvoudig te implementeren is. De gedachte achter mergesort is weer dat een rij die bestaat uit één element al gesorteerd is, en dat bovendien twee gesorteerde rijen van gelijke lengte kunnen worden samengevoegd tot één gesorteerde rij van de dubbele lengte door steeds het kleinste element van de twee rijen eruit te halen en op te slaan. Aangezien de twee rijen gesorteerd zijn, kan het kleinste element in $O(1)$ tijd gevonden worden en dus kunnen de rijen tot één rij worden samengevoegd in $O(n)$ tijd. De sorteeralgoritme bestaat dan uit het herhaaldelijk in twee gelijke (of bijna gelijke) delen splitsen van een rij, de twee verkregen deelrijen op dezelfde manier sorteren, en vervolgens de twee gesorteerde rijen samen te voegen. In pseudotaal:

```
1: Sort(row[1,...,n])
2: if n > 2 then
3:     row1=row[1, ..., ⌊n/2⌋];
4:     row2=row[⌊n/2⌋ + 1, ..., n];
5:     row1=Sort(row1);
6:     row2=Sort(row2);
7:     return(Merge(row1,row2);
8: else
9:     return(row);
10: end if
```

De routine Merge wordt dan als volgt.

```
1: Merge(row_1[1, ..., n], row_2[1, ..., m])
2: while notempty(row_1) and notempty(row_2) do
3:     pick the minimal element from row_1 and row_2;
4:     add it to row;
5: end while
6: return(row);
```

Voor het sorteren van een rij van lengte n wordt dus twee keer de routine Sort op rijen van lengte ongeveer $n/2$ aangeroepen en één keer de routine Merge op twee rijen van lengte ongeveer $n/2$. De recurrente betrekking die hierbij hoort is $T(n) = 2T(n/2) + O(n)$, waaruit een $O(n \log n)$ worst case bovengrens volgt.

sorteren: $\Omega(n \log n)$ ondergrens

Het sorteren volgens MergeSort levert een worst case bovengrens die gelijk is aan de ondergrens die geldt voor sorteren in het algemeen, als dat sorteren tenminste gebaseerd is op vergelijken van twee elementen uit de te sorteren rij. Er zijn nog efficientere methoden denkbaar, bv. radix sort[1], maar deze sorteermethoden zijn doorgaans op andere principes gestoeld dan de vergelijking van twee elementen in de rij.

Voor sorteermethoden die gebaseerd zijn op vergelijken en verplaatsen van elementen is $\Omega(n \log n)$ een ondergrens. We kunnen dit als volgt inzien. Beschouw de te sorteren rij als een rij getallen die in een boomstructuur gevoerd wordt, waarbij elke rij langs een pad naar beneden loopt. In elke knoop langs het pad wordt op een rij een operatie (vergelijking, verplaatsing) uitgevoerd, waardoor uiteindelijk in het blad horende bij het gevolgde pad de gesorteerde rij staat. Twee verschillende rijen moeten een verschillend pad volgen. Immers als op twee verschillende rijen verschillende operaties uitgevoerd worden, kunnen ze

[1] Bij radix sort wordt er net zoveel geheugen gereserveerd als de grootte van het maximale element in de rij. Vervolgens wordt voor ieder element dat wordt aangetroffen in de rij een boolese waarde in het bijbehorende geheugen op true gezet, en vervolgens wordt het hele array nagelopen en wordt het adres van de elementen die op true staan in een gesorteerde rij geconcateneerd. Soms is deze methode efficient. Immers de complexiteit is lineair in de waarde van het grootste element, en bovendien is het geheugengebruik ook lineair in de waarde van dit getal. Als die waarde vergelijkbaar is met het aantal elementen in de rij, dan hebben we dus een lineaire sorteermethode. Echter, aangezien een getal n met $\log n$ bits te representeren is, kan deze sorteermethode ook exponentieel zijn.

niet allebei door dezelfde serie operaties (verandering van de volgorde) gesorteerd worden. Aangezien er $n!$ volgorden zijn, moet de boom dus $n!$ bladeren hebben. Dientengevolge is er een pad in de boom van lengte $\Omega(\log n!) = \Omega(n \log n)$.

2.2.3 Sommen

1. Laat $T[1..n]$ een gesorteerd array van verschillende integers zijn, die positief of negatief mogen zijn. Geef een $O(\log n)$ algoritme die een index i vindt met $T[i] = i$, als zo'n index bestaat.

2. Stel er zijn drie algoritmen om hetzelfde probleem aan te pakken.

   - Algoritme A lost het probleem op door het in vijf deelproblemen te verdelen, die recursief op te lossen en de oplossingen in lineaire tijd te combineren.
   - Algoritme B lost problemen van grootte n op door twee deelproblemen van grootte $n-1$ op te lossen en vervolgens de oplossingen in constante tijd te combineren.
   - Algoritme C lost problemen van grootte n op door ze in negen deelproblemen van groote $n/3$ te verdelen, die recursief op te lossen en dan de oplossingen in $O(n^2)$ tijd te combineren.

   Welke algoritme kiest je en waarom?

3. Een staafdiagram wordt door een programma geproduceerd op een $n \times n$ vierkant. Bedenk een algoritme die de langste staaf in het diagram vindt in $O(n)$ (dus *niet* $O(n^2)$) tijd.

4. Een onverlaat heeft de potjes babyvoeding in een supermarkt geïnfecteerd. Er wordt besloten het geïnfecteerde potje te vinden door proefdieren in te zetten. De incubatietijd van het virus is 48 uur. De dierenbescherming ziet de noodzaak van het zo snel mogelijk identificeren van het besmette potje en het onderzoek op vingerafdrukken wel in, maar eist toch dat er niet meer dan $O(\log n)$ proefdieren per n potjes worden gebruikt. Kan het besmette potje binnen 50 uur gevonden worden?

5. Een spel van n kaarten bevat verschillende afbeeldingen. Het spel heeft een grote deelverzameling als $n/2$ of meer kaarten dezelfde afbeelding hebben. Geef een $O(n \log n)$ algoritme die bepaalt of het spel een grote deelverzameling heeft.

Programmeren

1. Pluk een paar flinke teksten van het web en vergelijk de verschillende sorteeralgoritmen, mergesort, quicksort en insertion sort.

2. Matrixvermenigvuldiging kan gebruikt worden om de transitieve afsluiting van een graaf te berekenen. Maak een random graaf op 25 knopen en bereken de transitieve afsluiting van de graaf. Doe dat op twee manieren: de standaardvermenigvuldiging en Strassen's methode. Vergelijk de tijden.

2.3 Dynamisch Programmeren

Een belangrijke techniek in de optimalisatie is de techniek van het dynamisch programmeren. Het is in tegenstellng tot de top down aanpak van de verdeel-en-heersstrategieën een bottom up aanpak die probeert optimale oplossingen voor deelproblemen te vinden. Deze worden vervolgens samengesteld tot een optimale oplossing voor het hele probleem. Een beetje zoals een legpuzzel wordt opgelost. Telkens wordt een stukje van de legpuzzel opgelost. Als de stukken grotere herkenbare verbanden gaan vormen worden ze in elkaar gepast. In 1 zagen we al, bij het berekenen van Fibonaccigetallen dat het opslaan van tussenresultaten soms tot dramatische verbering kan leiden. Het berekenen van Fibonaccigetallen is echter geen optimalizeringsprobleem. Daarom is dit niet een typisch voorbeeld voor de aanpak van een probleem met dynamisch programmeren. Een ander probleem dat we eerder hebben gezien, dat van het geven van wisselgeld kan *wel* worden aangepakt met dynamisch programmeren. Dat zullen we hier eerst uitwerken. In Som 2 op pagina 43

hebben we gezien dat niet in elk systeem de gulzige algoritme voor geldteruggave tot een optimale oplossing leidt. De hieronder beschreven aanpak met dynamisch programmeren doet dat wel.

Stel we hebben een muntsysteem dat munten van de waarden $w_1, \ldots, w_n$ heeft, en we moeten een bedrag B teruggeven. We nemen even aan dat de w_1 de eenheid in het muntsysteem is en dat B dus in ieder geval altijd bereikt kan worden. Nu vinden we een optimale oplossing als volgt. Stel dat we een optimale oplossing weten die bestaat uit muntwaarden alleen uit $w_1, \ldots, w_i$, dan kunnen we een optimale oplossing met muntwaarden uit $w_2, \ldots, w_{i+1}$ bereken door te observeren dat we de altijd de keuze hebben wel of geen munten van waarde w_{i+1} te gebruiken. Als we *geen* muntwaarde van type w_{i+1} gebruiken, dan is de optimale waarde dezelfde als die we hadden berekend bij teruggave met alleen waarden uit $w_1, \ldots, w_i$. Als we *wel* munten van type w_{i+1} gebruiken, dan kunnen we hoogstens $W = B/w_{i+1}$ van deze munten gebruiken, en een optimale waarde vinden we dus door voor $j \leq W$ te berekenen wat het optimale aantal munten voor het bedrag $B - j \times w_{i+1}$ is plus j. Dus $\text{Opt}(B, i+1) = \min\{\text{Opt}(B, i)\} \bigcup \{\text{Opt}(B - j \times w_{i+1}, j) + j : j \leq \lfloor B/w_{i+1} \rfloor\}$

Waarbij natuurlijk $\text{Opt}(B, 1)$ gelijk aan B is voor alle B. We kunnen ons een tabel voorstellen waarbij we de volgende regel (met muntwaarden w_{i+1} samenstellen uit door een aantal keren w_i in de vorige regel terug te kijken en dan te zien of we een kleiner getal kunnen krijgen. Misschien is een getallenvoorbeeld hier illustratief.

Voorbeeld 2.3.1: Neem een muntsysteem met waarden 1,4,5, en 7 en laat 9 het terug te geven bedrag zijn. We krijgen dan:

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

Merk overigens op dat de greedy algoritme het bedrag 9 zou vormen uit 7, 1, en 1. □

Het hier beschreven patroon is een generiek patroon voor algoritmen met dynamisch programmeren. Het is alleen de dimensie van de tabel en de manier waarop deze wordt ingevuld (rijgewijs, kolomsgewijs, of langs de diagonalen) die verschilt. We bekijken nog een bekende.

2.3.1 Dynamisch programmeren en Knapsack

Het knapsackprobleem konden we met een gulzige algoritme oplossen gegeven dat de objecten op elke manier gedeeld kunnen worden. Dat is natuurlijk waar voor koffie en brood, maar is minder waar voor bijvoorbeeld kristallen kroonluchters. Het knapsackprobleem wordt beduidend moeilijker als de keuze beperkt wordt tot het wel of niet meenemen van objecten. Deze variant wordt ook wel de 0-1 variant van knapsack genoemd.

We kunnen 0-1 knapsack oplossen met dynamisch programmeren als volgt. Bij elk voorwerp kunnen we de beslissing nemen het voorwerp wel of niet mee te nemen. Stel dat we weten wat de optimale oplossing is voor het probleem als we het probleem van het wel of niet meenemen van de voorwerpen beperken tot de eerste i voorwerpen *voor alle getallen tussen* 0 *en* b. Als we bij het $i+1$e voorwerp zijn aangekomen, dan kunnen we dit voorwerp niet meenemen (dus blijft de optimale oplossing voor b geldig voor de eerste i voorwerpen) *of* het voorwerp *wel* meenemen, in welk geval we nog $b - w_{i+1}$ te besteden hebben aan de eerste i voorwerpen. Onze beslissing hangt nu alleen maar af van de vraag of $\text{Opt}(\{1, \ldots, i\}, b)$ groter of kleiner is dan $\text{Opt}(\{1, \ldots, i\}, b - w_{i+1}) + w_{i+1}$. Zo kunnen we onze oplossingsverzameling uitbreiden tot we alle voorwerpen hebben opgenomen, en $\text{Opt}(\{1, \ldots, n\}, b)$ vertelt ons welke oplossing optimaal is voor de verzameling van alle voorwerpen en b. We kunnen dit in een tabelvorm opschrijven, waarbij we langs de horizontale as de voorwerpen uitzetten en langs de verticale as de gewichten. Wellicht laat zich dit illustreren aan de hand van een voorbeeld. Om het probleem wat te vereenvoudigen stellen we de waarde van een object gelijk aan het gewicht ervan.

Voorbeeld 2.3.2:

	1	2	3	4	5	6	$b = 7$
$w_1 = 3$	0	0	3	3	3	3	3
$w_2 = 2$	0	2	3	3	5	5	5
$w_3 = 5$	0	2	3	3	5	5	7
$w_4 = 4$	0	2	3	4	5	5	7
$w_5 = 6$	0	2	3	4	5	6	7

Als de waarde in een kolom van rij i naar rij $i+1$ verandert, betekent dat dat het gunstiger is het nieuw toegelaten object *wel* mee te nemen. □

2.3.2 Tekstwijzigen

Een tekstverwerker zoekt (een groot gedeelte van de tijd) naar een patroon in een tekst. Een tekst is hierbij een lange rij tekens, en een patroon is een woord of een gedeelte van een zin (korte rij tekens). De matching hoeft hierbij niet volledig te zijn. Er kunnen, zowel in het patroon als de tekst letters zijn weggelaten, of veranderd.

Meer precies: we zoeken een patroon P in een text T over hetzelfde alfabet. De *edit afstand* tussen P en T is het kleinste aantal veranderingen dat moet worden aangebracht om een deelrij van tekens van T te veranderen in P. De veranderingen kunnen zijn:

1. *Substitutie* - twee overeenkomstige letters kunnen verschillen, bijvoorbeeld *liggen* → *leggen*.
2. *Invoeging* - We kunnen een letter aan T toevoegen die in P voorkomt, bijvoorbeeld *gat* → *gaat*.
3. *Weglating* - We kunnen een letter uit T weglaten die niet in P voorkomt, bijvoorbeeld *plop* → *pop*.

Er zijn natuurlijk legio andere operaties op P denkbaar die een deelstring van T kunnen opleveren—denk bijvoorbeeld aan verwisseling van letters—maar deze operaties vormen een basis voor andere operaties. Dat wil zeggen, andere operaties die van P een deelstring van T maken zijn kunnen uit deze drie operaties worden samengesteld (in feite zelfs uit de laatste twee).

Laten we proberen dit probleem met dynamisch programmeren op te lossen. We hebben dan één of andere functie nodig die geoptimaliseerd dient te worden. Natuurlijk is het aantal verschillen tussen het patroon en enige deelstring van T een duidelijke kandidaat. P moet op enige plaats langs T gelegd worden, en we vergelijken de letters van P, P_1,...,P_n, met een aantal letters van T—dit aantal kan zowel groter als kleiner zijn dan n. Laten we het minimaal aantal verschillen tussen $P_1 \dots P_i$ en een deelstring van T die eindigt in j bijhouden in een array $D[i, j]$. Stel dat we al een gedeelte van P, zeg $P_1 \dots P_i$ hebben gepast op een deelstring van T en dat we bij T_j zijn aangekomen. Hoe breiden we nu het passende patroon uit? Wat is $D[i, j]$? Er zijn drie gevallen.

1. Als $P_i = T_j$ dan is $D[i, j] = D[i-1, j-1]$, als $P_i \neq T_j$ dan kunnen we substitutie plegen en $D[i, j] = D[i-1, j-1] + 1$.
2. $D[i-1, j] + 1$, dat wil zeggen we voegen een letter in het patroon toe om het patroon op de tekst te laten lijken.
3. $D[i, j-1] + 1$, we laten een letter uit het patroon weg om het patroon op de tekst te laten lijken.

Met deze regels kunnen we een matrix vullen, het getal waarin we geinteresseerd zijn, dat de kosten geeft om het patroon helemaal gelijk te maken aan de tekst, staat dan rechts onderin. Als we alleen geïnteresseerd zijn in gelijkheid van het patroon aan een substring *ergens* in T, zoals in een edit programma, dan vinden we de kosten door het minimum van de laatste rij te nemen. We kunnen niet alleen de kosten vinden, maar door terugredeneren ook de plaatsing en de bijbehorende operaties terugvinden. Bekijk het volgende voorbeeld.

Voorbeeld 2.3.3: In de tabel hieronder is de dynamisch programmeren algoritme toegepast op de tekst "...zijn spam-filters handig..." en het patroon "spam-fillers"—de werkelijke tekst is uiteraard veel langer, maar de bijbehorende matrix zou niet prettig zijn voor de layout van de tekst.

```
 |    z  i  j  n     s  p  a  m  -  f  i  l  t  e  r  s     h  a  n  d  i  g
-------------------------------------------------------------------------------
 | 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
s| 1  1  2  3  4  5  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
p| 2  2  2  3  4  5  6  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
a| 3  3  3  3  4  5  6  6  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21
m| 4  4  4  4  4  5  6  7  6  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
-| 5  5  5  5  5  5  6  7  7  6  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
f| 6  6  6  6  6  6  6  7  8  7  6  5  6  7  8  9 10 11 12 13 14 15 16 17 18
i| 7  7  6  7  7  7  7  7  8  8  7  6  5  6  7  8  9 10 11 12 13 14 15 16 17
l| 8  8  7  7  8  8  8  8  8  9  8  7  6  5  6  7  8  9 10 11 12 13 14 15 16
l| 9  9  8  8  8  9  9  9  9  9  9  8  7  6  6  7  8  9 10 11 12 13 14 15 16
e|10 10  9  9  9  9 10 10 10 10 10  9  8  7  7  6  7  8  9 10 11 12 13 14 15
r|11 11 10 10 10 10 10 11 11 11 11 10  9  8  8  7  6  7  8  9 10 11 12 13 14
```

We zien dat in de onderste kolom de minimale waarde 6 optreedt. Als we de diagonaal van dit getal terugvolgen zien we dat deze één keer verkregen wordt door substitutie (de t wordt in een l veranderd), en vijf keer door weglating van respectievelijk z,i,j,n, en een spatie.

□

2.3.3 Kortste paden 2: Floyd-Warshall

Bij de gulzige algoritmen in 2.1.3 hebben we al gezien hoe we de het kortste pad kunnen berekenen tussen een punt a en alle overige punten uit de graaf. Stel nu dat we willen weten, bijvoorbeeld om een afstandstabel tussen steden te bouwen, wat het kortste pad is tussen *alle paren* punten van een graaf. We kunnen dan Dijkstra's algoritme toepassen op alle punten van de graaf en zo de afstandstabel opbouwen. Er is echter ook nog een andere methode, die past in dit hoofdstuk omdat het een toepassing van dynamisch programmeren is. De observatie die een dynamic programming aanpak mogelijk maakt is dat we elk punt in de graaf afzonderlijk wel of juist niet toe kunnen laten op een af te leggen pad van a naar b. Als we geen enkel intermediair punt toelaten op het pad van a naar b dan is het kortste pad simpel de lengte van de kant van a naar b (of ∞ als die kant niet bestaat), en als we het kortste pad weten tussen alle paren knopen in de graaf dat loopt over punten uit $\{0,\ldots,i\}$ dan kunnen we het kortste pad in de graaf berekenen dat loopt over punten uit $\{0,\ldots,i+1\}$, door het korste pad wel of niet door $i+1$ te laten lopen. Het minimale pad van a naar b is dan *of* het minimale pad van a naar b dat alleen loopt langs knopen uit $\{0,\ldots,i\}$ *of* het minimale pad dat loopt van a naar $i+1$ (uiteraard alleen via knopen uit $\{0,\ldots,i\}$) geplakt voor het kortste pad dat loopt van knoop $i+1$ naar b. Als i gelijk aan n is geworden, dan hebben we het minimale pad dat kan lopen langs alle knopen van de graaf, dus het minimale pad.

Voorbeeld 2.3.4: We beschrijven een graaf op 5 punten door een matrix waarin de gewichten van de kanten staan. De graaf die we hier gebruiken heeft relatief veel kanten zodat een tekening van de graaf wat onoverzichtelijk wordt. Die tekening is voor de illustratie van de algoritme ook niet zo belangrijk.

$$\begin{pmatrix} \infty & 6 & 3 & 8 & 4 \\ 7 & \infty & 2 & 2 & 9 \\ 7 & \infty & \infty & 4 & 8 \\ 7 & 8 & 3 & \infty & 3 \\ 9 & 8 & \infty & 1 & \infty \end{pmatrix}$$

De algoritme heeft 5 slagen waarin nieuwe knopen in het kortste pad worden toegelaten. Dat gaat als volgt. Telkens wordt een nieuwe knoop in het pad toegelaten. De corresponderende kolom en rij worden cursief afgedrukt. Als het pad *via* de nieuwe knoop korter is dan wat we al hadden (het cursieve getal in dezelfde rij opgeteld bij het cursieve getal in dezelfde kolom), vervangen we de waarde van het kortste pad door de nieuwe waarde. Aangezien de lengte van het pad naar de nieuwe knoop niet kan veranderen door toevoeging van de nieuwe knoop blijven de cursieve rij en de cursieve kolom dezelfde getallen houden. De knopen waarvoor de afstand een nieuwe waarde krijgt, drukken we vet af.

$$\begin{pmatrix} \infty & 6 & 3 & 8 & 4 \\ 7 & \mathbf{13} & 2 & 2 & 9 \\ 7 & \mathbf{13} & \mathbf{10} & 4 & 8 \\ 7 & 8 & 3 & \mathbf{15} & 3 \\ 9 & 8 & \mathbf{12} & 1 & \mathbf{13} \end{pmatrix} \quad \begin{pmatrix} \mathbf{13} & 6 & 3 & 8 & 4 \\ 7 & 13 & 2 & 2 & 9 \\ 7 & 13 & 10 & 4 & 8 \\ 7 & 8 & 3 & \mathbf{10} & 3 \\ 9 & 8 & \mathbf{10} & 1 & 13 \end{pmatrix} \quad \begin{pmatrix} \mathbf{10} & 6 & 3 & \mathbf{7} & 4 \\ 7 & 13 & 2 & 2 & 9 \\ 7 & 13 & 10 & 4 & 8 \\ 7 & 8 & 3 & \mathbf{7} & 3 \\ 9 & 8 & 10 & 1 & 13 \end{pmatrix}$$

$$\begin{pmatrix} 10 & 6 & 3 & 7 & 4 \\ 7 & \mathbf{10} & 2 & 2 & \mathbf{5} \\ 7 & \mathbf{12} & \mathbf{7} & 4 & \mathbf{7} \\ 7 & 8 & 3 & 7 & 3 \\ \mathbf{8} & 8 & \mathbf{4} & 1 & 4 \end{pmatrix} \quad \begin{pmatrix} 10 & 6 & 3 & \mathbf{5} & 4 \\ 7 & 10 & 2 & 2 & 5 \\ 7 & 12 & 7 & 4 & 7 \\ 7 & 8 & 3 & \mathbf{4} & 3 \\ 8 & 8 & 4 & 1 & 4 \end{pmatrix}$$

□

We voeren in deze algoritme dus n slagen uit waarin we de informatie in de matrix over minimale verbindingen updaten. Elke update kost $O(n^2)$ stappen. Per element doen we slechts twee vergelijkingen. De algoritme kost in totaal dus $O(n^3)$.

2.3.4 Matrixvermenigvuldiging

Voor het vermenigvuldigen van twee matrices hebben we een efficiente algoritme gezien in 2.2.1. We beschouwen hier het probleem van het vermenigvuldigen van een rij matrices, die niet allemaal vierkant zijn. Twee matrices A en B kunnen met elkaar vermenigvuldigd worden als het aantal *kolommen* van A overeenkomt met het aantal *rijen* van B. Immers op plaats i, j in de productmatrix komt het inproduct van de i de rij van A en de j-de kolom van B te staan. De i-de rij van A moet voor die berekening dus net zoveel elementen hebben als de j-de kolom van B, en het aantal elementen van de i-de rij van A is gelijk aan het aantal kolommen van A. Evenzo is het aantal elementen in de j-de kolom van B gelijk aan het aantal rijen van B. Laat A een $n \times m$ matrix zijn en B een $m \times p$ matrix zijn. Het resultaat is dan een matrix C met n rijen en p kolommen. Het aantal vermenigvuldigingen dat hiervoor nodig is, is met de standaard matrixvermenigvuldigingsalgoritme $n \times m \times p$.

Stel nu eens dat we een drietal matrices moeten vermenigvuldigen $A \times B \times C$, waarbij A een 30×40 matrix is, B een 40×50 en C een 50×60 matrix. We kunnen dit, aangezien matrixvermenigvuldiging associatief is op twee manieren doen $(AB)C$ of $A(BC)$. In het eerste geval voeren we eerst $30 \times 40 \times 50$ vermenigvuldigingen uit om een 30×50 matrix te krijgen. Vervolgens vermenigvuldigen we deze 30×50 matrix met C ten koste van $30 \times 50 \times 60$ vermenigvuldigingen. In het tweede geval vermenigvuldigen we eerst B en C ten koste van $40 \times 50 \times 60$ vermenigvuldigingen en de resulterende 40×60 matrix vermenigvuldigen we met A ten koste van $30 \times 40 \times 60$ vermenigvuldigingen. Beide operaties leiden tot een 30×60 productmatrix, de ene keer ten koste van $60000 + 90000 = 150000$ vermenigvuldigingen en de andere keer ten koste van $120000 + 72000 = 184000$ vermenigvuldiginen. De volgorde van vermenigvuldigen is hier van belang.

Stel dat we een rij matrices $A_1, \ldots, A_n$ hebben die we met elkaar willen vermenigvuldigen, waarbij gegeven is dat als A_i als A_i een $d_{i-1} \times d_i$ matrix is, dat dan A_{i+1} een $d_i \times d_{i+1}$ matrix is, m.a.w. vermenigvuldiging is mogelijk. Hoe zullen wij de volgorde van vermenigvuldigen (i.e. het plaatsen van de haakjes) bepalen? Doorwerken van enige voorbeelden (zie Som 2.3.5) laat zien dat gulzige methoden niet werken. De oplossing voor dit probleem bereiken we met de dynamisch programmeren aanpak. Immers, stel dat we optimaal uit de matrices d_i tot en met d_k een productmatrix met d_{i-1} rijen en d_k kolommen willen maken. Het buitenste paar haakjes moet ergens tussen i en k gezet worden, zeg op plaats j. Het aantal vermenigvuldigingen is nu gelijk aan $d_{i-1} \times d_j \times d_k$ *plus* het aantal vermenigvuldigingen dat nodig is om optimaal de rij matrices van i tot j te vermenigvuldigen *plus* het aantal vermenigvuldigingen dat nodig is om optimaal de rij matrices van j tot k te vermenigvuldigen. Door deze getallen uit te rekenen voor j lopend van i naar k kunnen we achter het optimale getal komen, waarbij het optimale aantal vermenigvuldigingen om twee *opeenvolgende* matrices te vermenigvuldigen vast ligt. Dit nodigt uit tot een recursieve algoritme, ware het niet dat de recursieve boom die aldus onstaat alle mogelijke plaatsing van haakjesparen in een knoop representeert. Het

aantal mogelijkheden dat dit oplevert is een bekend getal, het zogenaamde *Catalaanse getal* en is ongeveer $4^n/n^{3/2}$. Dit levert dus geen efficiente algoritme op. Als we echter voor alle paren j tussen i en k het optimale getal kennen, dan kunnen we met bovenbeschreven methode in ongeveer $k-i$ stappen uitrekenen wat het minimum is voor i tot k. Deze methode nodigt dus ook uit tot de dynamisch programmeren aanpak, en een handige manier om deze te implementeren levert een $O(n^2)$ algoritme als volgt.

Neem een vierkante matrix P van $n \times n$ getallen. Omdat we weten hoeveel vermenigvuldigingen optimaal zijn om de matrices op plaats i en $i+1$ met elkaar te vermenigvuldigen ($d_{i-1} \times d_i \times d_{i+1}$) kunnen we al deze getallen op plaats $i, i+1$ in de matrix invullen (formeel kan ook de diagnonaal gevuld worden met 0). Nu vullen we de rest van de matrix met de formule

$$P[i,j] = \min\{P[i,k] + P[k+1,j] + d_{i-1} \times d_k \times d_j : i < k < j\}.$$

Omdat we de waarden $P[i, i+1]$ in deze matrix weten, kunnen we hem in $O(n^2)$ stappen (waarbij elke stap niet meer dan het vinden van het minimum in een rij van lengte n kost) verder invullen. De complexiteit van de algoritme is dus $O(n^3)$.

Voorbeeld 2.3.5: Laat gegeven een zijn een rijtje matrices met de volgende dimensies

$$18 \times 31, 31 \times 40, 40 \times 29, 29 \times 9, 9 \times 11, 11 \times 8, 8 \times 20, 20 \times 44, 44 \times 31$$

Uit de bovenbeschreven algoritme komt de volgende matrix.

$$\begin{pmatrix}
0 & 22320 & 43200 & 26622 & 28404 & 26544 & 29424 & 39920 & 48960 \\
 & 0 & 35960 & 21600 & 24669 & 22080 & 27040 & 40032 & 47720 \\
 & & 0 & 10440 & 14400 & 12160 & 18560 & 33280 & 40032 \\
 & & & 0 & 2871 & 2880 & 7452 & 20128 & 28024 \\
 & & & & 0 & 792 & 2232 & 10152 & 20976 \\
 & & & & & 0 & 1760 & 10912 & 20680 \\
 & & & & & & 0 & 7040 & 17952 \\
 & & & & & & & 0 & 27280 \\
 & & & & & & & & 0
\end{pmatrix}$$

Waaruit we kunnen zien dat de meest gunstige manier om deze matrices te vermenigvuldigen 48960 vermenigvuldigingen kost. Om de volgorde te bepalen waarmee dat lukt is het nodig tussenresultaten op te slaan. In ons geval is de optimale volgorde:

$$(1 \times (2 \times (3 \times 4 \times (5 \times 6)))) \times ((7 \times 8) \times 9)$$

Naast de aantallen vermenigvuldigingen, moeten we daarvoor ook bijhouden *welke* matrices met elkaar vermenigvuldigd worden. In de volgende matrix staan op plaats (i,j) telkens het hoogste niveau haakjes. Een $\times$ betekent dat twee naast elkaar gelegen matrices rechtstreeks vermenigvuldigd moet worden en $i \rightharpoonup j$ betekent dat op plaats i,j gekeken moet worden hoe deze rij optimaal vermenigvuldigd kan worden.

$$\begin{pmatrix}
1 \times 2 & (1 \times 2)3 & 1(2 \rightharpoonup 4) & (1 \rightharpoonup 4)5 & 1(2 \rightharpoonup 6) & (1 \rightharpoonup 6)7 & (1 \rightharpoonup 6) \times (7 \times 8) & (1 \rightharpoonup 6) \times (7 \rightharpoonup 9) \\
 & 2 \times 3 & 2(3 \times 4) & (2 \rightharpoonup 4)5 & 2(3 \rightharpoonup 6) & (2 \rightharpoonup 6)7 & (2 \rightharpoonup 6) \times (7 \times 8) & (2 \rightharpoonup 6) \times (7 \rightharpoonup 9) \\
 & & 3 \times 4 & (3 \times 4)5 & 3(4 \rightharpoonup 6) & (3 \rightharpoonup 6)7 & (3 \rightharpoonup 6) \times (7 \times 8) & (3 \rightharpoonup 6) \times (7 \rightharpoonup 9) \\
 & & & 4 \times 5 & 4(5 \times 6) & 4(5 \rightharpoonup 7) & (4 \rightharpoonup 6) \times (7 \times 8) & (4 \rightharpoonup 6) \times (7 \rightharpoonup 9) \\
 & & & & 5 \times 6 & (5 \times 6)7 & (5 \rightharpoonup 7)8 & (5 \times 6) \times (7 \rightharpoonup 9) \\
 & & & & & 6 \times 7 & 6(7 \times 8) & 6(7 \rightharpoonup 9) \\
 & & & & & & 7 \times 8 & (7 \times 8)9 \\
 & & & & & & & 8 \times 9
\end{pmatrix}$$

□

2.3.5 Sommen

1. Laat zien met behulp van voorbeelden dat gulzige algoritmen voor matrix-ketting-vermenigvuldiging gebaseerd op

   (a) Neem de kleinste producten (minimaliseer $d_{i-1} \times d_i \times d_{i+1}$ eerst).

   (b) Neem de grootste producten (maximaliseer $d_{i-1} \times d_i \times d_{i+1}$ eerst).

   niet werken. Kun je nog andere gulzige heuristieken bedenken?

2. Vind dynamisch programmeren oplossing voor de volgende problemen. Het volstaat de recurrentie te vinden.

   (a) Een gokker begint met 3 euro en wil in drie keer gokken vijf euro hebben. Bij elke gok is de kans op winst 2/3 en bij winst wordt twee keer de inzet uitbetaald (bij verlies is de inzet verloren). Bepaal met een een dynamisch programma de strategie (hoeveel moet zij inzetten) om de kans te maximaliseren.

   (b) Langste stijgende subrij. Gegeven een rij getallen, vind de langste deelrij waarvan de elementen monotoon stijgend zijn (ze hoeven niet naast elkaar te staan). Dus bijvoorbeeld van $S = (9, 5, 2, 8, 7, 3, 1, 6, 4)$ heeft de langste stijgende deelrij lengte drie en is $(2, 3, 4)$ of $(2, 3, 6)$.

   (c) Maximale som. Vind in een rij getallen de deelrij van naast elkaar liggende getallen met maximale som (bijvoorbeeld in $\{31, -41, 59, 26, -53, 58, 97, -123, 93, 84\}$ is dat de rij van 59 t.e.m. 97).

   (d) Kleuren. n auto's hebben c kleuren en kunnen k plaatsen versleept worden. Zoek de rij met het minimaal aantal kleurwisselingen. Bijvoorbeeld: 011200121 kan met $k = 2$ worden veranderd in 110002211.

   (e) Play Offs. Team A en team B spelen een serie van niet meer dan $2n - 1$ wedstrijden tegen elkaar. Een team heeft de serie gewonnen zodra het n of meer wedstrijden gewonnen heeft. Er bestaat geen gelijkspel en de wedstrijden zijn onafhankelijk van elkaar. Voor elke wedstrijd is er een constante waarschijnlijkheid p dat team A wint (en dus een constante waarschijnlijkheid $1 - p$ dat team B wint). Bereken de kans dat team A wint.

   (f) Shuffle. Stel je krijgt drie rijtjes letters:$X = x_1, x_2, \ldots, x_m$, $Y = y_1 y_2, \ldots, y_n$ en $Z = z_1, z_2, \ldots, z_{m+n}$. Z is een *shuffle* van X en Y als Z gemaakt kan worden door om beurten een stukje van X en een stukje van Y te nemen, waarbij de letters in volgorde blijven staan. Bijvoorbeeld, *cchocohilaptes* is een shuffle van *chocolate* em en *chips*, maar *cchocolhilaptes* is dat niet. Gegeven X, Y, Z, m en n. Is Z wel of niet een shuffle van X en Y?

Programmeren

1. Maak een ongerichte volledige graaf op 25 knopen en geef de kanten random waarden tussen de 1 en de 100. Bereken tussen elk tweetal punten de lengte van het kortste pad. Doe vervolgens hetzelfde met de algoritme van Floyd-Warshall en vergelijk de uitkomsten.

Hoofdstuk 3

Grafenalgoritmen

Een belangrijk onderdeel van de algoritmiek wordt gevormd door de specifieke toepassing van algoritmen in de grafentheorie. Een aantal typische graafalgoritmen hebben we al besproken in Hoofdstuk 2 omdat ze de in die sectie besproken techniek uitstekend toelichten. Een aantal typische grafenproblemen zullen nog uitgebreid aan bod komen in 6.5 vanwege het feit dat er geen efficiente algoritme voor het algemene probleem bekend is. Voor een aantal eenvoudig te beschrijven en veelvoorkomende grafenproblemen weten we geen andere oplossing te bedenken dan alle mogelijke oplossingen te genereren en daaruit een passende te kiezen. Omdat het totaal aantal mogelijke oplossingen vaak exponentieel is, zeggen we dat zo'n grafenprobleem ondoenlijk is en zoeken we naar inperkingen van de soort grafen waarop het aantal mogelijke oplossingen beperkt is, of waarop de algoritme verder kan worden gespecialiseerd, zodat niet alle oplossingen bekeken hoeven worden. Te denken valt hierbij aan grafen met beperkte graad, beperkte diepte (padlengte), beperkte boombreedte of andere speciale eigenschappen zoals planariteit. Het onderzoek naar grafenalgoritmen beweegt zich voor een belangrijk deel op dit gebied. In dit hoofdstuk zullen we nog een paar algemene methoden bespreken voor het doorzoeken en klassificeren van grafen. We beginnen met doorzoeken.

3.1 Zoeken in Grafen

Een graaf bestaat uit knopen en kanten en de structuur van een graaf kunnen we onderzoeken door van de ene knoop naar de andere knoop te lopen met behulp van een algoritme. Aangezien een knoop doorgaans door meerdere kanten aan meerdere andere knopen verbonden is, moeten we in elke knoop een keuze maken welke knoop de volgende in de rij is (of meer precies welke knoop na de volgende knoop in de rij zal volgen). Er zijn twee hoofdkeuzen die de soort zoekalgoritme bepalen, de overige methoden zijn variaties daarop.

3.1.1 Depth First Search

In de diepte eerst methode, wordt het volgende te bezoeken punt gekozen door eerst zo diep mogelijk in de graaf te gaan en pas daarna nog onbezochte knopen die buren zijn van eerder bezochte knopen te onderzoeken. We nemen aan dat de knopen van de graaf genummerd zijn, zodat we in een knoop als volgende knoop de knoop met het laagste nummer kunnen kiezen. Ook nemen we aan dat we een merkteken "bezocht" op een knoop kunnen aanbrengen.

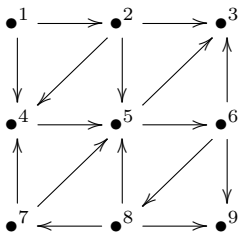
1: Procedure DFS(v: knoop)
2: Markeer v bezocht.
3: **while** $\exists w$ buurknoop van v die nog niet bezocht is **do**
4: Neem zo'n w met minimale index;
5: DFS(w);
6: **end while**

De procedure voor diep zoeken eerst, Depth First Search (DFS), is efficiënt en bijzonder eenvoudig op te schrijven. Als we aannemen dat het aflopen van een kant in constante tijd kan, dan neemt DFS niet meer tijd dan er knopen zijn, en is dus $O(n)$. Zoiets zou bijvoorbeeld kunnen wanneer de graaf gegeven is als een adjacency matrix en we dus directe toegang tot de kanten hebben. Andere implementaties vereisen misschien eerst sorteren van de knopen ($O(n \log n)$), of van de kanten ($O(m \log m) \subseteq O(n^2 \log n)$). Hoe ook, de procedure blijft buitengewoon efficient. Met behulp van deze zoekmethoden kunnen we diverse eigenschappen van een graaf identificeren.

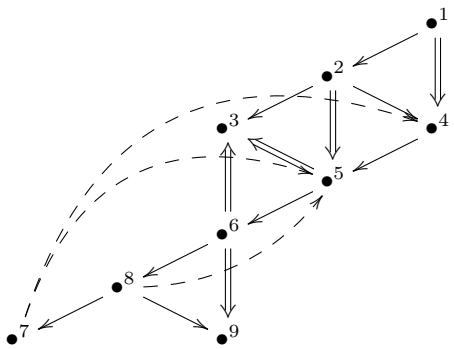
De procedure DFS selecteert een aantal kanten in de graaf en stelt de graaf voor als een boom. Doorgaans heeft de graaf natuurlijk nog veel meer kanten. De kanten die door DFS gebruikt worden om de graaf te doorlopen zullen we *boomkanten* (tree-edges) noemen. De kanten in de graaf die van een knoop v terugwijzen naar een knoop w die een voorouder is van v in de DFS boom zullen we *terugkanten* (back-edges) noemen en de kanten die niet in deze categorie vallen (dus verschillende paden van de boom met elkaar verbinden) zullen we *kruiskanten* noemen. In het voorbeeldje hieronder zullen we de boomkanten van de graaf aangeven met pijlen, de terugkanten met gestreepte pijlen en de kruiskanten met $\Rightarrow$.

Voorbeeld 3.1.1: Bekijk de graaf G op 9 knopen met 16 kanten. Deze graaf wordt door DFS in de figuur DFS(G) als boom beschreven.

G



DFS(G)



□

Cykels

Met DFS kan worden bepaald of in een graaf cykels zitten. Immers, als er een cykel in een graaf zit, dan komen we met DFS langs een pad aan op een knoop die reeds gemarkeerd is, en anders komen we alleen terug uit de recursie bij reeds gemarkeerde knopen. Een enkele DFS run kan beslissen of er een cykel is die door een gegeven knoop voert. Om te beslissen of er een cykel in de graaf is, kunnen we DFS achtereenvolgens uit alle knopen uit de graaf starten.

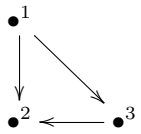
Topological Sort

Een acyclische graaf kan de voorstelling zijn van een partiële ordening [1] . Soms kan het nodig zijn van een partiële ordening een totale ordening te maken. Te denken bijvoorbeeld valt aan machinevolgordeproblemen waarbij afhankelijkheden bestaan tussen taken uit te voeren door een machine. Deze afhankelijkheden definiëren een partiële ordening die platgeslagen dient te worden tot een lineaire ordening om op een machine te kunnen worden uitgevoerd. Ook zien we het topological sort probleem vaak optreden bij het gebruik van databases, waarbij transacties die gebruik maken van dezelfde resources op elkaar moeten wachten. Het

[1] Een ordening op een verzameling kan partiëel of totaal zijn. Als een ordening, $<$, totaal is, dan moet voor elk paar elementen a en b gelden $a < b$ of $b < a$. Bij een partiële ordening hoeft dat niet. Een acyclische graaf kan een partiële ordening voorstellen omdat niet tussen elk tweetal knopen een pad hoeft te zijn. Als er een cykel in de graaf zit, dan kan *zowel* $a < b$ als $b < a$ gelden. De ordening is dan niet meer *antisymmetrisch*.

probleem is dan het vinden van een volgorde waarin de transacties kunnen worden uitgevoerd, of waarin de transacties zouden kunnen zijn uitgevoerd als ze door een machine zouden zijn behandeld (serializability).

We zouden kunnen proberen voor dit probleem DFS in te zetten, maar DFS respecteert niet noodzakelijk alle afhankelijkheden in de graaf. Een graafje als bijvoorbeeld



wordt door DFS in de volgorde 1, 2, 3 doorlopen, maar de afhankelijkheid $2 \leftarrow 3$ maakt dat deze volgorde niet legitiem is voor topological sort.

We hebben dus een andere algoritme nodig voor het vinden van deze ordening. De observatie die we maken is dat er in een acyclische graaf altijd minstens één knoop moet zijn met ingraad 0. Begin in een willekeurige knoop, kies een willekeurige inkomende kant en ga naar de knoop die aan de andere kant zit. Aangezien de graaf acyclisch is, moet deze actie altijd een nieuwe knoop opleveren, en dat kan niet meer dan n keer. De knoop met ingraad 0 hoeft niet later in de ordening te komen dan enige andere knoop in de graaf en kan dus de eerste in de ordening zijn. Deze knoop geven we dus nummer 1 in de volgorde, we verwijderen de knoop en voeren de actie opnieuw voor de resterende knopen (die ook een acyclische graaf vormen) uit.

```
1: Procedure TopSort(G,n)
2: if G ≠ ∅ then
3:     Find v with indegree 0;
4:     Give v number n;
5:     TopSort(G − {v}, n + 1)
6: end if
```

Verbonden Componenten

Een verbonden component in een graaf is een deel van de graaf met de eigenschap dat tussen elk tweetal punten tenminste één pad loopt. Zowel gerichte als ongerichte grafen hebben verbonden componenten, maar in ongerichte grafen loopt tussen elk tweetal punten natuurlijk zowel een pad heen als een pad terug. Dat is in gerichte grafen niet altijd het geval. Als in een deel van een gerichte graaf tussen elk tweetal punten zowel een pad heen als een pad terug loopt, noemen we zo'n deel dubbelverbonden of sterkverbonden (bi-connected of strongly connected). Depth First Search is natuurlijk een uitgelezen algoritme om verbonden componenten te vinden. Immers, als we een Depth First Search in een punt starten, dan vinden we alle punten in de graaf die door een pad met dat punt verbonden zijn, ofwel we vinden alle punten in de graaf die in dezelfde verbonden component zitten als dit punt. DFS kan echter ook gebruikt worden om de dubbelverbonden componenten van een gerichte graaf te ontdekken.

Articulation Points

Een articulation point in een graaf is een punt dat de graaf in twee delen verdeelt, zo dat elk pad van het ene deel van de graaf naar het andere deel van de graaf door dit punt moet gaan. Articulation points in netwerken zijn vaak de kwetsbare punten. Neem je de articulation points uit een verbonden graaf weg, dan valt de graaf uiteen in een aantal verbonden componenten. In dataverkeer tussen computers willen we niet graag articulation points hebben omdat uitval van dit punt het einde van de communicatie betekent. Als articulation points om welke reden dan ook onvermijdelijk zijn, dan willen we er graag zo weinig mogelijk hebben.

In 3.1.1 hebben we al gezien hoe we met behulp van DFS verbonden componenten konden identificeren. Net zo makkelijk kunnen we natuurlijk articulation points met behulp van DFS identificeren. Bijvoorbeeld: neem zo'n punt uit de graaf. Twee buren van dit punt komen dan in verschillende verbonden componenten te zitten. Aan de volledige DFS boom van een verbonden component kunnen we echter de articulation points nog gemakkelijker herkennen. Bekijk een DFS boom en neem een punt dat niet de wortel van de boom is.

Als dit punt een articulation point is, dan moet de graaf na het wegnemen van dit punt uit tenminste twee componenten bestaan die niet met elkaar verbonden zijn. Alle punten in de graaf die niet in de subboom van dit punt zitten, kunnen vanuit de wortel bereikt worden. De graaf kan na weglating van dit punt dus alleen maar uit twee niet met elkaar verbonden componenten bestaan, als de subboom van dit punt geen andere verbinding (terugkanten of kruiskanten) heeft met de rest van de graaf. Dit kunnen we met een recursieve procedure nagaan.

3.1.2 Breadth First Search

De alternatieve manier om recursief een graaf te doorzoeken is de "breedte eerst" methode. In plaats van eerst zo diep mogelijk in de graaf te gaan, bezoeken we eerst de hele buurt van de knoop waar we zijn, om daarna alle knopen die van deze buren weer buren zijn te bezoeken. Depth First Search kan worden gekarakterizeerd door een stapel (de eerste buur van een knoop wordt ook het eerstvolgend bezocht, waarna diens buren op de stapel gezet worden enzovoort) en kan dus met een recursieve procedure beschreven worden. Breedte eerst heeft veel meer het karakter van een rij. Zo zullen we ook deze methode in pseudocode beschrijven.

```
1: Procedure BFS(Q);
2: if Q ≠ ∅ then
3:     remove first element v from Q;
4:     mark v visited;
5:     enqueue all neighbors of v in Q;
6:     BFS(Q);
7: end if
```

3.2 Kortste paden 3: De A^* algoritme

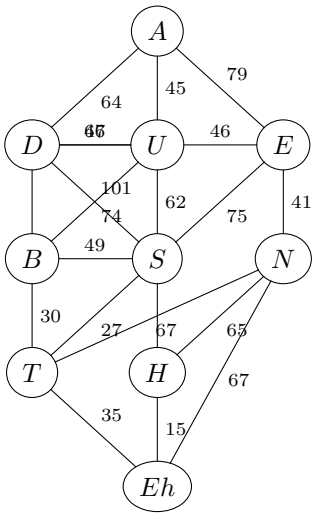
In de AI, met name de robotica, is een algoritme voor het zoeken van paden in grafen, waarbij obstakels vermeden dienen te worden zeer populair. Deze algoritme staat tegenwoordig alom bekend als de A^* algoritme. Het is een variant op Dijkstra's algoritme die in de praktijk vaak aanzienlijk sneller is, maar in theorie zelfs niet hoeft te convergeren. Daarom is deze algoritme in het algoritmenonderzoek veel minder bekend dan in de AI. De algoritme heeft een beetje een merkwaardige naam. In het oorspronkelijke artikel [HNR68] werd deze algoritme aangeduid met "Algoritme A". De algoritme is gebaseerd op een techniek die in de AI "heuristiek"wordt genoemd (zie ook 6.7.3) en als een optimale heuristiek in de AI is gevonden wordt dat aangeduid met een $*$, vandaar de A^* algoritme.

In tegenstelling tot de algoritme van Dijkstra is de A^* algoritme er niet per se op gericht het kortste pad van A naar B te vinden. Vaak is men al dik tevreden als er *een* pad gevonden is. Als de gebruikte heuristiek bekend is, dan is het vaak ook mogelijk de gebruikte variant van de algoritme te verleiden een pad langs *alle* mogelijke locaties te produceren.

