Berekenbaarheid, onberekenbaarheid en complexiteit: een aanvullende studie

Gijs Vermeulen
gijs.vermeulen@gmail.com

25 augustus 2005

Inhoudsopgave

Voorwoord

Bij deze wil ik graag een deel mensen bedanken voor hun steun en hulp bij het bekomen van inzicht. In de eerste plaats zou ik heel graag prof. Jan Van Den Bussche willen bedanken omdat hij het mogelijk heeft gemaakt dat ik mijn thesis in dit studiegebied heb mogen verwezenlijken en omdat hij mij heeft geholpen met uitleg bij onderwerpen die ik niet direct begreep. Ook wil ik mijn vrienden bedanken, vooral Joris en Tim, die me aanmoedigde om deze thesis te verwezenlijken.

Natuurlijk hebben ook mijn ouders een belangrijke rol gespeeld in heel mijn opleiding. Zonder hun steun en geloof zou ik nooit zo ver geraakt zijn als waar ik nu sta.

Hoofdstuk 1

Inleiding

Deze tekst behandelt verschillende onderwerpen uit de theoretische informatica. Omdat ik tijdens mijn opleiding al een ruime inleiding tot de theoretische informatica heb gehad, zal dit een uitbreiding zijn van wat ik reeds gestudeerd heb. Om die reden zullen de meeste delen die ik ga behandelen losstaande stukken zijn of uitbreidingen van reeds geziene leerstof. Sommige delen zullen dus dieper ingaan op bepaalde aspecten waarvan ik uitga dat ze alreeds gekend zijn.

Ik zal eerst een hoofdstuk wijden aan 'niet context vrije talen', waarbij het deel context vrije talen' al in mijn opleiding aan het licht gebracht is. Hierin zal het 'pumping lemma voor contextvrije talen' besproken worden, waarmee we van verschillende talen kunnen aantonen dat het niet contextvrije talen zijn.

In het volgende deel, hoofdstuk 3, zullen er onberekenbare problemen aan het licht gebracht worden. Eerst worden er inleidende voorbeelden gegeven waar de onberekenbaarheid van problemen wordt aangetoond door een contradictie. Later in het hoofdstuk zal er een methode, die reduceerbaarheid genoemd wordt, geïntroduceerd worden.

Hoofdstuk 4 zal een aantal onderwerpen uit de berekenbaarheidstheorie behandelen. Het eerste onderwerp zal het recursie theorema zijn. Dit theorema stelt dat er Turing Machines bestaan die zijn eigen beschrijving kunnen opvragen en daarmee kunnen verder werken. Vervolgens zal ik logische theoriën behandelen. Het derde onderdeel gaat over Turing reduceerbaarheid, een sterkere vorm van reduceerbaarheid dan mapping reduceerbaarheid uit hoofdstuk 3. Het laatste onderwerp zal dan een mogelijke defintie voor informatie behandelen.

Het volgende deel, hoofdstuk 5, zal over PSPACE-compleetheid handelen, waarin een bewijs wordt gegeven dat het TQBF probleem PSPACE-compleet is. Eenmaal dat bewezen is, kan men via reducties aantonen dat andere problemen ook PSPACE-compleet zijn. Zo zal er aangehaald worden dat de meeste spellen PSPACE-compleet zijn.

In hoofdstuk 6 komen vervolgens problemen aan bod die niet van praktisch nut zijn. Er wordt van deze problemen aangetoond dat ze te veel geheugen of tijd in beslag nemen, in de zin dat ze er meer als polynomiaal veel van gebruiken. Hiervoor worden eerst de hiërarchie stellingen bewezen om de polynomiale klassen

van de exponentiële te scheiden.

Vervolgens komen we tot hoofdstuk 7, waarin ik een kleine inleiding geef tot alternerende Turing Machines.

Ten slotte zal ik in hoofdstuk 8 even uitwijden over parallelle berekeningen, waarbij parallelle algoritmes worden gemodelleerd door booleaanse circuits die in hoofdstuk 6 aan het licht werden gebracht.

Hoofdstuk 2

Niet context vrije talen

In mijn opleiding heb ik reeds geleerd wat een context vrije taal(CVT) is. Nu bestaan er ook manieren om aan te tonen dat een bepaalde taal niet context vrij is. De manier die ik hier ga bespreken steunt op het pumping lemma voor context vrije talen.