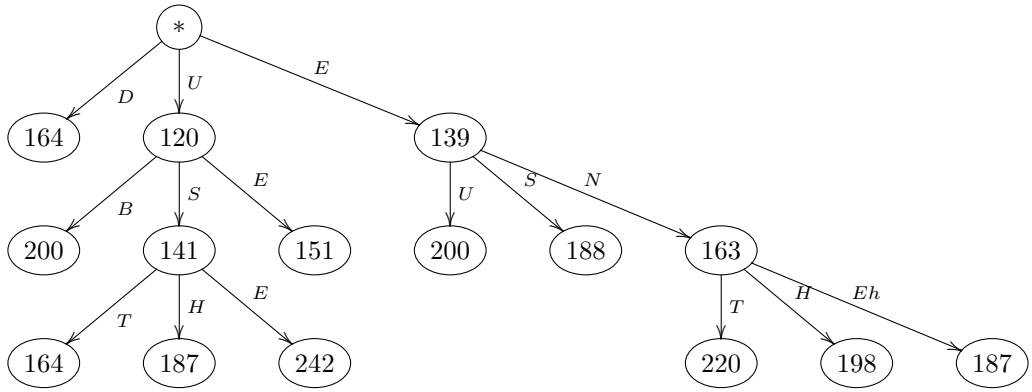
In overeenstemming met de algoritme van Dijkstra houdt de A^* algoritme een verzameling punten bij waarvan het de afstand tot het startpunt A "kent", dwz een pad van A naar dit punt gevonden heeft. De afstand van A naar een punt P wordt aangeduid met $g(P)$. Verder gebruikt de A^* algoritme een heuristiek h die gegeven een punt P in de graaf een schatting geeft (vaak een onderschatting) van de afstand van dat punt naar het doelpunt B, $h(P)$. De "verwachte" afstand van A tot B is dan de afstand van A naar P plus de afstand van P naar B, $f_P(B) = g(P) + h(P)$. De algoritme maakt dan steeds een update door voor de knoop mpet minimale $f_P(B)$, de buren aan de bekende verzameling toe te voegen, net zo lang tot B aan de verzameling is toegevoegd.

Voorbeeld 3.2.1: We berekenen de afstand van Amsterdam naar Eindhoven over verschillende tussenstations. Als heuristiek gebruiken we de afstand in centimeters tussen een gegeven stad en Eindhoven, vermenigvuldigd met de schaal van de kaart. De tussenstations die we meenemen zijn: Delft, Utrecht, Ede, Breda, 's-Hertogenbosch, Nijmegen, Tilburg, en Helmond. De werkelijke afstanden tussen deze steden over de weg zijn gegeven in de volgende graaf (afgerond overgenomen van een internet routeplanner). De af-

standen die we (hemelsbreed) van de kaart kunnen aflezen van de buren van Amsterdam naar eindhoven, vermelden we in een tabel ernaast. Deze worden gebruikt als schatting voor de resterende afstand.



stad	afstand
Delft	100
Utrecht	75
Ede	60
Breda	54
's-Hertogenbosch	34
Nijmegen	43
Tilburg	30
Helmond	13



Op dit punt is een weg naar Eindhoven gevonden over Ede en Nijmegen met een lengte van 187km. De algoritme kan nu stoppen, of proberen een kortere weg te vinden, want zowel de weg over Delft, als de weg over Utrecht, 's-Hertogenbosch en Tilburg zouden nog korter kunnen zijn.

□

3.3 Netwerken en Stromen

Met grafen kunnen we van alles modelleren. Vaak worden grafen gebruikt om relaties tussen individuele objecten te modelleren. We zien dan vaak ongerichte grafen. Vaak ook worden geordende transitieve relaties gemodelleerd. Paden in de graaf worden dan belangrijker dan individuele kanten. We beperken de klasse van grafen onder beschouwing dan vaak tot gerichte grafen (en zouden eigenlijk „bogen" moeten zeggen in plaats van kanten, maar waar dit niet tot verwarring leidt blijven we toch ook graag in het gerichte geval het

woord „kant" gebruiken). De paden in een gerichte graaf modelleren vaak een soort infrastructuur. Zulke grafen zijn het onderwerp van deze sectie.

Een voorbeeld van het gebruik van de paden in grafen is het probleem van de netwerkstromen. We hebben gegeven een netwerk dat bestaat uit een graaf en twee functies, de capaciteitsfunctie c en de stroomfunctie f. We zullen van de graaf aannemen dat hij gericht is. Dat is geen beperking van de algemeenheid, want een weg die in twee richtingen kan worden genomen, kan altijd worden voorgesteld door een tweetal kanten. Beide functies f en c zijn functies van de kanten naar de reële getallen die aan voorwaarden verbonden zijn.

1. f en c zijn allebei groter dan of gelijk aan 0 voor elke kant.
2. Voor elke v geldt $\sum_{e \to v} f(e) = \sum_{e \leftarrow v} f(e)$. Behalve voor twee specifieke knopen s en t die we de *bron* en het *doel* zullen noemen. Dat wil zeggen voor alle knopen behalve de bron en het doel is de totale inkomende stroom gelijk aan de totale uitgaande stroom.
3. Voor elke kant e geldt $f(e) \leq c(e)$.

De capaciteitsfunctie is constant voor het probleem en we definieren verschillende stroomfuncties. Het probleem is het vinden van de stroomfunctie die onder de hierbovenbeschreven voorwaarden de maximale inkomende stroom in de knoop t geeft. Deze stroom is vanwege voorwaarde 2 gelijk aan de totale uitgaande stroom uit s, en dit getal zullen we (als die bestaat natuurlijk) "de" maximale stroom in het netwerk noemen. We zullen een algoritme behandelen die de maximale stroom vindt. Eerst zullen we bewijzen dat zo'n maximale stroom bestaat.

3.3.1 Min Cut Max Flow

Eerst definieren we een *doorsnijding van het netwerk* als een graaf waaruit zoveel kanten zijn verwijderd dat precies twee verbonden componenten ontstaan. De ene component bevat s, en de andere component bevat t. Ofwel als $G = (V, E)$, dan is de doorsnijding van de graaf de definitie van twee verzamelingen V_1, V_2 zodat $V_1 \cup V_2 = V$, $V_1 \cap V_2 = \emptyset$, $s \in V_1$, en $t \in V_2$. De capaciteit van een doorsnijding is de som van de capaciteiten van de kanten die van de component van s naar de component van t lopen, ofwel $\sum\{c(e) : e = (v, w) \wedge v \in V_1, w \in V_2\}$.

Stelling 3.3.1 (min-cut-max-flow) *De maximale stroom in een netwerk is gelijk aan het minimum genomen over de capaciteiten van alle doorsnijdingen.*

$f \leq$ **min-cut.** Allereerst is het duidelijk dat voor elke doorsnijding geldt dat de stroom in het netwerk kleiner moet zijn dan de capaciteit van die doorsnijding. Aangezien er een eindig aantal doorsnijdingen is, moet dus zeker gelden dat de maximale stroom *kleiner* dan of gelijk is aan de minimale capaciteit van een doorsnijding.

$f \geq$ **min-cut.** Om in te zien dat in elk netwerk ook een stroom kan lopen waarvan de waarde (= de totale stroom uit s = de totale stroom in t) gelijk is aan de capaciteit van een doorsnijding, en vanwege de vorige alinea de minimale, geven we een constructief bewijs. Dit „bewijs" is wat lang en eindigt pas op pagina 64, maar omdat het constructief is, is het nergens echt ingewikkeld.

We definiëren een algoritme die die maximale stroom berekent, en bewijzen dat de algoritme pas stopt als die, dan maximale, stroom bereikt is. Deze algoritme is vernoemd naar Ford en Fulkerson [FF56], die deze algorime voor het eerst presenteerden.

De algoritme begint met een stroom 0 door het netwerk en zoekt dan zogenoemde stroomvergrotende paden. Dat zijn paden van s naar t waardoorheen nog extra stroom "geduwd" kan worden. In deze paden worden kanten in de richting van s naar t opgenomen, maar *ook* kanten in de tegenovergestelde richting. De kanten in de richting van s naar t kunnen in zo'n pad worden opgenomen als de stroom in de graaf door die kant zijn maximum (capaciteit) nog niet heeft bereikt. Extra stroom door die kant voeren zou dan de stroom langs het pad vergroten, en stroom langs een kant in de tegenovergestelde richting kan vergroot

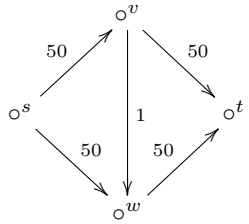
worden, als de stroom niet gelijk aan nul is. Door die stroom te verkleinen, vergroten we de stroom langs het pad. Hebben we zo'n stroomvergrotend pad gevonden, dan hebben we voor elke kant apart uitgerekend hoeveel extra stroom er nog bij zou kunnen langs het pad, en dan kunnen we door het minimum te nemen de stroom langs het pad door alle kanten berekenen. Het vinden van een stroomvergrotend pad gaat vanuit s met DFS naar t. Als t niet bereikt wordt, dan is de stroom in het netwerk maximaal. We zullen een verzameling van gemarkeerde knopen bijhouden M, met in het begin alleen $s \in M$ We zullen voor kanten in de richting van het pad het verschil tussen de stroom en de capaciteit, en voor kanten tegen de richting van het pad in de stroom zelf de *restcapaciteit* van zo'n kant noemen.

```
1: procedure FFK(node v, real minval)
2:  Add v to M;
3: if v = t then
4:     print(t,minval);
5:     return(true);
6: end if
7: for (∀w ∈ V − M)[[(v, w) ∈ E ∧ f((v, w)) < c((v, w))] ∨ [(w, v) ∈ E ∧ f((w, v)) > 0]] do
8:     Let p be the residual capacity of this edge.
9:     Let q be min{p, minval};
10:    if FFK(w, q) then
11:        print(w);
12:        return(true);
13:    end if
14: end for
15: return(false);
```

Deze procedure zoekt met DFS een pad van s naar t. Als een pad in t eindigt, dan wordt t samen met de minimale vergrotingswaarde langs het gevonden pad afgedrukt, bovendien worden omdat de recursieve aanroepen achtereenvolgens allemaal true terug geven het pad waarlangs de stroom kan worden vergroot in omgekeerde volgorde afgedrukt. Eindigt een pad in een knoop omdat geen vervolgknoop wordt gevonden, dan zal deze knoop geen recursieve aanroep hebben die true teruggeeft en dus niet worden afgedrukt.

Elke keer dat een stroomvergrotend pad gevonden wordt, wordt de stroom met de minimale restcapaciteit langs dat pad vergroot. Bovendien weten we dat de stroom niet de capaciteit van enige doorsnijding kan overstijgen. Betekent dat nu dat de maximale stroom ook zal worden bereikt door deze algoritme? Helaas niet. Er bestaan netwerken waarin de restcapaciteit langs stroomvergrotende paden die door deze algoritme gevonden worden een machtreeks vormen met een factor kleiner dan 1, en waarin de werkelijke maximale stroom nooit bereikt wordt. Wel kunnen we inductief afleiden dat als de capaciteiten in het netwerk gehele getallen zijn, de stroom steeds met een gehele waarde toeneemt. Dat betekent dat het aantal stappen in een netwerk met geheeltallige capaciteiten eindig is. Omdat een getal n in $\log n$ bits gerepresenteerd kan worden geeft deze observatie echter niet een betere dan exponentiële bovengrens voor de complexiteit van het maximaliseren van de stroom in een netwerk. Ook met geheeltallige capaciteiten kan de algoritme van Ford en Fulkerson nog onacceptabel traag kan zijn, getuige het volgende voorbeeld.

Voorbeeld 3.3.1: Beschouw de volgende graaf



Een stroomvergrotend pad is, als de stroom 0 is het pad s,v,w,t. Hierlangs kan de stroom met 1 vergroot worden. Daar*na* is echter het pad s,w,v,t een stroomvergrotend pad geworden. Immers de stroom kan worden vergroot door 1 eenheid langs dit pad te sturen. De stroom van v naar w wordt daardoor weer 0.

De algoritme die stroomvergrotende paden vindt zal dus in dit netwerk 50 stappen kunnen doen voordat de maximale stroom gevonden wordt. Als echter meteen de goede paden gekozen worden, dan zal de algoritme slechts twee keer een stroomvergrotend pad hoeven te selecteren. □

De algoritme geeft wel een bewijs voor het tweede onderdeel van de min-cut-max-flow stelling, namelijk een maximale stroom in een netwerk is minstens gelijk aan de minimale capaciteit van een doorsnijding. Stel namelijk maar dat een maximale stroom bereikt is. Er is kan dan geen stroomvergrotend pad meer worden gevonden. Dat betekent dat de algoritme t en mogelijk een aantal andere knopen niet aan M zal toevoegen in regel 2. Er is dus een doorsnijding te definieren, nl. het paar $(M, V - M)$, zo dat $v \in M$, en $t \in V - M$. Deze doorsnijding heeft één of andere capaciteit. Er loopt geen kant e van M naar $V - M$ waarvoor $f(e) < c(e)$, noch loopt er een kant e' van $V - M$ naar M waarvoor $f(e') > 0$. Als dit wel het geval was, dan was M namelijk minstens één knoop groter geweest. Dat wil zeggen dat de stroom in het netwerk tenminste gelijk is aan de capaciteit van de doorsnijding $(M, V - M)$.

De mogelijke traagheid van de Ford-Fulkerson algoritme ligt in het feit dat we onhandig stroomvergrotende paden kunnen kiezen. Er is geen duidelijke eigenschap van het netwerk dat het mogelijk maakt op een handige manier een stroomvergrotend pad te kiezen. Wel geven netwerk en stroom *samen* een methode die het totaal aantal stroomvergrotende paden dat gevonden moet worden zo klein mogelijk houdt. Netwerk en stroom geven namelijk samen een methode om een zogenoemd *gelaagd netwerk* te definiëren, vanwaaruit een nieuwe (grotere) stroom gevonden kan worden. Deze techniek leidt tot maximalisering van de stroom in een aantal stappen dat begrensd wordt door een functie van het aantal knopen. Een dramatische verbetering van de complexiteit.

3.3.2 Gelaagde Netwerken

Stroomvergrotende paden in een netwerk worden gevonden door paden van s naar t te vinden. In die paden nemen we kanten in de richting van s naar t op als er een stroom doorheen loopt die kleiner is dan de capaciteit. Kanten in de richting van t naar s worden opgenomen waardoorheen een stroom loopt die groter is dan 0. In plaats van elke keer zo'n kant te kiezen en met die kanten een pad te construeren, geven we nu een methode die als het ware alle stroomvergrotende paden *tegelijkertijd* vindt.

Laat een netwerk N bestaande uit een graaf G met knopen s en t, een capaciteitfunctie c en een stroomfunctie f gegeven zijn. We definiëren inductief het *gelaagde netwerk* $L(N)$ als volgt.

Laag 0 van $L(N)$ bestaat uit de verzameling $\{s\}$. Verder bestaat de $i + 1$ste laag van $L(N)$ uit alle knopen w die *nog niet zijn opgenomen in een eerdere laag* waarvoor er een knoop v in de i de laag zit met.

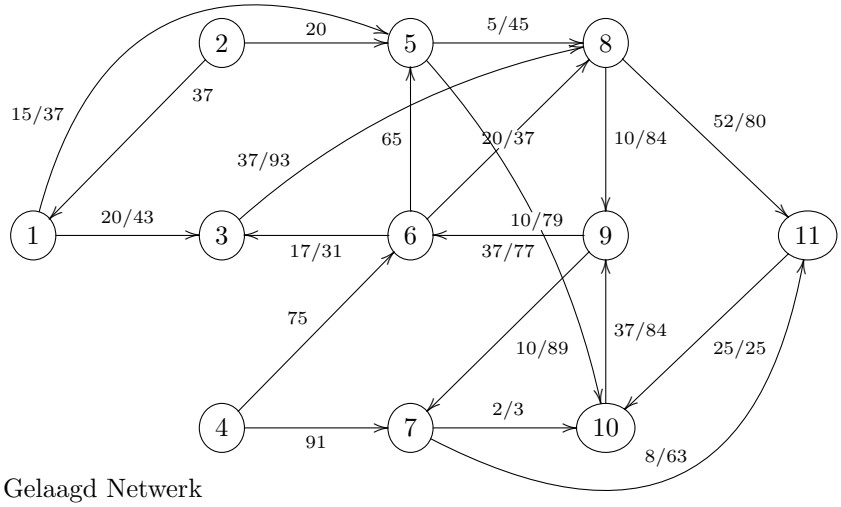
1. Er is een kant van v naar w met $f((v, w)) < c((v, w))$, of

2. Er is een kant van w naar v met $f((v, w)) > 0$.

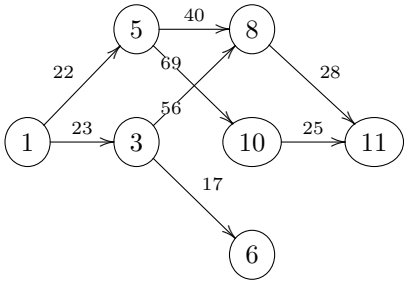
De laatste laag van het netwerk bestaat uit de knoop t. Van laag i naar laag $i + 1$ voegen we steeds de kanten toe die tot opname van een knoop in laag $i + 1$ hebben geleid, met als capaciteit $c(e) - f(e)$ of $f(e)$ afhankelijk van de richting waarin de kant in N loopt, totdat we het stadium bereiken dat t in één of andere laag opgenomen wordt. De knoop t zit altijd alleen in de laatste laag van het netwerk. Om de constructie te verduidelijken geven we een klein voorbeeldje. Merk op dat als een knoop v in laag i zit, dat er dan geen kanten zijn van knoop v van of naar knopen in laag $j < i-1$ met de bovenbeschreven eigenschappen. Immers, als die er wel zouden zijn, dan zou knoop v al opgenomen zijn in laag $j + 1$. De hierbeschreven methode geeft een beschrijving van een netwerk waarin altijd *nette* stroomvergrotende paden lopen, die bestaan uit

kanten van laag i naar laag $i+1$.

Voorbeeld 3.3.2: Netwerk



Gelaagd Netwerk



□

Als nu de lagen van het nieuwe netwerk $V_1, V_2, \ldots, V_k$ zijn, dan lopen alle stroomvergrotende paden door opeenvolgende lagen. We kunnen nu, beginnend met de kant met minimale capaciteit tussen twee lagen, de stroom vergroten net zolang tot de capaciteit van één van de doorsnijdingen bereikt is. We noemen zo'n maximale stroom in een gelaagd netwerk een *blokkerende* stroom en we vinden zo'n stroom als volgt. Laat de restcapaciteit van een knoop het minimum zijn van de som van de restcapaciteiten van zijn uitgaande kanten en de som van de restcapaciteiten van de ingaande kanten. We zoeken de knoop met minimale restcapaciteit, en voegen deze capaciteit eerst toe als stroom aan de uitgaande kanten van de knoop. Vervolgens duwen we deze stroom door het netwerk als volgt. Eerst zorgen we ervoor dat in lagen *hoger* dan de laag waarin deze knoop zit de behoudswetten weer gelden, en daarna zorgen we ervoor dat ook in de lagen *lager* dan de laag waarin deze knoop zit de behoudswetten weer gaan gelden. We beginnen met deze knoop en zijn ingaande kanten. Minimaal één knoop in het gelaagde netwerk is nu verzadigd. We halen deze knoop *met* met al zijn ingaande en uitgaande kanten uit het netwerk en passen de stroom aan zodat de behoudswetten weer gelden. Als er nog een stroomvergrotend pad van s naar t is, dan herhalen we de procedure, anders hebben we een blokkerende stroom gevonden.

De blokkerende stroom kan nu worden opgeteld bij de stroom in het netwerk en een nieuw gelaagd netwerk kan worden berekend. Een nieuw gelaagd netwerk kan niet al te vaak worden berekend. We kunnen bewijzen dat elk nieuw gelaagd netwerk dat wordt berekend na een nieuwe, blokkerende stroom, bij de stroom te hebben opgeteld tenminste één laag dieper is dan het vorige. Aangezien er hoogstens V knopen zijn, betekent dit dat ten hoogste V zulke fasen in de volgende algoritme zitten.

1: **repeat**
2: compute layered network $Y = LN(N, f)$;

3: **if** $t \in Y$ **then** compute blocking flow in Y;
4: **else**
5: write flow at maximum; exit
6: **end if**
7: add blocking flow to f
8: **until** exit occurs

In ons voorbeeld gelaagde netwerk is eerst knoop 5 de knoop met de kleinste capaciteit (22). Na aanpassing van het netwerk en verwijdering van knoop 5, is knoop 8 de knoop met de kleinste capaciteit (6). Als knoop 8 verwijderd is, is er geen pad meer van s naar t. We kunnen in totaal 28 eenheden bij de stroom optellen en een nieuw gelaagd netwerk berekenen. De algoritme die een maximale stroom vindt met behulp van blokkerende netwerken wordt MKM algoritme genoemd, naar de bedenkers Malhotra, Kumar en Maheshwari [MKM78].

3.4 Toepassingen van Network Flow

De algoritmen die we besproken hebben voor het maximaliseren van de stroom kunnen we gebruiken om andere problemen op te lossen. Veel toepassingen zijn denkbaar, we noemen er hier drie.

3.4.1 Perfect Matching

Als eerste voorbeeld bekijken we het probleem PERFECT MATCHING, dat we ook verderop in deze tekst in een wat ingewikkelder vorm tegen zullen komen. Gegeven is een ongerichte tweedelige graaf $G = ((V_1, V_2), E)$ en gevraagd is een deelverzameling van de kanten te vinden, zo dat elke knoop in het ene deel van de graaf verbonden is met precies één knoop in het andere deel van de graaf. Dit probleem is met het maximaliseren van stromen op te lossen als volgt. Geef alle kanten in G een richting zodat ze *van* V_1 naar V_2 lopen. Voeg aan V twee knopen s en t toe. Verbind s met alle knopen uit V_1, en t met alle knopen uit V_2, waarbij de nieuwe kanten *uit* s en *in* t gericht zijn. Tenslotte krijgen alle kanten capaciteit 1. We kunnen nu bewijzen dat zowel de Ford-Fulkerson algoritme als de gelaagde netwerk algoritme een maximale stroom vindt waarbij elke kant stroom 1 of 0 heeft, en dat door elke knoop in V_1 en V_2 precies één enheid stroom gaat. Immers, de restcapaciteit bij elke update van de flow algoritme is steeds een geheel getal en de totale inkomende capaciteit van elke knoop in V_1 is 1 terwijl de totale uitgaande capaciteit van een knoop in V_2 eveneens 1 is. Als we nu alle kanten tussen V_1 en V_2 selecteren waarlangs een stroom 1 loopt, dan hebben we een perfecte matching gevonden.

3.4.2 Connectivity

Een belangrijk probleem dat samenhangt met het al eerder besproken probleem van het vinden van articulation points is het vinden van de connectiviteit van een graaf. De connectiviteit van een graaf is het minimum aantal *kanten* dat verwijderd moet worden om tenminste twee componenten over te houden. Dit probleem kunnen we [ET75] oplossen met de algoritme voor het maximaliseren van de stroom in een netwerk, als volgt. Laat een graaf $G = (V, E)$ gegeven zijn. We maken netwerk $N = (V, E, f)$ door 1 als source te gebruiken en een willekeurige knoop j als sink. Vervolgens vervangen we elk paar $(v, w) \in E$ door een kant in beide richtingen die allebei capaciteit 1 hebben en berekenen een maximale stroom F_j. Het maximum van de getallen $F_1, \ldots, F_n$ is de edge connectivity van G.

3.4.3 Matrixsommen

De laatste toepassing van stromen in netwerken die we bespreken is het probleem van de matrixsommen.

NAAM: Matrix Som
GEGEVEN: Twee verzamelingen getallen R en C

GEVRAAGD: Is het mogelijk een 0/1 matrix op te stellen zodat de getallen in R precies de rijsommen zijn en de getallen in C precies de kolomsommen zijn?

Ook dit probleem kan met netwerk flow worden opgelost. Stel maar dat de rijsommen $r_1, \ldots, r_m$ zijn en de kolomsommen $s_1, \ldots, s_n$. We maken een netwerk N als volgt. De knoop s is verbonden met knopen $x_1, \ldots, x_n$ door kanten met capaciteit $r_1, \ldots, r_n$ respectievelijk, en de vanuit de knopen $y_1, \ldots, y_m$ lopen kanten met capaciteit $s_1, \ldots, s_m$ respectievelijk naar t. Verder loopt van elke x_i een kant naar elke y_j met capaciteit 1. Maximaliseer de stroom en zet een 1 op plaats (i, j) in de matrix als in de maximale stroom de stroom door de corresponderende kant 1 is.

3.4.4 Sommen

1. Bewijs dat een graaf met n knopen en meer dan $n - 1$ kanten een cykel heeft.

2. Kijk naar de volgende algoritme voor *ongerichte* grafen $G = (V, E)$.

   1: **repeat**
   2: Find nodes v, v' such that there is no path from v to v';
   3: Add (v, v') to E;
   4: **until** no such pair can be found

   Als $\|V\| = n$, dan is de scherpste afschatting voor het aantal *keren* dat de repeat loop wordt uitgevoerd $O(\sqrt{n})$, $O(n)$, $O(n^2)$ of $O(n^3)$? Waarom?

3. $n > 1$ personen staan op een veld, gewapend met een taart. Iedereen heeft een unieke dichtstbijstaande buur die met 100% zekerheid geraakt wordt, en iedereen gooit tegelijkertijd. Als iemand niet geraakt wordt noemen we die persoon „overlevende".

   (a) Geef een voorbeeld waarbij er geen overlevenden zijn, en een voorbeeld waarbij er meer dan één overlevende is.

   (b) Laat zien dat voor oneven n er altijd minstens één overlevende is.

4. Een gerichte acyclische graaf is een *tralie* als hij een knoop s heeft, zodat van s naar elke andere knoop in de graaf een pad loopt en een knoop t, zodat vanuit elke andere knoop in de graaf een pad naar t loopt.

   (a) Beschrijf een algoritme die voor een input gerichte acyclische graaf beslist of het een tralie is.

   (b) Wat is de worst-case looptijd van de algoritme?

   (c) Gegeven een graaf met gewogen kanten. Laat A^* algoritme gebruik maken van de heuristiek h, die de afstand van twee punten op het papier waarop de graaf getekend is berekent. Bedenk een graaf met twee punten A en B zo dat de algoritme een pad tussen A en B produceert dat langs alle punten gaat, en niet het optimale pad is.

5. Een toernooi (ook wel volledige gerichte graaf genoemd) is een graaf waarbij elk tweetal punten door een kant verbonden wordt. Een kampioen in een toernooi is een punt dat afstand hoogstens twee tot elk ander punt heeft. Dwz. k is een kampioen als voor elke x in het toernooi er ofwel een kant van k naar x is ofwel er een y in het toernooi is, zo dat er een kant van k naar y en een kant van y naar x bestaat. Bewijs (met inductie) dat elk toernooi een kampioen heeft. Is zo'n kampioen uniek?

6. Bewijs met inductie dat in een netwerk met geheeltallige capaciteiten, de algoritme van Ford en Fulkerson altijd convergeert.

7. Laat zien dat er netwerken bestaan waarin de algoritme van Ford en Fulkerson niet convergeert.

8. Bewijs dat na de berekening van een blokkerende stroom en aanpassing van de stroom in het originele netwerk het nieuwe gelaagde netwerk altijd tenminste één laag dieper is.

9. Het probleem System of Distinct Representatives (SDR) wordt als volgt gedefinieerd.

   NAAM: SDR
   GEGEVEN: Een collectie deelverzamelingen $\{S_i\}_{i=1}^m$ van een verzameling $U = \{u_1, \ldots, u_n\}$
   GEVRAAGD: Bestaat er een $S' = \{u'_1, \ldots, u'_m\}$, zó dat $u'_i \in S_i$ en $u'_i \neq u'_j$ voor $i \neq j$?

   Geef een algoritme voor dit probleem m.b.v. netwerkstromen.

10. Een stel families gaan samen uit eten. Om de sociale interactie te vergroten willen ze een tafelarrangement bedenken zo dat geen twee leden van dezelfde familie aan dezelfde tafel zitten. Stel dat er p families zijn en dat de i-de familie $a(i)$ leden heeft. Stel ook dat er q tafels beschikbaar zijn en dat de j-de tafel capaciteit $q(j)$ heeft. Formuleer dit probleem als een network-flow probleem.

programmeren

1. Een reisbureau heeft over de hele wereld 1000 vestigingen, die tegelijkertijd toegang hebben tot een gedistribueerd databasesysteem om plaatsen op bepaalde vluchten te reserveren. De plaatsen zijn aangegeven alleen door stoelnummers, maar een stoelnummer kan natuurlijk ook de vlucht waarvan zij deel uitmaakt identificeren. Als reserveringen verschillende stoelnummers betreffen, kan de database ze tegelijkertijd (in één batch) vastleggen, maar sommige vluchten zijn populairder dan andere, en dus komt het vaak voor dat twee of meer reserveringen voor hetzelfde stoelnummer gemaakt moeten worden. Die moeten dan in volgorde van binnenkomst worden afgehandeld. Implementeer een batch van 100000 aanvragen, waarbij stoelnummers random tussen de 1 en de 500 worden gekozen. Het programma moet een verdeling maken van de transacties zo dat zo min mogelijk batches voorkomen.

2. Stel eens dat ons reisbureau alleen maar van paren transacties kan vaststellen dat ze niet gelijktijdig gedraaid kunnen worden. Dus niet per se omdat ze dezelfde stoel willen reserveren maar bijvoorbeeld omdat ze een printer in hetzelfde kantoor willen gebruiken, dezelfde vluchttabel willen raadplegen of wat dan ook. Nu kan dus alleen van twee transacties T_1 en T_2 worden vastgesteld dat ze niet tegelijkertijd uitgevoerd kunnen worden, maar niet noodzakelijk van grotere verzamelingen. Hoe pak je nu het probleem aan?

3. Maak een netwerk van ongeveer 100 knopen met willekeurige capaciteiten tussen de 1 en de 100. Vergelijk op dit netwerk de snelheden van de Ford-Fulkerson algoritme en de MKM algoritme.

Hoofdstuk 4

Numerieke Algoritmen

4.1 De Euclidische Algoritme en Uitbreidingen

Het wordt wel „de oudste algoritme" genoemd, de algoritme om de grootste gemene deler van twee getallen te bepalen. Voor het eerst beschreven in „Elementen" van Euclides, ongeveer 300 voor Christus en oorspronkelijk geformuleerd als een meetkundig probleem: zoek een gemeenschappelijke maat voor twee lijnstukken. Deze algoritme echter was bijna zeker al 200 jaar eerder bekend bij Eudoxus van Cnidus (375vChr) en ook Aristoteles verwijst er in 330 vChr al naar in zijn Topica. De algoritme die wij kennen als „de Euclidische Algoritme" is dus waarschijnlijk net zo min van Euclides als de „Stelling van Pythagoras"van Pythagoras was (het is bekend dat deze stelling door Pythagoras gestolen is van de Babyloniers). Niettemin zullen wij de algoritme uit deze sectie „de Euclidische Algoritme noemen" en de pas veel later bekend geworden uitbreiding die we nodig hebben in Sectie 4.5 als de „Uitgebreide Euclidische Algoritme". Waar ging het ook alweer over? Als we twee getallen a en b hebben, dan is er een grootste getal dat beide getallen deelt. Als dat getal 1 is, dan zeggen we dat a en b *relatief priem* zijn, anders kunnen wij een groter getal dan 1 vinden dat beide getallen deelt, maar hoe? We zouden bijvoorbeeld de beide getallen kunnen ontbinden in (priem)factoren, en dan de kleinste machten van de gemeenschappelijke priemfactoren kunnen uitvermenigvuldigen. Bijvoorbeeld $1400 = 2^3 \cdot 5^2 \cdot 7$ en $6860 = 2^2 \cdot 5 \cdot 7^3$, waaruit volgt dat de grootste gemene deler van 1400 en 6860 gelijk is aan $2^2 \cdot 5 \cdot 7 = 140$. Dat geeft zeker de grootste gemene deler, maar is, zeker voor grote getallen, een moeizaam proces. Veel simpeler is de algoritme gebaseerd op de volgende observatie. Als je twee getallen a en b hebt en $a = qb + t$, dan deelt elk getal dat zowel t als b deelt ook a. De grootste gemene deler van t en b is dus ook de grootste gemene deler van a en b. Als $b = 0$, dan is a de grootste gemene deler van a en b, want elk getal deelt 0. Dit suggereert een recursieve algoritme: óf één van a en b is 0 en dan is het klaar, óf $a > b$ en dan schrijven we $a = qb + t$ met t zo klein mogelijk en gaan verder met b en t. We schrijven dit in pseudocode in Figuur 4.1

```
1: r = a mod b
2: while r ≠ 0 do
3:     a = b
4:     b = r
5:     r = a mod b
6: end while
```

Figuur 4.1: grootste gemene deler

Als r gelijk aan 0 is geworden, dan is b gelijk geworden aan de grootste gemene deler van de oorspronkelijke getallen. Deze algoritme is aanzienlijk efficiënter dan het ontbinden in factoren. Aangezien in elke iteratie de getallen minstens door 2 gedeeld worden kan de loop niet vaker dan log keer doorlopen worden, in het aantal bits van de input is dat dus lineair. Hierbij nemen we echter aan dat elke deling één stap kost, wat

bij grote getallen misschien niet een reële aanname is.

4.1.1 De Uitgebreide Euclidische Algoritme en Inversen

Neem twee getallen $a > b$. Als de grootste gemene deler van a en b gelijk is aan 1, dan heeft b een multiplicatieve inverse modulo a, ofwel, er is een getal x zodanig dat $bx \mod a = \text{ggd}(a, b) = 1$. Dit getal x kan bij het bepalen van de grootste gemene deler van a en b door de Euclidische algoritme gevonden worden. Dat gaat als volgt.

De Euclidische algoritme is gebaseerd op de waarneming dat de grootste gemene deler van twee getallen a en b gelijk is aan de grootste gemene deler van b en $a \mod b$, ofwel als $a = qb + r$ en $r = a \mod b$, dan geldt $d = \text{ggd}(a, b) = \text{ggd}(b, r)$. Stel nu eens dat de recursieve algoritme gebaseerd op deze observatie niet alleen de ggd terug zou geven maar ook twee getallen k en l zodat $d = kb + lr$. Dan hebben we dat $d = kb + lr = kb + l(a - qb) = la + (k - lq)b$ en dus $d = xa + yb$. Als d gelijk is aan 1 en we rekenen modulo a, dan staat daar $1 = (k - lq)b$. Het probleem is alleen, dat we de grootste gemene deler pas op de bodem van de recursie tegen komen. Op de bodem van de recursie staat echter altijd $b = 0$ dus hier kunnen we een standaard antwoord geven, namelijk a is de ggd en $k = 1$ en $l = 0$.

```
1: function Egcd(a, b)
2: if b = 0 then
3:     return (a,1,0)
4: else
5:     r = a mod b;
6:     q = (a − r)/b;
7:     (d, k, l) = Egcd(b, r);
8:     return(d, l, k − lq);
9: end if
```

Figuur 4.2: de uitgebreide Euclidische algoritme

Voorbeeld 4.1.1: Een getallenvoorbeeld is altijd illustratief. We berekenen dus de inverse van 17 modulo 93 met deze algoritme. We beginnen met het uitvoeren van de uitgebreide Euclidische algoritme.

$$\begin{array}{rcl} a & = & q \times b + r \\ 93 & = & 5 \times 17 + 8 \\ 17 & = & 2 \times 8 + 1 \end{array}$$

De volgende stap is 0. De vergelijkingen $d = kb + lr = kb + l(a - qb) = la + (k - lq)b$ vertellen nu achtereenvolgens $1 = 1 \times 17 - 2 \times 8$, $1 = 1 \times 17 - 2 \times (93 - 5 \times 17)$, en $1 = -2 \times 93 + 11 \times 17$, waaruit volgt dat 11 de inverse van 17 modulo 93 is.

Dus $d = 1 = 11 \times 17 - 2 \times 93$ waaruit volgt dat 11 het gezochte getal is. □

Het resultaat van deze algoritme is een getal l dat modulo b de multiplicatieve inverse is van a (uiteraard alleen als de grootste gemene deler van a en b gelijk is aan 1).

4.1.2 sommen

1. Schrijf een niet-recursieve versie van de Euclidische algoritme. Generaliseer dit tot een niet-recursieve versie van de uitgebreide Euclidische algoritme.

2. Bewijs het bestaan van additieve inversen in Z_p. Dat wil zeggen laat zien dat er voor elke $x \in Z_p$ een $y \in Z_p$ bestaat zó dat $x + y \mod p = 0$.

3. Maak een vermenigvuldigingstabel voor Z_5, waar het element op plaats (i, j) in de tabel gelijk is aan $i * j \mod 5$.

4. Bereken de multiplicatieve inversen van de getallen 435, 234, en 534 in Z_{947}.

4.2 Vermenigvuldiging van Grote Getallen, DFT

4.2.1 Inleiding

Computers zijn rekenautomaten en rekenen, naast zoeken en sorteren, is nog steeds hun belangrijkste activiteit. Van de rekenkundige operaties: optellen, aftrekken, vermenigvuldigen, delen, machtsverheffen en worteltrekken worden sommige veel vaker uitgevoerd dan andere, en zijn sommige complexer, dwz. vereisen meer rekenkracht dan andere.

In veel gevallen beschouwen we een optelling of vermenigvuldiging als een atomaire operatie, en tellen voor elke vermenigvuldiging één stap. Dat is niet altijd gerechtvaardigd, zie ook 5.2.2. Omdat machines beperkte registergrootte hebben (meestal 8, 16, 32 of 64 bits), kan een vermenigvuldiging van twee willekeurig grote getallen niet in één stap worden uitgevoerd, maar moeten delen van de getallen (vaak individuele cijfers) net zoals wanneer je met de hand en kladpapier vermenigvuldigt, apart worden uitgevoerd. Het resultaat van de vermenigvuldiging volgt dan vaak uit een optelling van de tussenresultaten. In deze sectie zullen we verschillende vermenigvuldingsalgoritmen bekijken, beginnend met één die we ook voor vermenigvuldiging met de hand gebruiken, en de complexiteit van deze algoritmen analyseren. Zoals we gewend zijn, gaan we voorbij aan constante factoren in de complexiteit, maar de lezer zij bijvoorbaat gewaarschuwd dat in de algoritmen die in deze sectie gepresenteerd worden deze factoren *wel degelijk* een rol spelen. De meest efficiënte algoritme voor vermenigvuldiging, die aan het eind gepresenteerd wordt, is daarom alleen zinvol bij het vermenigvuldigen van zeer grote getallen—enkele honderden cijfers. Gelukkig heeft het vermenigvuldigen van zeer grote getallen tegenwoordig toepassing gevonden in de cryptografie, zoals we verderop in dit hoofdstuk zullen zien. Public key cryptosystemen maken gebruik van sleutels die het resultaat zijn van de vermenigvuldiging van priemgetallen van enkele honderden cijfers. Om een boodschap te coderen, wordt een getal dat de boodschap in binaire code voorstelt tot de macht zo'n groot getal genomen. Hiervoor zijn dus meerdere vermenigvuldigingen met grote getallen nodig, evenals een efficiente manier van machtsverheffen. Hierop komen we later terug.

4.2.2 De eerste algoritmen

De algoritme voor vermenigvuldiging die mensen het meest gebruiken is het „vierkante schema" dat we op de basisschool leren. Stel dat we de getallen 981 en 1234 willen vermenigvuldigen. Dan schrijven we deze getallen onder elkaar als.

```
    981
   1234
  ------
   3924
  29430
 196200
 981000
------
1210554
```

Overigens zij opgemerkt dat dit cultuurbepaald is. Op de Engelse scholen wordt dit vermenigvuldigingsschema „andersom" geleerd als:

```
    981
   1234
  ------
 981000
 196200
  29430
   3942
------
1210554
```

Hoeveel elementaire operaties (vermenigvuldiging van twee ééncijfergetallen) kost dit? Als we een getal van n cijfers met een getal van m cijfers vermenigvuldigen, dan kunnen we in het vierkant van vermenigvuldiging hierboven zien dat er $n \times m$ vermenigvuldigingen van ééncijfergetallen gedaan worden. Dit is niet optimaal.

4.3 Betere algoritmen voor vermenigvuldiging

Een andere manier om 2 getallen met elkaar te vermenigvuldigen wordt gegeven door het onderstaande schema.

981	1234	1234
490	2468	
245	4936	4936
122	9872	
61	19744	19744
30	39488	
15	78976	78976
7	157952	157952
3	315904	315904
1	631808	631808
		1210554

Het maakt gebruik van de simpele observatie dat je twee operanden met elkaar kunt vermenigvuldigen door de ene operand herhaald te halveren terwijl de andere herhaald verdubbeld wordt. Daarbij moet, omdat naar beneden wordt afgerond wel op de oneven delingen gelet worden. Daarom worden in de meest rechtse kolom alle getallen waarbij de linkerkolom oneven wordt, bij elkaar opgeteld om het eindresultaat te krijgen.

Enige analyse leert dat deze algoritme nog steeds n^2 bewerkingen doet om twee getallen van n cijfers met elkaar te vermenigvuldigen. Voor computers heeft deze vermenigvuldiging „à la russe" zoals hij genoemd wordt echter een zeer groot voordeel. Het betreft alleen vermenigvuldigen met en delen door 2. In machines is dat een simpele shift operatie.

Een zeer populaire manier om algoritmen te ontwikkelen is de „verdeel-en-heersmethode" (zie Sectie 2.2). Het probleem wordt uitgedrukt in twee of meer kleinere deelproblemen die dan vervolgens worden opgelost door ze weer in kleinere problemen te verdelen. Op de „bodem" van deze aanpak ligt het oplossen van problemen van grootte 1 of kleiner. Die problemen kennen een triviale oplossing.

We kunnen deze methode natuurlijk ook toepassen op vermenigvuldigingen. De bodem van de vermenigvuldiging is immers het vermenigvuldigen van individuele cijfers. Hiervoor hebben wij op de bassisschool tabellen (tafels) geleerd, dus dat kost geen inspanning. In ons voorbeeld ziet dat er (op het één na laagste niveau) zo uit:

	vermenigvuldig		schuif	resultaat
i)	09	12	4	1080000
ii)	09	34	2	30600
iii)	81	12	2	97200
iv)	81	34	0	2754
				1210554

We merken hier op dat we een truuk kunnen toepassen analoog aan de truuk die door Strassen is ontwikkeld voor matrixvermenigvuldiging (zie 2.2.1). Het vermenigvuldigen van 2 getallen van 4 cijfers kost namelijk op deze manier 4 vermenigvuldigingen van getallen van 2 cijfers (de schuifoperaties daargelaten)

als:

$$\begin{array}{rcrcrcr} 34 & \times & 81 & & & = & 2754 \\ 12 & \times & 81 & \times & 100 & = & 97200 \\ 34 & \times & 09 & \times & 100 & = & 30600 \\ 12 & \times & 09 & \times & 10000 & = & 1080000 \\ & & & & & & 1210554 \end{array}$$

Bijgevolg kost het vermenigvuldigen van twee getallen van n cijfers op deze manier $4^{log_2 n} = n^{\log_2 4} = n^2$, dus in complexiteit winnen we weer niets. Echter met het volgende schema kunnen de vier vermenigvuldigingen vervangen worden door drie vermenigvuldigingen: $(a \times 100 + b)(c \times 100 + d) = a \times c \times 100 \times 100 + b \times c \times 100 + a \times d \times 100 + b \times d$. Echter $b \times c + a \times d = (a + b)(c + d) - ac - bd$. Dus als we $(a + b)(c + d)$, ac, en bd uitrekenen, hebben we alles wat we nodig hebben (in drie vermenigvuldigingen). In getallen:

$$\begin{array}{rcrcr} 09 & \times & 12 & = & 108 \\ 81 & \times & 34 & = & 2754 \end{array}$$

en

$$(09 + 81)(12 + 34) = 90 \times 46 = 4140$$

en

$$981 \times 1234 = 1080000 + (4140 - 108 - 2754) \times 100 + 2754 = 1210554$$

waardoor de complexiteit daalt tot $n^{log_2 3}$ vermenigvuldigingen. Het is geen grote winst, maar in geval miljoenen van deze vermenigvuldigingen moeten worden gedaan, of heel grote getallen met elkaar vermenigvuldigd moeten worden, zeker de moeite waard.

4.3.1 De kampioen

In deze sectie zullen we een algoritme presenteren die vooralsnog als de snelste algoritme voor de vermenigvuldiging wordt beschouwd. De algoritme is gebaseerd op de Fast Fourier transformatie, in de vorige eeuw bedacht door Cooley en Tukey.

4.3.2 Getallen als polynomen

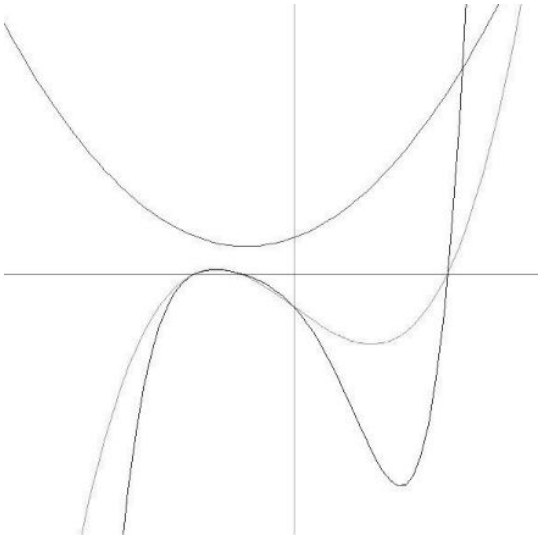
Kijk naar het getal 10230495. We kunnen het schrijven als

$$1 \times 10^7 + 0 \times 10^6 + 2 \times 10^5 + 3 \times 10^4 + 0 \times 10^3 + 4 \times 10^2 + 9 \times 10^1 + 5.$$

Elk getal van n cijfers is een polynoom van de graad $n - 1$ uitgerekend in het punt 10. In plaats van te onderzoeken hoe duur het vermenigvuldigen van getallen is, zouden we dus de algemenere vraag kunnen stellen: „Hoe duur is het vermenigvuldigen van twee polynomen, en hoe duur is het uitrekenen van een polynoom in een punt?"

We merken eerst op dat een polynoom van de graad $n - 1$ geheel bepaald is als we de waarden van het polynoom in n punten weten. Een lijn (eerstegraadspolynoom) ligt vast als we er twee punten van kennen, en een parabool ligt vast als we er drie punten van kennen, immers het invullen van drie waarden voor x en y in een vergelijking van de vorm $y = ax^2 + bx + c$ geeft drie lineaire vergelijkingen met drie onbekenden. Algemener: het invullen van n punten in een vergelijking $y = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_0$ geeft n vergelijkingen met n onbekenden en dus hoogstens 1 oplossing voor $a_{n-1}, \ldots, a_0$.

Dus, als we $n + m - 1$ punten van een polynoom weten dat het product is van een polynoom van de graad $n - 1$ en een polynoom van de graad $m - 1$ dan ligt dat polynoom, omdat het van de graad hoogstens $n - 1 + m - 1$ is, daarmee vast. Wanneer we echter voor een x de waarde van een polynoom p_1 in dat punt weten *en* we weten de waarde van een polynoom p_2 in dat punt, dan is het heel gemakkelijk te achterhalen wat de waarde van een polynoom $p_1 \times p_2$ in dat punt is. Vermenigvuldig namelijk alleen die waarden met elkaar.



Figuur 4.3: $x^2 + x + 1$, $x^3 - 2x - 1$, en $(x^2 + x + 1)(x^3 - 2x - 1)$

Als we dus $n + m - 1$ punten van p_1 hebben en $n + m - 1$ punten van p_2 in *dezelfde* rij x-waarden, dan krijgen we met $n + m - 1$ vermenigvuldigingen een verzameling punten die het productpolynoom volledig vastleggen.

Voorbeeld 4.3.1: In Figuur 4.3 zien we de grafieken van $x^2 + x + 1$ en $x^3 - 2x - 1$. Het product van deze polynomen is een vijfdegraads polynoom. Om dit polynoom vast te leggen hebben we zes punten nodig.

x	$x^2 + x + 1$	$x^3 - 2x - 1$	$\times$
-3	7	-22	-154
-2	3	-5	-15
-1	1	0	0
0	1	-1	-1
1	3	-2	-6
2	7	3	21

Een vijfdegraadspolynoom ziet eruit als $ax^5 + bx^4 + cx^3 + dx^2 + ex + f$. Om het productpolynoom te bepalen moeten we dus de volgende vergelijkingen oplossen:

$$\begin{aligned} -243a + 81b - 27c + 9d - 3e + f &= -154 \\ -32a + 16b - 8c + 4d - 2e + f &= -15 \\ -a + b - c + d - e + f &= 0 \\ f &= -1 \\ a + b + c + d + e + f &= -6 \\ 32a + 16b + 8c + 4d + 2e + f &= 21 \end{aligned}$$

Lossen we deze vergelijkingen op, dan vinden we $a = 1$, $b = 1$,$c = -1$, $d = -3$, $e = -3$, en $f = -1$. Het productpolynoom wordt dus

$$x^5 + x^4 - x^3 - 3x^2 - 3x - 1.$$

□

De algoritme voor polynoomvermenigvuldiging: „reken de polynomen uit in $n + m - 1$ punten → reken de punten van het productpolynoom uit → vertaal de punten terug naar coëfficienten" dringt zich op. De vraag is alleen: hoe duur zijn de buitenste vertaalslagen?