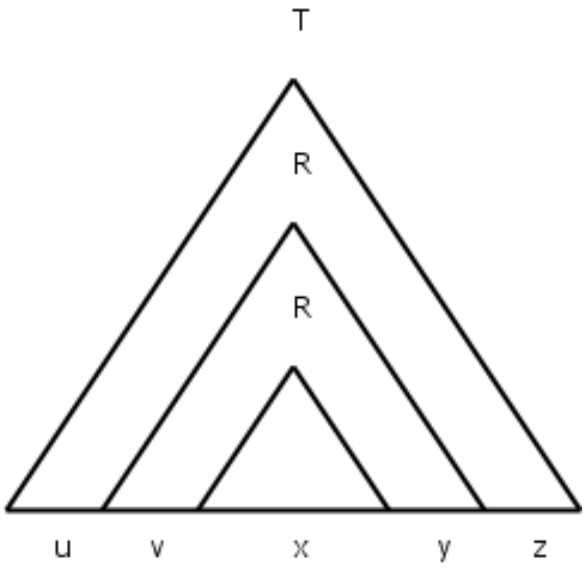
2.1 Pumping lemma voor context vrije talen

pumping lemma voor context vrije talen: Als A een context vrije taal is dan bestaat er een getal p zodat voor elke string s in A met lengte groter dan p de volgende puntjes gelden.

- $s = uvxyz$
- $\forall i \geq 0 : uv^ixy^iz \in A$
- $|vy| \geq 0$
- $|vxy| \leq p$

Vooraleer het technische bewijs te geven, volgt hier de intuïtie achter het lemma. Eerst en vooral werken we met een CVT A die gegenereerd wordt door een context vrije grammatica (CVG) G. Als s nu een lange string uit A is, dan heeft deze een afleidingsboom in G. Omdat s lang genoeg is, moet er in G een variabel symbool R zijn dat minstens twee keer voorkomt in de afleidingsboom voor s. Als we dan de deelboom horende bij het tweede voorkomen van R, vervangen door de deelboom die hoort bij het eerste voorkomen van R, krijgen we nog altijd een legale afleiding. Om deze reden kunnen we s dus in vijf stukken verdelen zoals in figuuur 2.1, en het tweede en vierde deel herhalen zoveel we willen, maar beiden evenveel.

Bewijs: Laat G een CVG voor de CVT A zijn. Stel dat b het aantal symbolen aan de rechterkant is van de afleidingsregel met het meeste symbolen aan de rechterkant, en laat b ook groter of gelijk zijn aan 2. Dan genereert een



Figuur 2.1: Afleidingsboom

afleidingsboom van hoogte h een string van maximale lengte b^h. Laat $|V|$ het aantal variabelen in G voorstellen. Definieer p dan gelijk aan $b^{|V|+2}$. Omdat $b \geq 2$, moet ook $p > b^{|V|+1}$.

Veronderstel dan nu dat s een string in A is van lengte minimum p. Stel dat τ een afleidingsboom is voor s. Als er meerdere afleidingsbomen bestaan dan kiezen we τ de afleidingsboom met minimaal aantal knopen. Omdat $|s| \geq p$, moet de afleidingsboom τ minstens hoogte $|V| + 2$ hebben. Dus er bestaat een pad van wortel tot blad van lengte minstens $|V| + 2$, en dit pad moet dus minstens $|V| + 1$ variabele symbolen bevatten (omdat enkel de bladeren uit terminale symbolen kunnen bestaan). Vermits er maar $|V|$ verschillende variablen in G zijn, moet er een variable R zijn dat minstens 2 keer voorkomt op dit pad. Nu kunnen we s verdelen in $uvxyz$ zoals in figuur 2.1. Het bovenste voorkomen van R in de boom is verantwoordelijk voor het genereren van het deel vxy van s. Op dezelfde manier genereert het onderste voorkomen van R het deel x. Omdat beide delen door dezelfde variabele worden gegenereert, mogen we de ene door de andere veranderen en toch nog een geldige afleidingsboom behouden. Als we herhalend de kleinste deelboom door de grootste vervangen, krijgen we afleidingsbomen voor uv^ixy^iz.

Om de derde conditie uit de stelling te voldoen, merken we op dat als v en y beiden gelijk zouden zijn aan ε, de afleidingsboom, verkregen door de kleine deelboom door de grote te vervangen, minder knopen zou bevatten

als τ, en toch nog s genereren. Dit kan niet omdat we τ gekozen hebben als de afleidingsboom voor s met minimaal aantal knopen.

Volgens de laatste conditie mag de lengte van vxy niet groter zijn dan p. Om hieraan te voldoen kiezen we de R, die vxy genereert, op die manier dat beiden voorkomens van R zich tussen de laatste $|V|+1$ variabelen op het pad bevinden. Dus die deelboom heeft maximale hoogte $|V|+2$, en een boom van deze hoogte kan strings van maximale lengte $b^{|V|+2} = p$ genereren.

2.2 Voorbeelden

Zoals eerder vermeld kunnen we het Pumping lemma gebruiken om aan te tonen dat bepaalde talen niet context vrij zijn. We veronderstellen eerst dat de bepaalde taal L wel context vrij is, om nadien tot een contradictie te komen. Dus we kunnen dan het Pumping lemma gebruiken om aan de lengte p van de stelling te komen. Vervolgens tonen we aan dat er aan string s is groter als p is, die toch niet aan alle voorwaarden van de stelling voldoet, door alle mogelijkheden om s in $uvxyz$ te verdelen af te gaan, en voor elke mogelijkheid aan te tonen dat er niet voldaan is aan minimum één voorwaarde uit de stelling.

Als eerste voorbeeld zullen we aantonen dat de taal $B = \{a^n b^n c^n | n \geq 0\}$ niet context vrij is.

We veronderstellen dat B wel contextvrij is en dat p de lengte is uit de stelling. We kiezen de string $s = a^p b^p c^p$. s zit dan duidelijk in B en heeft lengte mimstens p. We tonen nu dat op eender welke manier we s in $uvxyz$ verdelen, er altijd minstens één conditie van de steliing niet voldaan is. Er zijn twee gevallen: ofwel bevatten v en y enkel één type symbool, dus v bestaat uitsluitend uit a's of b's of c's, en hetzelfde geldt voor y. Ofwel bevat minstens één van beiden meerdere symbolen.
1. Als v en y slechts één type symbool bevatten, dan kan de string uv^2xy^2z niet evenveel a's, b's en c's bestaan, want $vy \neq \varepsilon$, en moet dus één van beiden niet leeg zijn, en dus minstens één symbool bevatten.
2. Als v of y uit meerdere symbolen bestaat, dan zal uv^2xy^2z de symbolen niet in de juiste volgorde bevatten om nog tot B te behoren.

Eén van deze twee gevallen moet zich voordoen en in beide gevallen hebben we aangetoond dat er een contradictie is. Dus het is onmogelijk dat B context vrij is.

Een volgend voorbeeld is iets moeilijker en is de taal $C = \{w\#x | w$ is een substring van x, met $x, w \in \{a, b\}^*\}$.

We veronderstellen dat C wel contextvrij is en dat p de lengte uit de stelling is. We kiezen dan de string $s = a^p b^p \# a^p b^p$ dat duidellijk een element van c moet

zijn. Er zijn vier verschillende mogelijke manieren hoe we s in $uvxyz$ kunnen verdelen.

1. Als v of y het symbool # bevat, dan zal de string uv^0xy^0z niet meer tot C behoren, want deze string bevat het hekje niet meer.

2. Als v zich voor het hekje bevindt onderscheiden we volgende gevallen:

2.1. v bestaat enkel uit het symbool a: y moet zich dan ook nog voor het hekje bevinden, want $|vxy| \leq p$. Als we dan nu de string uv^2xy^2z bekijken, dan zien we dat het deel wat zich nu voor het hekje bevindt meer a's bevat als het deel achter het hekje. Dus hetgeen voor het hekje komt kan zeker geen substring van hetgeen na het hekje komt.

2.2. v bestaat uit a's en b's: y kan dan niet verder geraken dan de a's die zich achter het hekje bevinden. Als we dan da string uv^2xy^2z bekijken, zien we dat zich er meer b's voor het hekje bevinden als achter het hekje. Dus het deel voor het hekje kan onmogelijk een substring zijn van het deel achter het hekje.

2.3. v bestaat enkel uit b's: y kan dan weeral niet verder geraken dan de $a's$ die zich achter het hekje bevinden. De string uv^2xy^2z zal dan meer b's voor het hekje bevatten dan na het hekje, waardoor het deel voor het hekje onmogelijk een substring van het deel achter het hekje kan zijn.

2.4. Als v de lege string is, dan is zeker y niet de lege string. Er kunnen zich de volgende gevallen voordoen:

2.4.1 y bevindt zich voor het hekje. Hier geldt dezelfde redenering als in puntjes 2.1 tot 2.3.

2.4.2 y bevindt zich achter het hekje. We nemen de string uv^0xy^0z. Deze bevat ofwel minder als p a's ofwel minder als p b's, terwijl hetgeen voor het hekje staat juit p a's en p b's bevat. Hierdoor kan het deel voor het hekje geen substring zijn van het deel dat achter het hekje staat.