Voor het uitrekenen van polynomen in punten zijn verschillende algoritmen bekend, bijvoorbeeld $x^5 + 2x^4 + 3x^3 + x^2 + x + 1$ in 2 als $2^5 + 2 \cdot 2^4 + 3 \cdot 2^3 + 2^2 + 2 + 1$ of $((((2+2)2+3)2+1)2+1)2+1$. De snelste algoritmen geven echter toch ongeveer n vermenigvuldigingen voor één punt. Dat betekent voor $n+m-1$ punten nog steeds kwadratische complexiteit. We moeten zoeken naar een methode waarbij we de evaluatie van polynomen in n punten *tegelijkertijd* kunnen doen voor minder dan $O(n^2)$ operaties. Dat vereist een stel punten met speciale eigenschappen. Zo'n stel punten is de verzameling van de complexe n-de machts eenheidswortels.

4.3.3 Complexe n-de machts eenheidswortels

De complexe n-de machts eenheidswortels zijn de oplossingen van de vergelijking $x^n - 1 = 0$. In de reële getallen heeft deze vergelijking hoogstens 2 oplossingen, namelijk 1 en -1 en dan alleen nog voor even waarden van n. Laten wij echter complexe getallen toe van de vorm $a + bi$, waarbij a en b reële getallen zijn en i een getal met de eigenschap $i^2 = -1$, dan heeft de vergelijking plotseling n oplossingen.

In 1.4.3 zagen we de reeksen voor sinus, cosinus en e-machten:

$$\begin{array}{lcl} e^x & = & \sum_{m=0}^{\infty} x^m/m! \\ \sin x & = & \sum_{r=0}^{\infty} (-1)^r x^{2r+1}/(2r+1)! \\ \cos x & = & \sum_{r=0}^{\infty} (-1)^r x^{2r}/(2r)! \\ \log \frac{1}{1-x} & = & \sum_{j+1}^{\infty} x^j/j \end{array}$$

Laten we het zojuist gedefinieerde getal i eens gebruiken en eens invullen ix invullen in de reeksen voor sin en cos. We zien dat $\cos x + i \sin x = \sum_{r=0}^{\infty} (-1)^r x^{2r}/(2r)! + i \sum_{r=0}^{\infty} (-1)^r x^{2r+1}/(2r+1)! = \sum_{m=0}^{\infty} (ix)^m/m! = e^{ix}$.

Vanwege de periodiciteit van sin en cos geldt nu dus voor elke n en $0 < j < n$ dat $(e^{2\pi ij/n})^n = \cos 2j\pi + i \sin 2j\pi = 1$. Voor elke n zijn de n verschillende getallen $\omega_{j(n)} = e^{i2\pi j/n} = \cos 2\pi j/n + i \sin 2\pi j/n$ de n verschillende oplossingen van de vergelijking $x^n - 1 = 0$. We noemen deze getallen de *complexe n-de machts eenheidswortels* en deze getallen hebben een aantal interessante eigenschappen. Als n er niet to doet laten we n vaak weg uit de notatie en schrijven we ω_j.

Laat $\omega = \omega_1 = e^{2\pi i/n}$, dan is $\omega_j = \omega^j$.

1. $\omega_j^{n-1} = \omega_j^{-1}$ want $\omega_j^{n-1} \times \omega_j = \omega_j^n = 1$.

2. Voor $n > 1$: $\sum_{k=0}^{n-1} \omega_j^k = 0$ want

$$\sum_{k=0}^{n-1} \omega_j^k = \frac{\omega_j^n - 1}{\omega_j - 1} = \frac{1-1}{\omega_j - 1} = \frac{0}{\omega_j - 1} = 0.$$

3. als ω_j een $2n$-de machts eenheidswortel is, dan is ω_j^2 een n-de machts eenheidswortel, want $\omega_j^{2n} = (\omega_j^2)^n = 1$.

4. Voor even $n > 0$ geldt $\omega^{n/2} = -1$ want

$$0 = \sum_{k=0}^{n-1} \omega^{(n/2)k} = \omega^0 + \omega^{n/2} + \ldots + \omega^0 + \omega^{n/2} = (n/2)(1 + \omega^{n/2})$$

4.3.4 De Fast-Fourier transform

Precies deze eigenschappen geven genoeg voor het snel evalueren van polynomen in n punten. We nemen voor het gemak aan dat n een macht van 2 is. Dat is geen beperking van de algemeenheid, want als n geen tweemacht is, dan ligt er een tweemacht dicht bij n (kleiner dan $n \times 2$ en we kunnen een polynoom altijd uitbreiden tot zijn graad een tweemacht is (door 0-coëfficienten op te nemen). Beschouw nu het polynoom $p = a_{n-1}x^{n-1} + \ldots + a_0$. We kunnen p onderverdelen in een even polynoom p_e en een oneven polynoom p_o

als $p_e = a_{n-2}x^{n-2} + \ldots + a_0$ en $p_o = a_{n-1}x^{n-1} + \ldots + a_1 x$. Verder is $p_o = x(a_{n-1}x^{n-2} + \ldots + a_1)$. We gebruiken nu de eigenschap 3 uit het rijtje eigenschappen en merken op dat het evalueren van een polynoom in een n de machts eenheidswortel kan worden gedaan ten koste van de evaluatie van dat polynoom in een $n/2$-de machts eenheidswortel plus één vermenigvuldiging, en dat deze eigenschap recursief is.

Dit betekent dat we de evaluatie van de polynomen kunnen uitstellen totdat er niet zoveel eenheidswortels zijn (bijvoorbeeld 1) en dan vervolgens de evaluatie van de polynomen daar kunnen gebruiken om hogerop de evaluatie van grotere polynomen in ten koste van 1 vermenigvuldiging te krijgen. De algoritme is dan als volgt.

Voor evaluatie van een polynoom van graad $n-1$ in de n-de machts eenheidswortels. Evalueer p_e en p_o in alle $n/2$ de machts eenheidswortels en doe telkens 1 extra vermenigvuldiging. Probleem is dat er natuurlijk maar half zoveel $n/2$e machts eenheidswortels zijn als n-de machts eenheidswortels, maar dat probleem wordt opgelost door de periodiciteit van de laatste. In het volgende voorbeeld lichten we dat toe.

Voorbeeld 4.3.2: Stel we evalueren het vijfde graads polynoom

$$2x^5 + 2x^4 + 4x^3 + 2x^2 + 8x + 1$$

in de achtste macht éénheidswortels en we zijn bij de berekening bij $\omega_5 = e^{2\pi i 5/8}$. We splitsen het polynoom in

$$p_e = 2x^4 + 2x^2 + 1$$

en

$$p_o = 2x^5 + 4x^3 + 8x = x(2x^4 + 4x^2 + 8).$$

Invullen geeft

$$p_e(e^{2\pi i 5/8}) = 2(e^{2\pi i 5/8})^4 + 2(e^{2\pi i 5/8})^2 + 1$$

en

$$p_o(e^{2\pi i 5/8}) = (e^{2\pi i 5/8})(2(e^{2\pi i 5/8})^4 + 4(e^{2\pi i 5/8})^2 + 8).$$

We zien dat

$$p_e(e^{2\pi i 5/8}) = 2(e^{2\pi i 5/4})^2 + 2(e^{2\pi i 5/4}) + 1$$

en

$$p_o(e^{2\pi i 5/8}) = (e^{2\pi i 5/8})(2(e^{2\pi i 5/4})^2 + 4(e^{2\pi i 5/4}) + 8).$$

We zien dus dat we de vierdemachts éénheidswortels nodig hebben, ware het niet dat er geen vijfde vierdemachts eenheidswortel bestaat. Gelukkig is echter $e^{2\pi i 5/4} = e^{2\pi i 1/4}$ vanwege de periodiciteit van de éénheidswortels. □

De complexiteit van de hierbovenbeschreven algoritme is telkens n vermenigvuldigingen *plus* de kosten van de evaluatie van de polynomen van de halve lengte. Waarbij het evalueren van een polynoom van de graad 0 in alle 1 de machts eenheidswortels uiteraard als eenheidsstap gerekend mag worden. In een recurrente betrekking: $T(n) = n + 2T(n/2)$. De oplossing van deze betrekking is $T(n) = O(n \log n)$.

4.3.5 De inverse Fourier transform

We zijn nu halverwege. We kunnen de beide polynomen voor de prijs van $n \log n$ vermenigvuldigingen op n punten evalueren, en dan ten koste van n vermenigvuldigingen met elkaar vermenigvuldigen om n punten van het productopolynoom in handen te krijgen. Dan moeten we echter terug. Gelukkig is de terugweg net zo eenvoudig als de heenweg. We beweren dat we de coëfficienten van het productpolynoom in handen kunnen krijgen door dezelfde algoritme toe te passen, maar nu met de verkregen punten als coëfficienten, en de evaluatie van de polynomen in de punten ω_j^{-1}. Om dat in te zien beschouwen we de Fourier transformatie als matrixvermenigvuldiging. In Figuur 4.4 zien we de matrix van de Fouriertransformatie en de inverse van die matrix onder elkaar.
Immers

$$F \times a = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega_1 & \cdots & \omega_1^{n-1} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & \omega_{n-1} & \cdots & \omega_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

$$F^{-1} = \frac{1}{n} \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega_1^{-1} & \cdots & (\omega_1^{-1})^{n-1} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & \omega_{n-1}^{-1} & \cdots & (\omega_{n-1}^{-1})^{n-1} \end{pmatrix}.$$

Figuur 4.4: De Fourier transformatie en inverse

$(F^{-1} \times F)[i,j] = \frac{1}{n} \sum_{k=0}^{n-1} \omega_i^k \times (\omega_j^{-1})^k$.
De entries van deze matrix zijn $\frac{1}{n} \sum_{k=0}^{n-1} (\omega_i \times (\omega_i^{-1}))^k$
Dit is

$$\begin{cases} \frac{1}{n} \sum_{k=0}^{n-1} 1^k = 1 & \text{als } i = j \text{ en} \\ \frac{1}{n} \sum_{k=0}^{n-1} (\omega_i \times (\omega_j^{-1}))^k = \frac{1}{n} \sum_{k=0}^{n-1} \omega_m^k = 0 & \text{als } i \neq j \end{cases}.$$

De algoritme voor de inverse Fourier transformatie kan dus dezelfde zijn en dus is die ook van dezelfde complexiteit. In totaal is er dus voor de vermenigvuldiging $O(n \log n + n + n \log n) = O(n \log n)$ nodig.

4.4 Snelle Machtsverheffing

In 4.5 hieronder zullen we niet alleen grote getallen met elkaar moeten vermenigvuldigen, maar we zullen zelfs machten van grote getallen moeten berekenen. Gegeven een snelle vermenigvuldigingsalgoritme is er ook een snelle algoritme om machten te berekenen. Deze algoritme is van het type „verdeel en heers" en is één van de oudst bekende algoritmen van deze soort. Eerst merken we op dat de naïeve berekening van a^n ongeveer n vermenigvuldigingen van a met zichzelf vereist, en dat bovendien de operanden in die vermenigvuldiging in ongeveer elke stap twee keer zo lang worden, zodat zeker na een klein aantal stappen de lengte van de getallen mee moet worden genomen in de complexiteit van de algoritme. Nu geldt dat $a^n = (a^{n/2})^2$ als a even is, en $a^n = a^{n-1} \times a$ als a oneven is. Dit betekent dat een verdeel-en-heersalgoritme voor a^n de volgende complexiteit heeft.

$$T(n) = \begin{cases} 0 & \text{als } n = 1 \\ T(n/2) + 1 & \text{als } n \text{ even is} \\ T(n-1) + 1 & \text{als } n \text{ oneven is} \end{cases}$$

Dit is een eigenaardige functie. Bijvoorbeeld

$$\begin{aligned} T(31) &= T(30) + 1 = T(15) + 2 = T(14) + 3 = T(7) + 4 = \\ &= T(6) + 5 = T(3) + 6 = T(2) + 7 = T(1) + 8 = 8 \\ T(32) &= T(16) + 1 = T(8) + 2 = T(4) + 3 = T(2) + 4 = T(1) + 5 = 5 \end{aligned}$$

Voor $T(2^n)$ zien we n halveringsstappen, zodat $T(2^n) = n$, terwijl voor $T(2^n - 1)$ we $n - 1$ halveringsstappen *en* $n - 1$ stappen $T(i) \to T(i-1)$, zodat $T(2^n - 1) = 2(n-1)$. Met inductie:

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

Omdat deze functie niet uiteindelijk niet dalend is, kunnen we niet toepassen wat we in Som 6(in 1.3.2) op pagina 26 hebben bewezen. We merken echter op dat de functie lineair is, dus misschien kunnen we

boven en ondergrenzen vinden die wel monotoon zijn. Als $n > 1$ oneven is, dan is $T(n) = T(n-1) + 1 = T((n-1)/2) + 2 = T(\lfloor n/2 \rfloor) + 2$. Als n even is, dan is $T(n) = T(\lfloor n/2 \rfloor) + 1$, dus altijd geldt:

$$T(\lfloor n/2 \rfloor) + 1 \leq T(n) \leq T(\lfloor n/2 \rfloor) + 2.$$

T wordt dus van beneden, resp. van boven begrensd door de functies T_1 en T_2 met

$$T_i(n) = \begin{cases} 0 & \text{als } n = 1 \\ T_i(\lfloor n/2 \rfloor) + i & \text{anders} \end{cases}$$

Omdat $T_1 \leq T \leq T_2$ en *beide* functies in $\theta(\log n)$ zitten geldt ook $T(n) \in \theta(\log n)$. Ofwel we kunnen machtsverheffing doen met een aantal vermenigvuldigingen dat *lineair* is de lengte van de exponent. De efficiënte methode voor vermenigvuldigen hierboven beschreven geeft dat machtsverheffing in tijd $O(n^2 \log n)$ waarin n een bovengrens is voor de lengte van de exponent en de grootste te vermenigvuldigen getallen.

4.4.1 sommen

1. Het produkt van twee polynomen f en g waarbij de de coëfficienten van $x^k - i$ in f en x^i in g met elkaar vermenigvuldigd worden en vervolgens voor alle $i \leq k$ bij elkaar opgeteld worden om in het produktpolynoom de coëfficient van x^k te vormen, wordt ook wel „convolutieproduct" genoemd. (Dit begrip heeft in de algebra een nog veel ruimere betekenis.) Met de DFT kunnen we nu zo'n produkt snel uitrekenen. Gebruik de DFT voor de polynomen met coëfficienten 1,2,3,4 en 4,3,2,1.

2. Voor elke verzameling $\{x_0, \ldots, x_{n-1}\}$ van reële getallen is er precies één n-de graads monadisch (dwz. de coëfficient van de hoogstegraadsterm is 1) polynoom, dat voor precies die waarden 0 wordt, namelijk $(x - x_0)(x - x_1) \ldots (x - x_{n-1})$. Geef een Verdeel-en-heersalgoritme die dit polynoom in coëfficienten geeft in tijd $O(n \log^2 n)$.

4.5 Cryptografie

Bij het uitwisselen en bewaren van gevoelige gegevens is het van belang dat deze niet tussentijds door derden kunnen worden gelezen en/of bewerkt. Behalve het letterlijk achter slot en grendel houden van de gegevens (dat houdt in zorgen dat niemand erbij kan door de deur waarachter zich de gegevens bevinden gesloten te houden), is ook de versleuteling van de gegevens van belang. Een goed cryptosysteem kan ervoor zorgen dat, zelfs als een derde persoon fysiek toegang krijgt tot de data, zij er niets mee kan doen. Cryptosystemen hebben de eigenschap dat geauthoriseerde personen gemakkelijk, dat wil zeggen computationeel eenvoudig, toegang hebben tot de data, terwijl niet geauthoriseerde personen moeilijk (bij voorkeur geen) toegang hebben tot de data. Aangezien aangenomen moet worden dat beiden toegang hebben tot de bron van de (bewerkte) data, moeten geauthoriseerde personen dus beschikken over een hoeveelheid extra informatie, die de brondata zinvol maakt.

Deze extra informatie wordt sleutel genoemd. Als meerdere personen toegang moeten hebben tot de informatie, of als het bijvoorbeeld over informatie gaat die over een onbetrouwbare lijn moet worden verstuurd wordt het ingewikkeld. Dan moeten meerdere personen zo'n sleutel delen, verschillende sleutels moeten toegang bieden tot dezelfde informatie, of de informatie moet op verschillende manieren versleuteld kunnen worden. In het laatste geval kan de ene partij versleutelde informatie versturen zonder kennis te hebben van de sleutel van de andere partij. In het eerste geval is de versleuteling symmetrisch (de sleutel die nodig is om te versleutelen is dezelfde als de sleutel nodig om de informatie terug te vinden). De laatste situatie is asymmetrisch, de sleutel nodig om de informatie te versleutelen is een andere dan die nodig is om de informatie terug te winnen. In het eerste geval is het noodzakelijk de sleutel geheim te houden bij de partijen die informatie uitwisselen (dit systeem wordt daarom "private key" genoemd) in het tweede geval is het niet altijd nodig de sleutel die gebruikt wordt om te versleutelen geheim te houden. Deze zou door meerdere partijen kunnen worden gebruikt om een versleutelde boodschap te sturen, mits deze versleuteling wel de eigenschap heeft dat zij alleen met de juiste decodeersleutel kan worden teruggedraaid (het is gebruikelijk de codeersleutel publiek te maken en daarom worden zulke systemen wel "public key" genoemd).

4.5.1 Private Key

One Time Pad

De enige werkelijk veilige methode voor versleuteling waarbij beide partijen een sleutel delen is ook een bijzonder simpele, de zogenoemde one-time pad. Hierbij kiezen beide partijen een string random gegenereerde bits als sleutel die net zo lang is als de te coderen boodschap. Vervolgens wordt de versleutelde boodschap berekend door bitsgewijs de XOR te nemen met de sleutel. De ontvangende partij neemt bitsgewijs de XOR met dezelfde sleutel en krijgt de oorspronkelijke boodschap terug. Omdat de XOR van een random bitstring met IEDERE bitstring weer een random bitstring oplevert, kan een afgeluisterde boodschap nooit worden gedecodeerd. Nadeel van deze methode is wel dat een zeer lange sleutel eerst via een veilig kanaal moet worden uitgewisseld. In de praktijk is deze methode daarom onbruikbaar.

kortere sleutels

Hoe lang de sleutel moet zijn om (enige) bescherming te bieden hangt af van de gebruikte versleutelingsalgoritme en van de computationele kracht van de tegenstander. Sleutels worden daarom langer naarmate die computationele kracht (en de eigen computationele kracht) groeit, en versleutelingsalgoritmen worden ingewikkelder. Aangezien het uitwisselen van geheime sleutels een zeldzame en dure operatie is, wordt dezelfde sleutel vaak hergebruikt, veelal zelfs in één en dezelfde boodschap. Deze boodschap wordt daarom onderverdeeld in blokken van hetzelfde aantal bits, waarop de versleuteling wordt toegepast. Vanwege deze onderverdeling worden dergelijk algoritmen „block cypers" genoemd. Er zijn veel block cyphers bekend[1], en veel van deze zijn gebaseerd op een zogenoemd Feistel netwerk beschreven door Horst Feistel van IBM in 1973. Een Feistel netwerk is een algoritme die een blok data van een bepaald aantal bits versleutelt in een aantal rondes, waarbij de volgende bewerking in iedere ronde wordt toegepast.

1. verdeel het blok in twee helften;
2. pas een rondefunctie F toe op de rechterhelft;
3. neem de XOR van de versleutelde rechterhelft en de linkerhelft; Dat is de nieuwe linkerhelft.
4. wissel de linker en rechterhelft om.

Ontsleuteling van de boodschap gebeurt door het toepassen van de versleutelingsalgoritme in omgekeerde volgorde, met gebruikmaking van F^{-1} als rondefunctie.

4.5.2 Sleutel Delen

Een van de eerste benodigdheden bij het gebruik van een private key cryptosysteem is het delen van een sleutel. Partijen A en B kunnen zo'n sleutel natuurlijk vantevoren afspreken, vervolgens uit elkaar gaan en de sleutel gaan gebruiken, maar geen enkele sleutel kan veilig onafgebroken worden gebruikt. Regelmatig moet een sleutel worden vernieuwd om het systeem veilig te houden. Nieuwe sleutels kunnen natuurlijk worden uitgewisseld door elkaar fysiek te ontmoeten en een nieuwe sleutel af te spreken. Dit is echter altijd duur en lang niet altijd uitvoerbaar. Ook kan natuurlijk zo lang de eerste sleutel nog veilig is, een nieuwe sleutel worden aangemaakt en gecodeerd worden verzonden. Dit heeft echter het nadeel dat een deel van de veiligheid die de sleutel heeft wordt aangewend om een nieuwe sleutel te versturen (en dus niet voor het coderen van boodschappen) en houdt bovendien het risico in dat, omdat je niet zeker kunt weten wanneer een sleutel niet meer kan worden gebruikt, de sleutel door derden is gevonden *voor*dat een nieuwe sleutel wordt verstuurd (waardoor de nieuwe sleutel al meteen is gevonden).

[1] onder andere: 3-Way,AES, Akelarre, Anubis, Blowfish, C2, Camellia, CAST-128, CAST-256, CMEA, CS-Cipher, DEAL, DES, DES-X, FEAL, FROG, G-DES, GOST, Hasty Pudding Cipher, ICE, IDEA, IDEA NXT, Iraqi, KASUMI, KHAZAD, Khufu and Khafre, Libelle, LOKI89/91, LOKI97, Lucifer, MacGuffin, Madryga, MAGENTA, MARS, MISTY1, MMB, NewDES, Noekeon, RC2, RC5, RC6, REDOC, Red Pike, S-1, SAFER, SEED, Serpent, SHACAL, SHARK, Skipjack, SMS4, Square, TEA, Triple DES, Twofish, XTEA

Diffie en Helman [DH76], naar een idee van Paul Merkle gebruikten in 1976 de volgende methode om een geheime sleutel te delen over een onvelig kanaal. Neem een groot priemgetal p. Een getal a kleiner dan p heet een voortbrenger of primitief element van $1, \ldots, p$ als $\{1, \ldots, p\} = \{a, a^2, a^3, \ldots, a^{p-1}\}$ De machten van a lopen dan (mod p) langs alle getallen modulo p. Voor elk priemgetal p bestaat er minstens één zo'n a. Stel dat er een p, priemgetal, en een a voortbrenger van $1, \ldots, p$ gegeven zijn. Twee partijen A en B kunnen nu als volgt een sleutel uitwisselen over een publiek kanaal. A kiest een willekeurig getal $X_A < p$ en berekent $Y_A = a^{X_A} \mod p$. B kiest een willekeurig getal $X_B < p$ en berekent $Y_B = a^{X_B} \mod p$. De waarden Y_A en Y_B worden vervolgens uitgewisseld, en X_A en X_B worden geheim gehouden. A berekent $(Y_B)^{X_A} \mod p$ en B berekent $(Y_A)^{X_B} \mod p$. Zo krijgen ze allebei dezelfde waarde K in handen. Immers

$$\begin{aligned} K &= (Y_B)^{X_A} \mod p \\ &= (a^{X_B} \mod p)^{X_A} \mod p \\ &= (a^{X_B})^{X_A} \mod p \\ &= (a^{X_A})^{X_B} \mod p \\ &= (a^{X_A} \mod p)^{X_B} \mod p \\ &= (Y_A)^{X_B} \mod p \end{aligned}$$

Omdat andere partijen niet de beschikking hebben over de X waarden is de publiek gemaakte informatie, aangenomen dat het vinden van de discrete logaritme modulo p computationeel ondoenlijk is, waardeloos.

4.5.3 Public Key

Hoewel het vermenigvuldigen van getallen een doel op zich is voor rekenautomaten is het vermenigvuldigen van echt grote getallen zeer veel meer in de belangstelling komen te staan na de introductie van public key cryptosystemen als Rivest-Shamir-Adleman (RSA) uit 1978 [RSA78]. Het principe achter RSA is gebaseerd op de aanname dat de inverse operatie van vermenigvuldiging, het *ontbinden* van een getal in zijn factoren een moeilijke operatie is, vooral wanneer het getal groot is (tegenwoordig zo'n vierhonderd cijfers) en zelf het product is van twee priemgetallen. De best bekende algoritme voor het ontbinden van een getal in factoren is tegenwoordig de zogenoemde *number field sieve* (zie bijvoorbeeld [Pom96]). De snelste methoden tot nu toe zijn echter nog steeds exponentieel en daarom kunnen we bij voldoende grote getallen er vanuit gaan dat het lang duurt om een factorisatie te vinden. Een cryptosysteem dat op deze aanname gebaseerd is werkt als volgt.

De Euler totient functie, $\phi(n)$, is het aantal getallen kleiner dan of gelijk aan n dat relatief priem met n is.

$$\phi(n) = \sharp\{i \mid i < n \wedge ggd(i, n) = 1\}$$

Als n zelf een priemgetal is, dan is dus $\phi(n)$ gelijk aan $n - 1$. Als $n = p \times q$, een getal e relatief priem is met $\phi(n) = (p - 1)(q - 1)$ en d de inverse is van e in $Z_{\phi(n)}$, dan kan het drietal d, e, n gebruikt worden in een public key cryptosysteem, waarbij n en e gepubliceerd worden en d de geheime decodeersleutel is. Immers een plaintext M in Z_n kan gecodeerd worden als $C = M^e \mod n$. De gecodeerde boodschap C kan dan gedecodeerd worden als $M = C^d \mod n = (M^{de} \mod n) = M^1 \mod n = M$, waarbij de tweede gelijkheid uit de stelling van Euler volgt[2] Om ervoor te zorgen dat d niet gemakkelijk uit e kan worden berekend is het echter noodzakelijk dat p en q uit veel cijfers bestaan en dus dat de vermenigvuldiging efficiënt gedaan moet worden. Bovendien moet er voor iedere gebruiker van het systeem een nieuw paar p, q worden aangemaakt en is de algemene regel dat een sleutel voor een coderingssysteem regelmatig moet worden ververst natuurlijk ook van toepassing.

Het recept voor het krijgen van een bruikbaar cryptosysteem is dus.

1. Genereeer twee grote priemgetallen p en q. Meestal worden deze random gekozen omdat de kans een priemgetal te treffen onder de getallen kleiner dan n vrij groot is. Er zijn ongeveer $n/\log n$ priemgetallen kleiner dan n, dus de kans voor een willekeurig getal is al $1/\log n$, maar voor speciale getallen als 3 mod 4 is de kans dat je een priemgetal treft nog aanzienlijk groter.

[2]De stelling van Euler zegt dat voor positieve a en n die relatief priem zijn geldt dat $a^{\phi(n)} = 1 \bmod n$. Dus geldt $M^{de} = M^{1+k\phi(n)} = M$, tenzij $M = p$ of $M = q$.

2. Kies een getal e dat relatief priem is met $(p-1)(q-1)$. Ook hier kun je volstaan met een random getal waarbij je met de uitgebreide algoritme van Euclides razendsnel kunt testen dat je inderdaad een getal hebt gevonden dat relatief priem met $(p-1)(q-1)$ is en ook nog een d kunt vinden die de inverse van e is.

3. Publiceer n en e en gebruik d voor het decoderen.

Een voorbeeldje met kleine getallen is misschien illustratief. **Voorbeeld 4.5.1**: Kies twee priemgetallen

3 en 5, dan is $n = 3 \times 5 = 15$ en $\phi(15) = 2 \times 4 = 8$. De enige twee getallen kleiner dan 8 die relatief priem zijn met 8 zijn 3 en 5 en die zijn hun eigen inverse, zoals bijvoorbeeld blijkt uit de uitgebreide euclidische algoritme:

$$8 = 1 \times 5 + 3$$
$$5 = 1 \times 3 + 2$$
$$3 = 1 \times 2 + 1$$

Dus $1 = 3 - 2 = 3 - (5 - 3) = (8 - 5) - (5 - (8 - 5)) = 2 * 8 - 3 * 5$, waaruit volgt dat $-3 \bmod 8$ de inverse is mod8 van 5 dus 5 is zijn eigen inverse mod8. Inderdaad, als we bijvoorbeeld 7 willen versturen, dan is $7^5 \bmod 15 = 16807 = 7$. Niet elk priemgetal werkt even prettig. □

4.5.4 Priemgetallen

Sinds kort is bewezen dat het voor het testen van het al dan niet priem zijn van een getal een efficiënte algoritme bestaat.[AKS04] Deze algoritme is echter nog niet voldoende uitgekristalliseerd om te hebben geleid tot industriële toepassingen. In de praktijk is het ook voldoende met redelijke zekerheid te weten dat de getallen die gebruikt worden voor het RSA systeem priemgetallen zijn. We zullen daarom alleen kort de meest in gebruik zijnde methode voor het verifiëren van de primaliteit van een getal in deze tekst bespreken. Deze verificatie gaat met behulp van een algoritme die gebruik maakt van een random generator. De algoritme heeft, als hij gebruikt wordt op een invoer die niet een priemgetal is, een redelijke kans te ontdekken dat de invoer niet priem is (ongeveer 1/2). Enige malen herhalen van de algoritme met nieuwe willekeurig getrokken getallen maakt dus de kans dat de invoer een priemgetal is groter (nauwkeuriger, enige malen herhalen van de algoritme maakt de kans dat de algoritme niet de uitvoer *geen priemgetal* zou geven onder de aanname dat de invoer geen priemgetal is kleiner; natuurlijk is de invoer wel of juist niet een priemgetal en is er geen sprake van kansen).

De test is gebaseerd op de stelling van Fermat die (toegepast op priemgetallen) zegt.

Stelling 4.5.1 *Als n een priemgetal is, dan geldt voor elke $b < n$ dat $b^{n-1} = 1 \mod n$.*

De kracht van deze test is dat je kunt bewijzen dat als n geen priemgetal is, dan geldt $b^{n-1} = 1 \mod n$ voor hoogstens de helft van de getallen kleiner dan n. Dit bewijs geven we hier niet, maar kan op veel plaatsen gevonden worden in standaard tekstboeken.

4.5.5 sommen

1. Op veel plaatsen (bijvoorbeeld www.prime-numbers.org) kunnen integers van beperkte grootte gevonden worden die priem zijn, en tevens kunnen zelf ingevoerde getallen op priemaliteit worden getest. Schrijf een programma dat een random getal trekt dat met grote kans priem is, en test het vervolgens op zo'n site. (Getallen van de vorm $3 \mod 4$ zijn vaak priem.) Test het vervolgens met behulp van Fermat's stelling een paar keer).

2. Genereer nog een priemgetal als in de vorige opgave en gebruik deze priemgetallen om een RSA systeem op te zetten.

4.6 Beveiliging van gegevens

Cryptografische methoden kunnen gebruikt worden voor verschillende vormen van gegevensbeveiliging. Computers zijn tegenwoordig vaak via netwerken met elkaar verbonden en wisselen over die netwerken gegevens met elkaar uit. Niet elk pakketje data dat door computers wordt uitgewisseld moet door iedereen die toevallig ook op die lijn aanwezig is kunnen worden gelezen, en als we een pakketje data ontvangen van een computer, dan willen we graag weten van welke computer dat pakketje afkomstig is. We kunnen de computer aan de andere kant van de lijn niet zien dus zouden we graag enige vorm van identificatie verlangen voordat we bijvoorbeeld een opdracht van die computer aanvaarden. Een derde vorm is nog commitment. Stel bijvoorbeeld dat de computer van een beleggingsmaatschappij een aandelenhandelaar opdracht geeft groot in te kopen in een bepaald bedrijf. Vervolgens gaat het slecht met dat bedrijf en de aandelen kelderen. De beleggingsmaatschappij zou nu kunnen beweren dat de opdracht nooit verstuurd is en dat de handelaar deze opdracht zelf verzonnen heeft.

Cryptografische methoden, in het bijzonder de public key cryptosystems, bieden voor al deze problemen een oplossing en we zullen hier schetsen hoe deze oplossingen werken. Het beveiligen van data tegen ongewild lezen door derden is de standaardtoepassing, dus zullen we dat onderwerp niet nogmaals behandelen.

4.6.1 Identificatie

Als we een opdracht van persoon P krijgen om uit te voeren, zouden we graag willen vaststellen dat de persoon P inderdaad is wie ze zegt te zijn. De opdracht zelf zal gecodeerd zijn met onze eigen publiek gemaakte sleutel c_m, dus deze zullen we kunnen decoderen met onze eigen, geheim gehouden, sleutel d_m. Hoe weten we dat de boodschap afkomstig is van P?

P heeft, net als alle andere deelnemers, in het publieke domein een sleutel c_p gedeponeerd, of deze sleutel is daar gezet door een vertrouwde derde partij. Verder heeft P haar eigen geheime decodeersleutel d_p. De identificatie van de boodschap M wordt nu bereikt met het omgekeerde coderingsproces. P verstuurt de boodschap $(P)^{d_p}$ als *onderdeel* van M. P^{d_p} kan nu worden gelezen met behulp van de *codeersleutel* van P, die publiek is $((P^{d_p})^{c_p} = P)$. Niemand anders had deze boodschap kunnen versturen zonder kennis van d_p

4.6.2 Commitment

Een soortgelijk schema kan worden gebruikt voor commitment. De boodschap $M =$„Koop 5000 aandelen” kan worden gecodeerd met de publieke sleutel c_p maar ook ondertekend met de hierbovenbeschreven identificatieprocedure. Aangezien de identificatie uniek is kan de gecodeerde boodschap ook dienen als bewijs dat de boodschap verzonden is. De ontvanger kan immers niet zelf de identificatie geproduceerd hebben.

4.6.3 sommen

1. Gebruik het RSA systeem uit som 2 om een document van een handtekening te voorzien.

Deel II

Complexiteitstheorie

In dit deel van de tekst abstraheren we van specifieke algoritmen voor problemen en kijken naar de complexiteit van de problemen zelf. Om uitspraken te doen over alle mogelijke algoritmen die voor een probleem kunnen bestaan, en de complexiteit daarvan hebben we een eenvoudig maar generiek toepasbaar model voor berekening nodig. Hiervan zijn diverse voorbeelden voorhanden, die allemaal modulo de juiste overhead uitwisselbaar zijn. Vervolgens kunnen we problemen onderverdelen in complexiteitsklassen. De klasse NP en de volledige problemen in die klasse spelen in dit deel een speciale rol.

Hoofdstuk 5

Modellen voor Berekening

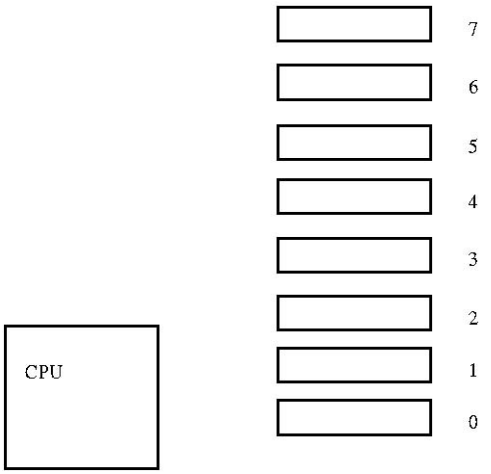
In het eerste deel van deze tekst hebben we een informeel model voor berekening gebruikt. We hebben de complexiteit van algoritmen opgehangen aan zogenoemde spil-operaties (pagina 16) die niet nader genoemde kosten hadden, maar waarvan we aannamen dat ze op een andere computer ook konden worden uitgevoerd voor kosten die niet meer dan een constante keer zo groot zijn, waarbij de aanname dan ook nog was dat die constante beperkt zou zijn. In dit deel van de tekst zullen we wat preciezer worden over het model dat we voor de berekening gebruiken, omdat we het hier over problemen hebben en bewijzen willen gaan geven die voor alle algoritmen gelden die bij die problemen horen. In het bijzonder willen we van problemen graag bewijzen dat er geen efficiënte algoritmen voor bestaan. Natuurlijk mag zo'n bewijs nooit afhangen van de specifieke computer waarop zo'n algoritme wordt uitgevoerd. We zullen dus in het bijzonder een probleem als ondoenlijk classificeren als er geen efficiënte algoritme bestaat voor dit probleem op een *redelijk* machinemodel. Voor effiënte algoritmen hebben we in het eerste deel al betoogd, dat deze klasse beperkt is tot de algoritmen waarvan de tijdgrenzen beperkt zijn tot polynomen. In ieder geval houden we vast dat een algoritme waarvan de rekentijd niet begrensd is door een polynoom niet efficiënt is. In het kader van de machinemodellen zullen we naar analogie een klasse van machinemodellen aanwijzen die elkaar kunnen simuleren in polynomiaal begrensde tijd. Dat wil zeggen dat wat op de ene machine in n stappen kan worden uitgevoerd, kan op de andere machine in $p(n)$ stappen worden uitgevoerd voor p een polynoom van bij voorkeur lage graad. Als een machine bijvoorbeeld in $\log n$ stappen kan doen wat onze machine in n stappen doet, dan zullen we die machine niet zien als een redelijke machine. De uitspraak „alle redelijke machinemodellen simuleren elkaar in polynomiale tijd begrensde overhead en constante factor overhead in ruimte" staat bekend als de *sequential computation thesis*. Omdat dit nogal wat ruimte inneemt zullen we deze uitspraak voortaan aanduiden met „redelijkheidsaanname".

5.1 Redelijke Machinemodellen

Er zijn talloze voorbeelden van redelijke machinemodellen. Twee hiervan zullen we in dit hoofdstuk bespreken. De Random Acces Machine en de Turing Machine.

5.1.1 De Random Access Machine

Een in veel boeken gebruikt model voor berekening is de Random Access Machine, omdat het model veel lijkt op in de praktijk nog steeds gebruikte machines. De Random Access Machine (Figuur 5.1) bestaat uit een centrale verwerkingseenheid, de processor, aangevuld met een onbegrensd geheugen dat bestaat uit een onbegrensd aantal registers. In elk register kan een natuurlijk getal worden opgeslagen. De Random Access Machine heeft een programma bestaande uit genummerde instructies. De machine begint altijd met het uitvoeren van instructie nummer 1 en stopt als zij de instructie HALT tegenkomt. Verder heeft zij de volgende set instructies.



Figuur 5.1: Random Access Machine

LOAD i : Haal wat er in register i zit op en sla dat op in de processor

STORE i : Stop wat er in de processor staat in register i.

ADD i : Tel wat er in de processor staat op bij wat er in het register i staat en sla dat op in de processor.

SUB i : Trek wat er in register i staat af van wat er in de processor staat en sla dat op in de processor.

JUMP a : Spring naar opdrachtregel a.

JGTZ a : Als wat er in de processor staat groter is dan 0 spring naar opdrachtregel a, ga anders door met de volgende opdrachtregel.

Verder kunnen alle getallen in het programma worden voorafgegaan door een # wat een indirectie inhoudt. Dat betekent dat niet dat getal moet worden gebruikt maar de *inhoud* van het register dat door dat getal wordt aangeduid. Bijvoorbeeld $JGTZ\#0$ betekent, als de inhoud van de processor groter dan 0 is, dan spring je naar de opdrachtregel die het nummer draagt van het getal dat in register 0 is opgeslagen. Om het model compleet te maken zal de Random Access Machine niet alleen stoppen als de instructie HALT bereikt wordt, maar ook als er een onzinnige opdracht verwerkt moet worden, bijvoorbeeld het springen naar een opdrachtregel die niet bestaat of het aflagen van een register waar al 0 in staat. Dit is de zogenoemde CRASH opdracht.

Tijd

Een maat voor de tijd die door een Random Access Machine wordt besteed aan het uitvoeren van een programma is de opdrachtregel. Elke uitvoering van een opdrachtregel is één stap. In sommige gevallen wordt ook de lengte van de operanden van ADD en SUB instructies geteld, omdat in een register nu eenmaal een willekeurig groot getal staat en in een realistisch machinemodel niet willekeurig grote getallen in één stap bij elkaar kunnen worden opgeteld.

Geheugen

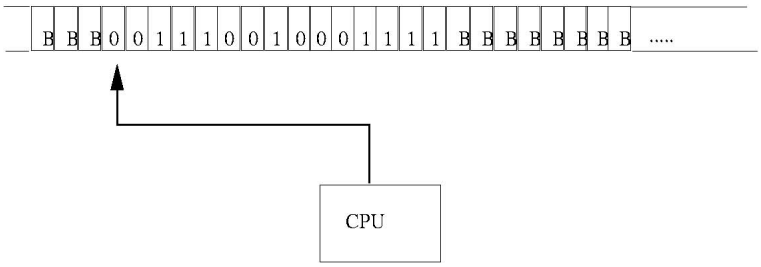
Een maat voor het gebruikte geheugen van een Random Access Machine is de hoeveelheid gebruikte registers. Aan het begin van de berekening is de inhoud van elke register 0 als er een getal in zo'n register wordt geschreven, dan telt vanaf dat moment het register mee in de hoeveelheid gebruikt geheugen. In sommmige gevallen wordt ook de som van de logaritmen van de inhouden van de registers die niet nul zijn geteld als

geheugen. Dit is om rekening te houden met het feit dat in een register een willekeurig natuurlijk getal kan worden opgeslagen.

5.1.2 De Turing Machine

Eén van de eerste modellen voor berekenbaarheid is de Turing machine, voor de eerste keer gepresenteerd in het artikel „On computable numbers, with an application to the Entscheidungsproblem." door Alan M. Turing in 1936 [Tur36]. Vóór de presentatie van het Turing machine model was de vraag wat precies een algoritme of algoritmische bewerking is een centraal onderwerp van discussie. Het verschil tussen algoritmisch en niet algoritmisch denken is onderwerp van filosofisch debat. Turing ging uit van de volgende vooronderstelling. Zie de wiskundige als een automaat die een vel ruitjespapier tot zijn beschikking heeft. Het ruitjespapier is in vier richtingen onbegrensd, maar de wiskundige kan slechts een beperkt aantal ruitjes (zeg één) tegelijkertijd lezen. Als zij een ruitje gelezen heeft, kan zij dat onthouden, een nieuwe waarde in het zojuist gelezen ruitje schrijven en naar een andere plaats op het papier bewegen. Het aantal waarden dat zij kan onthouden is begrensd door één of andere constante.

Tegenwoordig wordt het ruitjespapier vervangen door een tweezijdig onbegrensde band (zie Figuur 5.2). In elke cel van de band kan per bezoek één symbool geschreven worden en het onthouden van een symbool wordt gekarakterizeerd doordat de machine (wiskundige) in een begrensd aantal toestanden kan zijn. Het programma dat de machine uitvoert is dan steeds: „Als in toestand p symbool a gelezen wordt, dan komt de machine in toestand q, schrijft symbool b en beweegt naar links of naar rechts." Het programma eindigt als er een combinatie gelezen symbool/toestand actueel is waarvoor geen opvolger in het programma staat, ofwel het gaat net zolang door tot het niet meer verder kan.



Figuur 5.2: Het standaard Turing machinemodel

Representaties

Formeel bestaat het Turing machine model uit:

1. Een eindige verzameling toestanden Q. Een speciale toestand $q_0 \in Q$ is aangemerkt als de *begintoestand*.
2. Een eindige verzameling bandsymbolen Σ deze verzameling omvat altijd het blanco symbool B.
3. Een toestandsovergangsfunctie $\delta : Q \times \Sigma \mapsto Q \times \Sigma \times \{L, R, \emptyset\}$

Bij aanvang van de berekening bevat de band een aaneengesloten eindige rij niet blanco symbolen en staat het eerste symbool van links op de posititie waar zich de tapekop bevindt. In sommige gevallen wordt de Turing machine nog uitgebreid door een aantal van de toestanden „eindtoestanden" te noemen en af te spreken dat de Turingmachine stopt als zij in één van deze eindtoestanden komt. Dit is echter niet noodzakelijk.

Voorbeeld 5.1.1: Er zijn verschillende manieren om een Turingmachine te representeren, we zullen dit aan de hand van drietal voorbeelden laten zien waarbij in elk voorbeeld het programma gebruikt wordt dat bij zijn invoer die uit 0 en 1 bestaat 1 optelt. Aangezien de kop wegens aanname op het eerste niet blanco

symbool van links staat, moet zij eerst naar rechts tot het einde van de invoer lopen om aan die kant te proberen 1 bij de invoer op te tellen. De Turingmachine begint links op de band bij het eerste symbool. Dat is een 1 of een B. Als het een B is, dan schrijft zij een 1 en stopt. In het andere geval gaat de machine over in toestand q_1 en beweegt naar rechts. In toestand q_1 beweegt de Turingmachine net zo lang naar rechts totdat een B wordt gelezen. Dan keert de machine om en komt in toestand q_2. In toestand q_2 wordt elke 1 onder de kop veranderd in een 0. De éérste 0 of B die tegengekomen wordt, verandert in een 1, waarna de machine stopt.

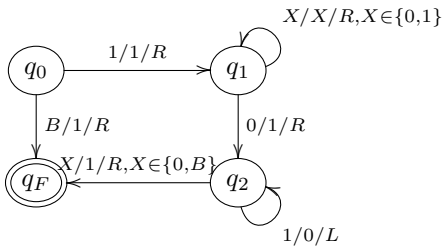
1. Allereerst kunnen we natuurlijk de Turing machine representeren door de toestandsovergangsfunctie expliciet te geven. Dat ziet er als volgt uit.

$$\begin{array}{rcl} \delta(q_0, 1) & = & \langle q_1, 1, R\rangle \\ \delta(q_0, B) & = & \langle q_F, 1, R\rangle \\ \delta(q_1, 0) & = & \langle q_1, 0, R\rangle \\ \delta(q_1, 1) & = & \langle q_1, 1, R\rangle \\ \delta(q_1, B) & = & \langle q_2, B, L\rangle \\ \delta(q_2, 0) & = & \langle q_F, 1, L\rangle \\ \delta(q_2, 1) & = & \langle q_2, 0, L\rangle \\ \delta(q_2, B) & = & \langle q_F, 1, L\rangle \end{array}$$

2. Een andere veelgebruikte manier is het schrijven van de toestanden en de symbolen in een tabel, waarbij de opdrachten de entries van de tabel zijn.

	0	1	B
q_0	–	$q_1, 1, R$	$q_F, 1, R$
q_1	$q_1, 1, R$	$q_1, 0, L$	q_2, B, L
q_2	$q_F, 1, L$	$q_2, 0, L$	$q_F, 1, L$

3. Tenslotte is er nog de grafische manier van representeren.



Al deze methoden zijn equivalent en komen in de literatuur ongeveer even vaak voor. □

Een paar variaties van Turingmachines

In het hieronderstaande stuk over de simulatie van een Turingmachine op een RAM zullen we gaan beschrijven hoe een Turingmachine met een éénzijdig oneindige band op een RAM kan worden gesimuleerd. We hebben echter juist hierboven afgesproken dat de Turingmachineband *tweezijdig* oneindig is. Gelukkig is het Turingmachinemodel een zeer vergevingsgezind model wat betreft wijzigingen in de standaard. De vele bekende variaties hebben allemaal de eigenschap dat ze elkaars berekeningen efficient kunnen simuleren. We bespreken er enkele, waaronder de tweezijdige band op een éénzijdige band. We stellen twee Turingmachines M_1 en M_2 in de onderstaande paragraphen voor. Hier is telkens M_1 de Turingmachine waarvan een berekening op M_2 gesimuleerd gaat worden.

Meer tracks op één tape Als voorbeeld nemen we een tape die uit twee sporen bestaat, een bovenspoor en een onderspoor. De toestandsovergang is nog steeds afhankelijk van de bandinhoud, maar nu worden *twee* symbolen gelezen en geschreven. Het is duidelijk dat elke berekening die op een turingmachine met een ééntracksband kan worden uitgevoerd in dezelfde hoeveelheid stappen ook op een Turingmachine met een tweetracksband kan worden uitgevoerd. De simulerende machine zal immers slechts één van beide tracks gebruiken. De omgekeerde bewering is ook waar. De tweetracks Turingmachine heeft een bepaald bandalfabet Σ. Gegeven M_1 bouwen we een ééntracks Turingmachine M_2 met een groter (maar nog steeds eindig) bandalfabet Γ, zo dat Γ precies $(|\Sigma| \times |\Sigma| + 1/2)$, voor elk paar uit $\Sigma \times \Sigma$ één. We hebben nu een codering voor elk paar symbolen uit $\Sigma \times \Sigma$, en kunnen deze code gebruiken om het programma van M_1 te vertalen.

Eénzijdig oneindige band Nu we hebben vastgesteld dat meerdere tracks op een band op één band gesimuleerd kunnen worden kunnen we dit gebruiken om een tweezijdig oneindige band op een éénzijdige band te simuleren. Hiertoe markeren we het begin van de band met een speciaal teken en gebruiken we verder een éénzijdig oneindige band met twee sporen. Deze tapecel zal in de simulatie niet worden gebruikt, maar alleen worden gebruikt om te zien of in de simulatie van de berekening van M_1 de tapekop van M_1 zich links of rechts van een speciaal aangewezen cel, die we „de oorsprong" zullen noemen bevindt. Het bovenste spoor stelt de linkerkant van de tweezijdig oneindige band voor terwijl het onderste spoor de rechterkant van de tweezijdig oneindige band voorstelt. De toestandsverzameling van M_2 is twee keer zo groot als die van M_1 om aan te geven of de machine links van de oorsprong of rechts van de oorsprong is. Aan één van beide kanten van de oorsprong (bijvoorbeeld rechts) beweegt M_2 zich hetzelfde als M_1 en leest en schrijft het onderste spoort. Als M_1 door de oorsprong naar de linkerkant gaat, dan beweegt M_2 zich steeds in tegengestelde richting en leest en schrijft het bovenste spoor. Zo voeren ze in essentie dezelfde berekening uit.

Meer koppen op een band Een machine met meerdere koppen op een band kan meerdere symbolen tegelijkertijd lezen en interpreteren. Vervolgens kunnen de koppen meerdere symbolen in één stap schrijven en de koppen kunnen tegelijkertijd een beweging naar links en naar rechts uitvoeren. Om dit te simuleren rusten we M_2 uit met een band met twee sporen. In het onderste spoor houden we de symbolen van M_1 bij en in het bovenste spoor zetten we een speciaal symbool, $*$, als we willen aangeven dat één van de gesimuleerde koppen zich op die plaats op de band bevindt. Nu bestaat de simulatie van één berekeningsstap op M_1 uit een veeg van links naar rechts totdat alle informatie over symbolen die onder de gesimuleerde koppen verzameld is. Vervolgens komt een tweede update veeg, waarbij de markers$(*)$ naar links of naar rechts geplaatst worden en de symbolen die onder de markers stonden vervangen worden door nieuwe.

Om te laten zien dat deze simulatie efficiënt is, merken we op dat de koppen van M_1 per stap niet meer dan 2 cellen verder uit elkaar kunnen komen (één kop beweegt naar links en de andere naar rechts), en dus in n stappen niet verder dan $2n$ uit elkaar kunnen komen. Elke veeg kost dus $O(n)$ stappen op M_2 (Misschien is er wat locaal heen en weer geschuif nodig om de markers te updaten). Gevolg is dat simulatie van n stappen op M_1 niet meer dan $O(n^2)$ stappen op M_2 kan kosten.

Meer banden De simulatie van meer banden op 1 band is eigenlijk dezelfde als die van meer koppen met 1 kop. Voor k banden nemen we $2k$ sporen en zetten op de oneven sporen (van onderen af) de symbolen van M_1, terwijl we op de even sporen markers zetten voor de plaats van de koppen. Een paar vegen zorgt, net als in het vorige geval, voor de update van de informatie. Net als in het vorige geval is de ruimte die een veeg moet overbruggen ook weer begrensd door $O(n)$, dus kost de simulatie van n stappen ook hier niet meer dan $O(n^2)$.

Tweedimensionale band De tweedimensionale band laat zich op de 1 dimensionale band simuleren door een handige nummering van de cellen van het platte vlak. Er zijn verschillende oplossingen voor het probleem denkbaar.

		91	90	89	88	87	86	85	84	83	82		
		92	57	56	55	54	53	52	51	50	81		
		93	58	31	30	29	28	27	26	49	81		
		94	59	32	13	12	11	10	25	48	79		
		95	60	33	14	3	2	9	24	47	78		
		96	61	34	15	4	1	8	23	46	77		
		97	62	35	16	5	6	7	22	45	76		
		98	63	36	17	18	19	20	21	44	75		
		99	64	37	38	39	40	41	42	43	74		
			65	66	67	68	69	70	71	72	73		

Figuur 5.3: De spiraalmethode voor de tweedimensionale band

a	*t*	*v*	*e*	*r*	*o*	*e*	*o*	*s*	*e*
	↑								
t	*a*	*c*	*c*	*u*	*s*	*a*	*m*	*u*	*s*
								↑	
e	*t*	*i*	*u*	*s*	*t*	*o*	*o*	*d*	*i*
		↑							

Figuur 5.4: veel banden op één band

1. We kunnen voor elke cel op M_2 een aantal aangrenzende cellen reserveren waarin we de coordinaten van deze cel in de band van M_1 bijhouden. Aangezien deze coordinaten op M_1 niet meer dan met 1 per stap kunnen groeien volgt dat deze coordinaten in n stappen niet meer dan $\log n$ tapecellen in beslag kunnen nemen, dus dat de tape van M_1 niet meer dan $n \log n$ groot zal zijn. Bijgevolg kost de update per stap niet meer dan $n \log n$. De simulatie van n stappen kan dus in $O(n^2 \log n)$.

2. We kunnen een spiraalmethode voor de nummering van de cellen in het platte vlak bijhouden.

3. We kunnen het platte vlak onderverdeeld denken in stroken. Zo houden we een vierkant bij op de band van M_2 waarin de simulatie zich afspeelt. Telkens wanneer M_1 zich buiten het vierkant begeeft, worden alle stroken met 1 cel uitgebreid.

5.1.3 Simulaties tussen RAMS en Turingmachines

Turingmachines en RAMs kunnen elkaars berekeningen uitvoeren, waarbij de overhead zowel in tijd als in geheugen begrensd is.

Een Turingmachineberekening op een RAM

Bij de simulatie van een Turingmachineberekening op een RAM maken we gebruik van de indirecte adresseermogelijkheid van de RAM. De machine die we simuleren is een éénbands Turingmachine met een halfoneindige band. De inhoud van de cellen van de Turingmachine slaan we op in registers $1, 2, \ldots$, en in register 0 houden we een getal bij dat de positie van de kop betekent. Een beweging van de Turingmachine naar rechts simuleren we door de inhoud van register 0 met 1 te verhogen, en een beweging naar links door de inhoud van het register met 1 te verlagen. De bandalfabetsymbolen worden 1 op 1 afgebeeld naar natuurlijke getallen. Het schrijven van een nieuw bandsymbool onder de kop kan gebeuren door een STORE $*0$ instructie en het lezen van een bandsymbool door een LOAD $*0$ instructie. Verder worden toestandsovergangen gesimuleerd door kleine stukjes programma van de volgende vorm:

$q:$	LOAD	$*0$
$q+1$	JZERO	Q_{i_0}
$q+2:$	SUB	1
$q+3:$	JZERO	Q_{i_1}
$q+4:$	SUB	1
$q+5:$	JZERO	Q_{i_2}
$q+..:$	$\ldots$	
$q+2k-1:$	SUB	1
$q+2k:$	JZERO	$Q_{i_{k-1}}$
$q+2k+1:$	JUMP	Q_{i_k}

Als het register geladen wordt, dan bevindt tenminste één van de k symbolen uit het bandalfabet (inclusief het blanco symbool, dat wordt gesymboliseerd door de 0, eerste instructie) zich in het register. Door het register telkens af te lagen en naar de juiste plaats in het programma te springen als de inhoud 0 geworden is, kunnen we een stap van de Turingmachine simuleren. Op de overeenkomstige plaats in het RAM programma wordt vervolgens de juiste waarde met indirecte adressering in het juiste register opgeslagen en wordt tapekop beweging gesimuleerd door register 0 aan te passen. De simulatie van een stap van de Turingmachine kost een constant aantal stappen op de RAM (begrensd door de grootte van het bandalfabet), derhalve kan de simulatie van n stappen van de Turingmachine in $O(n)$ stappen op de RAM gedaan worden.

Een RAM berekening op een Turingmachine

Bij de simulatie van de RAM op de Turingmachine, moeten we de inhoud van de registers, van de RAM opslaan op de band van de Turingmachine. We gebruiken een driebands Turingmachine voor de simulatie. Op de eerste band houden we de inhoud van de registers bij in de vorm $\#\#b_1 b_2 \ldots b_n \# b_{n+1} \ldots b_m \#\#$, waarbij de eerste serie bits een binaire representatie is van het registeradres, en het tweede adres de inhoud

van het register voorstelt. Een LOAD instructie is nu het kopiëren van een gedeelte van de inhoud van band 1 naar band 2, een STORE instructie het omgekeerde, waarbij het opzoeken van het juiste adres met een bit voor bit vergelijking kan gebeuren. De diverse vormen van JUMPs houden een toestandsverandering in, de Turingmachine gaat naar het overeenkomstige deel van zijn programma. ADD en SUB kunnen op een Turingmachine worden uitgevoerd, hiervoor is een implementatie van de overeenkomstige algoritme in Turingmachinecode noodzakelijk, maar dit kan lineair in het aantal bits van de inhoud van het register. De duurste operatie heeft plaats wanneer aan een register een extra bit moet worden toegevoegd. De rest van de band moet dan worden opgeschoven. Omdat we een extra band daarvoor hebben, kan dit gebeuren met een aantal bewerkingen dat begrensd wordt door de lengte van de band, het totaal aantal cellen dat in gebruik is. Om hier een afschatting van te krijgen bedenken we het volgende. Het aantal registers dat in gebruik is, is natuurlijk begrensd door het aantal stappen dat gesimuleerd moet worden. De inhoud van de registers wordt begrensd doordat in elke stap ten hoogste de inhoud van twee registers bij elkaar opgeteld kan worden. Aan het begin van het programma zijn de enige aanwezige getallen de getallen die in het programma zijn opgeschreven het grootste getal is daardoor begrensd door één of andere constante c. In n stappen is dus het grootste getal dat we kunnen maken c^n. Dit getal kan in $O(n)$ bits gerepresenteerd worden.

Het gevolg is dat de duurste operatie, het opschuiven van de band, $O(n)$ stappen op de Turingmachine kost. Bijgevolg kunnen n operaties van de RAM in $O(n^2)$ stappen op de Turingmachine worden gesimuleerd.

5.2 Onredelijke Machinemodellen

Dat we een klasse van redelijke machinemodellen invoeren, doet vermoeden dat er ook een klasse van onredelijke machinemodellen bestaat. We zullen enige aandacht hieraan besteden, hoewel deze modellen niet in deze tekst centraal staan.

5.2.1 Onbegrensd Parallellisme

De eerste klasse van onredelijke machinemodellen is die van de machines met onbegrensd parallellisme. De hoeveelheid hardware in het universum is begrensd en het zou redelijk zijn te vooronderstellen dat we in polynomiaal begrensde tijd slechts polynomiaal veel hardware aan het werk kunnen zetten. Dat ligt echter niet in het standaardmodel opgesloten. Als we beschikken over een onbegrensd aantal processoren, dan zou je je een berekening kunnen voorstellen die begint met een processor 0 die twee andere processoren aan het werk zet, in de tweede stap van de berekening zetten die twee andere processoren dan elk weer twee nieuwe processoren aan het werk enzovoort. In polynomiaal veel tijd kun je dan exponentieel veel processoren aan het werk hebben. Een Random Access Machine met deze eigenschap wordt Parallel RAM (PRAM) genoemd. In het PRAM model hebben alle processoren toegang tot hetzelfde werkgeheugen (de registers) en kunnen ze dus in die registers waarden aan elkaar doorgeven. Zo kunnen exponentieel veel processoren tegelijkertijd aan hetzelfde probleem gezet worden.

Verderop in 7.1 zullen we een klasse van problemen invoeren waarvan het onwaarschijnlijk is dat ze in redelijke (polynomiale) tijd op een redelijk machinemodel kunnen worden berekend. Standaardprobleem in deze klasse is de klasse van ware boolese formules met kwantoren, QBF. De problemen in deze klasse bestaan uit proposities $F(x_1, \ldots, x_n)$ voorafgegaan door afwisselende kwantoren $\exists$ en $\forall$. De vraag is voor een gegeven formule $Q_1 x_1 Q_2 x_2 \ldots Q_n x_n F(x_1, \ldots, x_n)$ of zij waar is of niet. Een PRAM kan zoiets gemakkelijk uitrekenen. Neem voor het gemak even aan dat $Q_i = \exists$ als i even is en $Q_i = \forall$ als i oneven is. Het programma van processor P_i doet dan het volgende:

1. Als $2^n \leq i \leq 2^{n+1}$ dan beschouwt P_i de laatste n bits van zijn index als n binaire waarheidswaarden voor $x_1, \ldots, x_n$. Die vult zij in in $F(x_1, \ldots, x_n)$ en ziet of deze toewijzing F waar maakt. Is dat het geval dan schrijft zij een 1 in register i, en anders een 0.

2. Als $i < 2^n$, dan schrijft P_i de waarde -1 in registers P_{2i} en P_{2i+1} en start P_i de processoren P_{2i} en P_{2i+1} vervolgens leest zij elke stap registers P_{2i} en P_{2i+1} uit totdat daar 0 of 1 in staat. De machine start doordat P_1 deze actie uitvoert.

3. Als i een oneven getal is dan schrijft P_i een 1 in register i als registers $2i$ en $2i+1$ allebei een 1 krijgen, anders 0. Als i een even getal is, dan schrijft P_i een 1 in register i als tenminste één van $2i$ en $2(i+1)$ een één krijgen. Dwz P_i berekent dus de $\forall$ resp de $\exists$ van de waarden van P_{2i} en P_{2i+1}

De PRAM berekent zo de waarde van een QBF uit en gebruikt daarvoor ongeveer $O(n)$ stappen.

5.2.2 Oneerlijk tellen

Van Random Access Machines hebben we gezegd dat in elke cel van het geheugen een willekeurig natuurlijk getal kan worden opgeslagen. Aangezien er oneindig veel natuurlijke getallen bestaan is het natuurlijk onredelijk om aan te nemen dat hele grote natuurlijke getallen in één geheugencel kunnen worden opgeslagen. In een bandcel van een Turingmachine kan immers maar één enkel symbool worden opgeslagen. Omdat in een polynomiaal aantal stappen niet al te grote getallen kunnen worden gemaakt door een RAM, is het niettemin mogelijk de berekeningen van een polynomiale tijd begrensde RAM op een polynomiale tijd begrensde Turing machine te simuleren, zoals we hebben gezien in 5.1.3. Dat komt omdat een enkele optelling niet veel meer kan doen dan een getal verdubbelen, en verdubbelen van een getal geeft slechts één extra bit over een binair bandalfabet.

Anders wordt de situatie als we de RAM ook laten vermenigvuldigen *en* voor de vermenigvuldiging van willekeurig grote getallen slechts één operatie in rekening brengen. Dan kunnen de aanwezige getallen met elkaar vermenigvuldigd worden en kan in n stappen het getal 2^{2^n} gemaakt worden. Dit getal neemt exponentiële ruimte in als het gerepresenteerd moet worden op een Turingmachineband en derhalve kan RAM die in één stap vermenigvuldigingen van willekeurig grote getallen kan uitvoeren niet in polynomiale tijd door een Turing machine worden gesimuleerd. Dit model van Random Access Machine noemen we Multiplication RAM of ook wel MRAM.

Net als bij de PRAM wordt kan van de MRAM worden aangetoond dat deze een berekening van een machine die mogelijk veel krachtiger is dan de polynomiale tijd begrensde Turing machine kan simuleren in polynomiale tijd. Het gaat om de polynomiaal geheugen begrensde machine. Een Turing machine heeft een accepterende berekening als de initiële configuratie bij een bepaalde invoer in minder dan exponentieel veel stappen een accepterende configuratie kan bereiken. Dit kan worden aangetoond door alle configuraties in een matrix te schrijven en te laten zien dat deze matrix wanneer hij ongeveer exponentieel vaak met zichzelf vermenigvuldigd wordt een 1 krijgt op de plaats die het pad van de beginconfiguratie naar de accepterende configuratie aangeeft. Een MRAM kan zo'n matrixvermenigvuldiging in één stap doen. Omdat niet alle tussenliggende matrices hoeven te worden berekend—telkens de gevonden matrix kwadrateren is voldoende—hoeven er niet exponentieel *veel* vermenigvuldigingen gedaan worden (de matrix moet nog wel als één getal in een register worden voorgesteld, en matrixvermenigvuldiging moet nog worden aangepast, maar dat zijn details), maar slechts polynomiaal veel. Details van dit bewijs zijn, zoals de lezer zich misschien kan voorstellen, behoorlijk ingewikkeld (zie [HS76]).

Voor zowel de MRAM als de PRAM is het onbekend of er ook werkelijk een *probleem* is dat door een MRAM kan worden opgelost (taal die kan worden herkend) dat niet in polynomiale tijd door een Turing machine kan worden opgelost. Immers, voor een vermenigvuldiging kan een recursieve procedure worden opgeschreven waarvan de te bewerken onderdelen slechts half zo groot zijn als de oorspronkelijke. In polynomiaal veel recursieve stappen kunnen dus exponentieel grote getallen worden opgeknipt tot onderdelen van lengte 1 die dan gemakkelijk ook op een Turingmachine met elkaar vermenigvuldigd kunnen worden. In totaal kan het geheugengebruik worden beperkt tot polynomiale afmetingen, omdat de stukken kort zijn en de recursiediepte beperkt. Het is dus niet op voorhand uitgesloten dat de berekening van een MRAM kan worden gesimuleerd door een Turingmachine in polynomiaal geheugen. Omdat het probleem of polynomiaal geheugenbegrensde Turingmachines gesimuleerd kunnen worden door polynomiale tijdbegrensde Turingmachines nog steeds open is, kunnen we dus ook niet zonder meer zeggen dat de MRAM een onredelijk machinemodel is.

5.2.3 Sommen

1. Pas het programma van de Turingmachine uit voorbeeld 5.1.2 zo aan dat het werkt op een machine met half-oneindige band.

2. Hoeveel kost de simulatie van één stap van machine M_1 die een tweedimensionale band heeft op machine M_2 als we de spiraalmethode van nummering van cellen gebruiken? Wat als we de strokenzaagmethode gebruiken (in dit geval is een analyse van de uitgesmeerde complexiteit aan de orde).

3. Een EDIT RAM is een random access machine die kan werken met textfiles en de volgende operaties heeft:

   (a) Een symbool uit een file lezen bij een textpointer (cursor)

   (b) Een symbool in een file schrijven op de plaats van een textpointer

   (c) Een texpointer aan het einde van een file schrijven

   (d) Een texpointer op een plaats in de file zetten die overeenkomt met een getal in een register

   (e) Vervangen van *text1* door *text2* overal tegelijkertijd in een file

   (f) Aan elkaar plakken van files

   (g) Delen van textfiles copiëren aan de hand van de positie van textpointers

   (h) Delen van files verwijderen die door de positie van textpointers bepaald worden.

   Al deze operaties kunnen in één stap door de Edit-RAM gedaan worden. Laat zien dat de Edit-RAM in polynomiale tijd de waarheid van een QBF kan bepalen.

Hoofdstuk 6

Centrale Complexiteitsklassen

Alle redelijke machinemodellen kunnen elkaar simuleren in Polynomiale Tijd, is onze centrale aanname waar het machinemodellen betreft. Als we dus uitspraken willen doen over problemen en de complexiteitsklassen waar ze in thuis horen, dan zullen we in ieder geval alle problemen die in complexiteit niet meer dan een polynoom verschillen in dezelfde complexiteitsklasse moeten plaatsen. Deze complexiteitsklasse noemen we P. De klasse P zal als het ware de 0 vormen van onze voorlopige hierarchie van complexiteitsklassen. Een plafond zullen we vooralsnog niet definiëren, maar om een speelbaar veld van complexiteitsklassen te verkrijgen zullen we minstens moeten zien dat er een complexiteitsklasse bestaat die niet gelijk is aan P. In Hoofdstuk 1 van Deel I maakten wij onderscheid tussen problemen waarvoor polynomiaal begrensde algoritmen bestaan en problemen waarvoor alleen exponentiële tijd begrensde algoritmen bekend zijn. Dat was uiteraard niet toevallig. We zullen zien dat de klasse van problemen waarvoor alleen exponentiële tijd begrensde algoritmen bekend zijn, die we EXP zullen noemen, ook problemen bevat waarvoor *geen* polynomiale tijd begrensde algoritmen *bestaan* en dus dat deze klasse *werkelijk* verschilt van de klasse P.

Als machinemodel nemen we de Turingmachine. Er zijn twee duidelijke complexiteitsmaten aan een Turingmachineberekening te geven, tijd en geheugen. Een variant van de Turingmachine, de zogenoemde nondetermistische Turingmachine zal verderop in het verhaal ook een rol spelen. Bij een nondeterministische Turingmachine is er geen sprake van een toestandsovergangsfunctie, maar van een toestandsovergangsrelatie. Bij een bepaalde configuratie zijn er meerdere vervolgconfiguraties mogelijk. Een nondeterministische Turingmachine heeft op een bepaalde invoer meerdere *mogelijke* berekeningen. We spreken af dat een nondeterministische Turingmachine de invoer x accepteert als er een berekening op x *mogelijk* is die accepteert. Dit heeft het voordeel dat we expressies als $(\exists x)[P(x)]$ met een algoritmisch model kunnen representeren. Vele vragen uit de AI, bijvoorbeeld de vraag of er een pad is waarlangs de robot van A naar B kan komen, kunnen dan als algoritmisch probleem worden gesteld. Doorgaans is dan de nondeterministische algoritme van geringe complexiteit. Er hoeft alleen aangetoond te worden dat de oplossing bestaat *gegeven de oplossing*. Dat voor het vinden van de oplossing vaak niets beters te vinden is dan exhaustive search is er de oorzaak van vele interessante onderzoeksgebieden in de AI.

Het verband tussen tijd en geheugen is bijzonder interessant. Vaak kunnen we door wat meer geheugen te gebruiken (bijvoorbeeld bij zoeken en sorteren) belangrijke tijdwinst boeken. Geheugen is echter ook duurder dan tijd. Tijd krijg je vanzelf meer door te wachten. Geheugen moet je gaan kopen. Tegenwoordig hebben we vaak de beschikking over hoeveelheden geheugen waarvan we vroeger slechts konden dromen. Databases van meer dan een petabyte zijn geen uitzondering meer en schijven van terabytes zijn zo langzamerhand retail artikelen. Op zoveel detail zullen we hier echter niet ingaan. We zullen net als met tijd ook de geheugencomplexiteit beschouwen als functie van de lengte van de invoer. Polynomiaal geheugen wordt dan aangegeven met PSPACE. Ook hier is een nondeterministische variant, maar in 7.2 zullen we de opmerkelijke stelling tegenkomen dat voor geheugengebruik het nondeterministische model geen extra rekenkracht geeft. Het is duidelijk dat een grens op het geheugen liberaler is dan een zelfde grens op de tijd. Immers in één stap kunnen we niet meer dan één nieuwe geheugencel aanspreken. Derhalve kunnen we in polynomiaal geheugen altijd minstens evenveel als in polynomiale tijd. Aan de andere kant heeft een Turingmachine, omdat

deze een eindig aantal toestanden, en een eindig alfabet heeft niet meer dan exponentieel veel verschillende configuraties op een bepaalde geheugengrens. Vandaar dat in exponentiële tijd minstens net zoveel gedaan kan worden als in polynomiaal geheugen.

Vanwege de simulatie van machinemodellen is wat betreft geheugencomplexiteit logarithmisch geheugen de kleinste maat die onafhankelijk van het machinemodel gepresenteerd kan worden. Complexiteitsklassen die een centrale rol spelen noteren we in Figuur 6.1 in oplopende volgorde van inclusie. De klasse volgend op deze hierarchy is PRIM, de klasse van primitief recursieve functies, waarin tijd en geheugengrenzen worden losgelaten.

LOGSPACE	logaritmisch begrensd geheugen
NLOGSPACE	nondeterministisch logaritmisch begrensd geheugen
P	polynomiaal begrensde tijd
NP	nondeterministisch polynomiaal begrensde tijd
PSPACE	polynomiaal begrensd geheugen
EXP	exponentieel begrensde tijd
NEXP	nondetermistisch exponentieel begrensde tijd
EXPSPACE	exponentieel begrensd geheugen
DEXP	dubelexponentieel begrensde tijd ($2^{2^{p(n)}}$)
$\vdots$	$\vdots$
ELEMENTARY	tijd begrensd door $\underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}}_{n}$

Figuur 6.1: Complexiteitsklassen

6.1 Polynomiale en Exponentiële Tijd

Om aan te tonen dat er problemen zijn die wel in exponentiële tijd op te lossen zijn, maar niet in polynomiale tijd, moeten we tenminste één probleem definiëren dat op geen enkele machine in polynomiale tijd is op te lossen. Wegens onze aanname dat redelijke machines elkaar in polynomiale tijd kunnen simuleren, zijn we in de gelukkige omstandigheid dat we dat alleen maar voor Turing machines hoeven te doen. Dan nog hebben we een probleem. Er bestaan oneindig veel Turing machines en er bestaan oneindig veel polynomen, dus hoe maken we een probleem dat door geen van die oneindig veel Turing machines in tijd begrensd door één van die polynomen kan worden opgelost? De oplossing hiervoor is codering, of „de Universele Turing Machine". We maken *een* Turing machine die *elke* andere Turing machine kan simuleren in tijd begrensd door een polynoom in de tijd waarin de te simuleren machine de berekening volooit. Dan hoeven we alleen nog maar een probleem te verzinnen dat voor geen enkel polynoom p in tijd begrensd door p op die universele machine kan worden opgelost, en we zijn klaar.

6.1.1 De Universele Turing Machine

Zoals we in 5.1.2 hebben gezien kunnen we een Turing machine programma dat bestaat uit een verzameling toestanden Q en een toestandsovergangsfunctie δ beschrijven door een opsomming van de vorm $\langle q_0, a, q_1, b, R\rangle \ldots$ te geven. We zullen deze opsomming coderen als lijst getallen die straks door onze universele machine gebruikt kan worden om de berekening gedaan door *deze* machine na te spelen. Dus, laat de toestands verzameling $Q = \{q_0, \ldots, q_k\}$ zijn en het bandalfabet $\{0, 1, B\}$. We coderen L als 0, en R als 1. Voor natuurlijk getal i, laat $bin(i)$ de binaire representatie van i zijn. Een instructie $q_i, a \to q_j, b, R$, met $a, b \in \{0, 1, 2\}$, waarin 2 de representatie van het blanco symbool is, kan dan bijvoorbeeld geschreven worden als $bin(i)\#a\#bin(j)\#b\#1$. Als we elke instructie *inst*, op deze manier naar een representatie rep(*inst*) vertalen in het alfabet $\{0, 1, \#\}$ dan kunnen we het hele programma schrijven als $rep(inst_1)\# \ldots repinst_n\#$. Tot slot kunnen we elke 0 vervangen door 00, elke 1 door 11 en elke # door 01 en een 1 voor de hele codering

zetten om een éénduidige afbeelding van Turing machines *in* de natuurlijke getallen te krijgen. Om van deze representatie een bijectie te maken spreken we af dat binair geschreven natuurlijke getallen die niet, als hierboven beschreven, een zinvol Turing machine programma coderen altijd een Turing machine programma coderen dat de lege verzameling herkent. Ook de invoer van een Turing machine kan op deze manier worden meegenomen. Een representatie $rep(prog)$ van een Turing machine programma als boven beschreven kan met een natuurlijk getal i gecombineerd worden asl $rep(prog)\#\#\#bin(i)$. Opnieuw vertalen we 0 naar 00, 1 naar 11 en # naar 01 en krijgen zo een representatie in de binaire getallen van de paren programma,invoer. Een Turing machine, die we de universele Turing machine noemen, kan als zij een getal als invoer op de band aantreft, dit getal op de hier beschreven manier interpreteren als paar (programma, invoer), en het aldus beschreven programma op de invoer simuleren.

Voorbeeld 6.1.1: Als voorbeeld vertalen we het programma uit Voorbeeld 5.1.2 naar een binaire string. De toestandsovergangsfunctie zag er als volgt uit.

$$\begin{array}{rcl}
\delta(q_0,1) & = & \langle q_1,1,R\rangle \\
\delta(q_0,B) & = & \langle q_F,1,R\rangle \\
\delta(q_1,0) & = & \langle q_1,0,R\rangle \\
\delta(q_1,1) & = & \langle q_1,1,R\rangle \\
\delta(q_1,B) & = & \langle q_2,B,L\rangle \\
\delta(q_2,0) & = & \langle q_F,1,L\rangle \\
\delta(q_2,1) & = & \langle q_2,0,L\rangle \\
\delta(q_2,B) & = & \langle q_F,1,L\rangle
\end{array}$$

We hebben te maken met vier toestanden. Deze krijgen representaties $bin(0)$, $bin(1)$, $bin(2)$ en $bin(3)$: 00, 01, 10 en 11. Het bandalfabet is $\{0,1,B\}$, wat we coderen als: 00, 01 en 10. De bewegingen L en R coderen we vervolgens als 0 en 1. De instructies zien er na deze coderingen uit als:

$$\begin{array}{l|rcl}
00\#01\#01\#01\#1 & \delta(q_0,1) & = & \langle q_1,1,R\rangle \\
00\#10\#11\#01\#1 & \delta(q_0,B) & = & \langle q_F,1,R\rangle \\
01\#00\#01\#00\#1 & \delta(q_1,0) & = & \langle q_1,0,R\rangle \\
01\#01\#01\#01\#1 & \delta(q_1,1) & = & \langle q_1,1,R\rangle \\
01\#10\#10\#10\#0 & \delta(q_1,B) & = & \langle q_2,B,L\rangle \\
10\#00\#11\#01\#0 & \delta(q_2,0) & = & \langle q_F,1,L\rangle \\
10\#01\#10\#00\#0 & \delta(q_2,1) & = & \langle q_2,0,L\rangle \\
10\#11\#11\#01\#0 & \delta(q_2,B) & = & \langle q_F,1,L\rangle
\end{array}$$

Nu schrijven we alle instructies achter elkaar en krijgen de string:

00#01#01#01#1#00#10#11#01#1#01#00#01#00#1#01#01#01#01#1
01#10#10#10#0#10#00#11#01#0#10#01#10#00#0#10#11#11#01#0

die we kunnen vertalen naar de binaire rij:

000001001101001101001101110100000111000111110100110111010011
010000010011010000011101001101001101001101001101110
001101110001110001110001000111000100000111110100110 1
000111000100110111000100000100011100011111011111010011 0100

□

Ook begrensde berekeningen kunnen op een dergelijke manier door de universele Turing machine worden uitgevoerd. Als de Turing machine door één of andere functie f in tijd begrensd wordt, dan kunnen we deze functie meecoderen door in plaats van het paar programma#invoer het drietal programma#invoer#f als getal te coderen, mits er voor f een efficiënte codering bestaat. In het geval dat f een polynoom is, dan is alleen het rijtje coëfficiënten van de machten van x nodig om het polynoom éénduidig vast te leggen. De universele Turing machine kan in dat geval beginnen met $f(|x|)$ uit te rekenen op een aparte band en dit

getal na elke gesimuleerde stap met 1 aflagen. Als dit getal gelijk wordt aan 0, is de gesimuleerde berekening geëindigd. We spreken af dat, als de Turing machine bij het bereiken van de 0 op de tijdband nog niet geaccepteerd heeft, de invoer wordt verworpen. Op deze manier krijgen we *geklokte* Turing machines. Het is duidelijk dat er bij hanteren van geschikte tijdgrenzen, geen verschil bestaat in de complexiteitsklassen gedefinieerd door geklokte Turing machines en tijdbegrensde Turing machines. Kijk bijvoorbeeld naar de klasse P. Voor elke taal in P is er een geklokte machine die deze taal herkent. Immers er is een polynomiaal, begrensde machine die die taal herkent. Laat $p(n)$ de tijdgrens zijn. Als we zo'n machine nemen en hem $p(n)$ of groter begrenzen hebben we de gewenste geklokte machine. Omgekeerd is het evident dat elke $p(n)$ geklokte Turing machine een taal in P herkent.

Deze codering en simulatie is een stap die we moeten beschrijven voor het volgende onderwerp, maar is ook een stap waaraan de informaticus natuurlijk al lang gewend is. Elk programma dat de informaticus schrijft wordt in de machine (bijvoorbeeld door de compiler) gerepresenteerd als een lange aaneengesloten binaire rij en dus als een getal. Het aantal manieren waarop dit kan gebeuren is groot. Het is zelfs denkbaar een programma te representeren als een rij met daarin maar één symbool, zo'n representatie noemen we een tally representatie. Echter, de tally representatie heeft als nadeel dat ze exponentieel langer is dan representaties waarin meer dan 1 symbool gebruikt wordt, terwijl deze laatste representaties onderling niet meer dan een constante factor van elkaar in lengte verschillen.

6.1.2 Het Padding Lemma

In de vorige sectie hebben we laten zien dat er een $1-1$-relatie bestaat tussen Turingmachine programma's en binair geschreven natuurlijke getallen. Er bestaat uiteraard ook een $1-1$ relatie tussen Turing machine programma's en functies van de natuurlijke getallen naar $\{0,1\}$ door te zeggen dat het resultaat van de berekening 1 is dan en slechts dan als de Turing machine eindigt in een accepterende toestand (of überhaupt eindigt) en anders 0. Elke functie naar $\{0,1\}$ is de representatie van een deelverzameling van het origineel van die functie. Als het origineel een verzameling strings is, dan representeert zo'n functie een taal, door te zeggen dat alles wat afgebeeld wordt op 1 in de taal zit, en alles wat afgebeeld wordt op 0 niet in de taal zit. Zo kan dus elk natuurlijk getal een taal of een probleem representeren. Deze representatie is echter zeer redundant. Elke taal wordt door *oneindig veel* Turing machine programma's gerepresenteerd. Immers, stel dat programma *prog* een taal L representeert, dan wordt L ook gerepresenteerd door programma $prog\#bin(i)$ voor *elk* natuurlijk getal i. Deze observatie, die in de berekenbaarheidstheorie bekend staat als het Padding Lemma, is een belangrijk hulpmiddel bij een belangrijke techniek om complexiteitsklassen te scheiden, de diagnonalisatie.

6.1.3 P $\neq$ EXP

Om aan te tonen dat EXP werkelijk verschilt van P zullen we een taal maken die door een Turing machine M in exponentiële tijd herkend wordt, maar die door geen enkele Turing machine in polynomiale tijd herkend kan worden. Hiervoor is het handig dat we in de vorige subsectie hebben aangetoond dat er een 1-1 verband bestaat tussen Turing machines en natuurlijke getallen, tussen paren Turing machines-input en natuurlijke getallen en zelfs tussen drietallen Turing machine-input-polynoom en natuurlijke getallen, m.a.w. dat we elk natuurlijk getal kunnen interpreteren als een drietal, Turing machine, invoer, polynoom en vervolgens de executie van die Turing machine op die invoer voor een aantal stappen begrensd door dat polynoom kunnen simuleren. Aangezien elk drietal is af te beelden op een natuurlijk getal krijgen we door alle getallen als invoer te bekijken een opsomming van alle talen in P, c.q. van machines die een taal herkennen. De techniek die we hiervoor gebruiken is die van de diagonalisatie, voor het eerst gebruikt door G. Cantor. Cantor bewees dat er meer reële getallen zijn dan natuurlijke getallen als volgt. Neem aan dat er net zoveel reële getallen als natuurlijke getallen zijn. Dan kun je de reële getallen tussen 0 en 1 op volgorde zetten (zie figuur 6.2). Die reële getallen kun je voorstellen door een decimale punt gevolgd door een oneindige rij nullen en eenen (een binaire breuk). Nu kan ik een nieuw getal maken door langs de diagonaal van die opsomming te lopen en overal waar een 0 staat een 1 noteren en omgekeerd. Mijn nieuwe getal krijgt dus een 1 op plaats i als er een 0 op plaats (i,i) staat in de opsomming en een 0 als er een 1 op die plaats staat. Dat nieuwe getal is

1 0001001001001000100 10.....
2 0101010010100100010 001.....
3 1001011110101111000 01.....
4 1111010010111000101 011.....
5 0101011011101010010 11.....
6 0100100100101010100 11.....
...
diag: 101110.....

Figuur 6.2: Cantor's diagonaal

óók een getal tussen 0 en 1, maar staat *nergens* in die rij. Dus kan zo'n opsomming niet gemaakt worden.

Voordat we diagonalisatie over tijdbegrensde klassen presenteren zullen we de gelegenheid te baat nemen om te laten zien dat er problemen zijn die *helemaal* niet door Turingmachines kunnen worden opgelost, functies die niet kunnen worden uitgerekend, of talen die niet kunnen worden herkend. Zulke problemen heten *onbeslisbaar* of *onberekenbaar* en er zijn er veel van. Je kunt dat op (minstens) twee manieren inzien.

Allereerst geeft Cantor's diagonaalargument een rechtstreeks, maar niet constructief, bewijs. Elke Turingmachine herkent maar één taal, berekent maar één functie of lost maar één probleem op. Dus zijn er hoogstens zoveel berekenbare functies als er Turingmachines bestaan en dat zijn er hoogstens zoveel als er natuurlijke getallen zijn, vanwege onze codering hierboven. Aan de andere kant is elke functie van de natuurlijke getallen naar $\{0,1\}$ op te vatten als een oneindige rij 0'en en 1'en net zoals in Cantor's diagonaalargument, en omgekeerd, bijv. de rij $f(0)f(1)f(2)\ldots = 101\ldots$. Tussen de functies van de natuurlijke getallen naar $\{0,1\}$ moeten dus functies zitten die niet door Turingmachines kunnen worden uitgerekend.

Er is nog een tweede bewijs, door Turing in 1936 gegeven, dat ook een diagonaalargument is, maar meer lijkt op de bewijzen die we ook voor complexiteitsklassen kunnen gebruiken. Kijk naar de taal HALT die gedefinieerd is als paren getallen. $\langle x, y\rangle \in$ HALT als en alleen als het programma x, dwz het getal x opgevat als Turingmachineprogramma, in eindige tijd stopt op invoer y. Er is geen Turingmachineprogramma, dat altijd stopt en het antwoord JA geeft op $\langle x, y\rangle$ als $\langle x, y\rangle$ in HALT zit en NEE als dat niet zo is.

Het bewijs hiervoor lijkt (en is ook) een bewijs uit het ongerijmde dat, als je nauwkeurig kijkt, ook een diagonaalargument is. Het gaat als volgt.

Stel dat er wel zo'n machine zou zijn. Noem hem M_H. Maak dan een nieuwe machine M_D die als volgt werkt.

1: input x
2: Simuleer M_H op invoer $\langle x, x\rangle$
3: **if** M_H zegt JA **then**
4: Loop
5: **else**
6: Stop
7: **end if**

M_D is ook een Turingmachine, dus is er één of ander getal x_D dat het programma van M_D codeert, maar kijk wat er gebeurt als M_D invoer x_D krijgt. Er zijn twee mogelijkheden: M_D stopt op invoer x_D, dat kan als M_H NEE zegt tegen invoer $\langle x_D, x_D\rangle$, maar dan heeft M_H het fout. Dus mag M_D niet stoppen op invoer x_D, maar dat kan alleen als M_H JA zegt tegen $\langle x_D, x_D\rangle$ en dus heeft M_H het dan ook fout. De conclusie moet zijn dat een machine M_H als door ons gedroomd niet kan bestaan.

Ons bewijs dat er een taal in EXP is die niet in polynomiale tijd herkend kan worden loopt ook over de diagonaal van Turingmachine/polynoom en invoer paren, en is eigenlijk hetzelfde bewijs als hierboven, met meer administratie.

We stellen ons voor dat we een figuur met een x en een y as hebben waarlangs de natuurlijke getallen staan. Langs de x as interpreteren we de getallen als de combinatie van natuurlijk getal/polynoom, en langs de y as interpreteren we de getallen gewoon als natuurlijke getallen. De machine die we voorstellen krijgt een paar x, y als invoer en voert het programma in Figuur 6.3 uit.

1: input x;
2: schrijf op een aparte band $2^{|x|+|y|}$ keer een 1.
3: Laag het aantal 1 af en vervang in de volgende stappen steeds een 1 door een B. Als deze band tussentijds leeg raakt, stop en verwerp.
4: decodeer x in P en q waar P een Turing machine programma is en q een polynoom;
5: simuleer $\{P\}$ op x voor $q(|x|)$ stappen.
6: **if** $\{P\}$ heeft nog niet geaccepteerd **then** accepteer
7: **end if**
8: Verwerp;

Figuur 6.3: Een taal in EXP − P

We beweren enerzijds dat dit programma voor invoer lengte n in tijd $O(2^n)$ loopt en anderzijds dat er geen enkel programma M, en polynoom p bestaat, zo dat M precies de invoeren accepteert die dit programma accepteert in tijd begrensd door p. De eerste bewering is evident, aangezien we een geklokte Turing machine gebruiken die na ongeveer 2^n stappen stopt. Deze Turing machine moet daarvoor eerst wel 2^n berekenen, maar dat kan gemakkelijk in 2^n stappen (hoe?). We nemen nog iets meer afstand en nemen aan dat q een polynoom is zo dat elke n stappen van M op invoer van lengte n in $q(n)$ stappen gesimuleerd kunnen worden. Stel nu dat er een machine M is en een polynoom p, zo dat M de bovenbeschreven taal herkent in hoogstens $p(n)$ stappen. Deze machine M heeft een programma dat als binaire string gecodeerd kan worden. Wegens het padding lemma is er een codering P voor dit programma, zo dat $q(p(n))$ (veel) kleiner is dan 2^n. Dus als we het programma P op invoer van lengte n simuleren, dan stopt het programma *altijd*. Laten we eens zien wat M met een getal als invoer dat het paar P, p codeert. Het voert P uit op invoer P, p voor $p(|P, p|)$ gesimuleerde stappen, wat dus minder dan $q(p(|P, p|)) < 2^{|P,p|}$ werkelijke stappen kost. Maar dan accepteert zij de invoer P, p dan en slechts dan als P de invoer P, p verwerpt. Onze aanname dat P, p (altijd) door $p(n)$ begrensd is moet dus fout zijn.

6.1.4 sommen

1. Een functie f heet *tijdconstrueerbaar* als er een Turingmachine bestaat en voor elke n een invoer van lengte n waarop M precies $f(n)$ toestandsovergangen doormaakt en dan stopt. Laat zien dat de volgende functies tijdconstrueerbaar zijn. Laat zien dat n, n^2, 2^n en $n!$ tijdconstrueerbaar zijn. Zijn er ook functies die *niet* tijdconstrueerbaar zijn?

2. Toon aan dat het programma uit Figuur 6.3 zo kan worden aangepast dat ermee bewezen wordt dat er een taal bestaat die wel in tijd $O(n^{\log n}$ herkend kan worden maar niet in polynomiale tijd.

3. Pas het programma zo aan dat ermee bewezen wordt dat er een taal is die in geheugen n^2 herkend kan worden die niet in geheugen $O(\log n)$ herkend kan worden.

6.2 NP Problemen

Exponentiële tijd begrensde algoritmen zijn alleen interessant voor zeer kleine instanties. Meestal komt het uitvoeren van een exponentiële tijd begrensde algoritme neer op het bekijken van alle mogelijke oplossingen van een probleem en daaruit de goede of meest geschikte oplossing te selecteren. Wij kennen deze procedure ook wel onder de naam „exhaustive search" of, in het Russisch, „perebor". Het bewijzen dat een probleem alleen kan worden opgelost door het toepassen van exhaustive search wordt in het algemeen geaccepteerd als een bewijs dat het probleem ondoenlijk is en dat alleen zeer kleine instanties van het probleem kunnen worden opgelost. Ook wordt zo'n bewijs geaccepteerd als een excuus om te zoeken naar algoritmen die niet in alle gevallen een oplossing voor het probleem geven en/of slechts een benadering van de optimale oplossing. De problemen die bewijsbaar niet kunnen worden opgelost vallen buiten het bereik van deze tekst.

Problemen waarvoor zo'n bewijs niet bestaat, maar wel een efficiënte benaderingsalgoritme komen verderop in deze tekst nog wel aan de orde.

We komen eerst toe aan de identificatie van een klasse van problemen die er alle schijn van hebben dat ze tot de klasse van zeer moeilijke problemen horen, maar waarvoor tot op heden geen bewijs bestaat dat dat zo is. Van al deze problemen weten we dat ze kunnen worden opgelost met exhaustive search. Tot op heden weten we echter niet of ze *alleen* maar met exhaustive search kunnen worden opgelost. Deze problemen zijn sterk aan elkaar verwant en de vraag of exhaustive search de enige mogelijke oplossing is, is bovendien dringend. Het zijn namelijk problemen die in de praktijk vaak voorkomen. Voordat we de eigenschappen van deze problemen in detail gaan ontleden, bekijken we eerst een aantal voorbeelden. Hiervoor gebruiken we een vast formaat. Eerst geven we de naam waaronder het probleem bekend is, vervolgens beschrijven we de invoer van het probleem zoals deze aan de Turing machine wordt voorgesteld en tenslotte stellen we de vraag die door de Turing machine moet worden beantwoord. Dit antwoord is altijd van de ja/nee vorm. De Turing machine moet accepteren of verwerpen. We geven eerst een voorbeeld van hoe zo'n probleem gesteld wordt.

NAAM: KRAL
GEGEVEN: Het Tora Bora gebergte bevat een groot aantal grotten en n vliegende draken. Van elke draak is een deelverzameling grotten bekend waarin hij zich kan bevinden. Als een draak zich in de grot bevindt wanneer een ridder binnenkomt, is de ruimte te krap om goed te bewegen en is de draak kansloos. Als de ridder zich in de grot bevindt wanneer de draak binnenkomt, vult de draak de grot met vuur en is de ridder kansloos. In het boek KRAL staan voor ieder uur van de dag m drietallen. Als je van elk drietal twee getallen kunt wegstrepen zodat n verschillende getallen overblijven, zijn dat de grotten waarin zich op dat uur van de dag een draak bevindt.
GEVRAAGD: Kan de koningin van het land n ridders erop uitsturen die op een bepaald uur van de dag alle draken tegelijkertijd verslaan?

We zullen veel van deze problemen in deze tekst behandelen. De bovengegeven representatie is een makkelijke manier om de problemen te definieren. Elk probleem is altijd een oneindige verzameling van *instanties* (voor elk gebergte kan het aantal draken, grotten en ridders variëren), en we zijn geinteresseerd in hoe moelijk een eventuele algoritme is die voor elke instantie van het probleem gebruikt kan worden (eindige problemen zijn, vanwege onze eerder aannamen over constanten natuurlijk niet interessant). We kunnen lang niet alle interessante problemen van deze vorm behandelen. Dat zijn er eenvoudig te veel. In ieder geval zullen wel de volgende problemen in onze te behandelen lijst voorkomen.

1. NAAM: Traveling Salesperson
   GEGEVEN: Een volledige graaf $G = (V, E)$ met een gewichtsfunctie $g : E \mapsto R$, en een getal K
   GEVRAAGD: Heeft G een Hamilton circuit waarvan het totale gewicht kleiner dan K is?
2. NAAM: Kleurbaarheid
   GEGEVEN: Een graaf $G = (V, E)$ en een getal K
   GEVRAAGD: Bestaat er een functie $c : V \mapsto K$, zo dat voor geen paar $(v, w) \in E$ geldt $c(v) = c(w)$?
3. NAAM: Boedelscheiding
   GEGEVEN: Een verzameling getallen $\{w_1, \ldots, w_n\}$
   GEVRAAGD: Bestaat er een indexverzameling $I \subseteq \{1, \ldots, n\}$ zodat geldt $\sum_{i \in I} w_i = \sum_{i \notin I} w_i$?
4. NAAM: Exacte Overdekking
   GEGEVEN: Een stel deelverzamelingen $S_1, \ldots, S_m$ van $U = \{1, \ldots, n\}$
   GEVRAAGD: Bestaat er een indexverzameling $I \subseteq \{1, \ldots, m\}$ zo dat $\bigcup_{i \in I} S_i = U$ terwijl voor alle $i \neq j$ geldt $S_i \cap S_j = \emptyset$?
5. NAAM: Knapsack
   GEGEVEN: Een verzameling getallen $\{w_1, \ldots, w_n\}$ en een getal b
   GEVRAAGD: Bestaat er een indexverzameling I zo dat $\sum_{i \in I} w_i = b$?
6. NAAM: Knoopoverdekking
   GEGEVEN: Een graaf $G = (V, E)$ en een getal K

GEVRAAGD: Is er een deelverzameling $V' \subseteq V$ met $\|V'\| \leq K$ zo dat voor elk paar $(v, w) \in E$ geldt $\{v, w\} \cap V' \neq \emptyset$?

7. NAAM: Vervulbaarheid
GEGEVEN: Een propositie van n boolese variabelen $F(x_1, \ldots, x_n)$
GEVRAAGD: Is er een toewijzing van waarheidswaarden aan $\{x_1, \ldots, x_n\}$ die F waar maakt?

8. NAAM: Hamilton Circuit
GEGEVEN: Een graaf G
GEVRAAGD: Heeft G een enkelvoudige cykel langs alle knopen?

9. NAAM: Clique
GEGEVEN: Een graaf $G = (V, E)$ en een getal K
GEVRAAGD: Heeft G een volledige ondergraaf van grootte tenminste K

10. NAAM: Onafhankelijke Verzameling
GEGEVEN: Een graaf $G = V, E$ en een getal K
GEVRAAGD: Heeft G een onafhankelijke verzameling van grootte minstens K?

11. NAAM: Betegeling
GEGEVEN: Een $N \times N$ vierkant met kleuren aan de rand, en een verzameling gekleurde tegels
GEVRAAGD: Kan het vierkant met deze verzameling tegels worden volgelegd?

Deze lijst kan nog eindeloos worden verlengd met problemen van dezelfde soort. In 1979 hebben Michael Garey en David Johnson [GJ79] de toenmalige stand van zaken opgetekend in hun boek en een lijst van ongeveer 500 van deze problemen daarin opgenomen. Sindsdien is deze lijst op alle gebieden van de wetenschap alleen maar gegroeid.

Wat is de eigenschap die deze problemen bindt? Het zijn meerdere eigenschappen.