3. Als y zich voor het hekje bevindt, dan bevindt v zich zeker ook voor het hekje. Dus geldt puntje 2.

4. Als v en y zich beiden achter het hekje bevinden, kunnen we de string uv^0xy^0z nemen om tot een contradictie te komen. Eén van beiden is zeker niet leeg, dus ofwel zijn er in deze string minder dan p a's, of minder dan p b's. Hierdoor kan hetgeen voor het hekje staat, dat uit p a's en p b's bestaat, onmogelijk een substring zijn van hetgeen achter het hekje staat.

Het volgende voorbeeld toont aan dat er niet CVT's bestaan waarvoor het pumping lemma ook geldt.

$$L = \{a^i b^j c^k d^l | i = 0 \text{ of } j = k = l\}.$$

Deze taal is niet contextvrij. Toch voldoet het aan het pumping lemma: als we $s = b^j c^k d^l$ kiezen en $s = uvxyz$, dan is het altijd mogelijk om u, v, x, y en z te kiezen zodat $uv^m xy^m z$ terug in L zit, zorg er bijvoorbeeld voor dat vxy enkel uit b's bestaat. Als we $s = a^i b^j c^k d^l$ kunnen we ervoor zorgen dat uvx enkel uit a's bestaat.

Hoofdstuk 3

Reduceerbaarheid

In dit hoofdstuk zal ik het vooral hebben over onberekenbare problemen. Met onberekenbaar bedoel ik problemen waarvoor geen Turing Machine (TM) bestaat die ze berekent. Turing machines zijn uitvoerig besproken tijdens mijn opleiding, dus zal ik er hier niet verder over uit wijden. Wel haal ik even aan dat we al reeds weten dat A_{TM}, het probleem om na te gaan of een bepaalde Turing Machine een gegeven input aanvaardt of niet onbeslisbaar is. Dit is belangrijk want om aan te tonen dat bepaalde problemen onberekenbaar zijn, zal ik in dit hoofdstuk gebruik maken van reduceerbaarheid.

Een reductie is het herleiden van één probleem naar een ander, op een manier dat een oplossing van het tweede kan gebruikt worden voor het eerste op te lossen. een voorbeeld van zo een reductie in het alledaagse leven kan het volgende zijn: veronderstel dat je vuile kleren hebt en dat je die wilt wassen. Je weet dat dit vrij eenvoudig gaat met een wasmachine. Dus we kunnen het probleem om kleren te wassen herleiden tot het vinden van een wasmachine.

Reduceerbaarheid is vrij belangrijk, want als A reduceerbaar is tot B, en B is beslisbaar dan is A ook beslisbaar, zoals verder in dit hoofdstuk zal blijken. Omgekeerd, als A onberekenbaar is en reduceerbaar tot B, dan is B onberekenbaar.

Omdat tot nu toe A_{TM} het enige probleem is waarvan we weten dat het onberekenbaar is, zullen we dit in onze reducties moeten gebruiken om aan te tonen dat andere problemen ook onberekenbaar zijn.

3.1 Inleidende voorbeelden

Zoals eerder vermeld weten we al dat A_{TM} onberekenbaar is. We gaan dit nu gebruiken om aan te tonen dat andere problemen ook onbeslisbaar zijn.

Het eerste voorbeeld zal $HALT_{TM}$ zijn, het probleem dat moet nagaan of een bepaalde Turing Machine stopt op een gegeven input (door te aanvaarden of niet te aanvaarden, dus niet in een loop te gaan).

$$HALT_{TM} = \{\langle M, w\rangle | M \text{ is een TM en } M \text{ stopt op input } w\}$$

Stelling: $HALT_{TM}$ is onbeslisbaar.

Intuïtief bewijs: We gaan een bewijs uit het ongerijmde geven voor deze stelling te bewijzen. Stel dus dat we wel zouden kunnen beslissen of een bepaalde Turing Machine stopt op een gegeven input. We tonen dan aan dat A_{TM} ook beslisbaar is, en komen zo tot een contradictie.

Als we eerst het algoritme R dat $HALT_{TM}$ beslist op de input $\langle M, w\rangle$ voor A_{TM} laten lopen, weten we of M stopt op w of niet. Als het niet stopt, weten we ook dat het w niet kan aanvaarden, als het wel stopt, kunnen we M op w laten lopen om te checken of het aanvaardt of niet.

Dit toont dus aan dat als we $HALT_{TM}$ kunnen beslissen, we ook A_{TM} kunnen beslissen, wat niet kan want we weten al dat A_{TM} onbeslisbaar is. We mogen dus besluiten dat $HALT_{TM}$ ook onbeslisbaar is.

Bewijs: Veronderstel dat er een TM R bestaat die $HALT_{TM}$ beslist. We maken een TM S die A_{TM} beslist.