- Allereerst kun je veel van deze problemen in de één of andere vorm in de praktijk tegenkomen.
  - Het kleurbaarheidsprobleem bijvoorbeeld staat voor een hele reeks van verdelingsproblemen van een aantal zaken in een beperkt aantal deelverzamelingen die geen punt gemeen hebben. Als bijvoorbeeld een aantal transacties op een computersysteem moet worden doorgevoerd waarvan sommige dezelfde resources gebruiken (en dus niet tegelijkertijd gestart kunnen worden) is de vraag met hoeveel kleuren zo'n verzameling gekleurd kan worden precies de vraag in hoeveel batches deze verzameling transacties kan worden uitgevoerd.
  - De vraag naar een exacte overdekking komt voor in het vliegverkeer, waar een vliegtuig behalve passagiers ook vliegpersoneel kan vervoeren naar een ander vliegveld waar een vliegtuig op ze staat te wachten. Een overdekking met minimale overlap is er dan één waarin zo weinig mogelijk crews (en dus zoveel mogelijk betalende passagiers) worden vervoerd.
  - Een knoopoverdekking zoek je als je een vertegenwoordigende verzameling wilt vinden die voor iedereen in de gemeenschap het woord kan voeren en een onafhankelijke verzameling als je een deelverzameling van zoekt waarin niemand met iemand anders in de verzamling ruzie heeft (handig voor een feestje).
- Verder hebben al deze problemen gemeen dat er geen algoritmen bekend zijn om de op te lossen anders dan de exhaustive search—probeer alle mogelijke oplossingen en kies degene die werkt (als er één is). Er is echter voor *geen* van deze problemen een bewijs dat dat de enige mogelijke manier is om ze op te lossen.
- Tenslotte komen we bij de definierende eigenschap van deze problemen. Dat is de volgende. Als een oplossing voor zo'n probleem *gegeven* wordt, dan is eenvoudig, in polynomiale tijd, te controleren dat het een oplossing *is*.
  - Als we bijvoorbeeld een functie krijgen die voor elke knoop in een graaf een kleur geeft, kunnen we gemakkelijk controleren dat deze functie aan elk tweetal knopen dat aan een kant vastzit verschillende kleuren toewijst.

– Als we een aantal kanten in een gewogen graaf krijgen kunnen we gemakkelijk controleren dat deze kanten een enkelvoudig pad langs alle knopen vormen en dat bovendien de som van de gewichten van deze kanten beneden een bepaalde grens ligt.

– Het meest algemene voorbeeld van dit soort problemen is wel de wiskundige stelling. Het kan lang, soms wel 300 jaar, duren om voor een interessante wiskundige stelling een bewijs te vinden. Als zo'n bewijs echter eenmaal gevonden is, kan elke wiskundige met voldoende scholing controleren dat het bewijs correct is in redelijke tijd. Dat er bewijzen in de literatuur zijn die incorrect zijn hoewel ze uitvoerig gecontroleerd zijn, ligt aan de vaak informele stijl van bewijsvoering. Als een bewijs in stricte logische formules gegeven wordt, dan is het controleren van zo'n bewijs alleen het mechanisch nagaan van de ene regel na de andere dat bewijsregels correct zijn toegepast. Zo'n controle zou door een (Turing) machine gedaan kunnen worden.

Om de probleemstelling te formaliseren hebben we een speciale variant van de Turingmachine—de nondeterministische Turing machine—nodig. We kwamen deze variant al eerder tegen in Hoofdstuk 6.1. We zullen dit model nu wat aandachtiger beschouwen.

6.3 Het Nondeterministische Model

Er is een aantal manieren om het nondeterministische model van de berekening in te voeren. De eenvoudigste vorm is die waarbij we een deterministische variant van de Turing machine een vantevoren op de band gegeven paar, probleem-oplossing laten controleren. Hierbij voeren we de beperking dan wel in dat de lengte van de oplossing polynomiaal is in de lengte van het probleem, zodat ook de lengte van de Turing machine berekening polynomiaal is in de lengte van het probleem.

De meest gebruikelijke karakterisering van nondeterminisme is het Turing machinemodel, waarbij de toestandsovergangsfunctie wordt veranderd in een relatie. In plaats van één mogelijke opvolger van het paar q, a is er nu een aantal mogelijke opvolgers, en de Turing machine mag in elke stap een keuze maken tussen de mogelijke opvolgers. Zo een serie van keuzes eindigt dan na polynomiaal veel stappen in een accepterende of verwerpende toestand, en elk van de rijen van keuzen is nu een mogelijke berekening op de invoer. We spreken af dat de Turingmachine accepteert, als er een rij keuzen *bestaat* die eindigt in een accepterende toestand.

Een derde manier om nondeterminisme te karakterizeren is met een existentiele quantor over een verzameling van strings in lengte begrensd door een polynoom in de lengte van de probleemstelling. We veronderstellen dat er een (deterministisch) polynomiale tijd berekenbaar predicaat R bestaat dat op invoer x, y het antwoord 0 of 1 kan geven en krijgen een karakterizering van nondeterminisme in de expressie $(\exists y)[|y| \leq p(|x|) \wedge R(x, y)]$.

Al deze manieren om nondeterminisme te beschrijven zijn equivalent, en het is aan de lezer om een keuze te maken die het best bij de intuïtie past. In deze tekst zullen we voor het nondeterministische Turing machinemodel kiezen, voornamelijk om historische redenen.

Het is niet nodig de lengte van de oplossing begrensd te houden tot polynomiale afmetingen. Als we de lengte van de oplossing exponentieel laten groeien, krijgen we de nondeterministische variant van exponentiële tijd (NEXP), maar ook grotere lengtes geven karakterizering van nondeterministische klassen (dubbel, drievoudig, tot en met n-voudig nondeterministisch exponentiële tijd). De eerste klasse waarbij nondeterminisme en determinisme gelijk zijn is de klasse ELEMENTARY, waar een rekentijd wordt toegestaan die een herhaalde macht van twee is $(2^{2^{2^{\cdots}}})$ en waarbij het aantal herhalingen lineair is. Uiteraard kunnen nondeterministische berekeningen gesimuleerd worden op een deterministische machine die exponentieel veel meer rekentijd heeft. Immers, deze machine kan alle mogelijke nondeterministische keuzen één voor één proberen om te zien of misschien één van deze keuzen eindigt in een accepterende toestand. Als we dus de klasse van problemen waarvoor een nondeterministische polynomiale tijd begrensde Turing machine bestaat aanduiden met NP, dan krijgen we voor de tot nu toe bekeken klassen de inclusierelatie $\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{EXP}$. Aangezien we hebben aangetoond dat $\mathrm{P} \neq \mathrm{EXP}$ *moet* één van beide inclusies echt zijn, het is echter tot op heden onbekend welke.

6.4 Reductie

Stel dat we in de winkel een programma kopen voor het oplossen van lineaire vergelijkingen. Het programma is een implementatie van de snelst bekende algoritme en het lost problemen van de vorm $Ax = b$ op, waarbij A een matrix is, b een vector en x het rijtje onbekenden. Thuisgekomen willen we natuurlijk graag dit programma inzetten om het ons al tijden ergerende probleem op te lossen, maar het werkt niet. Als we de kleine lettertjes in de gebruiksaanwijzing lezen, komen we erachter dat het programma alleen werkt voor *symmetrische* matrices, en er is ook geen garantie. Niet goed, geld kwijt. Het probleem is echter veel minder groot dan het lijkt. Immers in onze kast hebben we nog een algoritme voor matrixvermenigvuldiging, dat ook werkt voor matrix-vector vermenigvuldiging en we herinneren ons nog van de lessen lineaire algebra, dat we een matrix A symmetrisch kunnen maken door hem te vermenigvuldigen met zijn getransponeerde A^T. In plaats van $Ax = b$ op te lossen, lossen we nu het probleem $A^TAx = A^Tb$ op en vinden de gezochte oplossing.

We hebben het probleem van het oplossen van een stelsel van lineaire vergelijkingen *vertaald* in het probleem van het oplossen van een stelsel lineaire vergelijkingen waarvan de coëfficientenmatrix symmetrisch is. Behalve dat dit betekent dat we onze felbegeerde oplossing in handen krijgen, zegt dit ook iets over het probleem van het oplossen van een stelsel lineaire vergelijkingen. We hebben te maken met twee problemen. Het *algemene* probleem van het oplossen van een stelsel lineaire vergelijkingen en het *specifieke* probleem van het oplossen van een stelsel lineaire vergelijkingen met een symmetrische coëfficientenmatrix. Meestal kunnen we alleen maar zeggen dat het algemene probleem minstens zo moelijk moet zijn als het specifieke probleem. Immers een algoritme die een oplossing geeft voor het algemene probleem geeft ook een oplossing voor het specifieke probleem. Omdat we nu echter beschikken over een vertaling van het algemene probleem naar het specifieke probleem, kunnen we ook zeggen dat het algemene probleem niet *moeilijker is* dan het specifieke probleem. Immers gegeven een instantie van het algemene probleem, en een algoritme voor het specifieke probleem, kunnen we de vertaling en deze algoritme schakelen om zo een oplossing voor deze instantie te krijgen. De vertaling die we hebben gevonden, *reduceert* onze zoektocht naar een oplossing voor het algemene probleem naar het zoeken naar een oplossing voor het specifieke probleem. De reductie moet wel aan een paar voorwaarden voldoen. Stel we hebben een reductie van probleem A, het bronprobleem, naar probleem B, het doelprobleem. Deze reductie geeft voor elke x een vertaling $f(x)$ zo dat uit een oplossing voor $f(x)$ gemakkelijk een oplossing voor x kan worden afgeleid.

1. De reductie zelf mag niet al te ingewikkeld zijn. In het bijzonder mag de reductie nooit gebruik maken van eigenschappen van de oplossing voor probleem A, omdat je om die te in handen te krijgen eerst die oplossing zou moeten berekenen.

2. Het resultaat van de reductie, de vertaling, mag niet veel langer zijn dan het originele probleem. Anders is de samenstelling van de vertaling en de efficiënte algoritme voor het probleem waarnaartoe wordt gereduceerd geen efficiënte algoritme voor het originele probleem meer, zoals we hadden bedoeld. De efficiënte algoritme is immers efficiënt in de lengte van de vertaling.

Aan deze voorwaarden wordt voldaan door te eisen dat de vertaling/reductie polynomiale tijd begrensd is. Immers in polynomiale tijd kan de reductie niet meer dan polynomiaal veel uitvoer genereren. Als zowel de vertaling/reductie polynomiale tijd begrensd is als de algoritme voor het doelprobleem, dan is de samenstelling van de vertaling en deze algoritme een polynomiale tijd algoritme voor het bronprobleem.

6.5 NP-volledigheid

Het nondeterministische Turing machinemodel is (in polynomiale tijd begrensd) het definiërende model voor de complexiteitsklasse NP. Problemen in deze klasse worden gedefiniëerd door een Turing machine en een polynoom. Het polynoom, uitgerekend in de lengte van de invoer geeft een tijdgrens (is grens op het aantal stappen). Als er een berekening van de (nondeterministische) Turing machine *bestaat* die eindigt met "ja" en die korter is dan de tijdgrens, dan zeggen we dat de Turing machine de invoer accepteert, of ook wel

dat de invoer een instantie van het probleem is, of ook wel element van de verzameling die het probleem beschrijft. Dit betekent dat het aantal problemen in NP (net zoals overigens in alle andere door ons bekeken complexiteitsklassen) aftelbaar is. Ieder paar Turing machine-polynoom definieert een probleem in NP en omgekeerd bestaat er voor elk probleem in NP zo'n paar.

We weten niet of NP problemen herbergt die exponentiële rekentijd vergen. Elk lang genoeg tijdsinterval (denk aan een jaar) ziet wel een voorstel voor zo'n probleem, maar het bewijs is meestal van de vorm „...en daarom is er geen andere mogelijkheid een oplossing te vinden dan alle mogelijke oplossingen te proberen." Zulke bewijzen zijn—tot nu toe—allemaal fout gebleken. Er is echter een deelklasse van problemen in NP waarvan het niet duidelijk is, en zo langzamerhand vanwege alle pogingen onwaarschijnlijk wordt, dat ze deterministisch polynomiale tijd begrensde algoritmen hebben. Deze deelklasse heeft bovendien de eigenschap dat alle problemen die erin zitten allemaal wel, of juist allemaal *niet* een deterministische polynomiale tijd begrensde algoritme hebben. In het bijzonder geldt voor deze klasse dat als voor één van deze problemen een polynomiale tijd algoritme zou bestaan, dan bestaat er een polynomiale tijd algoritme voor *alle* problemen in NP. We noemen deze problemen NP-volledig, omdat ze de volledige complexiteit van de klasse NP in zich dragen.

Hoe zouden we van een probleem in NP bewijzen dat het deze eigenschap heeft? Hiertoe gebruiken we het hierboven geïntroduceerde middel van de reductie. Als we een polynomiale tijd berekenbare functie f hebben, zo dat voor elke x en twee deelverzamelingen A en B van $\{0,1\}^*$ geldt $x \in A$ dan en slechts dan als $f(x) \in B$, dan is elke algoritme die $y \in B$ kan beslissen een algoritme die $x \in A$ kan beslissen door $y = f(x)$ te kiezen. Als voor een probleem A in NP geldt dat er zo'n reductie f naar B bestaat, dan geldt $B \in \mathrm{P} \Rightarrow A \in \mathrm{P}$. Om aan te tonen dat een probleem B dus lid is van onze klasse van NP-volledige problemen hoeven we dus alleen maar aan te tonen dat er zo'n reductie f_A bestaat voor *elke* A in NP. Er zijn echter, zoals boven opgemerkt, aftelbaar oneindig veel problemen in NP, zodat we dit bewijs liever niet per stuk leveren. We beschrijven een schema voor de reductie, liever dan de reductie zelf, waarin elk afzonderlijk probleem in NP kan worden ingevuld om een reductie van dat probleem naar ons probleem te krijgen. Om zo'n schema te krijgen maken we gebruik van de eigenschap dat er een nondeterministische polynomiaal begrensde Turing machine voor probleem A bestaat als A een probleem in NP is.

Neem dus aan dat er zo'n Turing machine M_A bestaat en polynoom p, zodat voor elke x geldt $x \in A$ dan en slechts dan als er een berekening van M_A op invoer x bestaat van niet meer dan $p(|x|)$ stappen die eindigt in een accepterende berekening. We transformeren het drietal: programma van M_A dat een binair getal is, een beschrijving van het polynoom p, dat een ander binair getal is, en de invoer x naar een binaire rij y zodanig dat $y \in B$ dan en slechts dan als M_A invoer x kan accepteren in tijd begrensd door $p(|x|)$. Dit is het plan. Rest nog een geschikt probleem B te kiezen.

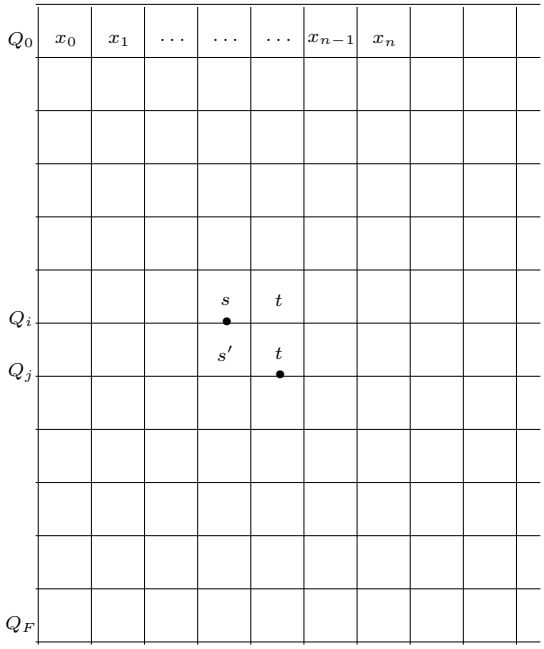
Het probleem B wordt het eerder genoemde probleem

NAAM: SATISFIABILITY
GEGEVEN: Een propositie $F(x_1, \ldots, x_n)$
GEVRAAGD: Is er een toewijzing van waarheidswaarden aan $x_1, \ldots, x_n$ die F waar maakt

.

Allereerst stellen we vast dat dit zeker een probleem in NP is. Immers een nondeterministische Turing machine kan in n opeenvolgende keuzen een toewijzing van waarheidswaarden bepalen, waarna in een deterministische fase deze toewijzing gecontroleerd kan worden. Equivalent: een gegeven oplossing kan in $O(n^2)$ stappen door een deterministische Turingmachine gecontoleerd worden.

Laat dus A een probleem in NP zijn en M_A een Turing machine die voor gegeven polynoom p op invoer x een accepterende berekening $M_A(x)$ van lengte $p(|x|)$ heeft dan en slechts dan als $x \in A$. We merken terzijde op dat we kunnen aannemen dat de accepterende berekening $M_A(x)$ niet slechts begrensd wordt door $p(|x|)$, maar *precies* lengte $p(|x|)$ heeft, dit kunnen we bereiken door het programma van M_A te wijzigen. Tevens merken wij op dat in tijd $p(|x|)$ niet meer dan $p(|x|)$ bandcellen kunnen worden gebruikt en dus nemen we aan dat in de berekening *precies* $p(|x|)$ bandcellen gebruikt worden. Nu zien we dat er een accepterende berekening van M_A op invoer x bestaat precies als we het vierkant van Figuur 6.4 kunnen invullen met symbolen uit het bandalfabet S, de toestandsverzameling Q en een merkteken dat aangeeft waar de kop is zodanig dat:



Figuur 6.4: Plaatje van de Turingmachineberekening met instructie $\delta(q_i, s) = \langle s', q_j, R\rangle$

1. In elke plaats in het bandvierkant precies één symbool uit het alfabet staat
2. In elke rij in het toestandsvak precies één toestand staat
3. In elke rij precies één cel is gemerkt als plaats van de bandkop
4. Voor elk paar opeenvolgende rijen geldt dat op de plaats waar de bandkop *niet* is de symbolen in dezelfde kolom gelijk zijn
5. Voor elk paar opeenvolgende rijen i, en $i+1$ geldt als de bandkop op plaats i, j is dan is de bandkop op plaats $i+1, j$, of op $i+1, j+1$ of op $i+1, j-1$ en deze plaats is in overeenstemming met de toestand in rij i, het symbool op plaats i, j en het programma van M_A.
6. Voor elk paar rijen i, $i+1$ geldt dat als de bandkop op plaats i, j is, dan is het symbool op plaats $i+1, j$ in overeenstemming met het symbool op plaats i, j, de toestand in rij i en het programma van M_A.
7. De eerste $|x|$ bandsymbolen op rij 1 zijn de symbolen van x.
8. De laatste toestand in rij $p(|x|)$ is de accepterende toestand.

Als deze condities allemaal vervuld zijn, dan heeft de machine een accepterende berekening op invoer x van lengte $p(|x|)$. We zullen nu laten zien dat we deze eisen kunnen vertalen in proposities die alleen waargemaakt kunnen worden als aan deze voorwaarden voldaan kan worden. De eis dat een er een accepterende berekening van de juiste lengte op invoer x bestaat, laat zich dan vertalen in het bestaan van een vervulling voor de conjunctie van deze proposities. We voeren de volgende boolese variabelen in:

1. Voor elk symbool uit het bandalfabet $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$ voeren we de variabelen S_{ijk} in met $i \in \{1, \ldots, m\}$, $j, k \in \{1, \ldots, p(|x|)\}$. Voor elk vakje in het symbolenvierkant van Figuur 6.4 één variabele, met de betekenis dat als S_{ijk} waargemaakt wordt om de uiteindelijke formule waar te maken, dan is in de accepterende berekening σ_i op plaats j, k ingevuld.

2. Voor elke toestand uit de toestandsverzameling $Q = \{q_1, \ldots, q_r\}$ een variabele Q_{ij} met $i \in \{1, \ldots, r\}$ en $j \in \{1, \ldots, p(|x|)\}$ met de betekenis dat als Q_{ij} waargemaakt wordt om de uiteindelijke formule waar te maken, dan is in de accepterende berekening q_j in rij i ingevuld.

3. Voor $i, j \in \{1, \ldots, p(|x|)\}$ een variabele T_{ij} met de betekenis dat als T_{ij} waargemaakt wordt om de uiteindelijke formule waar te maken, dan is in de accepterende berekening in rij i het merkteken voor de kop op plaats j gezet.

Om de formules overzichtelijk te houden, voeren we een aantal afkortingen in. Zo zal bijvoorbeeld $x_1 \wedge \ldots \wedge x_n$ vervangen worden door $\bigwedge_i x_i$, $x_1 \vee \ldots \vee x_n$ door $\bigvee_i x_i$ en $(x_i \vee \overline{x_j})_{i \neq j}$ als afkorting worden gebruikt voor de kwadratische hoeveelheid paren waarop dit past, gekoppeld door conjunctie. Ook zullen we operatoren $=$ en $\rightarrow$ gebruiken hoewel deze niet in de propositielogica voorkomen, maar uiteraard (efficiënt) in de propositielogica kunnen worden uitgedrukt. Immers $x_i = y_j$ d.e.s.d.a. $(x_i \wedge y_j) \vee (\overline{x_i} \wedge \overline{y_j})$ en $x_i \rightarrow y_j$ d.e.s.d.a. $\overline{x_i} \vee y_j$. Achtereenvolgens kunnen we nu de eisen voor een accepterende berekening vertalen in proposities:

1. $\bigwedge_{jk} \bigvee_i S_{ijk}$ en $\bigwedge_{jk}[(\overline{S_{ijk}} \vee \overline{S_{i'jk}})_{i \neq i'}]$
2. $\bigwedge_i \bigvee_j Q_{ij}$ en $\bigwedge_i (\overline{Q_{ij}} \vee \overline{Q_{ij'}})_{j \neq j'}$
3. $\bigwedge_i \bigvee_j T_{ij}$ en $\bigwedge_i (\overline{T_{ij}} \vee \overline{T_{ij'}})_{j \neq j'}$
4. $\bigwedge_{ij} \overline{T_{ij}} \rightarrow [S_{ijk} = S_{i+1jk}]$
5. $\bigwedge_{ijk} Q_{ij} \wedge T_{ik} \wedge S_{ik\ell} \Rightarrow [\bigwedge_{mj'k'} S_{i+1km} \wedge Q_{i+1j'} \wedge T_{i+1k'}]$ overeenkomstig het programma van M_A.
6. $\bigwedge_{j \leq n} S_{0j} = x_j \wedge \bigwedge_{j > n} S_{0j} = B$.
7. $Q_p(|x|) = Q_A$.

Als deze propositie een vervulling heeft, dan staat in de Figuur 6.4 in elk vakje precies één symbool, vanwege 1. Op elk moment is de machine in precies één toestand vanwege 2. De bandkop is op elk moment op precies één plaats vanwege 3. Onder elkaar staande symbolen zijn gelijk als de tapekop niet op die plaats is vanwege 4. Als onder elkaar staande symbolen van elkaar verschillen dan is dat in overeenstemming met het programma vanwege 5. De symbolen op de eerste rij zijn symbolen van x aangevuld met blanco symbolen, vanwege 6, en de laatste toestand is de accepterende vanwege 7. Kortom, als de door deze reductie geproduceerde formule een vervulling heeft, dan heeft het drietal programma, polynoom, invoer van waaruit hij geproduceerd is een accepterende berekening.

De reductie is zeker polynomiaal begrensd. Voor een gegeven machine M, een invoer x en een polynoom p is de grootste van de zeven formules ongeveer kwadratisch (paren variabelen) in het aantal variabelen. Het aantal variabelen is zelf maximaal kwadratisch in $p(|x|)$, hetgeen in totaal een $O(p(|x|)^4)$ begrensde reductie geeft. Voor kleine Turingmachineprogramma's is deze grens toch al te groot om een zinnig voorbeeld op te schrijven. Het minimale aantal te gebruiken bandsymbolen is 3 (0,1,B), zodat 10 bandcellen bij 10 tijdstappen al 300 variabelen S_{ijk}, wat dan ongeveer 900 paren in de eerste zin zou opleveren. Helaas moeten we dus hier wat betreft het toelichten van de reductie met een voorbeeld verstek laten gaan.

6.5.1 Meer NP-volledige problemen

Een reductie zoals we zojuist gezien hebben is een reductieschema dat voor alle problemen in NP tegelijkertijd werkt. Zo'n reductie noemen we ook wel een masterreductie. Vanwege de schakelbaarheid van polynomen (een polynoom van een polynoom is opnieuw een polynoom) hoeven we niet elke keer om de volledigheid van een probleem te bewijzen zo'n algemene reductie te geven. Immers, als we een reductie f hebben van probleem A naar probleem B en een reductie g van probleem B naar probleem C dan geeft dit een reductie van probleem A naar probleem C omdat we een gegeven instantie x eerst met f vertalen naar een instantie $f(x)$ met de eigenschap $x \in A \leftrightarrow f(x) \in B$ en we vervolgens $f(x)$ met g reducerent tot $g(f(x))$ met de

eigenschap dat $g(f(x)) \in C \leftrightarrow f(x) \in B \leftrightarrow x \in A$. Omdat g *en* f polynomiale tijd begrensd zijn, is gf dat ook en is gf een polynomiale tijd begrensde reductie van A naar C.

Deze observatie zullen we eerst gaan gebruiken om te laten zien dat een variant van SATISFIABILITY genaamd 3-SAT ook een NP-volledig probleem is.

NAAM: 3SAT
GEGEVEN: Een formule $F(x_1, \ldots, x_n)$ in conjunctieve normaalvorm met ten hoogste drie optredens van variabelen per zin
GEVRAAGD: Is er een toewijzing van waarheidswaarden die F waar maakt?

De reductie vertaalt een willekeurige formule F naar een formule F' in conjunctieve normaalvorm (dat is een conjuctie van disjuncties waarin alleen variabelen of hun ontkenning staan) met de eigenschap dat F' waargemaakt kan worden d.e.s.d.a. F waargemaakt kan worden. De reductie gaat in twee stappen. Eerst vertalen we een willekeurige formule naar een formule in conjunctieve normaalvorm en vervolgens vertalen we deze formule naar een formule waarin precies drie optredens van variabelen per zin staan.

Gegeven een formule F. We nemen aan dat F syntactisch correct is. Omdat de operatoren $\wedge$ en $\vee$ geen verschillen in prioriteit hebben is een formule als $x_1 \wedge x_2 \vee x_3$ niet syntactisch correct, maar een formule als $(x_1 \wedge x_2) \vee x_3$ is dat wel, evenals $x_1 \wedge (x_2 \vee x_3)$. We kunnen aannemen dat F van de vorm $F = (F_1) \vee \ldots \vee (F_k)$ is *of* van de vorm $F = (F_1) \wedge \ldots \wedge (F_k)$ met de eigenschap dat voor $i = 1, \ldots, k$ de formules F_i minder diep genest zijn (dwz minder haakjesparen hebben) dan de formule F. De reductie gaat dus met inductie naar het aantal haakjesparen, waarbij we ervoor zorgen dat, als er nog maar 1 niveau haakjes in de formule staat de formule in conjunctieve normaalvorm staat. De formule $F = (F_1) \wedge \ldots \wedge (F_k)$ is al van de gewenste vorm, zodat we ons in dat geval met de subformules kunnen gaan bezighouden. Deze zijn echter van de form $F_i = (G_1) \vee \ldots \vee (G_k)$ en worden dus op dezelfde manier behandeld als F wanneer deze van de vorm $F = (F_1) \vee \ldots \vee (F_k)$ zou zijn. We behandelen dus alleen dit geval. We voeren k nieuwe variabelen $y_1, \ldots, y_k$ in met de eigenschap dat F alleen waargemaakt kan worden als tenminste 1 van deze variabelen waargemaakt kan worden, en bovendien dat als y_i waar is, dat dan ook F_i waar is. De nieuwe formule wordt.

$$(y_1 \vee \ldots y_k) \wedge (\overline{y_1} \vee (F_1)) \ldots \wedge (\overline{y_k} \vee (F_k))$$

Het deel van de formule $(y_1 \vee \ldots \vee y_k)$ is van de juiste vorm. Alleen de stukken $\overline{y_i} \vee (F_i)$ zijn dat nog niet. Bovendien is het aantal haakjes in het tweede deel van de formule *toegenomen*. Dat is echter schijn. We merken op dat (F_i) van de vorm $((G_1^i) \wedge \ldots \wedge (G_{i_r}^i))$ is en dat elke (G_j^i) van de vorm $((H_1^{ij}) \vee \ldots \vee (H_{k_{ij}}^{ij}))$ is. De formules $(\overline{y_i} \vee (F_i))$ kunnen dus geschreven worden als $(\overline{y_i} \vee (((H_1^{i1}) \vee \ldots \vee (H_{k_1}^{i1})) \wedge \ldots \wedge ((H_1^{ir}) \vee \ldots \vee (H_{k_{ir}}^{ir}))))$. Merk op dat de zichtbare haakjesdiepte—die dus één niveau dieper is dan die van de formule F waarmee we begonnen nu vier is. Deze formule is echter equivalent met $((\overline{y_i} \vee (H_1^{i1}) \vee \ldots \vee (H_{k_{i1}}^{i1})) \wedge \ldots \wedge (\overline{y_i} \vee (H_1^{ir}) \vee \ldots \vee (H_{k_{ir}}^{ir})))$ waarvan de zichtbare haakjesdiepte slechts twee is. Effectief is de diepte van de structuur dus met één haakjespaar afgenomen.

De lezer met enige achtergrond in de logica zou kunnen opmerken dat vertalingen van conjunctieve naar disjunctieve normaalvormen en omgekeerd ook gedaan kunnen worden met de zogenoemde wetten van de Morgan. Voor ons betoog is deze transformatie echter niet geschikt omdat ze niet efficient is. Bekijk als voorbeeld de formule $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$. Passen we de distributieve wet van de Morgan toe, dan wordt deze formule als volgt in een equivalente conjunctie vertaald.

1. $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$
2. $(x_1 \vee (x_3 \wedge x_4)) \wedge (x_2 \vee (x_3 \wedge x_4))$
3. $(x_1 \vee x_3) \wedge (x_1 \vee x_4) \wedge (x_2 \vee x_3) \wedge (x_2 \vee x_4)$.

We zien dat de formule die eerst twee haakjesparen lang was, nu vier haakjesparen lang is geworden, dus in lengte verdubbeld. Dit fenomeen kent deze operatie voor de distributie van elke variabele. Voor n variabelen wordt de formule dus 2^n keer zo lang. Het verschil met de bovenbeschreven methode is dat we elke nieuw ingevoerde variabele slechts één keer over alle subformules hoeven te distribueren. Voor elke nieuwe variabele wordt de formule dus ten hoogste n sybmolen langer. Aangezien we slechts n symbolen per

stap invoeren wordt dus per stap de formule slechts n^2 symbolen langer, en aangezien we in elke stap een haakjespaar kwijtraken gebeurt dit hoogstens n keer. In totaal kan de lengte van de formule dus met $O(n^3)$ bits toenemen door onze transformaties.

We zullen tenslotte laten zien dat een formule die in conjunctieve normaalvorm staat kan worden getransformeerd naar een equivalente formule waarin elke zin precies drie variabelen heeft. Eerst bekijken we de zinnen van de vorm $Z = (x_1 \vee \ldots \vee x_n)$ met $n > 3$. Voor deze zinnen voeren we een nieuwe variabele z in en veranderen de zin in $Z' = (x_1 \vee x_2 \vee z) \wedge (\overline{z} \vee x_3 \vee \ldots \vee x_n)$. Als Z waargemaakt kan worden, kan 1 van de variabelen x_1 of x_2 waargemaakt worden *of* één van de variabelen $x_3, \ldots, x_n$. In het eerste geval maken we z niet waar en in het tweede geval maken we z waar, zodat ook Z' waargemaakt kan worden met de zelfde toewijzing aan x_i. Als Z' waargemaakt kan worden, dan *moet* z waar of niet waar gemaakt worden. In het eerste geval concluderen we dat ook één van $x_3, \ldots, x_n$ waargemaakt kan worden en in het tweede geval dat x_1 of x_2 waargemaakt kan worden. In beide gevallen kan dus ook Z waargemaakt worden.

Tot slot merken we op dat in de zin $(\overline{y_1} \vee \overline{y_2} \vee \overline{y_3}) \wedge (\overline{y_1} \vee y_2 \vee \overline{y_3}) \wedge (y_1 \vee \overline{y_2} \vee \overline{y_3}) \wedge (y_1 \vee y_2 \vee \overline{y_3})$ de variabele y_3 altijd onwaar gemaakt moet worden om een vervulling te kunnen bereiken. Twee van zulke variabelen kunnen dan altijd worden gebruikt om zinnen van lengte één of twee tot de gewenste lengte aan te vullen.

Vertex Cover, Independent Set en Clique

De drie volgende problemen die we zullen gaan bekijken zijn grafenproblemen die bijzonder dicht bij elkaar liggen. Dat wil zeggen dat de reductie van het ene naar het andere probleem zeer direct is en weinig ingewikkelde constructies behoeft. Allereerst hebben we echter de uitspraak nodig dat één van deze problemen NP-volledig is, en dat behoeft wel enig werk. We kiezen voor vertex cover waarvoor we een reductie geven van het zojuist geïdentificeerde probleem 3SAT. Eerst de probleemstelling.

NAAM: VERTEX COVER
GEGEVEN: Een Graaf $G = (V, E)$ en een getal k
GEVRAAGD: Bestaat er een deelverzameling $V' \subseteq V$ met de eigenschap dat $\|V'\| \leq k$ en voor elke e in E geldt $e \cap V' \neq \emptyset$?

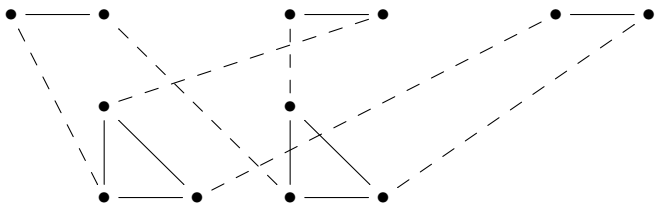
Gegeven een formule F met variabelen $x_1, \ldots, x_n$ en zinnen $C_1, \ldots, C_m$ waarbij elke C_i van de vorm $(\ell_{i_1} \vee \ell_{i_2} \vee \ell_{i_3})$ is met $\ell_{i_j} = x_{i_j}$ of $\ell_{i_j} = \overline{x_{i_j}}$. Dan maken we een graaf G als volgt. Voor elke variabele x_i voeren we twee knopen in x_{i_1} en x_{i_2} die we verbinden met een kant. Voor elke zin C_j voeren we drie knopen in y_{j_1}, y_{j_2} en y_{j_3} waarvan we een driehoek maken. Als x_i in C_j voorkomt op de $k$de plaats ($k \leq 3$), dan verbinden we x_{i_1} met y_{j_k} en als $\overline{x_i}$ in C_j voorkomt op de $k$de plaats dan verbinden we x_{i_2} met y_{j_k}.

Stel nu dat F vervulbaar is. Dan kunnen we dus een n-tal $\ell_1, \ldots, \ell_n$ aanwijzen, zodat in elke C_j minstens één van deze ℓ_i voorkomt. Dat betekent, dat voor elke driehoek tenminste één van de kanten die tussen de driehoeken en de knopen lopen *aan beide* kanten een ℓ_i hebben staan. We nemen van deze knopen degene die bij de paren hoort op in de vertex cover, alsmede de *andere* twee knopen van de driehoek. We hebben nu van de paren (zonodig door aan te vullen) één knoop, en van elke driehoek twee knopen in de vertex cover. Totaal dus $n + 2m$ knopen. Er geldt dat elke kant in de paren en de driehoeken een eindpunt in de vertex cover heeft. Van de kanten die tussen de paren en de driehoeken lopen zit *of* het eindpunt in de driehoek in de vertex cover, *of* het eindpunt in het paar, zodat onze vertex cover volledig is.

Stel omgekeerd dat onze graaf een vertex cover van grootte $n + 2m$ heeft. Van elk paar zit tenminste (en ook ten hoogste) één punt in de vertex cover en van elke driehoek zitten ten minste (en ook ten hoogste) twee punten in de vertex cover. Alle kanten hebben per definitie tenminste één eindpunt in de vertex cover. Omdat van elke driehoek tenminste één punt *niet* in de vertex cover zit, moet het wel zo zijn dat voor de kant die tussen dit punt en de paren loopt het andere eindpunt in de vertex cover zit. We kunnen dus bij de paren n punten vinden (van ieder paar één), zodat deze n punten samen verbonden zijn met *alle* driehoeken. Dat wil zeggen dat de optredens waarmee deze n punten gelabeled zijn in alle C_j voorkomen of dat we door

deze optredens waar te maken, de hele formule waar kunnen maken.

Voorbeeld 6.5.1: Laat gegeven zijn de formule $F(x_1, x_2, x_3) = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3)$. De reductie transformeert dit tot de volgende graaf.



In deze graaf hebben we kanten die tussen twee delen van de graaf lopen (van de variabelen naar de zinnen) onderbroken getekend voor de duidelijkheid. □

Nu we de volledigheid van VERTEX COVER hebben aangetoond, kunnen we ook de volledigheid van INDEPENDENT SET en CLIQUE bewijzen. Eerst de probleemdefinities.

NAAM: INDEPENDENT SET
GEGEVEN: Een graaf $G = (V, E)$ en een getal k
GEVRAAGD: Bestaat er een deelverzameling $V' \subseteq V$ met $\|V'\| \geq k$ zo dat tussen de knopen van V' geen kant zit $(\{(v, v') : v, v' \in V'\} \cap E = \emptyset)$?

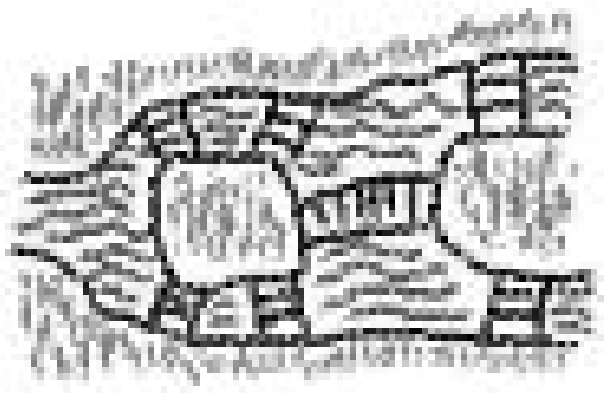
NAAM: CLIQUE
GEGEVEN: Een graaf $G = (V, E)$ en een getal k
GEVRAAGD: Bestaat er een deelverzameling $V' \subseteq V$ met $\|V'\| \geq k$ zó dat tussen *elk* paar knopen in V' een kant zit ($G|V'$ is een volledige ondergraaf)?

Eerst merken we op dat beide problemen in NP zitten. Gegeven een collectie van k knopen is het immers vrij eenvoudig te controleren dat tussen elk paar geen, of juist wel, een kant aanwezig is. Vervolgens zien we in dat de reductie *tussen* de beide problemen ook bijzonder simpel is. Als afbeelding van G nemen we in beide gevallen de graaf $\overline{G}$, de complementaire graaf. In de complementaire graaf is elke kant verwijderd en tussen elk paar knopen waartussen geen kant liep is een kant aangebracht. Ofwel als $G = (V, E)$, dan is $\overline{G} = (V, \overline{E})$ met $\overline{E} = V \times V - E$. Elke independent set in G wordt een clique in $\overline{G}$ en omgekeerd.

De reductie van VERTEX COVER naar INDEPENDENT SET is ook een bijzonder simpele reductie. Hiervoor veranderen we de graaf niet eens, we vinden alleen een andere waarde voor k. Immers, als er een verzameling V' met cardinaliteit hoogstens k is zo dat *alle* kanten een eindpunt in V' hebben, dan heeft *geen* kant beide eindpunten in $V - V'$, ofwel $V - V'$ is een independent set. Dus een graaf G heeft een VERTEX COVER van grootte hoogstens k dan en slechts dan als G een independent set heeft van grootte minstens $\|V\| - k$.

Rondjes in Grafen

Grafen worden vaak gebruikt om geografische problemen te representeren, waarbij paden in grafen dan de bereikbaarheidsrelatie aangeven. In het eerste deel van deze tekst hebben we ons al afgevraagd hoe moeilijk het is om de kortste weg tussen twee punten te vinden. Dat bleek een algoritmisch relatief eenvoudig probleem te zijn. Hier zullen we inzien dat het vinden van de *langste* weg tussen twee punten algoritmisch vele malen moeilijker ligt. Een rondje langs alle punten in een graaf vinden zonder in herhaling te vallen, zullen we leren kennen als een NP-volledig probleem. Het komt vaker voor dat moeilijke en eenvoudige algoritmische problemen dicht bij elkaar liggen. Zo zagen wij bijvoorbeeld dat 3-SAT een NP-volledig probleem is, omdat we elke formule in conjunctieve normaalvorm kunnen terugbrengen tot een formule waarin elke zin slechts drie optredens van variabelen kent. De oplettende lezer zal hebben opgemerkt dat de daar gepresenteerde reductie niet kan worden doorgezet totdat elke zin nog maar twee optredens van variabelen heeft. Dit heeft



Figuur 6.5: Euler's eilanden in Königsberg

een diepere betekenis, zoals we in de opgaven zullen zien. Ook in deze sectie zijn er subtiele verschillen tussen efficient oplosbare en NP-volledige problemen. Het verschil tussen kortste pad/langste pad noemden we al, maar we willen toch ook het voorbeeld van een volledige enkelvoudige cykel die alle kanten slechts 1 keer gebruikt hier niet voorbijlopen.

NAAM: EULER CYCLE
GEGEVEN: Gegeven een graaf G
GEVRAAGD: Is het mogelijk een pad te vinden dat alle kanten precies één keer gebruikt?

In het stadje Königsberg (tegenwoordig Kalingrad) in voormalig oost Pruisen waren zeven bruggen over de rivier die door de stad loopt. De burgers wandelden vaak op zondagmiddag van één van de oevers naar de eilanden en dan naar de andere oever en dan terug. De overtuiging heerste dat het onmogelijk is om al deze zeven bruggen over te gaan zonder één van de bruggen twee keer te gebruiken (zie Figuur 6.5.1). Euler loste niet allen dit probleem op, maar ook het algemenere geval, door te bewijzen dat een ronde in een graaf die elke kant precies één keer gebruikt (vanaf toen Eulerronde genoemd) alleen mogelijk is als elke knoop in de graaf een even graad heeft. Het bewijs is een eenvoudig inductiebewijs dat in elke grafentheorietekst gevonden kan worden. Voor de complexiteit van het probleem EULER CYCLE betekent dit dat het daarmee een deterministisch polynomiale tijd oplosbaar probleem geworden is. Immers, we hoeven slechts te controleren of elke knoop in de graaf een even graad heeft.

Het probleem wordt echter ineens *veel* moeilijker als we in de definitie het woord kant door knoop vervangen.

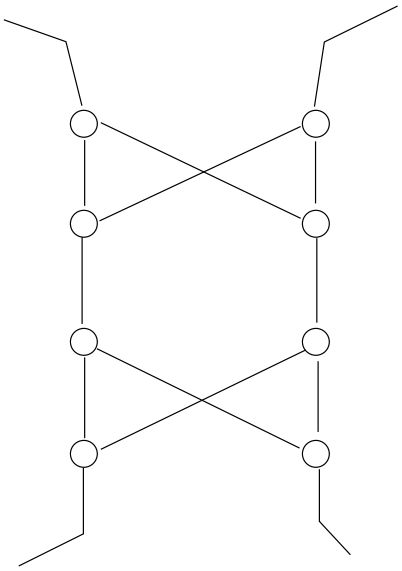
NAAM: HAMILTON CIRCUIT
GEGEVEN: Een graaf $G = (V, E)$
GEVRAAGD: Is het mogelijk een permutatie van de knopen te vinden $v_{i_1}, \ldots, v_{i_n}$ zo dat deze volgorde een enkelvoudig pad langs alle knopen voorstelt?

Opnieuw merken we eerst op dat dit een probleem in NP is. Immers gegeven zo'n permutatie is het eenvoudig te controleren dat elke knoop *precies* één keer voorkomt en dat tussen twee opeenvolgende knopen in de rij een kant in G zit.

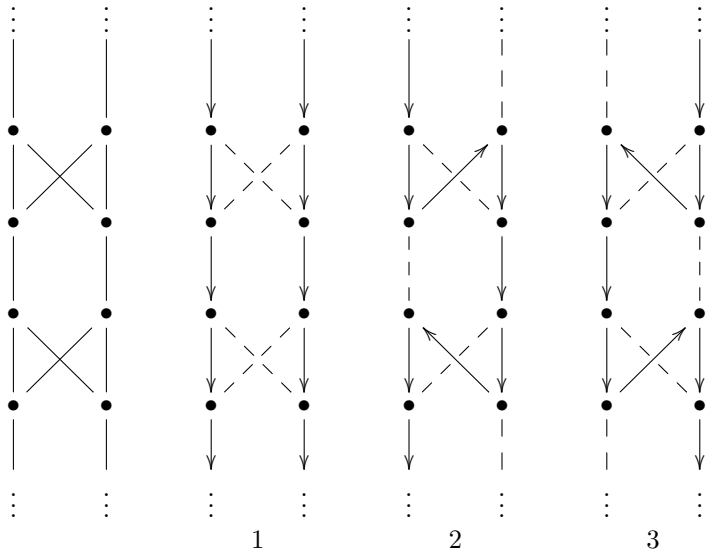
Het probleem is echter ook NP-volledig. Hiertoe beschrijven we een deterministisch polynomiale tijd begrensde reductie van het probleem VERTEX COVER.

Gegeven een graaf $G = (V, E)$ en een getal k, maken we een graaf G' met de eigenschap dat als G een vertex cover heeft van grootte hoogstens k, dan heeft G' een Hamilton circuit. Onze graaf G' bestaat uit k zogenoemde *keuzeknopen* die ons elk uit de verzameling knopen van G een knoop laten kiezen. Verder voeren we voor elke kant in G een deelgraafje van G' in, zoals in Figuur 6.6.

Deze deelgraafjes kunnen op precies drie manieren doorlopen worden zonder een knoop over te slaan en zonder een knoop twee keer aan te doen. Zie figuur 6.7



Figuur 6.6: Deelgraaf voor kanten uit G.



Figuur 6.7: 3 manieren om de deelgraaf te doorlopen

1. Je kunt eerst een keer links langs alle onder elkaar liggende knopen lopen en vervolgens bij een volgende keer rechts langs alle knopen.

2. Je kunt de eerste twee knopen van boven van de linkerkant pakken, vervolgens naar de eerste knoop rechtsboven oversteken, dan alle knopen aan de rechterkant langslopen, en tenslotte naar de derde knoop linksboven oversteken en de laatste twee knopen van de linkerkant meenemen.

3. Je kunt het bovenstaande pad ook gespiegeld volgen. Dus eerst de twee bovenste knopen rechts, dan het hele linkerrijtje en vervolgens de laatste twee knopen rechts.

Als zo'n deelgraaf een kant (v, w) representeert, verbinden we in de hele graaf *alle* deelgrafen die een kant (v, w') representeren in een lang pad met de linkerkant van deze deelgraaf. Evenzo verbinden we *alle* deelgrafen die een kant (v', w) voorstellen in een lang pad met de rechterkant van deze graaf. Een knoop wordt zo geïdentificeerd met een pad in de graaf. Tenslotte zijn al dit soort paden verbonden met de k keuze knopen.

Als nu G een vertex cover van grootte k heeft, kun je in keuzeknoop 1 beginnen en vervolgens alle deelgrafen doorlopen die een eindpunt in de vertex cover hebben (dus alle kanten) op drie manieren:

1. Als beide eindpunten in de vertex cover zitten, dan wordt de deelgraaf twee keer doorlopen.

2. Als het linkereindpunt in de vertexcover zit, komen we aan de linkerkant binnen, nemen vervolgens alle knopen op en gaan we er ook aan de linkerkant weer uit.

3. Idem voor de rechterkant.

Omgekeerd, als G' een Hamilton circuit heeft kan elke deelgraaf maar op één van de drie manieren doorlopen worden. Aan de manier waarop een deelgraaf doorlopen wordt, kunnen we zien welke knopen in de vertex cover horen. Het feit dat de hele graaf doorlopen wordt, garandeert een vertex cover van grootte hoogstens k.

Als we eenmaal de NP-volledigheid van Hamilton circuit hebben aangetoond kunnen we van daaruit weer de NP-volledigheid van het beroemde handelsreizigerprobleem aantonen.

NAAM: HANDELSREIZIGER
GEGEVEN: Een volledige graaf $K = (V, E)$, een gewichtsfunctie $w : E \mapsto R$, en een getal B
GEVRAAGD: Heeft K een Hamilton circuit waarvan het totale gewicht kleiner dan of gelijk is aan B

Ook dit is een probleem in NP, want als ons een Hamilton circuit gegeven wordt, is het eenvoudig te controleren dat het inderdaad een Hamilton circuit is, en na optellen van de gewichten te constateren dat het totale gewicht kleiner dan of gelijk aan B is.

Het is echter ook een NP-volledig probleem. Om dit te laten zien gebruiken we de volgende reductie van Hamilton circuit. Laat een graaf $G = (V, E)$ gegeven zijn op n knopen. Een Hamilton circuit in G heeft precies n kanten. We geven alle kanten in G gewicht 1, en maken G vervolgens volledig. Alle nieuwe kanten krijgen gewicht $n + 1$. Noem de nieuwe graaf K. Als nu gevraagd wordt "Heeft K een Hamilton circuit waarvan het totale gewicht kleiner dan of gelijk is aan n, dan kan het antwoord alleen maar bevestigend zijn indien ook G een Hamilton circuit heeft. Omgekeerd: als G een Hamilton circuit heeft, dan vormen de kanten in dit Hamilton circuit een Hamilton circuit in K waarvan het totale gewicht precies n is.

We concluderen dus dat K een tour heeft van lengte n dan en slechts dan als G een Hamilton circuit heeft, of $\langle K, n\rangle \in \text{TSP} \Leftrightarrow G \in \text{HAM}$.

Passen en Verdelen

„Met passen en meten wordt de meeste tijd versleten", is een oude wijsheid. Uit een berg onderdelen waarbij geen nette bouwtekening gegeven is, is het lastig iets zinnigs samen te stellen, zo dat zoveel mogelijk onderdelen gebruikt worden. Iedereen die wel eens een meubelstuk bij een doe het zelf zaak gekocht heeft, met lego gespeeld heeft, of een legpuzzel heeft opgelost, kent dit probleem. In dit gedeelte zullen we een aantal van dit soort problemen bekijken. Een masterprobleem is dat van het leggen van legpuzzels. We zullen zien dat een sterke vereenvoudiging van dit probleem nog NP-volledig is. Het wordt als volgt geformuleerd.

NAAM: BEGRENSDE BETEGELING

GEGEVEN: Een $N \times N$ vierkant met kleuren aan de rand en een verzameling tegeltypen, met een 1×1 oppervlakte die ook gekleurde randen hebben

GEVRAAGD: Is het mogelijk het vierkant zo vol te leggen, dat tegels niet gedraaid worden en aanliggende tegels langs aanliggende randen dezelfde kleur hebben?

Uiteraard is ook BEGRENSDE BETEGELING een probleem in NP immers, gegeven een invulling van het vierkant kunnen we gemakkelijk controleren dat dit vierkant aan alle eisen voldoet. BEGRENSDE BETEGELING is ook een NP-volledig probleem. Om dit te bewijzen gebruiken we opnieuw het middel van de master reductie. Gegeven een willekeurige nondeterministische Turing machine M, een polynoom p en een invoer x construeren we een betegelingsprobleem dat een oplossing heeft dan en slechts dan als M invoer x in ten hoogste $p(|x|)$ stappen kan accepteren.

Om technische redenen nemen we aan dat het programma van M zodanig is dat er geen instructie in staat waar M zowel naar links als naar rechts kan bewegen. Dit is geen beperking van de algemeenheid. Verder nemen we aan dat elke accepterende berekening van M eindigt in de meest linker bandcel in toestand q_F met onder de kop het blanco symbool. Verder zorgen we er met een extra instructie voor dat de laatste configuratie van M zo vaak als nodig herhaald kan worden.

De reductie gaat als volgt. We gebruiken kleuren die we de namen geven van achtereenvolgens de bandsymbolen s_i, de toestanden q_i en de combinaties van toestanden en bandsymbolen $\langle q_i, s_j \rangle$. Tenslotte hebben we nog één speciale kleur die we "wit" (w) zullen noemen. Het vierkant dat moet worden gevuld heeft afmetingen $p(|x|) \times p(|x|)$. Nu geven we de eerste n plaatsen langs de rand de kleur die overeenkomt met de symbolen van x, en langs de onderrand van het vierkant plaatsen we in het eerste vakje de kleur $\langle q_F, w \rangle$. Nu gebruiken we de volgende tegels:

1. Voor elk bandsymbool s_i (inclusief $s_i = w$) hebben we een tegel die zowel aan de bovenrand als aan de benedenrand de kleur s_i heeft.

2. Voor elke instructie $q_i, s_j \mapsto q_k, s_\ell R$ hebben we een tegel die aan de bovenrand de kleur $\langle q_i, s_j \rangle$ draagt, aan de onderrand de kleur s_ℓ en aan de rechterzijkant de kleur q_k

3. Idem voor elke instructie $q_i, s_j \mapsto q_k, s_\ell L$, maar dan met de kleur q_k aan de linkerzijkant.

4. Voor elke tegel van de vorige twee typen *en* elk bandsymbool s_i hebben we een tegel met aan de bovenrand de kleur s_i, aan de linker (resp. rechter) zijkant de kleur q_k en aan de onderrand de kleur $\langle q_k, s_i \rangle$.

Als er nu een berekening van M bestaat die accepteert in $p(|x|)$ stappen, dan kunnen we uit de achtereenvolgende configuraties van M een kleuring van het vierkant aflezen. Omgekeerd definieert elke kleuring van het vierkant ook één op één een accepterende berekening van M van lengte precies $p(|x|)$.

De NP-volledigheid van BEGRENSDE BETEGELING kunnen we gebruiken om het volgende probleem in de rij verdeelproblemen NP-volledig te bewijzen. Het gaat hier om een gegeven aantal deelverzamelingen waarmee we een gebied willen overdekken waarbij het aantal plaatsen dat dubbel gedekt wordt zo klein mogelijk gehouden wordt. Een bekende toepassing van dit probleem is het roosteren van flight crews. Vliegtuigen kunnen niet nadat ze geland zijn meteen weer opstijgen, omdat er allerlei technische controles moeten worden uitgevoerd, het vliegtuig moet worden bijgetankt, soms moet worden omgebouwd van passagiers naar vrachtvliegtuig enzovoort. Evenzo kan de crew niet altijd meteen met hetzelfde of met een ander vliegtuig weer opstijgen zodra ze geland zijn. Soms moet er worden gegeten of worden overnacht of willen ze zomaar een paar dagen vrij. Het gevolg hiervan is dat vliegpersoneel vaak met een vliegtuig van dezelfde of een andere maatschappij moet worden vervoerd naar een andere luchthaven om daar weer een vliegtuig te gaan besturen. Een luchtvaartmaatschappij wil uiteraard zoveel mogelijk van de beschikbare stoelen verkopen en dus graag zo weinig mogelijk van het eigen personeel vervoeren. Zaak is dus om de uit te voeren vluchten zoveel mogelijk te bezetten met één enkele crew. De andere constraint is natuurlijk dat elke vlucht wel minimaal één crew moet hebben. In onze notatie wordt dit probleem (als beslissingsprobleem) als volgt geformuleerd.

NAAM: EXACTE OVERDEKKING
GEGEVEN: Een universum $U = \{1, \ldots, n\}$ en een verzameling deelverzamelingen $S_i \subseteq U$.
GEVRAAGD: Bestaat er een indexverzameling I zó dat $\bigcup_{i \in I} S_i = U$ en $(\forall i \neq j \in I)[S_i \cap S_j = \emptyset]$?

Uiteraard is ook dit een probleem in NP. Gegeven de indexverzameling kunnen we gemakkelijk controleren dat elk element van het universum precies één keer voorkomt. Het probleem is ook NP-volledig. Hiervoor produceren we een reductie van begrensde betegelingen. Gegeven een begrensd betegelingsprobleem met een $N \times N$ vierkant en tegeltypen T_i. We noteren een verzameling kleuren HC die langs horizontale randen kunnen voorkomen, een verzameling VC van kleuren die langs verticale randen kunnen voorkomen. Nu bekijken we de bovenrand van het vierkant en introduceren de verzameling $R_B = \{\langle\langle 0, i\rangle, k_i\rangle : k_i$ is de i-de kleur langs de bovenrand $\}$. Evenzo voor de onderrand van het vierkant $R_O = \{\langle N+1, i, \mathrm{HC} - k_i\rangle : k_i$ is de i-de kleur langs de onderrand $\}$. Voor de linkerrand van het vierkant introduceren we de verzameling $R_L = \{\langle 0, i, k_i\rangle : k_i$ is de i-de kleur langs de linkerrand $\}$. Voor de rechterrand van het vierkant introduceren we de verzameling $R_R = \{\langle N+1, i, \mathrm{VC} - k_i\rangle : k_i$ is de i-de kleur langs de rechterrand $\}$. Tot slot: stel dat er een tegeltype T_k is met kleuren b_k, o_k, ℓ_k, r_k respectievelijk aan de boven-, onder-, linker- en rechterkant. Dan introduceren we de verzamelingen $T_{kij} = \{\langle i, j, \mathrm{HC} - b_k\rangle, \langle i+1, j, o_k\rangle, \langle i, j, \mathrm{VC} - \ell_k\rangle, \langle i, j+1, r_k\rangle\}$ voor $i, j \in \{1, \ldots, N\}$. Het universum is $U = \{\langle i, j, \mathrm{HC}\rangle, \langle i, j, \mathrm{VC}\rangle\}$ voor $i, j \in \{1, \ldots, N+1\}$.

Als het vierkant met tegels kan worden overdekt kunnen we aan het tegeltype T_k dat op plaats i, j ligt aflezen welke deelverzameling T_{kij} moet worden gekozen. Aan de andere kant, als een volledige overdekking kan worden gerealiseerd door verzamelingen T_{kij} en $T_{k'i+1j}$ in de overdekking te kiezen, dan betekent dat, dat er tegels T_k en $T_{k'}$ zijn die de kleur b_k langs de onder, resp bovenrand hebben en die dus op coordinaten i, j en $i+1, j$ gelegd kunnen worden voor gegeven i en j, ofwel dat er een kleuring van het vierkant mogelijk is.

De volgende twee problemen in de rij passen en meten zijn koppelingsproblemen. We hebben gegeven dat bepaalde elementen in groepen kunnen voorkomen, maar natuurlijk is voor elk element niet éénduidig bepaald in welke groep dit element terechtkomt. Dit probleem is oorspronkelijk bekend onder de naam „stabiele huwelijksprobleem". In deze (ouderwetse) vorm ging het erom paren aan elkaar te koppelen, zodat de totale verzameling paren een stabiele koppeling voor zou stellen. We hebben dit probleem eerder gezien in 3.4.1 en daar gezien dat het een eenvoudig oplosbaar probleem is. De situatie wordt heel anders wanneer we het probleem uitbreiden van paren naar trio's. De complexiteit van het probleem springt dan ineens van P naar NP-volledig. We zetten de twee probleemstellingen even onder elkaar om te zien hoe weinig de twee van elkaar verschillen.

NAAM: MATCHING
GEGEVEN: $U = \{1, \ldots, m\}$, $m \geq n$ en een verzameling paren $\{P_i\}_{i=1}^m$, $P_i \in U \times U$
GEVRAAGD: Is er een indexverzameling I met $\|I\| = n$ zo dat elke $u \in U$ precies één keer op elke coordinaatplaats voorkomt in $\bigcup_{i \in I} P_i$?

NAAM: 3-DIMENSIONAL MATCHING
GEGEVEN: $U = \{1, \ldots, m\}$, $m \geq n$ en een verzameling drietallen $\{D_i\}_{i=1}^m$, $D_i \in U \times U \times U$
GEVRAAGD: Is er een indexverzameling I, met $\|I\| = n$ zo dat zo dat elke $u \in U$ precies één keer op elke coordinaatplaats voorkomt in $\bigcup_{i \in I} D_i$?

Zoals vaker gebeurt bij de overgang van 2 naar 3—zoals we hierboven gezien hebben bij SATISFIABILITY en zoals we verderop zullen zien bij kleurenproblemen—zien we een verandering van de complexiteit van het probleem van doenlijk naar ondoenlijk. De bovengrens voor het probleem blijft polynomiaal. Immers, 3-DIMENSIONAL MATCHING is een probleem in NP. Wanneer ons zo'n indexverzameling gegeven wordt, kunnen we gemakkelijk nagaan dat voor alledrie de coordinaten de projectie van de verzamgeling drietallen op die coordinaat een permutatie van de getallen $1, \ldots n$ is. Dat 3-DIMENSIONAL MATCHING ook een NP-volledig probleem is tonen we aan door een reductie van EXACTE OVERDEKKING naar 3-DIMENSIONAL MATCHING te geven.

Gegeven een exact overdekkingsprobleem met $U = \{1, \ldots, n\}$ met verzamelingen S_i. We gaan een verzameling drietallen D en een universum T maken, zó dat als een deelverzameling van D exact $T \times T \times T$ overdekt, we uit die overdekking een deel van de verzamelingen S_i kunnen selecteren dat exact U overdekt.

Allereerst definieren we paren $\langle i, j\rangle$, voor alle i en j waarvoor i in S_j zit. Al deze paren vormen de verzameling T. Verder laten we $\alpha(i)$ de minimale $\langle i, j\rangle$ zijn waarvoor er zo een paar $\langle i, j\rangle$ bestaat. Tot slot hebben we nog een afbeelding π nodig die voor vaste j steeds een cyclische permutatie is van S_j. Dus als we een paar $\langle i_1, j\rangle$ hebben, dan is $\pi(\langle i_1, j\rangle)$ het paar $\langle i_2, j\rangle$ waarbij i_2 het element in S_j is dat volgt op i_1. De permutatie π loopt zo de hele verzameling S_j door, todat we weer bij i_1 terug zijn. Nu definieren we drietallen $D = \{(\alpha(i), \langle i, j\rangle, \langle i, j\rangle)\} \cup \{(\beta, \langle i, j\rangle, \pi(\langle i, j\rangle) : \beta \in T \wedge \beta \neq \alpha(\langle i, j\rangle)\}$.

De bewering is nu dat er een deelverzameling van deze drietallen bestaat die $T \times T \times T$ overdekt als en alleen als er een deel van de S_j bestaat dat exact U overdekt. Dus laten we maar eens aannemen dat er zo'n verzameling drietallen bestaat. Allereerst moeten de elementen $\alpha(i)$, allemaal in de eerste coordinaat van $T \times T \times T$ geraakt worden, dus we moeten voldoende paren uit het eerste deel van D kiezen. Die worden allemaal met één of ander paar $\langle i, j\rangle$ in de tweede coordinaat en *hetzelfde* paar $\langle i, j\rangle$ in de derde coordinaat gekozen. We beweren dat de verzameling van alle j's die op deze manier gekozen wordt zo is dat de S_j een exacte overdekking vormen van U. Het is duidelijk dat de vereniging van de S_j de verzameling U overdekt, het is dus voldoende te bewijzen dat de zo gekozen S_j een onderling lege doorsnede hebben. Stel dus maar dat we een paar $\langle i, j\rangle$ en een paar $\langle i', j'\rangle$ kiezen zo dat $S_j \cap S_{j'} \neq \emptyset$. Neem aan dat $S_j \cap S_{j'} \supset \{k\}$. Het drietal $(\alpha(k), \langle k, \ell\rangle, \langle k, \ell\rangle)$ kan zo gekozen worden dat $\ell = j$ *of* $\ell = j'$, maar niet allebei. Laten we aannemen dat $\langle k, j'\rangle$ gekozen wordt. Aangezien $k \in S_j$ zal ook het paar $\langle k, j\rangle$ ergens in de eerste, tweede en derde coordinaat moeten optreden. In het bijzonder is van belang dat het in de tweede coordinaat optreed. Als we namelijk $\langle k, j\rangle$ in de tweede coordinaat ergens kiezen, dan moet de derde coordinaat van dit drietal gelijk zijn aan $\pi(\langle k, j\rangle)$. Immers we kunnen dit paar niet meer met $\alpha(k)$ in de eerste coordinaat kunnen kiezen. Dit is echter een paar $\langle k', j\rangle$ waarbij k' het element van S_j is dat *volgt* op k. Dit betekent dat ook $(\alpha(k'), \langle k', j\rangle, \langle k', j\rangle)$ niet in de overdekking met drietallen gekozen kan worden en dus dat ook $\langle k', j\rangle$ alleen in de tweede coordinaat door de overdekking kan worden geraakt door een drietal $(\beta, \langle k', j\rangle, \pi(\langle k', j\rangle)$. Zo voortgaande schakelen we achtereenvolgens alle elementen van S_j uit als kandidaat om op te treden in een drietal $(\alpha(r), \langle r, j\rangle, \langle r, j\rangle)$, ofwel S_j is *niet* een verzameling die in de overdekking is gekozen. Tegenspraak.

Als omgekeerd een overdekking van de S_j bestaat die U overdekt, dan kunnen we voor elke i die in een S_j zit, het bijbehorende drietal $(\alpha(i), \langle i, j\rangle, \langle i, j\rangle)$ kiezen. Het feit dat de gekozen S_j een lege doorsnede hebben, maakt dat we de *rest* van $T \times T \times T$ kunnen overdekken met drietallen van de vorm $(\langle i, j\rangle, \langle i, j\rangle, \pi(\langle i, j\rangle)$.

Laten we dit ingewikkelde verhaal toelichten aan de hand van een voorbeeld.

Voorbeeld 6.5.2: Stel dat het universum $U = \{1, \ldots, 5\}$ en laten de verzamelingen S_i als volgt gegeven zijn.

$$\begin{array}{lcl} S_1 & = & \{1, 4, 5\} \\ S_2 & = & \{2, 3\} \\ S_3 & = & \{1, 3, 5\} \\ S_4 & = & \{2, 5\} \end{array}$$

Merk op dat S_1 en S_2 samen een exacte overdekking vormen. De paren $\alpha(u_i)$ voor $i = 1, \ldots, 5$ zijn dan $\{\langle 1, 1\rangle, \langle 2, 2\rangle, \langle 3, 2\rangle, \langle 4, 1\rangle, \langle 5, 1\rangle\}$

De verzameling T bestaat uit de paren:

$$\{\langle 1, 1\rangle, \langle 1, 3\rangle, \langle 2, 2\rangle, \langle 2, 4\rangle, \langle 3, 2\rangle, \langle 3, 3\rangle, \langle 4, 1\rangle, \langle 5, 1\rangle, \langle 5, 3\rangle, \langle 5, 4\rangle\}$$

en dus hebben we de volgende verzameling drietallen:

$$\begin{array}{lll} \{(\langle 1, 1\rangle, \langle 1, 1\rangle, \langle 1, 1\rangle), & (\langle 1, 1\rangle, \langle 1, 3\rangle, \langle 1, 3\rangle), & (\langle 2, 2\rangle, \langle 2, 2\rangle, \langle 2, 2\rangle), \\ (\langle 2, 2\rangle, \langle 2, 4\rangle, \langle 2, 4\rangle), & (\langle 3, 2\rangle, \langle 3, 2\rangle, \langle 3, 2\rangle), & (\langle 3, 2\rangle, \langle 3, 3\rangle, \langle 3, 3\rangle) \\ (\langle 4, 1\rangle, \langle 4, 1\rangle, \langle 4, 1\rangle), & (\langle 5, 1\rangle, \langle 5, 1\rangle, \langle 5, 1\rangle), & (\langle 5, 1\rangle, \langle 5, 3\rangle, \langle 5, 3\rangle), \\ (\langle 5, 1\rangle, \langle 5, 4\rangle, \langle 5, 4\rangle)\} & & \end{array}$$

en de drietallen

$$\begin{array}{lll} \{(\beta, \langle 1, 1\rangle, \langle 4, 1\rangle), & (\beta, \langle 4, 1\rangle, \langle 5, 1\rangle), & (\beta, \langle 5, 1\rangle, \langle 1, 1\rangle), \\ (\beta, \langle 2, 2\rangle, \langle 3, 2\rangle), & (\beta, \langle 3, 2\rangle, \langle 2, 2\rangle), & (\beta, \langle 1, 3\rangle, \langle 3, 3\rangle), \\ (\beta, \langle 3, 3\rangle, \langle 5, 3\rangle), & (\beta, \langle 5, 3\rangle, \langle 1, 3\rangle), & (\beta, \langle 2, 4\rangle, \langle 5, 4\rangle), \\ (\beta, \langle 5, 4\rangle, \langle 2, 4\rangle)\} & & \end{array}$$

voor β lopend over alle paren die niet $\alpha(i)$ zijn. Een overdekking van $T \times T \times T$ is nu

$$\{(\langle 1,1\rangle, \langle 1,1\rangle, \langle 1,1\rangle), \quad (\langle 2,2\rangle, \langle 2,2\rangle, \langle 2,2\rangle), \quad (\langle 3,2\rangle, \langle 3,2\rangle, \langle 3,2\rangle),$$
$$(\langle 4,1\rangle, \langle 4,1\rangle, \langle 4,1\rangle), \quad (\langle 5,1\rangle, \langle 5,1\rangle, \langle 5,1\rangle), \quad (\langle 1,3\rangle, \langle 1,3\rangle, \langle 3,3\rangle),$$
$$(\langle 2,4\rangle, \langle 3,3\rangle, \langle 5,3\rangle), \quad (\langle 3,3\rangle, \langle 5,3\rangle, \langle 1,3\rangle), \quad (\langle 5,3\rangle, \langle 2,4\rangle, \langle 5,4\rangle)$$
$$(\langle 5,4\rangle, \langle 5,4\rangle, \langle 2,4\rangle)\}$$

□

Het volgende probleem in deze serie is KNAPSACK. We hebben al eerder een variant van KNAPSACK gezien die kon worden opgelost met behulp van een gulzige algoritme. Ook hebben we gezien dat dynamisch programmeren helpt om de discrete variant van KNAPSACK op te lossen. De grens die we vonden voor een dynamisch programmeren algoritme voor een KNAPSACK met n objecten en grens b was $n \times b$. We merken op dat, hoewel deze algoritme voor begrensde b een polynomiale tijd algoritme is, de gevonden grens toch exponentieel in de lengte van de invoer kan zijn, omdat b nu eenmaal in $\log b$ bits in de invoer kan worden gepresenteerd. Het bewijs dat KNAPSACK NP-volledig is, dat we hieronder geven, is dan ook niet in tegenspraak met onze onwetendheid omtrent de relatie tussen P en NP.

NAAM: KNAPSACK
GEGEVEN: Gegeven een verzameling gewichten $w_1, \ldots, w_n$ en een grens b
GEVRAAGD: Is er een indexverzameling I zó dat $\sum_{i \in I} w_i = b$?

Natuurlijk is KNAPSACK een probleem in NP. Immers gegeven een indexverzameling kunnen we gemakkelijk controleren dat de som van de gewichten van de elementen in deze verzameling precies b is. KNAPSACK is echter ook NP-volledig. We laten dit zien door een reductie van EXACT COVER. Laat eens een universum $U = \{1, \ldots, n\}$ gegeven zijn, en een stel verzamelingen S_j. We definieren $u_{ij} = 1 \Leftrightarrow i \in S_j$. Als we $b = n$ kiezen, dan zien we hier al dat wanneer er een overdekking bestaat met de S_j die een onderling lege doorsnede hebben, de bijbehorende u_{ij} precies tot n sommeren. We kunnen nu echter ook een stel verzamelingen kiezen zodat de bijbehorende u_{ij} tot n sommeren *zonder* dat deze verzamelingen een overdekking vormen. We kunnen bijvoorbeeld n keer de verzameling $\{1\}$ kiezen. De oplossing wordt gevonden door te verhinderen dat optellingen van elementen een carry tot gevolg hebben. Als het aantal deelverzamelingen m is, dan kunnen we nooit meer dan m keer hetzelfde element kiezen. In plaats van $u_{ij} = 1$ kiezen we $u_{ij} = m^i$ als i in S_j zit en 0 anders. Voor elke S_j maken we nu het getal $s_j = \sum_i u_{ij}$ en $b = \sum_{i=1}^{n} m^i$. Als er een stel s_j bestaat dat tot b sommeert, dan betekent dat dat voor elke macht van m precies één j in dit stel is zo dat $u_{ij} = m^i$, anders zien we in de som m^i niet met coëfficient 1 optreden. Dus als er een oplossing voor het KNAPSACK probleem is, dan is er ook een exacte overdekking en omgekeerd.

Het laatste probleem in deze serie is boedelscheiding. Hier moet niet een verzameling gevonden worden die precies een bepaald gewicht heeft, maar moet een gegeven verzameling in tweeën gedeeld worden zodat beide delen precies evenveel waarde hebben. In de praktijk is de taak van de verdeler (rechter) natuurlijk om twee verzamelingen te vinden die in waarde zo weinig mogelijk van elkaar verschillen, en wordt die keuze nog bemoeilijkt doordat hetzelfde object voor verschillende personen verschillende (emotionele) waarde kan hebben, maar in onze ideale wereld van de wiskunde kunnen we dit probleem als beslissingsprobleem stellen, als volgt.

NAAM: BOEDELSCHEIDING
GEGEVEN: Gegeven een verzameling gewichten $w_1, \ldots, w_n$
GEVRAAGD: Is er een indexverzameling I zó dat $\sum_{i \in I} w_i = \sum_{i \notin I} w_i$?

Ook BOEDELSCHEIDING is een probleem in NP. Immers, gegeven een index verzameling kunnen we gemakkelijk controleren dat de beide deelverzameling gelijke som hebben. BOEDELSCHEIDING is ook NP-volledig. Om dit te bewijzen geven we een reductie van KNAPSACK.

Laat gegeven zijn een stel gewichten $w_1, \ldots, w_n$ en een getal b. We maken een stel gewichten $w_1, \ldots, w_n, w_{n+1}, w_{n+2}$ waarvoor een boedelscheiding bestaat dan en slechts dan als de KNAPSACK een oplossing heeft. De gewichten $w_1, \ldots, w_n$ blijven gelijk. De nieuwe gewichten w_{n+1} en w_{n+2} maken we allereerst zo dat ze bij een

boedelscheiding niet allebei aan dezelfde kant van de indexverzameling terecht kunnen komen. Daarvoor maken we w_{n+1} gelijk aan $b+1$ en w_{n+2} gelijk aan $\sum w_i - b + 1$. Als w_{n+1} en w_{n+2} nu allebei in I of juist niet in I zitten, dan kunnen de overgebleven elementen nooit een zo grote som hebben dat dit kan worden gecompenseerd.

Veronderstel nu dat er een boedelscheiding mogelijk is. Dus dat er een indexverzameling I bestaat, zodat

$$\begin{array}{l} w_{n+2} + \sum_{k\in I} w_k = \\ \sum_{i\leq n} w_i - b + 1 + \sum_{k\in I} w_k = \\ b + 1 + \sum_{\ell\notin I} w_\ell = \\ w_{n+1} + \sum_{\ell\notin I} w_\ell \end{array}$$

.

Dan is $2\sum_{i\in I} w_i = 2b$, ofwel $\sum_{i\in I} w_i = b$

Kleuren

De laatste familie van problemen die we in deze tekst bespreken is de familie van de kleurenproblemen. Het beroemdste lid van deze familie is misschien wel het vierkleurenprobleem voor planaire grafen. Gegeven is een landkaart met landen die niet in één punt aan elkaar grenzen. Gevraagd is de landkaart met vier kleuren in te kleuren. Verrassend genoeg lukt dit altijd. Pas in de vorige eeuw werd bewezen door Appel en Haken dat een planaire graaf (dat is een graaf die in het platte vlak kan worden ingebed) met vier kleuren altijd zo kan worden gekleurd dat geen twee knopen die aan dezelfde kant zitten dezelfde kleur krijgen. Dit noemen we een legale kleuring. Het bewijs reduceerde eerst de oneindige verzameling van planaire grafen tot een eindig aantal typen (een paar duizend) waarvan vervolgens met behulp van een computerprogramma kon worden aangetoond dat ze allemaal vierkleurbaar zijn. Een inzichtelijk bewijs voor deze stelling ontbreekt echter altijd nog.

Voor planaire grafen en $K \geq 4$ is het probleem „kan de graaf met K kleuren worden gekleurd" dus een simpel probleem. Het antwoord is ja. Voor algemene grafen, en *ook* voor $K = 3$ is het probleem NP-volledig, zoals we hieronder kunnen zien. Voor $K = 2$ is het probleem opnieuw simpel, hoewel dan voor sommige grafen het antwoord ja en voor andere het antwoord nee is.

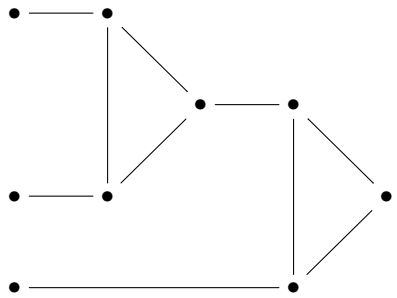
Het grafenkleurprobleem: „met hoeveel kleuren kan een graaf worden gekleurd?" is een uitbreiding van het onafhankelijke verzameling probleem. Immers een graaf moet worden onderverdeeld in een aantal onafhankleijke verzamelingen, en wat is het minimale aantal waarvoor dit mogelijk is? Het probleem wordt CHROMATIC NUMBER genoemd in Engelse teksten. Hier zullen wij het aanduiden met het KLEURGETAL van een graaf.

NAAM: KLEURGETAL
GEGEVEN: Gegeven een graaf G en een getal K
GEVRAAGD: Is er een legale kleuring van G met K kleuren?

Ook dit is een probleem in NP, want gegeven een oplossing kunnen we gemakkelijk controleren dat de regels van kleurbaarheid niet overschreden zijn en dat slechts K kleuren gebruikt zijn. Om aan te tonen dat het probleem ook NP-volledig is geven we een reductie van 3SAT.

Gegeven is dus een formule $F(x_1, \ldots, x_n)$, en we gaan een graaf maken die met $n+1$ kleuren te kleuren is dan en slechts dan als de formule vervulbaar is. We nemen zonder beperking der algemeenheid aan dat $n \geq 4$. Allereerst maken we een complete graaf op n knopen, B, die ervoor zorgt dat de meeste kleuren vergeven zijn. Om deze graaf te kleuren hebben we tenminste n kleuren nodig. Vervolgens voeren we $2n$ knopen x_i en $\overline{x_i}$ in die we paarsgewijs met elkaar verbinden, zodat ze niet dezelfde kleur kunnen krijgen. Deze knopen stellen de variabelen voor. Bovendien worden de knopen uit elk paar met alle knopen behalve 1 uit B verbonden. Hoogstens één van deze knopen kan dus de kleur van deze ene knoop uit B krijgen. De andere knoop moet de enig overgebleven $n+1$e kleur krijgen, deze kleur zal de kleur "onwaar" voorstellen. We voeren m knopen c_j in. Deze knopen stellen de zinnen voor. Tenslotte zullen c_j verbonden zijn met x_i ($\overline{x_i}$) als x_i ($\overline{x_i}$) *niet* in de zin c_j voorkomt.

We beweren nu dat de aldus geconstrueerde graaf gekleurd kan worden met $n+1$ kleuren dan en slechts dan als F vervulbaar is. Laten we eerst eens kijken hoe deze graaf gekleurd zou kunnen worden. We hadden



Figuur 6.8: Gadget voor 3-Kleurbaarheid

al gezien dat er n kleuren voor de deelgraaf B nodig zijn. Van de paren x_i, $\overline{x_i}$ is er één die de $n+1$e kleur moet krijgen. Alle andere moeten één van de n eerste kleuren krijgen. Verder kan geen van x_i, $\overline{x_i}$, x_j, $\overline{x_j}$ dezelfde kleur kan krijgen voor $i \neq j$. Omdat $n \geq 4$, is elke knoop c_j wel verbonden met zowel x_i als $\overline{x_i}$ voor één of andere i. Dus c_j kan niet met kleur $n+1$ gekleurd worden. In een legale kleuring kan c_j dus alleen de kleur van een x_i of een $\overline{x_i}$ krijgen die *wel* in de $j$de zin voorkomt en die *niet* de $n+1$e kleur (onwaar) gekregen heeft. Dat betekent dus dat als de graaf gekleurd kan worden, we in elke zin een x_i of $\overline{x_i}$ kunnen kiezen die waargemaakt kan worden en dus dat F waargemaakt kan worden. Dit recept, in omgekeerde vorm, geeft tevens bij iedere toewijzing die F waarmaakt een kleuring van de graaf.

NAAM: 3-KLEURBAARHEID
GEGEVEN: Gegeven een graaf G
GEVRAAGD: Is er een legale 3-kleuring van G?

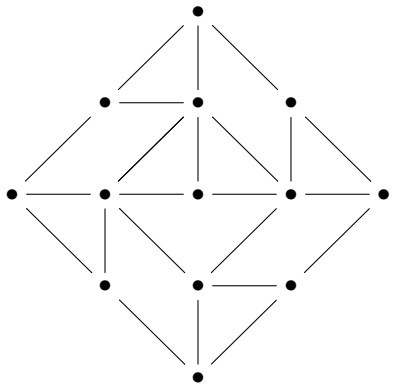
Als het algemene kleuringsprobleem een probleem in NP is, geldt dat natuurlijk ook voor het probleem als we het aantal kleuren tot drie beperken. Wat opmerkelijker is, is dat deze beperking het probleem niet makkelijker maakt. Om dit te bewijzen geven we weer een reductie van 3SAT. In dit geval hebben we, zoals in het geval van de reductie van VERTEX COVER naar HAMILTON CIRCUIT kleine graafjes nodig met een bijzondere eigenschap, zogenoemde gadgets. Hier maken we gebruik van de graafjes uit Figuur 6.8

Gegeven drie kleuren, bijvoorbeeld r, b en z heeft deze graaf de eigenschap dat als alle drie de knopen aan de linkerkant dezelfde kleur krijgen, bijvoorbeeld z, dan wordt de meest rechtse knoop ook met deze kleur gekleurd. Dit is een mooie karakterizatie voor 3-SAT. Als namelijk alle optredens van variabelen in een zin van 3-SAT de waarde *onwaar* krijgen, dan krijgt de zin ook deze waarde. Nu voeren we voor elke variabele x_i twee knopen x_i, en $\overline{x_i}$ in, die we met elkaar verbinden, zodat ze niet dezelfde kleur kunnen krijgen en we verbinden *al* deze knopen met een derde knoop R, zodat er maar twee kleuren overblijven om x_i en $\overline{x_i}$ te kleuren. Voor elke zin in de formule nemen we een graafje zoals in Figuur 6.8, en we verbinden alle rechterknopen van deze figuurtjes met R zodat de rechterkanten één van de twee kleuren moeten krijgen die we voor x_i en $\overline{x_i}$ gebruiken. We verbinden alle rechterkanten nog met een extra knoop F die zelf met R verbonden is, zodat er nog maar één kleur voor de rechterkanten overblijft. Nu krijgen alle linkerkanten van de figuurtjes de naam van het optreden van de variabele op die plaats in die zin (dus x_i of $\overline{x_j}$) en we smelten alle knopen die dezelfde naam hebben tot één knoop.

Als nu de resulterende graaf gekleurd kan worden, dan hebben alle rechterkanten van de figuurtjes dezelfde kleur, en deze kleur komt op minstens één plaats aan de linkerkant voor. Dat betekent, als we deze kleur even *waar* noemen, dat in elke zin een optreden van een variabele gekozen kan worden die waargemaakt kan worden, en dat betekent weer dat F waargemaakt kan worden. Omgekeerd geeft een toewijzing van waarheidswaarden die F waarmaakt met dit recept een legale driekleuring van de graaf.

NAAM: PLANAIRE 3-KLEURBAARHEID
GEGEVEN: Gegeven een planaire graaf G
GEVRAAGD: Is er een legale 3-kleuring van G?

We hebben een nog specialer geval van een probleem in NP dus ook dit probleem is een probleem in NP. Een gegeven oplossing kan gemakkelijk gecontroleerd worden. Zelfs planariteit maakt echter het probleem



Figuur 6.9: Crossover Component

niet makkelijker als het aantal kleuren beperkt wordt tot drie. Om dit te bewijzen laten we zien dat we elk paar kanten die de graaf niet planair maken, omdat ze niet in twee dimensies kunnen worden ingebed, kunnen worden vervangen door een component die wel planair is, zo dat de uiteinden van die component de kleur „bewaren". We maken een graaf die de eigenschap heeft dat hij planair en driekleurbaar is en dat de kleur van de meest linker knoop dezelfde moet zijn als de kleur van de meest rechter knoop en dat de kleur van de bovenste knoop dezelfde moet zijn als de kleur van de onderste knoop. Deze graaf wordt gegeven in Figuur 6.9. Als we een niet planaire graaf hebben, dan kunnen we de paren kanten die in drie dimensies kruisen paar voor paar vervangen door zo'n crossover component en we houden een platte graaf over. Elke driekleuring van de oorspronkelijke graaf geeft een driekleuring van de nieuwe graaf en omgekeerd, vanwege de eigenschap van deze crossover componenten.

6.5.2 Beslissen, Optimalisatie en Zelfreduceerbaarheid

NP-volledige problemen zoals we in hierboven hebben gezien komen in de praktijk vaak voor als optimaliseringsproblemen. Er wordt niet gevraagd: „is er een overdekking waarbij de paarsgewijze doorsneden leeg zijn?", maar: „wat is de overdekking waarbij het totale aantal vaker voorkomende elementen zo klein mogelijk is?" Niet: „is er een boedelscheiding waarbij de ene verzameling precies evenveel waard is als de andere?", maar: „bij welke boedelscheiding is het verschil zo klein mogelijk?", etc. etc. Ook willen we graag van een algoritme die een antwoord op ons probleem geeft graag wat meer informatie dan alleen maar ja/nee. Als we een algoritme hebben die beslist of een graaf op 500 knopen 32 kleurbaar is, dan is „ja" misschien niet een antwoord waar we erg gelukkig van worden. We willen zo'n kleuring ook graag in handen hebben.

Tussen beslissen en optimaliseren is gelukkig een eenvoudig verband. Een snelle beslissingsalgoritme geeft een snelle optimaliseringsalgoritme door tussenkomst van binair zoeken, en een snelle optimaliseringsalgoritme geeft triviaal een snelle beslissingsalgoritme. Toch valt er nog iets meer over te zeggen. Een algoritme voor het beslissen van VERTEX COVER geeft voor de paren G, k, waarbij G een graaf is en k een aantal knopen het antwoord "ja" als er een vertex cover van grootte hoogstens k in de graaf zit. Voor zo'n vertex cover bestaat een kort bewijs, bijvoorbeeld zo'n verzameling knopen. Het is echter helemaal niet duidelijk wat een kort bewijs zou zijn voor de uitspraak G heeft *geen* vertex cover van grootte k. Het complement van het probleem VERTEX COVER bestaant uit alle paren G, k waarvoor de kleinste vertex cover in G *groter* is dan k. Dit zou dus wel eens wezenlijk moeilijker kunnen zijn dan VERTEX COVER zelf. Uiteraard is niet het geval wanneer er een deterministisch polynomiale tijd algoritme voor VERTEX COVER zou zijn. Problemen die van de vorm zijn als in dit voorbeeld—de kleinste vertex cover in G heeft grootte k—vormen een aparte klasse. Ze kunnen worden opgelost door een deterministisch polynomiale tijd machine die toegang heeft tot vragen aan NP-volledige problemen. Deze klasse wordt Δ_2^P genoemd. Ook Δ_2^P heeft volledige problemen. Een bekend voorbeeld is ODDMINSAT, „de lexicografisch kleinste vervulling van een formule is

oneven".

Als we een algoritme voor een beslissingsprobleem hebben, dan willen we ook wel graag een oplossing in handen hebben (dit wordt het „witness probleem" genoemd). Ook hier is er bij de meeste NP-volledige problemen een eenvoudig verband aan te geven. De meeste NP-volledige problemen zijn zelfreduceerbaar, dwz. terug te voeren op kleinere instanties van zichzelf. Uit zo'n zelfreductie kunnen we een algoritme bouwen die ons een oplossing in handen geeft. We geven een paar voorbeelden.

1. SATISFIABILITY. Stel we hebben een algoritme die een antwoord kan geven op de vraag of een formule $F(x_1, x_2, \ldots, x_n)$ vervulbaar is. F is alleen vervulbaar als $F(0, x_2, \ldots, x_n)$ vervulbaar is *of* $F(1, x_2, \ldots, x_n)$ vervulbaar is. Een hypothetische beslissingsalgoritme voor SATISFIABILITY kan nu recursief gebruikt worden om een vervulling te vinden door de vervulbaarheid van $F(1, x_2, \ldots, x_n)$ en/of $F(0, x_2, \ldots, x_n)$ te beslissen en het antwoord hierop te gebruiken voor verdere recursieve aanroepen. De onderstaande algoritme vindt een vervulling voor een *vervulbare* formule $F(x_1, \ldots, x_n)$. Om de notatie eenvoudiger te maken nemen we aan dat we in F een bitstring s kunnen invullen, en dat $F(s)$ betekent F met x_i gelijk aan het $i$de bit van s. Verder is $s0$ de bitstring s met aan het einde een 0 geplakt en $s1$ de bitstring s met aan het einde een 1 geplakt. De aanroep SAT$(F, \emptyset)$; Het diepste niveau van de recursie drukt een bitstring van lengte n af die een vervulling voorstelt.

```
1: function SAT(F, s)
2: if |s| < n then
3:     if F(s0) ∈ SAT then
4:         SAT(F,s0);
5:     else
6:         SAT(F,s1);
7:     end if
8: else
9:     print(s);
10: end if
```

2. VERTEX COVER. Bij dit probleem merken we op dat van elk tweetal punten dat door een kant verbonden wordt, tenminste één in een VERTEX COVER zit. De recursieve aanroep van de procedure kiest dus één van die punten en vraagt of de graaf zonder dat punt een VERTEX COVER heeft die één punt minder bevat. In de onderstaande algoritme is $G - \{v\}$ de graaf G waaruit het punt v en alle kanten die v als eindpunt hebben verwijderd zijn. De algoritme vindt dan een verzameling van k punten die een vertex cover van G zijn, als we beginnen met een G die *inderdaad* zo'n VERTEX COVER heeft.

```
1: procedure VC(G, k)
2: if G has edges then
3:     Choose an edge (v, w) in G;
4:     if (G − {v}, k − 1) ∈ VERTEXCOVER then
5:         print(v); VC(G − {v}, k − 1);
6:     elseprint(w); VC(G − {w}, k − 1);
7:     end if
8: end if
```

3. HAMILTON CIRCUIT. Stel we kiezen een kant e in G. Ofwel de graaf $G - \{e\}$ heeft nog steeds een Hamilton circuit ofwel $G - \{e\}$ heeft geen Hamilton circuit meer. In het tweede geval (aangenomen dat G zelf wel een Hamilton circuit heeft, is e een kant in *elk* Hamilton circuit en heeft $G/\{e\}$—de graaf die ontstaat als we beide einpunten van e samentrekken tot één punt—weer *wel* een Hamilton circuit. Deze algoritme kunnen we gebruiken totdat een graaf overblijft die klein genoeg is om het overgebleven Hamilton circuit te identificeren (bijvoorbeeld als G tot een driehoek gereduceerd is).

```
1: procedure HAM(G);
2: if G is bigger than a triangle then
3:     Choose e in G
```

```
    if G − {e} ∈ HAMILTONCIRCUIT then
        HAM(G − {e});
    else
        print(e);
        HAM(G/{e})
    end if
else
    print(the edges in G);
end if
```

Op deze wijze kunnen we voor vrijwel alle besproken problemen een recursieve algoritme vinden die een certificaat produceert voor een instantie die tot het probleem behoort. In deze recursieve algoritmen wordt echter telkens in een **if** statement het oplossen van een NP-volledig probleem gevraagd. Omdat echter een certificaat wordt geproduceerd, kan deze **if** zelf echter weer vervangen worden door een recursieve aanroep, waardoor een recursieve algoritme voor het beslissingsprobleem zelf ontstaat. We nemen als voorbeeld VERTEX COVER. Nu is de aanroep VC(G, k) en de functie VC geeft nu 0 of 1 terug als de ingevoerde graaf geen, resp. wel een Vertex Cover van grootte k heeft.

```
function VC(G = (V, E), k)
if  thenk ≥ ‖E‖ return(1)
end if
if  then‖E‖ > 0
    Choose edge (v, w) in G
    if VC(G − {v}, k − 1) then
        return(1);
    else if VC(G − {w}, k − 1) then
        return(1);
    else
        return(0);
    end if
end if
```

Een aldus verkregen recursieve algoritme voor een NP-volledig probleem moet altijd tenminste twee recursieve aanroepen hebben, omdat anders het probleem in P terecht zou komen en we een P=NP bewijs hebben. In sommige gevallen, zoals bijvoorbeeld in het geval van Traveling Sales Person zien we zelfs meer dan één recursieve aanroep. De meeste bekende NP-volledige problemen laten op deze wijze een recursieve algoritme toe. De recursieve aanroepen zijn disjuncties. Als één van de recursieve aanroepen slaagt is het antwoord „ja". Bovendien kan het aantal aanroepen vaak tot twee (maar niet tot minder) beperkt worden. We noemen deze speciale vorm van zelfreduceerbaarheid twee-disjunctieve zelfreduceerbaarheid. Het is helaas onbekend of misschien alle NP-volledige problemen zelfreduceerbaar zijn. Omgekeerd is het wel bekend dat alle zelfreduceerbare problemen in de grotere complexiteitsklasse PSPACE zitten.

6.6 Sommen

1. Bewijs dat er deterministisch polynomiale tijd begrensde algoritmen bestaan voor

   (a) 2-SAT. SATISFIABILITY van proposities in conjunctieve normaalvorm met hoogstens twee optredens van variabelen per conjunctie.

   (b) Graph 2-colorability. Is het mogelijk een graaf te kleuren met slechts twee verschillende kleuren.

2. Bewijs de NP-volledigheid van de volgende problemen:

   (a) NAAM: DUAL-SATISFIABILITY
   GEGEVEN: Een formule $F(x_1, \ldots, x_n)$

GEVRAAGD: Zijn er minstens twee verschillende toewijzingen aan $x_1, \ldots, x_n$ die F waar maken?

(b) NAAM: DUAL-VERTEX COVER
GEGEVEN: Een graaf $G = (V, E)$ en een getal k
GEVRAAGD: Zijn er tenminste twee verschilende verzamelingen $V' \subseteq V$ met $\|V'\| \leq k$ die allebei een vertex cover van G zijn?

(c) NAAM: HITTING SET
GEGEVEN: Een stel verzamelingen getallen $S_i \subseteq V$ en een getal k.
GEVRAAGD: Is er een deelverzameling $V' \subseteq V$ met $\|V\| \leq k$ en $\|V' \cap S_i\| \geq 1$ voor alle i?

(d) NAAM: FEEDBACK VERTEX SET
GEGEVEN: Een graaf $G = (V, E)$ en een getal k.
GEVRAAGD: Is er een deelverzameling $V' \subseteq V$ met $\|V'\| \leq k$ zo dat elke cykel van G tenminste één knoop van V' raakt?

(e) NAAM: XXX
GEGEVEN: Een collectie deelverzamelingen $\{S_j\}_j$ van $U = \{1, \ldots, n\}$ en twee getallen k en ℓ
GEVRAAGD: Bestaat er een deelverzameling $S = \{u_{i_1}, \ldots, u_{i_k}\}$ zó dat voor ℓ verschillende j's geldt $S_j \subseteq S$?

(f) NAAM: YYY
GEGEVEN: Een collectie deelverzamelingen $\{S_j\}_j$ van $U = \{1, \ldots, n\}$ en twee getallen k en ℓ
GEVRAAGD: Bestaat er een deelverzameling $S = \{u_{i_1}, \ldots, u_{i_k}\}$ zó dat alle j's geldt $\|S_j \cap S\| \leq 1$?

3. Van de onderstaande problemen zijn er drie in P en is er één NP-volledig. Welke zijn dit?

   (a) Gegeven een graaf G bestaat er een onafhankelijke verzameling van grootte 3?

   (b) Gegeven een gerichte gewogen volledige graaf $K = (V, A)$ en een getal k, is de som van de gewichten der kanten in *elk* Hamilton circuit groter dan k?

   (c) Gegeven een ongerichte graaf $G = (V, E)$; bestaat er een $V' \subset V$, zó dat alle kanten uit E incident zijn met een knoop uit V'?

   (d) Gegeven een zoekboom horende bij een independent set probleem en een getal k; stelt één van de bladeren van de boom een onafhankelijke verzameling van meer dan k knopen voor?

   (e) Gegeven een graaf $G = (V, E)$ en een getal k; bestaat de grootste onafhankelijke verzameling in G uit minder dan k knopen?

   (f) Gegeven een collectie deelverzamelingen $\{S_i\}_i$ van $U = \{u_1, \ldots, u_n\}$; bestaat er een $W \subseteq U$ zo dat $\|W \cap S_i\| = 1$ voor alle i?

6.7 NP-problemen en de Praktijk

In de vorige sectie hebben we van een aantal problemen aangetoond dat ze NP-volledig zijn. Een NP-volledigheidsbewijs voor een probleem wordt doorgaans gegeven om aan te geven dat een algoritmische aanpak van het probleem weinig zin heeft. In de praktijk echter kunnen we vaak niet ervoor kiezen om een probleem maar te laten liggen als het algoritmisch moelijk blijkt te zijn. Een NP-volledigheidsbewijs is dan ook veel meer een aanleiding om naar alternatieve oplossingen voor het probleem te zoeken nu is gebleken dat zoeken naar een efficiente algoritme voor het algemene probleem waarschijnlijk hopeloos is. Een paar mogelijke uitwegen zijn.

1. Probeer een beperkte algoritme voor een speciaal geval van het probleem te vinden. Als bijvoorbeeld het probleem het vinden van een kleuring is, en het probleem beperkt kan worden tot planaire grafen, dan hoeft het probleem niet zo moeilijk te zijn.

2. Probeer een algoritme te vinden die in de meeste gevallen in polynomiale tijd werkt en alleen op een (zo klein mogelijk) deel van de mogelijke invoeren exhaustive search gebruikt (en dus exponentieel is). Een

bekend voorbeeld hiervan is de in de economie gebruikte simplex algoritme voor lineair programmeren. Voor een lineaire functie in n variabelen zoekt men een optimale waarde die voldoet aan een serie lineaire beperkingen. De lineaire beperkingen kunnen in n dimensies gezien worden als een polytoop waarop de lineaire functie gedefinieerd is. Een extreme waarde wordt dan altijd aangenomen in één van de hoekpunten van het polytoop. Een algoritme om het maximum te bepalen is dus van het ene naar het andere hoekpunt lopen om te kijken waar het maximum wordt aangenomen. Dit is precies wat de simplexalgoritme, met behup van een aantal lineaire transformaties, doet. De algoritme is eenvoudig te implementeren en te begrijpen en voor kleine gevallen (niet teveel beperkingen) zelfs met de hand makkelijk uit te voeren. Er zijn gevallen te construeren waarin de algoritme exponentiële looptijd heeft, maar deze komen in de praktijk niet voor. Daarom is deze methode, hoewel er al lang een polynomiaal begrensde algoritme voor dit probleem bestaat [Kha80], nog steeds in gebruik.

3. Probeer een algoritme te vinden die gebruik maakt van coinflips, zó dat die algoritme voor alle inputs in de meeste gevallen in polynomiale tijd werkt. Sinds enige tijd is bekend dat primaliteitstesten in polynomiale tijd kan [AKS04]. De algoritme hiervoor is echter nog niet wijdverbreid geïmplementeerd, misschien ook omdat de snelste algoritmen hoewel polynomiaal nog steeds van vrij hoge graad zijn. Probabilistische algoritmen voor primaliteitstesten als in deze tekst beschreven zijn wel wijdverbreid, gemakkelijk te implementeren en ruim voorhanden. Daarom zijn deze, hoewel in principe exponentieel, nog steeds zeer in trek.

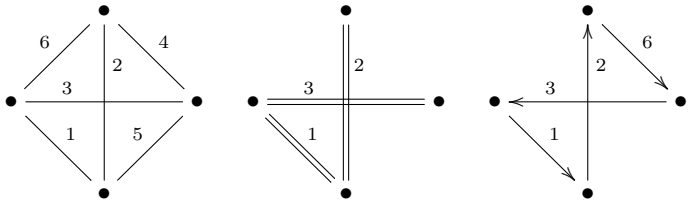
4. Probeer een algoritme te vinden die een suboptimale oplossing geeft. De problemen die we in dit hoofdstuk besproken hebben zijn alle beslissingsproblemen. Bestaat er, is er etc. Nauw verwant met deze problemen zijn de optimaliseringsproblemen. Wat is de kortste route door een volledige gewogen graaf langs alle punten, wat is de verzameling objecten in de knapsack die het grootste gewicht kleiner dan b geeft, wat is de kleinste verzameling knopen die een vertex cover van de graaf geeft etc. Het is duidelijk dat als het optimaliseringsprobleem polynomiale tijd begrensd is, dat dan ook het beslissingsprobleem polynomiale tijd begrensd is en omgekeerd (via binary search). Echter als het beslissingsprobleem moelijk is, dan hoeft het niet altijd zo te zijn dat het optimaliseringsprobleem ook moelijk is. Mits niet gevraagd wordt naar de beste oplossing, maar bijvoorbeeld naar een acceptabele oplossing.

6.7.1 SAT, KNAPSACK, en VC

Soms bestaan er eenvoudige gulzige algoritmen die weliswaar niet een optimaal resultaat geven, maar toch een resultaat in de buurt van een oplossing. We hebben al eerder gezien dat SAT met twee variabelen per zin een probleem in P is geworden, net zoals bijvoorbeeld tweekleurbaarheid. De NP-volledige variant van SAT, die met meer variabelen per zin, heeft weliswaar geen polynomiale algoritme, maar met een eenvoudige deterministische algoritme kan toch een verrassend resultaat behaald worden. Neem een SAT formule van m zinnen. Elke toewijzing van waarheidswaarden vervult een fractie van die zinnen. Als de formule vervulbaar is, dan is er minstens één toewijzing die ze allemaal vervult. Er is een deterministische algoritme die tenminste de helft van het maximaal aantal vervulbare zinnen vervult. Als volgt: Als $x_1 = 1$ meer dan de helft van de zinnen waarin x_1 voorkomt verult, zet dan $x_1 = 1$, anders zet $x_1 = 0$. Schrap alle zinnen die aldus waargemaakt zijn en ga door met x_2.

Voor 3SAT is de situatie nog interessanter dan voor SAT. Als je maar 3 variabelen per zin hebt, dan is er maar 1 van de 8 mogelijke toewijzingen van waarheidswaarden aan die variabelen, die die zin onwaar maakt. Dat betekent dat de kans dat een random toewijzing de zin waarmaakt 7/8 is. Wegens het feit dat de verwachtingswaarde een lineaire functie is, maakt dit dat de verwachting dat een willekeurige toewijzing van waarheidswaarden fractie waargemaakte zinnen door een willekeurige toewijzing van waarheidswaarden 7/8 is.

Voor Vertex Cover bestaat ook simpele algoritme. Begin met een lege verzameling VC. Neem een willekeurige kant, stop beide eindpunten in VC en schrap vervolgens deze kant en alle kanten die in die eindpunten uitkomen. Ga verder totdat de graaf geen kanten meer heeft. De geproduceerde verzameling VC bestaat uit een verzameling eindpunten van losse kanten, een matching. Omdat elke vertex cover (ook de minimale) van



Figuur 6.10: Van graaf naar TSP cykel

elke kant tenminste één eindpunt heeft, zit precies de helft van alle punten in VC ook in de minimale vertex cover (maar mogelijk meer) VC is dus een vertex cover die hoogstens twee keer zo groot is als de minimale.

Hoewel Vertex Cover en Independent Set heel gemakkelijk tot elkaar te reduceren zijn, kunnen we een benaderingsalgoritme voor Vertex Cover niet gebruiken om een benadering van Independent Set te verkrijgen. Stel dat we een VC kunnen krijgen die ρ te slecht is, dus voor zekere $\rho > 1$ krijgen we in een graaf met n knopen een vertex cover van ρK als er een vertex cover van K in zit. Als in een graaf van n knopen een IS van groote M zit, dan zit er in die graaf een VC van grootte $n - M$. Een ρ slechte benaderingsalgoritme zou dan een VC van grootte $\rho(n - M)$ geven, die weer een IS van grootte $n - \rho(n - M)$ zou geven. Hieraan zie je niet alleen dat M minstens een $(\rho - 1)/\rho)$ fractie van n moet zijn om ueberhaupt een garantie van meer dan 0 knopen in de independent set te krijgen, maar bovendien dat de verkregen benaderingsfractie afhangt van n. Er kan worden bewezen, maar valt buiten deze tekst, dat voor independent set niet een dergelijke benaderingsalgoritme kan bestaan, tenzij P=NP.

We hebben voor KNAPSACK al gezien dat, als b klein is, een polynomiale algoritme gemaakt kan worden met behulp van dynamic programming. Deze zelfde algoritme kan gebruikt worden om een benadering van de optimale waarde in polynomiale tijd te krijgen. We definieren eenvoudig een getal $K = \epsilon W/n$ waar W het maximale gewicht van alle objecten is, delen vervolgens alle waarden in het probleem door K en voeren de dynamic programming algoritme uit. Deze levert een optimale waarde O op die hoogstens een factor ϵ slechter is dan die van het oorspronkelijke probleem.

6.7.2 Een benadering voor TSP

Een probleem dat zowel onder de eerste als de laatste categorie hierboven valt is het handelsreizigerprobleem. Het algemene geval van het handelsreizigerprobleem is moeilijk, NP-volledig. In de praktijk hebben we echter vrijwel altijd te maken met een speciaal geval van het handelsreizigerprobleem, namelijk het probleem dat zich afspeelt op een kaart, de Euclidische versie van het probleem. Dit probleem heeft doorgaans de eigenschap dat de driehoeksongelijkheid geldt. De som van de afstanden van A naar B en van B naar C is altijd groter dan of gelijk aan de afstand van A naar C. Op een graaf waar de driehoeksongelijkheid geldt, kunnen we met behulp van een algoritme voor een bekend probleem een polynomiale tijd algoritme geven voor het handelsreizigerprobleem die *nooit* slechter is dan tweemaal de optimale route voor het echte probleem. De algoritme die we gebruiken is één van de algoritmen voor de opspannende boom van minimale kosten en gaat als volgt.

1: input een volledige gewogen graaf K waarin de driehoeksongelijkheid geldt.
2: Bereken een opspannende boom van K.
3: Verdubbel alle kanten.
4: Begin een cykel in een kant en loop telkens naar de volgende knoop en markeer deze als "bezocht".
5: Sla in de cykel reeds bezochte knopen over door naar de volgende knoop te springen. Daarbij verwijderen we de twee kanten in de opspannende boom en vervangen deze door één kant uit K. Zonodig wordt deze operatie herhaald uitgevoerd.

Figuur 6.10 geeft een voorbeeldje van de transformatie van graaf naar spanning tree naar cykel.

6.7.3 Branch en Bound

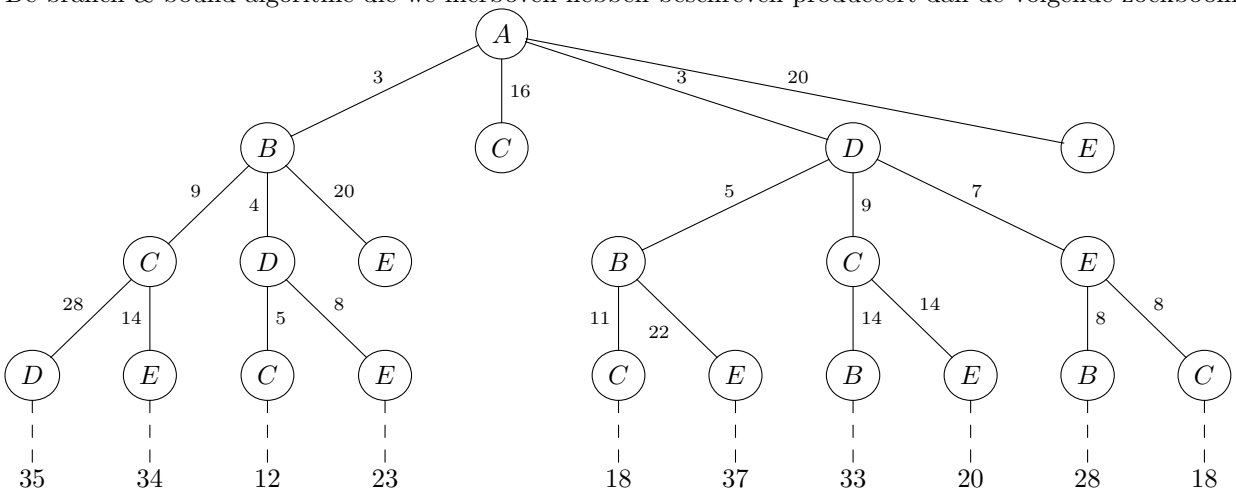
Het doorzoeken van alle mogelijke oplossingen voor een NP-volledig probleem komt vaak neer op het doorzoeken van een exponentieel grote boom. Bijvoorbeeld het zoeken van een oplossing voor SATISFIABILITY kunnen we opvatten als een boom doordat we zeggen dat we eerst voor de waarde $x_1 = 0$ alle mogelijke oplossingen voor $x_2, \ldots, x_n$ proberen om vervolgens voor alle oplossingen bij $x_1 = 1$ alle oplossingen voor x_2 t.e.m. x_n proberen. We reduceren als het ware het oplossen van het probleem SATISFIABILITY tot het oplossen van twee eenvoudiger problemen, dat wil zeggen problemen met minder variabelen. Zulke problemen zijn onder de NP-volledige problemen zeer talrijk. We noemen dit soort problemen *zelf-reduceerbaar*(zie 6.5.2). We kunnen ze oplossen door het antwoord te zoeken op problemen van dezelfde soort maar van kleinere omvang. Dit soort problemen leent zich uiteraard expliciet voor recursieve programma's, al kost zo'n recursief programma in het onderhavige geval natuurlijk wel exponentiële tijd.

In het geval van optimaliseringsproblemen kunnen we misschien wat aan die exponentiële tijd doen, omdat we, althans in de meeste gevallen, niet de gehele zoekruimte hoeven te doorlopen. Als voorbeeld nemen we weer het handelsreiziger probleem. Hoe is het handelsreizigerprobleem zelfreduceerbaar? We bespreken twee methoden. Allereerst moeten we van elke stad naar een andere stad totdat een rondje gemaakt is. Om een stad te bezoeken hebben we de keuze uit elk van zijn buren. Kiezen we een buur, dan betekent dat dat we alle andere buren niet kiezen (behalve op de terugweg natuurlijk). Als we kiezen voor b als buur van a dan betekent dat dat de minimaal mogelijke tour met het gewicht van de kant (a, b) toeneemt. Als we zo een aantal kanten gekozen hebben weten we dus dat de minimale tour langs die kanten groter dan of gelijk is aan de som van de gewichten van die kanten. De *branch* heuristiek laat ons telkens kiezen de knoop in de zoekboom verder te ontwikkelen waarvoor deze waarde minimaal is. Hebben we op deze manier eenmaal $n - 2$ knopen gekozen, dan ligt de laatste knoop in het rijtje vast en kunnen we de lengte van de gekozen tour uitrekenen, de *bound* heuristiek zegt dan dat geen enkele knoop in de boom waarvan de minimaal te verwachten waarde *groter dan of gelijk* is aan de waarde van zo'n complete tour verder uitgewerkt moet worden. Op deze manier voorkomen we in veel gevallen dat we de complete boom moeten doorzoeken. Aangezien alle mogelijke paden nog steeds in de boom voorkomen, is de worst-case complexiteit van deze aanpak echter nog steeds $n!$.

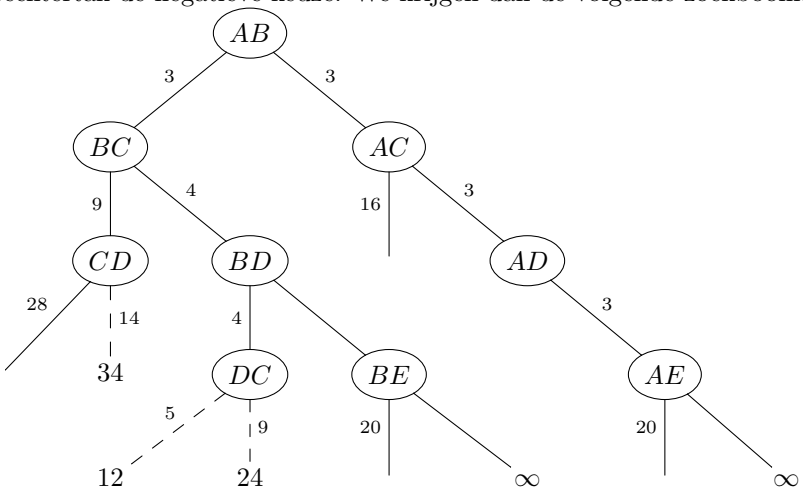
Voorbeeld 6.7.1: Stel een afstandstabel tussen vijf steden is als volgt gegeven:

	A	B	C	D	E
A	-	3	16	3	20
B	5	-	6	1	17
C	14	4	-	19	5
D	1	2	1	-	4
E	2	1	1	19	-

De branch & bound algoritme die we hierboven hebben beschreven produceert dan de volgende zoekboom:



We behandelen aan de hand van dit voorbeeld nog een andere methode om branch & bound te spelen met het handelsreizigerprobleem. In plaats van dat we telkens een buur van een stad kiezen kunnen we voor elke kant kiezen dat die wel of geen deel uitmaakt van een rondje. Dat heeft voor- en nadelen. Aan de ene kant wordt de breedte van de zoekboom kleiner (er zijn twee keuzemogelijkheden), aan de andere kant is de algoritme wat ingewikkelder. We nemen hetzelfde voorbeeld. De wel/niet keuze voor de kanten laten we langs de rijen lopen (dus eerst kant (A,B), dan (A,C) etc. De keuze voor kant (A,B) in het rondje sluit onmiddellijk de kanten (A,C), (A,D), en (A,E) uit en maakt de minimaal te bereiken tour tenminste 3, de keuze *tegen* (A,B) sluit (A,B) uit en maakt de minimaal te bereiken tour *ook* 3, maar nu omdat dat het minimum gewicht van de kanten (A,C), (A,D), en (A,E) is. Als vier kanten zijn opgenomen is de vierde verplicht en de lengte van de tour bekend. De diepte van de boom kan nu echter groter zijn dan 4, omdat negatieve keuzen zijn opgenomen. We laten steeds de linkertak de positieve keuze voor de kant zijn en de rechtertak de negatieve keuze. We krijgen dan de volgende zoekboom:



Ook hier zien we (gelukkig) dat de optimale route lengte 12 heeft en langs de kanten AB, BD, DC (en CE, en EA) voert. Ook krijgen we hier een werkelijke verbetering van de worst-case complexiteit. Als er n knopen in de graaf zijn, zijn er namelijk $O(n^2)$ kanten. Dat betekent dat onze algoritme een worst-case complexiteit heeft van $O(2^{n^2})$ nog steeds aanzienlijk, maar een opmerkelijke verbetering ten opzichte van $O(n!)$. □

De oplettende lezer zal hier opmerken dat de eerder besproken A^* algoritme (zie 3.2) niet veel meer is dan het toepassen van het Branch & bound principe op kortste paden in grafen.

Deel III

Extra Onderwerpen

In dit deel van de tekst behandelen we een aantal onderwerpen die te maken hebben met geheugencomplexiteit van problemen. Geheugencomplexiteit is een andere belangrijke maat voor de complexiteit van algoritmen en problemen, maar de onderwerpen die de tijdcomplexiteit betreffen vullen doorgaans een inleidende cursus geheel. Als er tijd over is omdat bijvoorbeeld aan het begin wat sneller gegaan kan worden als het publiek wat meer voorkennis heeft zouden deze onderwerpen niettemin aan bod kunnen komen. Ook valt te denken aan het bestuderen van deze onderwerpen achteraf in het kader van zelfstudie. In dit deel zijn vooralsnog geen sommen opgenomen.

Hoofdstuk 7

Meer Complexiteitsklassen

In dit hoofdstuk zullen we een aantal complexiteitsklassen bespreken die niet onmiddelijk onderwerp zijn in een inleidende cursus Algoritmen en Complexiteit. Tijd is een belangrijke complexiteitsmaat, maar er is natuurlijk ook nog een aantal andere maten van complexiteit die we aan algoritmen kunnen meten. Geheugencomplexiteit is daar één van, maar ook de grootte van circuits die nodig is om een probleem aan te pakken is een belangrijke complexiteitsmaat in het kader van de afmetingen van special purpose hardware. De diepte van een circuit (langste pad in een graaf) is een maat voor de parallelle complexiteit van een probleem, of de mate waarin een algoritme voor een probleem wel of niet geparallelliseerd kan worden. In Hoofdstuk 6 hebben we gezien dat we de moelijkheid van problemen kunnen aantonen door reductie van een NP-volledig probleem. Voor parallellizeerbaarheid is een soortgelijk gereedschap voorhanden. We kunnen de inherente sequentialiteit (ontmoediging voor het zoeken naar parallelle algoritmen) door een reductie van een circuitprobleem te geven die dan niet in polynomiale tijd, maar in logarithmisch begrensd geheugen is uit te voeren. Zo een reductie toont de P-volledigheid van een probleem aan. Veel van deze reducties kunnen gevonden worden in het boek [GHR95]. Vooralsnog bespreken we dit onderwerp niet in deze tekst.

7.1 Geheugenbegrensde Klassen

Als tegenhanger van Polynomiale tijd in het domein van de geheugenbegrensde theorie bekijken we Polynomiaal geheugen, PSPACE. Het is duidelijk dat alles dat in polynomiaal begrensde tijd kan ook in polynomiaal begrensd geheugen kan. Immers, een Turing machine kan in één stap niet meer dan één nieuwe geheugencel aanspreken. Aan de bovenkant is polynomiaal begrensd geheugen begrensd door exponentiële tijd. Dit is in te zien door naar het Turing machinemodel te kijken. Een standaard Turingmachine wordt op één bepaald moment in de tijd geheel beschreven door de inhoud van de band op dat moment, de toestand waarin de Turingmachine zich op dat moment bevindt, en de plaats waar de tapekop op dat moment op de band staat. Laat het bandalfabet $\{0, 1\}$ zijn, het toestandsalfabet Q en het aantal cellen op de band begrensd door het polynoom p, d.w.z. voor één of andere n kunnen nooit meer dan $p(n)$ cellen in gebruik zijn, dan volgt dat er niet meer dan $\|Q\| \times p(n) \times 2^{p(n)}$ van dit soort beschrijvingen bestaan. Een exponentiële tijd begrensde machine kan van de beginconfiguratie een voldoende aantal van deze configuraties doorrekenen om erachter te komen of onderweg een accepterende toestand bereikt wordt. Na hoogstens $\|Q\| \times p(n) \times 2^{p(n)}$ moet zo'n toestand immmers bereikt worden, of de machine raakt in een (oneindige) loop.

7.2 PSPACE

De natuurlijke tegenhanger van P in het geheugenbegrensde domein is de klasse van problemen die kunnen worden opgelost door Turingmachines waarbij het geheugengebruik begrensd wordt door een polynoom in de lengte van de invoer. Deze klasse noemen we PSPACE. Zoals hierboven beargumenteerd geldt P $\subseteq$ PSPACE *en* PSPACE $\subseteq$ EXP, maar omdat ook geldt P $\neq$ EXP (zie 6.1) geldt *niet* "P = PSPACE *en* PSPACE =

EXP". Eén van de twee inclusies moet een ongelijkheid zijn, maar welke van de twee is één van de grote open problemen van onze tijd. Voor polynomiaalbegrensde geheugencomplexiteit geldt echter wel invariantie onder nondeterminisme. Er is dus geen P = NP? probleem in de geheugencomplexiteit, ofwel NPSPACE bestaat niet als aparte entiteit, zoals de volgende stelling tot uitdrukking brengt.

Stelling 7.2.1 PSPACE = NPSPACE.

Om deze stelling te bewijzen merken we op, net zoals in het bewijs van PSPACE $\subseteq$ EXP dat als een geheugenbegrensde Turing machine een accepterende berekening heeft, er een exponentieel begrensde verzameling polynomiaal begrensde momentopnames van deze Turing machine moet zijn die achter elkaar gezet zo'n accepterende berekening vormen. Anders is er een cykel en zal de Turing machine niet accepteren. Voor een nondeterministische machine is dit argument een fractie subtieler. Als er een accepterende berekening bestaat die *langer* is dan een vantevoren te geven exponentieel aantal momentopnames, dan zit daar een cykel in, en is er dus ook een kortere accepterende berekening (waar die cykel niet in zit). De kortste accepterende berekening zal geen cyckel hebben en minder dan exponentieel veel momentopnames bevatten.

Stel dus er is een exponentiële verzameling $C_1, \ldots, C_{2^{p(n)}}$ momentopnames ieder van lengte hoogstens $p(n)$ die samen accepterende berekening van machine M op invoer x vormen. We beweren dat het bestaan van zo'n verzameling kan worden berekend door een polynomiaal geheugenbegrensde *deterministische* machine M', ofwel dat van elke nondeterministische polynomiaal geheugenbegrensde machine M en invoer x door een deterministische polynomiaal geheugenbegrensde machine M' kan worden bepaald of M' de invoer x kan accepteren.

Merk op dat een verzameling $C_1, \ldots, C_{2^{p(n)}}$ bestaat, als en alleen als er een $C_{2^{p(n-1)}}$ bestaat, zo dat $C_1, \ldots, C_{2^{p(n-1)}}$ een legitieme opeenvolging van momentopnames van M op invoer x is *en* $C_{2^{p(n-1)}}, \ldots, C_{2^{p(n)}}$ een legitieme opeenvolging van momentopnames van M op invoer x is. We gebruiken deze observatie voor de definitie van de volgende recursieve procedure.

```
1: Procedure Comp(C1,C2,n).
2: if n = 0 then
3:     if C2 is a legal successor of C1 according to M then
4:         return(1);
5:     else
6:         return(0);
7:     end if
8: else
9:     return(⋁{[Comp(C1, Ck, n − 1) ∧ Comp(Ck, C2, n − 1)] : Ck a legal configuration of M})
10: end if
```

Door deze routine aan te roepen met $\text{Comp}(C_0, C_{acc}, p(n))$ kunnen we beslissen of er een accepterende berekening bestaatn van $2^{p(n)}$ momentopnames. Vooropgesteld dat de recursieve boom in deze algoritme verstandig doorlopen wordt, kan ze geheel in polynomiaal begrensd geheugen worden uitgevoerd. Immers, elke C_k in regel 9 is slechts $p(n)$ groot, en verder is de recursiediepte begrensd door $p(n)$, zodat de totale berekening kan worden uitgevoerd in $O(p^2(n))$ begrensd geheugen.

7.2.1 Volledige Problemen

Ook voor PSPACE zijn er volledige problemen bekend. De meeste zijn zogenoemde two person perfect information games. Een master probleem voor PSPACE is het acceptatieprobleem van een lineair begrensde Turing machine, het model voor het herkennen van context sensitieve talen. Het eerste, meest bekende, karakterizerende probleem voor PSPACE is echter dat van de logische predikaten, quantified boolean formulae (QBF).

NAAM: QBF
GEGEVEN: Een gekwantificeerde logische formule $P = Q_1 Q_2 \ldots Q_n F(x_1, \ldots, x_n)$, met $Q_i \in \{\exists, \forall\}$.
GEVRAAGD: Is P waar?

De eerste stap is, is QBF een probleem in PSPACE? Dat is iets minder makkelijk dan de vergelijkbare stap voor NP-volledige problemen, maar ook weer niet heel erg moeilijk. Immers wanneer is $P = Q_1 Q_2 \ldots Q_n F(x_1, \ldots, x_n)$ waar? Dat hangt van Q_1 af. Als $Q_1 = \forall$, dan is P waar d.e.s.d.a. $Q_2, \ldots, Q_n F(0, x_2, \ldots, x_n)$ waar is *en* $Q_2, \ldots, Q_n F(0, x_2, \ldots, x_n)$ waar is, en als $Q_1 = \exists$ dan is P waar d.e.s.d.a. $Q_2, \ldots, Q_n F(0, x_2, \ldots, x_n)$ waar is *of* $Q_2, \ldots, Q_n F(0, x_2, \ldots, x_n)$ waar is. De volgende procedure beslist dus of een predicaat waar is.

```
1: Procedure QBF(Q_1 ... Q_n F(x_0, ..., x_n))
2: if Q_1 = ∀ then
3:     return(QBF(Q_2 ... Q_n(F(0, x_2, ..., x_n) ∧ Q_2 ... Q_n(F(0, x_2, ..., x_n))
4: else if Q_1 = ∃ then
5:     return(QBF(Q_2 ... Q_n(F(0, x_2, ..., x_n) ∨ Q_2 ... Q_n(F(0, x_2, ..., x_n))
6: else
7:     return(F(x_1, ..., x_n)) //no quantifiers left, all x have values.
8: end if
```

De recursiediepte is n, en elk element op de stapel van de aanroepen kan in polynomiaal geheugen worden opgeslagen. Het geheugengebruik is dus ongeveer kwadratisch, en dus zeker polynomiaal.

Dat elk probleem dat door een Turing machine kan worden opgelost vertaald kan worden naar een QBF die waar is als en alleen als de invoer geaccepteerd wordt zien we aan de hand van het eerder gegeven bewijs voor PSPACE $\subseteq$ EXP. Een momentopname van een Turingmachine kan worden vertaald naar een logische formule zoals we hebben gezien in 6.5. Twee configuraties kunnen naar een logische formule vertaald worden die alleen een vervulling heeft als de twee configuraties legale opvolgers zijn van elkaar volgens het programma van de Turing machine. Laat Q_0 een logische formule zijn die de begintoestand van M representeeert, en Q_F een logische formule zijn die de eindtoestand representeerd. Laat verder $S(F_1(x_1, \ldots, x_n), F_2(y_1, \ldots, y_m))$ zo zijn dat $(\exists x_1, \ldots, x_n, y_1 \ldots y_m)[S(F_1(x_1, \ldots, x_n), F_2(y_1, \ldots, y_m))]$ dan en slechts dan als $F_1(x_1, \ldots, x_n)$ en $F_2(y_1, \ldots, y_m)$ voor deze x en y opeenvolgende of dezelfde momentopnames van M representeren. We definieren de formules $S_k(P_1, P_2)$ voor predikaten met of zonder quantoren P_1 en P_2 als:

$$S_k = \begin{cases} S(P_1, P_2) & \text{if } k = 0 \\ (\exists P_3)[S_{k-1}(P_1, P_3) \wedge S_{k-1}(P_3, P_2)] & \text{anders} \end{cases}$$

De Turingmachine heeft een accepterende berekening van lengte 2^n dan en slechts dan als $S_n(Q_0, Q_F)$. Helaas is dit nog niet een polynomiale tijd begrensde vertaling. Immers S_n is twee keer zolang als S_{n-1} waardoor S_n lengte $\Omega(2^n)$ krijgt en dat mag niet in een polynomiaal begrensde reductie. We merken echter op dat S_{k-1} twee keer gebruikt wordt, en dit kan worden beperkt tot één keer door een extra universele quantor te gebruiken als volgt.

$$(\exists P_3)[(\forall P_4 P_5)[S_{k-1}(P_4, P_5) \vee [(P_4 \neq P_1 \vee P_5 \neq P_3) \wedge (P_4 \neq P_3 \vee P_5 \neq P2)]]$$

Als $P_1 = P_4$ *en* $P_3 = P_5$, dan moet $S_{k-1}(P_4, P_5)$ waar zijn, en dus ook $S_{k-1}(P_1, P_3)$. Hetzelfde geldt voor P_3 en P_2. De reductie heeft nu nog maar één optreden van de formule S_{k-1} en dus wordt de uiteindelijke formule ook polynomiaal in lengte in de lengte van de invoer.

7.2.2 Andere PSPACE Volledige Problemen

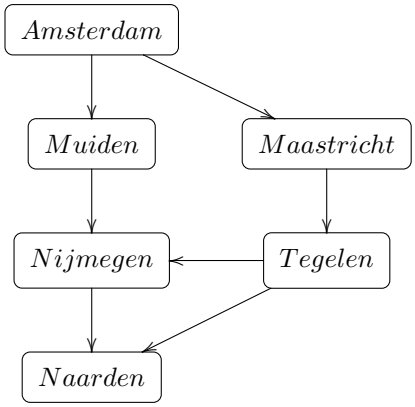
Zoals we al hebben opgemerkt zijn veel van de bekende PSPACE volledige problemen varianten van spelletjes, en wel de zogenoemde two person perfect information games. De intuitie hierbij is dat deze spellen vaak de volgende vorm hebben. Er is een winnende strategie alleen als er een zet voor speler 1 is, zodat voor elke zet van speler 2 er weer een zet voor speler 1 is zodat,... Kortom, deze spelen hebben de structuur van QBF en het is dan ook niet verwonderlijk dat QBF vaak vertaald kan worden naar de vraag „is er een winnende strategie" voor zo'n spel.

Het eerste spel dat we zullen bekijken wordt gespeeld op grafen en heet Generalized Geography. Geography is een bekend spel dat vaak gespeeld wordt om de verveling te verdrijven gedurende lange autoritten. De spelers mogen geografische namen (bijvoorbeeld steden) noemen die ergens in de wereld voorkomen. Het

gaat erom dat de volgende speler steeds een naam moet noemen die begint met de letter waarmee de vorige naam geëindigd is. De speler die geen nieuwe naam aan de lijst kan toevoegen verliest.

We modeleren het spel met een gerichte graaf waarin de knopen alle stedennamen ter wereld zijn, een kant gaat van stad A naar stad B als de naam van A eindigt met de letter waarmee de naam van B begint. Een van de steden wordt als startpunt aangewezen. Er zijn twee spelers P_1 en P_2 en de vraag is heeft P_1 een winnende strategie, d.w.z. kan zij voor elke strategie die P_2 kan hebben een verzameling knopen vinden zo dat P_2 vastloopt. We geven een klein voorbeeld.

Voorbeeld 7.2.1:



Voorbeeld van een mogelijk verloop van Geography. □

Generalized Geography wordt gespeeld op een gerichte graaf met een speciale knoop *start*. Speler 1 begint met het kiezen van een buur van *start* en om beurten kiezen spelers een buur van een knoop die de vorige speler gekozen heeft. Als een speler gedwongen is een knoop met uitgraad 0 te kiezen, of een knoop die al eerder gekozen is, dan heeft zij verloren. Het probleem is nu.

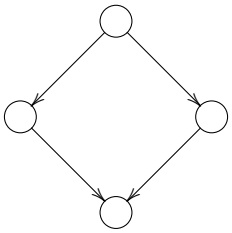
NAAM: Generalized Geography
GEGEVEN: Een gerichte graaf met startpunt s
GEVRAAGD: Heeft Speler 1 een winnende strategie

We moeten eerst bewijzen dat GG in PSPACE zit. Daartoe geven we een simpele recursieve algoritme die berekent of Speler 1 een winnende strategie heeft. Omdat elke knoop slechts één keer gekozen kan worden, en er eindig veel knopen in de graaf zitten heeft één van beide spelers altijd een winnende strategie. Een antwoord op de vraag dat een algoritme inhoudt is dus: „Speler 1 heeft een winnende strategie, dan en slechts dan als Speler 2 dat niet heeft." We vertalen dit als volgt naar een algoritme die de gegeven vraag onderzoekt op het bestaan van een winnende strategie.

```
1: GG(knoop)
2: if knoop has no outgoing edges then
3:     return(reject)
4: else if (∀ neighbors n of knoop)GG(n)=reject then
5:     return(accept)
6: else
7:     return(reject)
8: end if
```

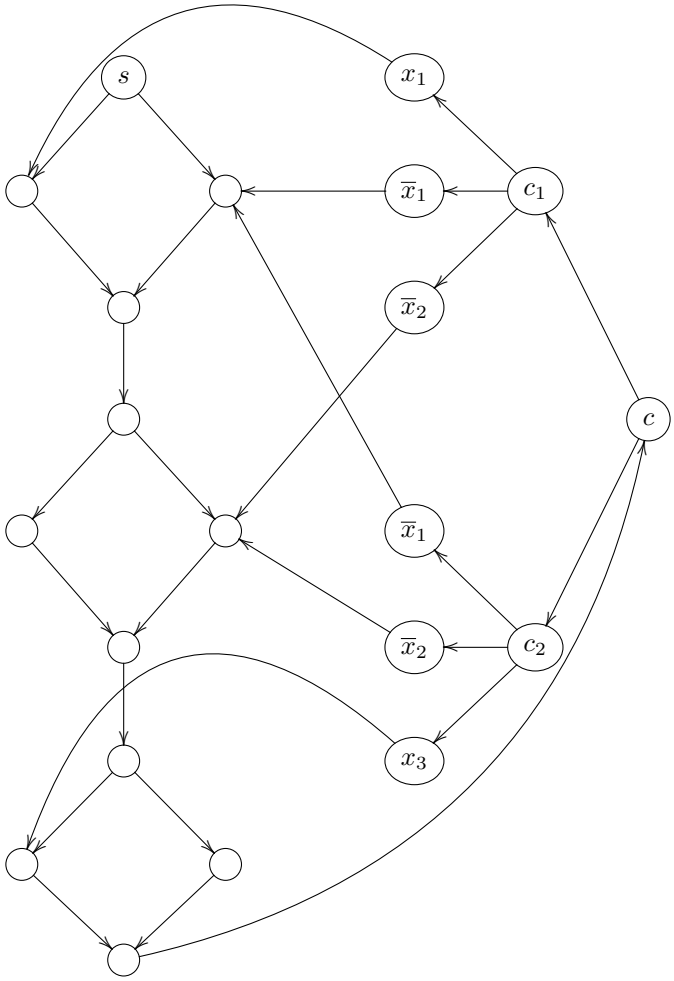
GG(start) onderzoekt of speler 1 vanuit de startknoop een winnende strategie heeft. Het is duidelijk dat deze algoritme in polynomiale ruimte kan worden uitgevoerd en dus dat ons probleem in PSPACE zit.

Vervolgens bewijzen we dat GG volledig is voor PSPACE. We doen dit door een reductie van een spelvorm van QBF. Gegeven een formule met kwantoren $Q_1x_1Q_2x_2\ldots Q_nx_nF(x_1,\ldots,x_n)$ dan maken we voor elke kwantor in de formule de volgende graaf.



Verder zijn er $2n$ knopen x_i en $\overline{x_i}$ voor de n variabelen en hun ontkenning, en m knopen voor de zinnen. De zin knopen c_j hebben uitgraad 3 en zijn verbonden met de variabele knopen of hun ontkenning al naar gelang ze in de zin voorkomen. De zin c_j knopen zijn via één enkele enkele kant verbonden met een keuzeknoop c. We geven weer een klein voorbeeld.

Voorbeeld 7.2.2: Laat gegeven zijn de formule $(\exists x_1 \forall x_2 \exists x_3)[(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)]$. De reductie hierboven gegeven vertaalt dit naar de volgende graaf.



Eerst stellen we even vast dat het predicaat in dit voorbeeld waar is. Immers als je x_3 waar kiest, dan doet het er niet toe wat de andere twee variabelen voor waarde aannemen. Dus P_1 moet een winnende strategie hebben. De strategie is als volgt. P_1 kiest in de deelgrafen die een existentiële kwantor vertegenwoordigen, de waarde van de variabele die zij waar wil maken om het predicaat waar te krijgen. P_2 kiest in de deelgrafen de waarde van de variabele die zij onwaar wil maken om het predicaat onwaar te krijgen. P_1 begint, door voor x_1 linksaf te gaan (waar) of rechtsaf te gaan (onwaar). In dit speciale geval doet dat er niet toe. Wat

P_2 bij de volgende deelgraaf doet, doet er niet toe. P_1 moet vervolgens bij de laatste deelgraaf wel linksaf slaan. P_1 kiest gedwongen de knoop c, en P_2 kan nu één van de zinnen kiezen door c_1 of c_2 te kiezen. Omdat de formule waar is, doet het er in dit geval niet toe welke van de twee zij kiest. Als ze c_1 kiest, dan moet vervolgens Speler 1 een variabele kiezen die waar is. Als zij x_1 waar gekozen heeft, dan kan zij knoop x_1 kiezen, als zij x_1 echter onwaar gekozen heeft, dan kan zij de knoop $\overline{x_1}$ kiezen. In beide gevallen verliest P_2 omdat de volgende knoop reeds eerder gekozen is. Kiest P_2 voor c_2 dan kiest P_1 vervolgens x_3 en bereikt hetzelfde resultaat. □

Een lange lijst van PSPACE volledige spelletjes bestaat. We noemen er een paar:

1. Amazons [Bur00, Hea05]
2. Go[LS80]
3. Hex [ET76]
4. Othello [IK94]
5. Rush Hour [FHN$^+$03]
6. Shanghai [CFLS97]
7. Sokoban [Cul99]

De werkelijke lijst is nog veel langer. Behalve in de wereld van de spelletjes komen we PSPACE volledigheid ook in de taalherkenning tegen. Naarmate een probleem meer uitdrukkingskracht heeft wordt het computationeel ook moelijker. Programmeertalen worden vaak zoveel mogelijk contextvrij gehouden omdat dat het compileren makkelijker maakt. Als voor de betekenis van een statement niet de context uitvoerig hoeft te worden onderzocht kan worden volstaan met een one-pass compiler of een interpreter. Natuurlijke talen zijn context-sensitief. Computationeel betekent dit een sprong van polynomiaal beslisbaar voor contextvrije talen naar PSPACE-volledigheid voor de context gevoelige talen.

Ook in de logica komen we PSPACE volledigheid op meer plaatsen tegen dan alleen in de predicatenlogica. Veel van de logica's die met modale of temporele kwantoren gedefinieerd worden zijn PSPACE volledig of soms nog complexer.

7.3 LOGSPACE

Zon beetje de kleinste klasse die we machinemodelonafhankelijk kunnen definiëren is de klasse van talen beslisbaar in (deterministische) logaritmische ruimte. Omdat de Stelling van Savitch, hierboven, een $O(n^2)$ overhead heeft van deterministisch naar nondeterministisch, is er ook een klasse NLOGSPACE van problemen beslisbaar in nondeterministisch logarithmische ruimte. Het probleem NLOGSPACE $\stackrel{?}{=}$ LOGSPACE is vooralsnog een open probleem, al zijn er deelresultaten behaald [Rei05].

Wel is bekend dat NLOGSPACE gesloten is onder complementatie. Dat wil zeggen dat het complement van elke context sensitieve taal zelf ook weer een context sensitieve taal is. Voor dit probleem (dat 23 jaar open heeft gestaan) is in 1988 door N. Immerman [Imm88] en onafhankelijk door R.Szelepscenyi [Sze87] een techniek ontwikkeld die sindsdien *inductive counting* genoemd wordt. We geven hier een schets van het bewijs.

Stel we hebben gegeven van een nondeterministische machine M een momentopname (configuratie) c en we vragen ons af of een bepaalde configuratie d vanuit c bereikt kan worden. We kunnen dan nondeterministisch een aantal tussenconfiguraties $c_1, c_2, \ldots, c_k$ met $c_1 = c$, $c_k = d$ en elke configuratie c_{i+1} is in één stap vanuit c_i te bereiken. Het aantal configuraties kan beperkt blijven tot n als M een machine is die in LOGSPACE werkt. Stel dat we precies weten *hoeveel* configuraties vanuit c bereikt kunnen worden, en

we willen bewijzen dat een configuratie d *niet* vanuit c bereikt kan worden. Een nondeterministische machine kan dan het gegeven aantal verschillende configuraties bepalen, bewijzen dat deze alle vanuit c bereikt kunnen worden en daarmee dus dat d *niet* zo'n configuratie is.

Stel nu dat we willen bewijzen dat een invoer x niet geaccepteerd wordt door M in een LOGSPACE begrensde berekening. We moeten dan bewijzen dat in LOGSPACE *geen* configuratie met een accepterende toestand vanuit de begintoestand kan worden bereikt. We *weten* hoeveel configuraties er vanuit de begintoestand in één stap kunnen worden bereikt en we beweren (inductie) dat we als we weten hoeveel verschillende configuraties uit de begintoestand in k stappen bereikt kunnen worden, dat we dan ook weten hoeveel configuraties er in $k+1$ stappen vanuit de begintoestand bereikt kunnen worden.

Het bewijs voor deze bewering is de volgende methode. Gegeven dat er bijvoorbeeld i configuraties in k stappen vanuit de begintoestand bereikt kunnen worden. Genereer achtereenvolgens al deze configuraties en bewijs dat dit die configuraties zijn door voor elk van die configuraties $k-1$ tussenconfiguraties te raden, waartussen de overgang in één stap kan gebeuren. Nu kunnen we voor elk van die configuraties apart bepalen hoeveel opvolgers deze configuratie heeft, en daarmee dus hoeveel configuraties in $k+1$ stappen kunnen worden bereikt.

De machine die het complement van een gegeven machine M berekent, doet nu op invoer x het volgende.

1: Bereken hoeveel mogelijke verschillende opvolgers de beginconfiguratie kan hebben in n stappen.
2: Genereer die configuraties (en bewijs dat het ze zijn)
3: **if** Geen van die configuraties accepterend is **then**
4: Accepteer
5: **else**
6: Verwerp
7: **end if**

7.3.1 Complete Problemen

Het generieke complete probleem voor nondeterministische logaritmische ruimte is het probleem ST-connectivity.

NAAM: ST-CONNECTIVITY
GEGEVEN: Een gerichte graaf met twee knopen S en T
GEVRAAGD: Is er en pad van S naar T

Dit probleem kan in nondeterministisch logaritmische ruimte worden beslist. Immers, als de graaf gegeven is, kan een Turingmachine beginnen met een 1 op de werkband en vervolgens een getal S raden. Daarna controleert de machine dat S, i een kant is in de graaf. De ruimte ingenomen door S kan vervolgens hergebruikt worden om een getal i_2 te raden en te controleren dat i, i_2 een kant is in de graaf. Daarna kan de ruimte gebruikt voor i worden hergebruikt. Zo voortgaande kan de nondeterministische machine uitkomen bij T als er een pad van S naar T bestaat en accpeteren. Als zo'n pad niet bestaat, dan faalt deze aanpak uiteraard. Alle gebruikte getallen zijn kleiner dan of gelijk aan het aantal knopen in de graaf, waardoor het gebruikte gegeugen begrensd is in de logaritme van de afmetingen van de representatie van de graaf op de invoerband. Hiermee is bewezen dat dit een probleem in LOGSPACE is.

Het probleem is echter ook LOGSPACE volledig. Om dit in te zien moeten we, gegeven een nondeterministische logarithmisch geheugenbegrense machine M en een invoer x, een graaf $G(M, x)$ maken met twee knopen S en T, zodat T vanuit S bereikbaar is dan en slechts dan als M een accepterende berekening heeft. Natuurlijk zijn de knopen van $G(M, x)$ configuraties van M op invoer x en is S de beginconfiguratie en T de eindconfiguratie. Elke configuratie heeft maar een paar mogelijke opvolgers, die door een deterministische logarithmisch geheugenbegrensde machine kunnen worden berekend. We mogen echter niet de uitvoerband als geheugen gebruiken, omdat deze groter dan logaritmisch wordt. Dit probleem kunnen we ondervangen door als representatie van de output te kiezen voor de adjacency matrix. Elk van de getallen van 1 tot $p(n)$ kan een legale codering van een configuratie van M zijn (dat hoeft natuurlijk niet, maar dat is voor de berekening van geen belang). De algoritme wordt:

1: **for** $i = 1$ to $p(n)$ **do**
2: **for** $j = 1$ to $p(n)$ **do**

```
3:        if  i to j is a legal transition then
4:            output 1;
5:        else
6:            output 0;
7:        end if
8:    end for
9: end for
```

Deze algoritme output een serie 0en en 1en die een representatie van de adjacencymatrix van de configuraties van M op invoer x is. Zonder beperking der algemeenheid kunnen we aannemen dat hierin knoop 0 de beginconfiguratie van M voorstelt en knoop $p(n)$ de eindconfiguratie. De vraag of M de invoer x accepteert is dan dezelfde als de vraag of vanuit knoop 0 knoop $p(n)$ bereikt kan worden. het geheugen dat nodig is, is slechts het geheugen voor de tellers i en j, dus zeker niet meer dan logarithmisch begrensd geheugen.

7.4 Interactieve Bewijzen

De vraag wat een bewijs is, is een heel oude vraag in de filosofie en misschien wel de reden waarom de logica bedreven wordt. Kern van de zaak is dat de één iets weet of denkt te weten, en dat zij de ander daarvan probeert te overtuigen. Doorgaans is er sprake van regels om ervoor te zorgen dat de bewijsvoering ordelijk verloopt. Een voorbeeld hiervan is elke willekeurige sport waar elk van beide tegenstanders de ander (en mogelijk ook een publiek) ervan probeert te overtuigen sterker te zijn. In sommige gevallen is er uiteraard ook geen sprake van regels, maar dan moeten achteraf de regels van de samenleving worden toegepast, en de politie eraan te pas komen.

Onderdeel van de regels is doorgaans een stelsel van axioma's. Dat zijn principes waarover partijen vantevoren een akkoord sluiten om ervoor te zorgen dat er tenminste een basis voor een gesprek is. Axioma's zijn *niet* algemeen geldende principes, getuige bijvoorbeeld de axioma's van Euclides voor de meetkunde. Slechts voor het onderhavige bewijs (en misschien een heel stel bewijzen) gelden ze als absoluut. Het spel bestaat eruit vanuit de axioma's met gebruikmaking van de regels te komen tot de uitspraak waarvan men de ander wil overtuigen. Dit spel heet dan „bewijzen" en het bewezene heet een stelling (of tautologie). Een groot probleem vormt het in de hand houden van de bewijsdrift. Hoeveel bewijs moet geleverd worden om de tegenstander te overtuigen? Vooral wanneer de stelling interessant is, dan kan de lengte van het bewijs nogal uit de hand lopen. Treffende voorbeelden hiervan zijn het bewijs van de vierkleurbaarheid van planaire grafen van Appel en Haken [AH77a, AH77b], Het bewijs van Robertson en Seymour over graaf minoren [RS83, RS04], en het bewijs van Wiles van de stelling van Fermat[CSS97]. Het probleem is het vinden van een trade-off tussen de mate van overtuigd raken van de waarheid van de tegenstander, en de inspanning die de tegenstander moet leveren bij het controleren van het bewijs. Natuurlijk zijn ook hier omgevingsfactoren van invloed. Bij de controle van het bewijs van de Stelling van Fermat is een schier onbegrensde inspanning ter controle gerechtvaardigd, maar bij de pinautomaat geldt een gemakkelijk te dupliceren code van vier cijfers als bewijs van identiteit.

Van nature hebben bewijzen dus een interactief karakter. De meest eenvoudige vorm hiervan is dat de ene partij het bewijs op het bord schrijft en de andere partij zegt „ja". Dit kan echter alleen gedaan worden als het bewijs klein genoeg is om op een bord te schrijven. Anders worden delen van het bewijs weggelaten en wordt het aan de andere partij gelaten over delen die zij niet begrijpt of niet gelooft door te vragen. Als bewijzen klein genoeg zijn (in onze traditie polynomiaal van lengte) dan geven we ze in één keer. De hele actie van bewijs vinden en verifiëren is dan een NP-proces geworden. Als bewijzen langer (exponentieel langer) worden, dan moeten we teruggrijpen op het presenteren van delen van het bewijs in een vraag antwoord spel. Zo'n bewijs is een echt interactief bewijs.

7.5 Zero Knowledge

Een zeer bijzondere tak van sport wordt gevormd door de zogenoemde zero-knowledge bewijzen. Het gaat er hierbij om een interactief bewijs te geven waarbij geen kennis over de aard van het bewijs wordt overgedragen.

Het protocol bestaat, zoals gebruikelijk in interactieve bewijzen uit een serie uitdagingen en antwoorden, maar de antwoorden moeten altijd zo zijn dat de uitdager die ook zelf had kunnen geven. De uitdager mag dus geen kennis kunnen ontlenen aan het gegeven antwoord. Dat deze vorm van bewijzen van grote waarde is in het interactieve dataverkeer behoeft nauwelijks betoog. Immers een afgeluisterde communicatie verschaft de afluisteraar bij dergelijke protocollen *per definitie* niet de informatie nodig om bij misleiding te gebruiken. Er zijn twee soorten van deze zero knowledge protocollen. Er is de absolute zero-knowledge—ongeacht hoeveel computing power de tegenpartij of afluisteraar heeft, zij zal nooit iets leren uit het protocol—en er is de relatieve zero-knowledge—als we ervanuit gaan de dat de computing power van de tegenpartij redelijk begrensd is, zij kan bijvoorbeeld geen NP-volledige problemen kraken, dan zal zij uit de communicatie niets leren dat zij tot haar voordeel kan gebruiken.

Voorbeeld 7.5.1: Als voorbeeld geven we een voorbeeld van relatieve zero-knowledge, gebaseerd op het HAMILTON CIRCUIT probleem. Zij gegeven een graaf $G = (V, E)$. Speler 2 beweert dat G een Hamilton circuit heeft en dat zij dat Hamilton circuit kent. Hoe zal zij speler 1 hiervan overtuigen zonder het Hamilton circuit prijs te geven. Dit gaat als volgt. Speler 2 produceert een random permutatie van G die zij vervolgens aan speler 1 stuurt. Deze heeft nu de keuze uit twee uitdagingen waaruit zij random kiest (een andere strategie is alleen voordelig voor een leugenachtige speler 2).

1. Uitdaging 1 is: produceer de permutatie die van G de graaf G' maakt.
2. Uitdaging 2 is: geef een Hamilton circuit in graaf G'.

Als speler 2 inderdaad een Hamilton circuit kent, dan kan zij in daarvan een Hamilton circuit in G' construeren. Aangenomen dat speler 1 niet voldoende rekenkracht heeft om het, moeilijk geachte, grafenisomorfie probleem op te lossen, zal speler 1 niets leren over het Hamilton circuit in G uit het antwoord op uitdaging 2. Uiteraard kan speler 1 ook niets leren uit het antwoord op uitdaging 1. Een random permutatie van G had zij immers zelf ook kunnen maken. □

Voorbeeld 7.5.2: Het in het vorige voorbeeld genoemde grafenisomorfie probleem kan worden gebruikt als identificatieprotocol als volgt. Speler 1 produceert een grote graaf G en een permutatie van deze graaf G'. Aangezien grafenisomorfie een moeilijk probleem is is het niet iedereen gegeven de isomorfie tussen G en G' te berekenen, maar Speler 1 kent deze natuurlijk. Nu worden G en G' tegenover Speler 2 gebruikt om Speler 1 te identificeren als volgt. Speler 1 produceert een derde graaf G'' die een permutatie van G of van G' is (dat maakt voor beide spelers niets uit).

1. Uitdaging 1: Laat zien dat G'' isomorf is aan G.
2. Uitdaging 2: Laat zien dat G'' isomorf is aan G'.

Als Speler 1 inderdaad de isomorfie tussen G en G' kent, dan kan zij aan beide uitdagingen voldoen. Als zij die niet kent, dan kan zij aan hoogstens één van beide voldoen. Bij random gekozen uitdagingen valt zij dus met kans 0.5 door de mand. Door herhaling van het protocol kunnen we de kans op onontdekt bedrog zo klein maken als we maar willen.

Een geringe variatie op dit protocol geeft een alternatief voor het eerder besproken bit-commitment probleem (zie 4.6.2). In dit geval is het Speler 2 die de grafen G en G' maakt en Speler 1 produceert een graaf G'' uit G als zij een 0 wil vastleggen en uit G' als zij een 1 wil vastleggen. Deze G'' wordt in handen van Speler 2 gegeven, die niet kan zien van welke de graaf afkomstig is. Immers G, G' en G'' zijn alle isomorf. Als de tijd komt om het bit te laten zien, kan Speler 2 de isomorfie waarmee G'' is verkregen prijsgeven. □

7.6 IP=PSPACE

Lange tijd is de vraag open gebleven hoeveel computationele kracht er in een interactief bewijs kan schuilen. Het is duidelijk dat voor alle problemen in NP korte interactieve bewijzen kunnen worden gegeven; deze klasse

is ongeveer zo gedefinieerd. Aan de andere kant is PSPACE een bovengrens voor de problemen waarvoor een interactief bewijs kan worden gegeven. Het standaardbewijs in de literatuur maakt gebruik van een variant van backward inductie, maar er is een simpeler manier om dit in te zien. Eerst geven we een nette definitie van interactieve bewijzen.

Een bewijzer P (voor „prover"), mag elk berekenbaar of onberekenbaar mechanisme zijn, dat een verzameling boodschappen produceert waarop een controleur V (voor „verifier") die polynomiale tijd begrensd is, maar wel gebruik mag maken van een random generator kan reageren met boodschappen voor de bewijzer. Gegeven één of andere invoer w (de te bewijzen stelling) bestaat interactie tussen deze twee, $P \leftrightarrow V$, uit een serie boodschappen $m_1, m_2, \ldots, m_{p(|w|)}$ die als laatste boodschap „accept" of „reject" heeft. In het eerste geval noteren we $P \leftrightarrow V = 1$, en in het tweede geval $P \leftrightarrow V = 0$. Omdat V gebruik maakt van een random generator, spreken we eerder van $Pr(P \leftrightarrow V = 1)$, waarbij de waarschijnlijkheid uniform genomen wordt over alle random strings met een lengte gelijk aan de lengte van de berekening van V op invoer w. De boodschappen met even index zijn afkomstig van de controleur, die met oneven index zijn afkomstig van de bewijzer.

We zeggen dat een taal L een interactief bewijssysteem heeft, of tot de klasse IP hoort als en alleen als er een polynomiale tijd begrensde machine V bestaat zo dat geldt:

1. $w \in L \leftrightarrow (\exists P)[Pr(P \leftrightarrow V = accept) > 2/3]$
2. $w \notin L \Leftrightarrow (\forall P)[Pr(P \leftrightarrow V = 1) < 1/3]$.

7.6.1 IP $\subseteq$ PSPACE

Volgens de definitie kan de vraag of w geaccepteerd wordt, worden vertaald in een vraag naar de maximum waarschijnlijkheid genomen over alle mogelijke P. Als deze waarschijnlijkheid namelijk groter is dan $2/3$ dan geldt $w \in L$, en als $w \notin L$, dan is deze waarschijnlijkheid gegarandeerd kleiner dan $1/3$. We laten zien dat gegeven w en V dit maximum door een PSPACE machine kan worden berekend. Aangezien er *geen enkele* begrenzing is aan de rekenkracht van P, is elke volgorde $m_1, m_3, \ldots, m_{p(|w|)-1}$ een legale bewijzer. Het valt op dat de bewijzer dus eigenlijk helemaal niet naar het gezeur van de controleur *hoeft* te luisteren.

We moeten laten zien dat de vraag wat het maximum is over alle mogelijke bewijzers berekend kan worden door een machine met polynomiaal begrensd geheugen. Hiertoe stellen wij ons een berekeningsboom voor met knopen waarin de Bewijzer en de Controleur afwisselend aan de beurt zijn. In de bladeren van de boom is de Controleur aan de beurt en moet zij, op grond van de conversatie die gevormd wordt door de uitgewisselde boodschappen op het pad dat in dit blad eindigt het bewijs accepteren of verwerpen. Als zij het bewijs accepteert, dan krijgt deze knoop de waarde 1, anders krijgt deze knoop de waarde 0. In alle andere knopen maken we de volgende berekening. Als de Bewijzer aan de beurt is, dan moeten we een maximum nemen over alle mogelijke boodschappen die de Bewijzer als antwoord op de conversatie op het bovenliggende pad kan geven, we noteren dus hier het *maximum* van alle onderliggende knopen die een waarde hebben gekregen. Als de Controleur aan de beurt is, dan moeten we een (gewogen) gemiddelde nemen over de waarden in de onderliggende knopen.

Dat deze boom in PSPACE uit te rekenen is, hoeft weinig betoog. Er zijn slechts polynomiaal veel boodschappen in een conversatie, en iedere boodschap heeft zelf slechts polynomiale lengte. De recursiediepte van een opgebouwde boom is dus slechts polynomiaal. Verder hoeft nooit meer dan één enkel pad van de boom in zijn geheel gexpandeerd te worden, want zowel het nemen van het maximum, als het nemen van een (gewogen) gemiddelde eist niet dat tussenresultaten worden onthouden. De boom kan dus gewoon van links naar rechts recursief doorlopen worden, waarbij telkens slechts één pad in het geheugen is opgeslagen.

7.6.2 PSPACE $\subseteq$ IP

Voor dit deel van het bewijs laten we zien dat een bekend PSPACE-volledig probleem namelijk Quantified Boolean Formulas een interactief bewijs heeft. We gebruiken hiervoor de arithmetische versie van een formule.

De waarheidswaarden voor een logische variabele x, kunnen we vervangen door een waarde uit $\{0, 1\}$ waarbij 0 de rol van FALSE speelt, en 1 de rol van TRUE. Neem logische formules ϕ en ψ en noem hun

herleiding tot 0 en 1 door het invullen van de variabelen respectievelijk Φ en Ψ. Hoe dat moet als ϕ en ψ uit individuele variabelen bestaan hebben we zojuist gezien. Inductief herleiden we nu $\phi \wedge \psi$ en $\phi \vee \psi$ tot 0 en 1 door respectievelijk $\Phi \times \Psi$ en $\Phi + \Psi - \Phi \times \Psi$ te nemen. Noem de waarde die je krijgt uit een formule ϕ met één variabele x die je krijgt door $x = i$ in te vullen $\Phi(i)$, dan is $\forall x \Phi(x)$ hetzelfde als $\Phi(0) \times \Phi(1)$ en $\exists x \Phi(x)$ hetzelfde asl $\Phi(0) + \Phi(1) - \Phi(0) \times \Phi(1)$.

We merken terloops op dat de term $\Phi(0) \times \Phi(1)$ eigenlijk wel kan worden weggelaten. We krijgen dan bij een ware formule een waarde groter dan 0 en bij een onware formule de waarde 0 krijgt.

Een quantified boolean formula kan nu op deze manier in zijn geheel vertaald worden naar een getal dat groter dan 0 is dan en slechts dan als deze formule waar is. Neem eerst k een vast getal en stel dat we een formule hebben met k variabelen, waarbij we de *eerste* variabele x_k onbepaald laten en we de overige variabelen invullen kortom, we nemen in plaats van de formule

$$P = Q_k x_n Q_{k-1} x_{k-1} \ldots Q_1 x_1 F(x_1, \ldots, x_k)$$

De formule

$$Q_{k-1} x_{k-1} \ldots Q_1 x_1 F(x_1, \ldots, x_k)$$

Met inductie en de vertaling hierboven zien we dat dit een polynoom p_k is van de graad 2^k. Bovendien geldt in het geval $Q_k = \exists$ dat $p_k(0) + p_k(1) > 0 \leftrightarrow P = true$ en in het geval $Q_k = \forall$ dat $p_k(0) + p_k(1) > 0 \leftrightarrow P = true$

Twee verschillende polynomen van de graad 2^k hebben „slechts" 2^k punten gemeen. Stel er geldt bijvoorbeeld *niet* $p_k(0) + p_k(1) = 1$, maar er is een ander polynoom q waarvoor wel geldt dat $q(0) + q(1) = 1$ wanneer we een willekeurig getal r uit $\{1, \ldots, n\}$ kiezen, dan is de kans dat $p_k(r) = q(r)$ gelijk aan $2^k/n$. Het getal r zou namelijk precies de x van een snijpunt moeten zijn om dit waar te maken.

De Bewijzer en de Controleur spelen nu het volgende spel—we nemen even $*$ als operator die een $+$ is in geval van $\exists$ en een $\times$ in geval van $\forall$. De bewijzer wordt steeds gevraagd een polynoom te leveren dat overeenkomt met de quantified boolean formula als de volgende quantor van de formule wordt afgehaald, noem het polynoom p_k voor ronde k. In de eerste ronde controleert de controleur alleen dat $p_1(0) * p_k(1)$

In volgende rondes is er een extra controle beschikbaar, doordat de controleur in elke ronde een random getal r_k kiest, dat doorgeeft aan de Bewijzer, die $p_k(r_k)$ uitrekent. In de volgende ronde moet de bewijzer dan steeds een polynoom p_{k+i} kiezen waarvoor geld $p_{k+i}(0) * p_{k+i}(1) = p_{k+i-1}(r_{k+i-1}$.

Met dit protocol bereik je twee dingen. Als de waarde van de quantified boolean formula werkelijk positief is (i.e., de formule is waar), dan kan de Bewijzer beginnen met het juiste polynoom en daarbij blijven. Als alle kwantoren zijn geweest en alle waarden ingevuld, zijn Bewijzer en Controleur het erover eens dat de waarde van het polynoom positief is (en dus de formule waar).

Als de waarde van het polynoom 0 is en de Bewijzer produceert in de eerste ronde een polynoom dat een positieve waarde geeft bij het invullen van 0 en 1, dan moet ergens onderweg naar het echte polynoom worden „overgestapt". Dat betekent dat ergens de polynomen p_i en p_{i+1} precies in r_i moeten snijden. Als we de verzameling random getallen groot genoeg kiezen is de kans daarop klein.

De graad van het polynoom is wat aan de hoge kant. Om de bewering PSPACE $\subseteq$ IP te bewijzen, zouden we graag ongeveer n quantoren in de boolean formula willen hebben, waarmee de graad van het polynoom op 2^n uitkomt, anders kunnen we slechts formules met een beperkt aantal quantoren aan [LFKN90]. Dat de graad van het polynoom beperkt kan worden (zelfs tot ongeveer 3) is een observatie die A. Shamir [Sha92] aan de theorie toevoegde. Dat betekent, gezien het feit dat er exponentieel veel getallen zijn van polynomiale lengte dat de kans dat een liegende bewijzer halverwege het bewijs zou kunnen overstappen op een snijdend polynoom exponentieel klein wordt.

Voorbeeld 7.6.1: Bekijk de (ware) quantified boolean formula

$$B = (\forall x_1)[\overline{x}_1 \vee (\exists x_2)(\forall x_3)[[x_1 \wedge x_2] \vee x_3]]$$

De uitdrukking

$$A = \prod_{z_1 \in \{0,1\}} \left[(1 - z_1) + \sum_{z_2 \in \{0,1\}} \prod_{z_3 \in \{0,1\}} (z_1 . z_2 + z_3) \right]$$

is de arithmetizering van B. De waarde van deze uitdrukking is 2. Als we de eerste quantor weglaten, krijgen we de uitdrukking

$$A(z_1) = \left[(1 - z_1) + \sum_{z_2 \in \{0,1\}} \prod_{z_3 \in \{0,1\}} (z_1.z_2 + z_3) \right]$$

Hierbij hoort het polynoom $q(z_1) = z_1^2 + 1$. Als P dit polynoom produceert, dan kan V eerst controleren dat inderdaad $q(1) \times q(0) = 2$, conform de bewering. Zouden we $z_1 = 3$ invullen, dan krijgen we

$$A(3) = \left[(1 - 3) + \sum_{z_2 \in \{0,1\}} \prod_{z_3 \in \{0,1\}} (3z_2 + z_3) \right],$$

met waarde 10. V heeft alleen q dus V vult in $q(3) = 9 + 1 = 10$. Hieruit kan V afleiden dat de uitdrukking

$$\sum_{z_2 \in \{0,1\}} \prod_{z_3 \in \{0,1\}}$$

de waarde $10 - (1 - 3) = 12$ moet hebben, en aan P vragen het polynoom dat hoort bij

$$A(z_2) = \prod_{z_3 \in \{0,1\}} (3z_2 + z_3)$$

te genereren. Dit polynoom is

$$q(z_2) = 9z_2^2 + 3z_2.$$

V controleert eerst dat $q(0) + q(1) = 12$ en kiest dan bijvoorbeeld $z_2 = 2$. Er moet nu gelden dat

$$A = \prod_{z_3 \in \{0,1\}} (6 + z_3) = q(2) = 42$$

. Het polynoom dat P verstuurt is

$$q(z_3) = z_3 + 6,$$

en V kan zien dat $q(0).q(1) = 6.7 = 42$. Tenslotte kiest V voor $z_3 = 5$ en controleert dat

$$A(z_3 = 5) = (6 + 5) = 5 + 6 = q(5).$$

□

Bijlage A

Begrippenlijst

In dit deel herhalen we een aantal begrippen uit de tekst met hun betekenis in alfabetische volgorde. Zo is het voor de lezer makkelijker om snel de betekenis van een begrip in de in de tekst gehanteerde vorm terug te vinden. Voor de context kan dan gebruik gemaakt worden van de index.

adversary argument. Bewijsmethode die wordt gebruikt om een ondergrens voor een algoritme te bewijzen. De uitvoering van de algoritme wordt gezien als een spel, waarbij een denkbeeldige tegenstander zo ongunstig mogelijke zetten doet. Voorbeeld: het zoeken in een ongesorteerde rij. De elementen van de rij worden aan de zoekalgoritme gepresenteerd in willekeurige volgorde, maar met het gezochte element als laatste in de rij. Zo wordt bewezen dat deze zoekmethode altijd minstens alle elementen van de rij langs moet alvorens te kunnen constateren dat het gezochte element niet aanwezig is.

amortized complexity. Uitgesmeerde complexiteit. De complexiteit van een ingewikkelde stap wordt verdeeld over *voorgaande* stappen van mindere complexiteit.

articulatiepunt. Knoop in de graaf waardoor alle paden tussen twee andere knopen gaan. Als deze knoop wordt verwijderd bestaat de resterende graaf uit tenminste twee verbonden componenten. Slechte knopen om in een communicatienetwerk te hebben.

bankmethode. Een methode om uitgesmeerde complexiteit te] berekenen. Bij elke stap wordt zowel de echte complexiteit als een "spaarbedrag" in rekening gebracht. Het spaarsaldo kan later opgenomen worden om voor de duurdere stappen te betalen.

barometer. Spil van de berekening. Een instructie die minstens net zo vaak uitgevoerd wordt als een willekeurige andere instructie. De complexiteit van de algoritme wordt uitgedrukt in het aantal keren dat deze instructie wordt uitevoerd.

begrensde betegeling.

NAAM: BOUNDED TILING
GEGEVEN: Een vierkant met kleuren langs de rand en een verzameling Tegeltypen
GEVRAAGD: Is het mogelijk het vierkant te betegelen met de gegeven tegeltypen zo dat de kleuren langs de randen van de tegels passen

NP-volledig probleem. Bewijs met Master reductie van Turing machine probleem.

beslissingsalgoritme. Algoritme die op elke invoer een ja of nee antwoord geeft.

beslissingsprobleem. Probleem dat vraagt om een ja of nee beslissing.

Bit Commitment . Probleem uit de Cryptografie. De ene partij stuurt naar de andere partij één bit informatie. Hoewel de andere partij zonder extra informatie er niet achter kan komen of dit bit 0 of 1 is, kan de zendende partij niet achteraf het bit dat verstuurd is veranderen.

blowfish. Cryptoalgoritme.

boedelscheiding.

> NAAM: PARTITION
> GEGEVEN: Een verzameling van n getallen $\{x_1, \ldots, x_n\}$
> GEVRAAGD: Is er een indexverzameling I zodat $\sum_{i \in I} x_i = \sum_{i \notin I} x_i$

NP-volledig probleem. Reductie van KNAPSACK.

branch & bound. Methode om de explosieve groei van een zoekboom te beperken. In elke knoop van de gedeeltelijk ontwikkelde zoekboom wordt een waarde uitgerekend die het optimum voorstelt dat in de subboom van die knoop nog te bereiken is. De knoop met de beste waarde wordt gesplitst (branch). Als een pad geheel ontwikkeld is, hoeft geen enkele knoop die een slechter optimum voorspelt dan de gevonden waarde verder te worden ontwikkeld (bound).

breadth first search. Manier om een zoekboom te doorlopen waarbij van elke knoop eerst alle knopen op gelijke diepte worden bezocht voordat verder in de boom gegaan wordt.

Catalaans getal. Groot getal dat in veel combinatorische problemen opduikt. Het is vernoemd naar de Belgische wiskundige Eugène Charles Catalan (1814–1894) en kan geschreven worden als

$$\binom{C_n = \frac{1}{n+1} 2n}{n.}$$

chromatic number Aantal kleuren dat minimaal nodig is om een graaf te kleuren, zodat geen twee knopen die aan dezelfde kant zitten dezelfde kleur krijgen. NP-volledig probleem. Reductie van satisfiability.

circuit. Gerichte Acyclische Graaf waarin elke knoop een poort (en, of, niet) voorstelt.

Circuit Value Problem . Probleem om gegeven een invoer, de waarden van de knopen met ingraad 0 in een circuit, de uitvoer, de waarden van de knopen met uitgraad 0 te berekenen. LOGSPACE volledig voor P, reductie van Turingmachineprobleem.

Clique

> NAAM: Clique
> GEGEVEN: Een graaf $G = (V, E)$ en een getal K
> GEVRAAGD: Is er een $V' \subseteq V$ met $\|V'\| \geq K$, zodat $V' \times V' \subseteq E$

complexiteitsanalyse. Analyse van de complexiteit van een algoritme of een probleem.

complexiteitsklasse Verzameling problemen van ongeveer dezelfde complexiteit.

complexiteitsmaat. Maat voor de complexiteit van een algoritme of een probleem. De meestgebruikte maten zijn tijd en geheugen. Er zijn echter aftelbaar veel verschillende complexiteitsmaten.

component, verbonden. Ondergraaf van een graaf G waarin tussen elk tweetal knopen een pad is.

connectivity. Mate van verbondenheid van een graaf, minimaal aantal knopen (node connectivity) of kanten (edge connectivity) om van de graaf tenminste twee componenten te maken.

cut, min. Minimale doorsnijding van een graaf (met capaciteitsfunctie).

> NAAM: MINCUT
> GEGEVEN: Een graaf $G = (V, E)$ met een capaciteitsfunctie $c : E \mapsto R$.
> GEVRAAGD: Verzameling knopen V' zo dat $\sum_{e \in V' \times V - V'} c(e)$ minimaal is. In stroomproblemen geldt dat de waarde van de minimale doorsnijding gelijk is aan de maximale stroom.

cyckel. Rij punten in een graaf zodat tussen twee opeenvolgende punten een kant aanwezig is, tevens is er een kant tussen de laatste in de eerste knoop in de rij. Als de cykel geen subcykels—deelrij die ook een cykel is—heeft, noemen we de cykel *enkelvoudig*

deelgraaf. ondergraaf. Deelverzameling van de verzameling knopen met alle kanten die tussen deze knopen aanwezig zijn.

depth first search. Methode om een zoekboom te doorlopen die in een knoop eerst de uitgaande kanten uit die knoop kiest. Als dat geen nieuwe knopen meer oplevert wordt met bactracking de rest van de graaf op dezelfde manier doorlopen.

DES. Data Encryption Standard. Versleutelingsalgoritme van het private key type. Veelgebruikte methode waar uitwisseling van sleutels relatief goedkoop is.

descriptieve complexiteit. Beschrijvende complexiteit, of complexiteit van de beschrijving. Hoeveel informatie is nodig om een bepaald object te produceren. Meestgebruikt voorbeeld: Kolmogorov complexiteit, grootte van het kleinste Turingmachine programma dat het object produceert.

diagonalisatie. Bewijsmethode langs "de diagonaal". Twee aftelbare verzamelingen, bijvoorbeeld programma's en invoeren worden langs de horizontale, resp. verticale as uitgezet. Een bewijs tegen het bestaan van element van de ene verzameling wordt op de diagonaal verkregen, door op coordinaat (i, i) steeds het tegenovergestelde gedrag neer te schrijven. Het vooronderstelde element kan dan op geen plaats op de horizontale lijn staan. Voorbeelden:

1. Overaftelbaarheid van de reële getallen. Schrijf een vooronderstelde aftelling van de reële getallen tussen 0 en 1 als decimale ontwikkelingen langs de horizontale as, waarbij de decimalen langs de verticale as gezet worden. Op de diagonaal komt telkens een ander cijfer dan het cijfer in het desbetreffende getal. Het resultaat is een oneindige decimale ontwikkeling die niet in de rij staat.
2. Onbeslisbaarheid van het Halting probleem. Schrijf langs de horizontale as alle Turingmachine programmas en langs de verticale as mogelijke invoeren. Op de diagonaal staat een 0 als machine i op invoer i stopt, en een 1 als machine i niet op invoer i stopt. De machine die niet in de rij kan voorkomen is de machine die niet op invoer i stopt als er een 0 op plaats (i, i) staat en wel stopt als er een 1 op plaats (i, i) staat.

discrete Fourier transformatie. Methode om een gegeven polynoom van graad n te vertalen in de m waarden die dit polynoom aanneemt in de m-de machts éénheidswortels $\{e^{2\pi ij/m}\}_{j=0,\ldots m-1}$.

dubbelverbonden component. Component van een gerichte graaf waarbij tussen elk tweetal punten a en b zowel een pad van a naar b als een pad van b naar a is.

dynamisch programmeren. Algoritmeontwerp voor optimaliseringsproblemen dat gebruik maakt van tabellen waarin optimale oplossingen voor deelproblemen staan. Deze optimale oplossingen kunnen worden samengevoegd tot oplossingen voor grotere deelproblemen, zodat uiteindelijk de optimale oplossing voor het hele probleem gevonden kan worden.

EDIT RAM. Random Access Machine die teksten bewerkt. In één operatie kan een EDIT RAM een stuk tekst (die in het programma staat of in een register is opgeborgen) vervangen door een ander stuk tekst in de hele invoer. Deze eigenschap maakt de EDIT RAM lid van de tweede machine klasse. Alles wat een Turing machine in polynomiaal geheugen kan doen, kan de EDIT RAM in polynomiale tijd en omgekeerd.

extented gcd. Vorm van de Euclidische algoritme waarbij als de gcd van de ingevoerde getallen a en b gelijk aan 1 is, de algoritme ook een c teruggeeft, zodat $ac \mod b = 1$. Het getal c is de multiplicatieve inverse van a.

ELEMENTARY. Eerste subrecursieve tijdbegrensde klasse die bekend gesloten is onder nondeterminisme. Op invoer van lengte n geldt een tijdgrens van $\underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}}_{n}$.

entscheidungsprobleem. De vraag of er een algoritme bestaat om de universele geldigheid van eerste orde uitspraken te beslissen. Turing gaf hier een negatief antwoord op.

Euler path. Pad in een graaf dat elke kant precies één keer bevat. Elke graaf waarin elke knoop een even graad heeft heeft zo'n pad, en alleen deze grafen hebben zo'n pad.

EXACTE OVERDEKKING.

NAAM: EXACT COVER
GEGEVEN: Een universum $U = \{1, \ldots, n\}$ met deelverzamelingen S_j
GEVRAAGD: Bestaat er een selectie deelverzamelingen die samen nog steeds een overdekking zijn, maar paarsgewijs een lege doorsnede hebben?

Het probleem of een collectie deelverzamelingen een exacte overdekking hebben is NP-volledig. Reductie van Begrensde Betegelingen.

EXP. Klasse van problemen waarvoor een deterministische exponentiële tijd begrensde algoritme bestaat.

Feedback vertex set.

NAAM: FEEDBACK VERTEX SET
GEGEVEN: Een graaf $G = (V, E)$ en een getal K
GEVRAAGD: Bestaat er een V' met $\|V'\| \leq K$ zodat elke cykel in G tenminste één knoop uit V' heeft. NP-volledig probleem. Reductie van VERTEX COVER

Ford-Fulkerson algoritme. Methode om een maximale stroom in een netwerk te vinden die gebruik maakt van het herhaald identificeren van stroomvergrotende paden. De algoritme is correct, maar convergeert niet altijd.

geheugencomplexiteit. Complexiteitsmaat die de hoeveelheid geheugen begrenst. De eenheid van geheugen verschilt van model tot model. Voor een Turingmachine is dat één bandcel, voor een Random Access machine één register. De sequential computation thesis eist dat redelijke machinemodellen elkaar in constante factor overhead in geheugen simuleren. Dit vergt soms een andere manier om geheugengebruik toe te rekenen.

geography, generalized. Spel waarbij de spelers om beurten een knoop mogen selecteren van een gerichte graaf en die in hun verzameling op te nemen. De geselecteerde knoop moet altijd een opvolger zijn van de door de andere speler gekozen knopen. De vraag is of de beginnende speler een winnende strategie heeft in een gegeven graaf. NP-volledig probleem. Reductie van QBF.

graaf. Verzameling knopen en kanten. De kanten zijn ongeordende paren knopen. De knopen worden ook wel punten genoemd. De kanten worden in gerichte grafen ook wel bogen genoemd. In gerichte grafen zijn de bogen geordende paren punten.

greedy algorithm (hier gulizge algoritme). Optimaliseringsalgoritme waarbij in elke stap een zo gunstig mogelijke keuze gemaakt wordt. Op een keuze wordt nooit teruggekomen. De problemen waarvoor met gulzige algoritmen een optimale oplossing gevonden kan worden zijn van een speciale soort. Bekend voorbeeld is de kortste pad algoritme van Dijkstra.

HALT. Het Halting probleem voor Turing machines. Gegeven een paar x, y waarbij x een Turingmachineprogramma is en y een invoer. Zal x op invoer y stoppen. Het probleem is onbeslisbaar. Dwz er is geen Turingmachineprogramma dat voor elk paar x, y stopt en ja zegt als x op y stopt en nee anders.

Hamilton circuit.

NAAM: HAMILTON CIRCUIT
GEGEVEN: Een graaf G.
GEVRAAGD: Vormt een deel van de kanten van G een enkelvoudige cykel langs alle knopen?

NP-volledig. Reductie van VERTEX COVER.

handelsreiziger probleem.

NAAM: TSP
GEGEVEN: Een volledige gerichte graaf G met een gewichtsfunctie op de kanten en een grens b.
GEVRAAGD: Bestaat er een enkelvoudige cykel in G waarvan het totale gewicht kleiner is dan b?

NP-volledig. Reductie van HAMILTON CIRCUIT.

hashtabel. Database met natuurlijke getallen als index. De plaats van een record in de database wordt bepaald door een zogenoemde hashfunctie die op grond van een veld in het record een waarde berekend. Doorgaans is deze functie cyclish en berekent hij niet een unieke waarde voor elk record. Als twee records hetzelfde adres wordt toegewezen ontstaat een collision die op verschillende manieren kan worden opgelost (collision resolution).

heuristiek. Op natte vingerwerk gebaseerde methode om keuzen te maken in optimaliseringsproblemen. Vaak dient een—niet werkende—greedy algoritme als heuristiek voor een optimaliseringsprobleem waarna suboptimale oplossingen worden verbeterd door bactracking of local search.

HITTING SET.

NAAM: HITTING SET
GEGEVEN: Een verzameling deelverzamelingen $\{S_j\}_j$ van $U = \{1, \ldots, n\}$ en een getal k
GEVRAAGD: Is er een deelverzameling $V \subseteq U$ met $\|V\| \leq k$, zodat voor alle j geldt $\S_j \cap V \neq \emptyset$?

NP-volledig, reductie van VERTEX COVER

INDEPENDENT SET.

NAAM: INDEPENDENT SET
GEGEVEN: Een graaf $G = (V, E)$ en een getal K
GEVRAAGD: Is er een deelverzameling $V' \subseteq V$ met $\|V'\| \leq K$ zdd $V' \times V' \cap E = \emptyset$?

inproduct Getal dat verkregen wordt door van twee vectoren v en w, beide van dimensie n, overeenkomstige coordinaten met elkaar te vermenigvuldigen en het resultaat van de vermenigvuldigingen bij elkaar op te tellen. Als het inproduct van v en w gelijk is aan 0, dan staan v en w loodrecht op elkaar.

integraalkenmerk van d'Alembert Kenmerk dat gebruikt wordt om de som van een reeks af te schatten. De som is altijd kleiner of groter dan de integraal van de functie, afhankelijk van welke grenzen worden genomen.

Khacian's algoritme. Algoritme voor lineaire programmeren. De meest gebruikte algoritme, de simplex algoritme is in sommige gevallen exponentieel. Khacian's algoritme, ook wel ellipsoide algoritme genoemd, is altijd polynomiaal.

kleurbaarheid. De vraag of een graaf zodanig met een gegeven aantal kleuren te kleuren is dat geen twee punten die door een kant verbonden zijn dezelfde kleur krijgen. 2-kleurbaarheid is polynomiaal, 3-kleurbaarheid is NP-volledig (reductie van 3SAT). 4-kleurbaarheid van planaire grafen is triviaal. 3-kleurbaarheid is ook voor planaire grafen NP-volledig.

kleurgetal. Minimale aantal kleuren dat nodig is om een gegeven graaf te kleuren. Ook wel CHROMATIC NUMBER genoemd.

KNAPSACK.

NAAM: KNAPSACK
GEGEVEN: Een verzameling objecten $o_1, \ldots, o_n$ met gewichten $w_1, \ldots, w_n$ en waarden $v_1, \ldots, v_n$ en twee getallen b en v.
GEVRAAGD: Bestaat er een verzameling I, zo dat $\sum_{i \in I} w_i \leq b$, terwijl $\sum_{i \in I} v_i \geq v$?

knoopoverdekking, VERTEX COVER.

NAAM: VERTEX COVER
GEGEVEN: Een graaf $G = (V, E)$ en een getal K.
GEVRAAGD: Bestaat er een $V' \subseteq V$ met $\|V'\| \leq K$ zdd $(\forall e \in E)[e \cap V' \neq \emptyset]$?

knowledge, zero. Type protocol waarbij de Verifier overtuigd raakt van de juistheid van de stelling terwijl niets wordt geleerd over het bewijs.

Kruskal, algoritme van. Methode om een mincost spanning tree te berekenen in een gewogen graaf. Bij deze algoritme worden een aantal bomen gevormd die samengroeien totdat er nog één boom over is.

LOGSPACE. Klasse van talen die kan worden herkend door een Turing machine waarvan het geheugengebruik wordt begrensd door de logaritme van de lengte van de invoer.

machtreeks. Reeks waarbij de termen machten van een grondtal zijn.

masterprobleem. Probleem dat volledig wordt bewezen in een klasse door een reductieschema te geven waarbij een Turing machine, een grens en een invoer de parameters zijn.

masterreductie. Reductieschema voor het volledig bewijzen van een masterprobleem.

matching, perfect Selectie van een aantal kanten van een graaf waarbij de graaf beperkt tot deze kanten twee (of meer) delig wordt.

matrixvermenigvuldigingsexponenent. Exponent van het polynoom dat de grens is voor de meest efficiente matrixverminigvuldigingsalgoritme.

minor. Ondergraaf die uit een graaf G kan worden verkregen door een kant weg te laten of samen te trekken.

MRAM. Multiplication RAM. Random Access machine waarbij de kosten van een vermenigvuldiging van willekeurig grote operanden op 1 wordt gesteld.

NP Klasse van problemen herkenbaar door nondeterministische Turingmachines in polynomiale tijd. Equivalent: de klasse van problemen waarvoor een gegeven oplossing in polynomiale tijd kan worden gecontroleerd.

ODDMINSAT. NAAM: ODDMINSAT
GEGEVEN: Een formule $F = F(x_1, \ldots, x_n)$.
GEVRAAGD: Is het laatste bit van de lexicografisch kleinste vervulling, gelezen als bitstring, gelijk aan 1?

Volledig voor P^{NP}, het eerste Delta level van de polynomiale hierarchie [Sto76].

ondergraaf. Graaf die verkregen wordt uit een graaf G door een aantal punten met de bijbehorende incidente kanten weg te laten.

ondergrens. Minimaal benodigde looptijd voor een algoritme op invoeren van lengte n, waarbij over alle invoeren van lengt n het maximum genomen mag worden.

opspannende boom. Deel van de kanten van een graaf dat een boom vormt waarin alle punten zijn opgenomen.

optimalisatiealgoritme. Algoritme die de optimale oplossing vindt voor een probleem.

optimaliseringsprobleem. Probleem waarbij naar een optimum gezocht wordt. Zie ook: beslissingsprobleem.

overdekking. Selectie van deelverzamelingen die samen alle elementen van een gegeven verzameling bevatten.

overdekkingsprobleem. Probleem dat het vinden van een overdekking stelt.

P. Klasse van problemen die zijn op te lossen met een deterministische polynomiale tijd begrensde Turing machine.

padding lemma Lemma uit de recursietheorie. Bij elke codering van Turingmachines als natuurlijke getallen, komt elke Turingmachine die een bepaalde functie berekent oneindig vaak voor.

parallellisme, onbegrensd. Machinemodel waarbij verschillende processoren tegelijkertijd werken. Met onbegrensd parallelle machines kunnen PSPACE volledige problemen in polynomiale tijd worden opgelost.

perebor. Russisch woord dat de inherente hardheid van combinatorische problemen karakterizeert.

potentiaalmethode. Methode om de uitgesmeerde complexiteit van een algoritme te bepalen.

PRAM. Parallelle Random Access Machine. Machinemodel met onbegrensd parallellisme, ook wel SIMD RAM genoemd.

priemfactoren. Factoren waardoor een getal deelbaar is die zelf alleen deelbaar zijn door 1 en zichzelf.

priemgetal. Getal dat alleen deelbaar is door 1 en zichzelf.

primaliteitstest

NAAM: PRIMALITY
GEGEVEN: Een getal p
GEVRAAGD: Is p priem

. Probleem in P. Bewijs door Agrawal, Kayal en Saxena [AKS04]. Lange tijd heeft men gedacht dat dit een probleem in $\mathrm{NP} \cap \mathrm{Co} - \mathrm{NP}$ was dat niet in P zat, hoewel er goede randomized algoritmen bestaan die de primaliteit van een getal kunnen testen. Deze algoritmen worden in de praktijk nog steeds gebruikt.

Prim, algoritme van. Methode om een mincost spanning tree te berekenen in een gewogen graaf. Bij deze algoritme is er steeds één deel van een opspannende boom die langzaam groeit totdat alle punten er deel van uitmaken.

primaliteitstest. Test om te bepalen of een getal een priemgetal is.

primitieve n-de machts éénheidswortel. Oplossing van de vergelijking $x^n - 1 = 0$.

probabilistische algoritmen Algoritmen die gebruik mogen maken van coinflips.

PSPACE. Klasse van problemen oplosbaar door Turingmachines die in geheugen begrensd zijn door een polynoom in de lengte van de invoer.

quantified boolean formulae. Predicaatlogische formules.

quantum computing. Berekeningen gebaseerd op principes van de quantummechanica. De invoer is een quantum state, meestal een superpositie van toestanden die ontwikkelt door middel van unitaire afbeeldingen. Gedurende berekeningen wordt de toestand een aantal malen geobserveerd, waardoor een gehele of gedeeltelijke collapse van de superpositie plaatsvindt.

quicksort Sorteermethode volgens het verdeel-en-heersprincipe. In een rij van n elementen wordt een spil gekozen, waarna alle elementen kleiner dan de spil links van de spil geplaatst worden, terwijl alle andere elementen rechts van de spil geplaatst worden. De twee deelrijen worden recursief op dezelfde wijze gesorteerd.

radix sort. Methode van sorteren gebaseerd op het achtereenvolgens sorteren van getallen op basis van de in die getallen voorkomende cijfers.

RAM. Random Access Machine. Machinemodel bestaande uit een CPU die een programma uitvoert dat uit een genummerde lijst van instructies bestaat. Het geheugen van de machine is een onbegrensd aantal registers die elk een natuurlijk getal kunnen bevatten.

recurrente betrekking. Vergelijking die de verhouding weergeeft van geïndexeerde variabelen. In deze tekst worden recurrente betrekkingen vooral gebruikt voor de complexiteitsanalyse van recursieve algoritmen.

reductie. Bewerking de de relatieve complexiteit van problemen bepaalt.

RSA. Public key cryptosysteem vernoemd naar de uitvinders: Rivest, Shamir en Adleman.

simplexalgoritme Methode om lineair programmeren problemen op te lossen. De afbeeldingen lopen van hoekpunt naar hoekpunt op een polytoop om het optimum van de lineaire functie op dat polytoop te vinden.

spil. Zie barometer.

sleutel. Onderdeel van een versleutelingsalgoritme. Met de sleutel is codering en decodering eenvoudig. Zonder sleutel is het moeilijk.

Strassen, algoritme van. Recursieve manier om vierkante matrices met elkaar te vermenigvuldigen. In plaats van $O(n^3)$ heeft deze algoritme een complexiteit $O(n^{^2 \log 7}$.

stroomvergrotende paden Paden in een netwerk waarlangs de capaciteit niet is uitgeput. Langs deze paden is het vergroten van de stroom nog mogelijk. Essentieel onderdeel in de Algoritme van Ford en Fulkerson.

Turing machine. Machinemodel bestaande uit een centrale verwerkingseenheid en een band. De Turingmachine werkt doordat in elke stap een symbool van de band gelezen wordt, een symbool op de band geschreven wordt en de machine van de ene toestand uit een eindige verzameling toestanden naar een andere toestand overgaat. Tenslotte beweegt de leeskop op de band naar links of naar rechts.

toestandsovergangsfunctie Functie (soms relatie) die op invoer een toestand en een symbool, de volgende toestand(en) geeft en het symbool of de symbolen die het gelezen symbool gaan vervangen, alsmede de beweging van de leeskop.

versleutelingsalgoritme Algoritme die gegeven een sleutel een boodschap codeert.

vierkleurenprobleem Is het mogelijk een gegeven planaire graaf met vier kleuren te kleuren, zodat geen twee met elkaar verbonden knopen dezelfde kleur krijgen. Antwoord: ja. Bewijs gevonden met behulp van een computer door Appel en Haken. Naar een inzichtelijk bewijs voor deze stelling wordt nog steeds druk gezocht.

waarheidswaarden True en False of 1 en 0. Waarden die aan variabelen in een propositie kunnen worden toegekend.

zelfreduceerbaarheid Eigenschap van bepaalde problemen. Het antwoord op een zelfreduceerbaar probleem kan worden gevonden door een aantal kleinere instanties van hetzelfde probleem op te lossen. Zelfreduceerbare problemen zijn typisch problemen die met recursieve algoritmen kunnen worden aangepakt.

zoekboom Boom die door een zoekalgoritme wordt doorlopen. Typisch voorbeeld is een access structuur in een database.

zoekruimte. Totaal van alle mogelijke elementen die door een zoekalgoritme zouden kunnen worden gevonden. De afmetingen van de zoekruimte zijn van groot belang voor de complexiteit van een zoekalgoritme.

Bibliografie

[AB98] M. Akra and L. Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10:195–210, 1998.

[AH77a] K. Appel and W. Haken. Every planar map is four colorable. part i. discharging. *Illinois J. Math.*, 21:429–490, 1977.

[AH77b] K. Appel and W. Haken. Every planar map is four colorable. part ii. reducibility. *Illinois J. Math.*, 21:491–567, 1977.

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in P. *Annals of Mathematics*, 160(2):781–793, 2004.

[BB96] G. Brassard and P. Bratley. *Fundamentals of Algorithmics*. Prentice Hall International Editions, 1996.

[Bur00] M. Buro. Simple amazons endgames and their connection to hamilton circuits in cubic subgrid graphs. In *Proc. 2nd Int. Conf. Computers and Games*, 2000.

[CDRH+96] J. Cowie, B. Dodson, R.Elkenbracht-Huizing, A.K. Lenstra, P.L. Montgomery, and J. Zayer. A world wide number field sieve factoring record: on to 512 bits. In *Advances in Cryptology–ASIACRYPT6*, pages 382–394, 1996.

[CFLS97] A. Condon, J. Feigenbaum, C. Lund, and P. Shor. Random debaters and the hardness of approximating stochastic functions. *SIAM Journal on Computing*, 26(2):369–400, 1997.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[CSS97] G. Cornell, J.H. Silverman, and G. Stevens, editors. *Modular Forms and Fermat's Last Theorem*. Springer-Verlag, 1997.

[Cul99] J. Culberson. Sokoban is pspace-complete. In *Int. Conf. Fun with Algorithms, Proceedings in Informatics 4*, pages 65–76, 1999.

[DH76] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22:644–654, November 1976.

[ET75] S. Even and R.E. Tarjan. Network flow and testing graph connectivity. *SIAM Journal on Computing*, 4:507–518, 1975.

[ET76] S. Even and R. E. Tarjan. A combinatorial problem which is complete in polynomial space. In *Proc. 7th ACM Symp. Theory of Computing*, pages 710–719, 1976.

[FF56] L.R. Ford and D.R. Fulkerson. Maximum flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[FHN+03] Henning Fernau, Torben Hagerup, Naomi Nishimura, Prabhakar Ragde, and Klaus Reinhardt. On the parameterized complexity of the generalized rush hour puzzle. In *Proc. 15th Canad. Conf. Comput. Geom.*, pages 6–9, 2003.

[GHR95] R. Greenlaw, H.J. Hoover, and W.L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory.* Oxford University Press, 1995.

[GJ79] M.R. Garey and D.S. Johnson. *Computing and Intractability, A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, 1979.

[GT02] M.T. Goodrich and R. Tamassia. *Algorithm Design.* Wiley, 2002.

[Har92] D. Harel. *Algorithmics, the Spirit of Computing.* Addison Wesley, 1992.

[Hea05] Robert A. Hearn. Amazons is pspace-complete, 2005.

[HNR68] P.E. Hart, N.J Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[HS76] Juris Hartmanis and Janos Simon. On the structure of feasible computations. *Advances in Computers*, 14:1–43, 1976.

[IK94] S. Iwata and T. Kasai. The othello game on an n×n board is pspace-complete. *Theoretical Computer Science*, 123:329–340, 1994.

[Imm88] N. Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17:935–938, 1988.

[JS04] R. Johnsonbaugh and M. Schaefer. *Algorithms.* Pearson, 2004.

[Kha80] L.G. Khacian. Polynomial algorithms in linear programming. *Zh. vychisl. Mat. mat. Fiz*, 20(1):51–68, 1980.

[Knu73] D.E. Knuth. *The Art of Computer Programming*, volume 3. Addison Wesley, 1973.

[LFKN90] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. In *Proc. 31st Symposium on Foundations of Computer Science*, pages 2–90, New York, 1990. IEEE.

[LS80] D. Lichtenstein and M. Sipser. Go is polynomial-space hard. *Journal of the ACM*, 27:393–401, 1980.

[Meh85] K. Mehlhorn. *Graph Algorithms and NP-Completeness.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1985.

[MKM78] Vishv M. Malhotra, M. Pramodh Kumar, and S. N. Maheshwari. An o(—v—) algorithm for finding maximum flows in networks. *Inf. Process. Lett.*, 7(6):277–278, 1978.

[Par99] R. Parkinson. *Cracking Codes, the Rosetta Stone, and Decipherment.* University of California Press, 1999.

[Pom96] C. Pommerance. A tale of two sieves. *Notices of the American Mathematical Society*, 43(12):1473–1485, 1996.

[Rei05] O. Reingold. Undirected ST-connectivity in LOGSPACE. In *Proceedings 37th Annual ACM Symposium on Theory of Computing*, pages 376–385, 2005.

[RS83] N. Robertson and P.D. Seymour. Graph minors. i. excluding a forest. *Journal of Combinatorial Theory Series B*, 35(1):39–61, 1983.

[RS04] N. Robertson and P.D. Seymour. Graph minors. xx. wagner's conjecture. *Journal of Combinatorial Theory Series B*, 92(2):325–357, 2004.

[RSA78] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[Sha92] A. Shamir. IP=PSPACE. *Journal of the ACM*, 4:869–877, October 1992.

[Ski98] S. Skienna. *The Algorithm Design Manual*. Springer Verlag, 1998.

[Sto76] L. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1976.

[Str69] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[Sze87] R. Szelepsényi. The method of forcing for nondeterministic automata. *Bulletin of the EATCS*, 33:96–100, 1987.

[Tur36] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathmatical Society*, 42(2):230–265, 1936.

[Wil86] H.S. Wilf. *Algorithms and Complexity*. Prentice-Hall International Editions, 1986.

Index