S = Op input $\langle M, w\rangle$:
1. Laat R op $\langle M, w\rangle$ lopen.
2. Als R niet aanvaardt, *aanvaard niet*.
3. Als R wel aanvaardt, simuleer M op w.
4. Als M aanvaardt, *aanvaard*, anders *aanvaard niet*.

We zien duidelijk dat als $HALT_{TM}$ beslist wordt door R, dat S dan A_{TM} beslist. Omdat A_{TM} onbeslisbaar is, moet ook $HALT_{TM}$ onbeslisbaar zijn.

Het volgende probleem waarvan we zullen aantonen dat het onberekenbaar is, is E_{TM}.

$$E_{TM} = \{\langle M\rangle |\ \text{M is een TM en } L(M)^1 = \emptyset\}$$

Stelling: E_{TM} is onbeslisbaar.

Intuïtief bewijs: Als we veronderstellen dat E_{TM} wel beslisbaar is door een TM R, dan gaan we proberen aan te tonen dat A_{TM} ook beslisbaar is.

Op een gegeven input $\langle M, w\rangle$, kunnen we R op M laten lopen. Als het aanvaardt, dan weten we dat $L(M)$ leeg is, dus dat M ook onmogelijk w kan aanvaarden. Als R niet aanvaardt, dan weten we enkel nog maar dat $L(M)$ niet leeg is, maar we weten niet met zekerheid of w ertoe behoort of niet. Dus we moeten R op een aangepaste versie van M laten lopen. Hoe dit gebeurt zien we in het echte bewijs.

[1] Met $L(M)$ bedoelen we de taal voortgebracht door TM M. Dit wil dus zeggen de verzameling van alle inputs die worden aanvaard door M

Bewijs: We veronderstelen dus dat E_{TM} beslisbaar is door TM R en gaan aantonen dat A_{TM} dan beslisbaar is, om zo tot een contradictie te komen.

Eerst en vooral zullen we de aangepaste versie van M definiëren als volgt:

M_1 = Op input x:
1. als $x \neq w$, *aanvaard niet*.
2. als $x = w$, laat R op w lopen en *aanvaard* als R aanvaardt.

Deze TM aanvaardt bijna niets. Enkel als de input w is, gedraagt het zich zoals M en anders aanvaardt het niet.

Nu zullen we een TM S construeren die A_{TM} beslist op de volgende manier:

S = Op input $\langle M, w \rangle$:
1. Gebruik M en w om M_1 te construeren.
2. Laat R op M_1 lopen.
3. Als R niet aanvaardt, *aanvaard*, anders, *aanvaard niet*.

Omdat A_{TM} onbelisbaar is moet ook E_{TM} onbeslisbaar zijn.

3.2 Reducties met behulp van computatie histories

De computatie historie van een Turing Machine op een input is gewoonweg de sequentie van configuraties waar de machine doorheen gaat als het de input verwerkt.

Definitie: Als M een Turing Machine is en w een input string, dan is een aanvaardende computatie historie voor M op w een sequentie configuraties, $C_1, C_2, \ldots, C_l$, waarbij C_1 de startconfiguratie is van M op w, C_l een aanvaardende configuratie is van M en elke C_i een legale opvolgende configuratie is op C_{i-1} volgens de regels van M.

Een niet aanvaardende computatie historie voor M op w is hetzelfde gedefinieerd, behalve dat C_l een niet aanvaardende configuratie is.

Computatie histories zijn eindige sequenties, dus als M niet stopt op input w, dan bestaat er geen computatie historie voor M op w. We zien dus in dat deterministische machines maximaal één computatie historie hebben voor een enkele input, en dat niet-deterministische machines er wel meerdere kunnen hebben. We houden ons hier even bezig met deterministische machines.

Ons eerste onbeslisbaarheids probleem betreft een soort machine dat lineair gebonden automaat heet.

Definitie: Een lineair gebonden automaat (LGA) is een Turing Machine dat niet toegelaten is om de tape head van het stuk tape, waar de input op staat, te bewegen. Als het toch voorbij één van beiden uiteinden probeert te bewegen, blijft het staan waar het staat.

Ik wil dan het probleem A_{LGA} bespreken, het probleem dat nagaat of een LGA een bepaalde inpunt aanvaardt of niet.

$$A_{LGA} = \{\langle M, w\rangle | M \text{ is een LGA die de } w \text{ aanvaardt}\}$$

Als eerste gedacht zullen sommigfen misschien denken dat A_{LGA}, net zoals A_{TM}, ook onbeslisbaar is, maar we zullen zien dat A_{LGA} wel beslisbaar is. In het bewijs hiervan zullen we gebruik maken van volgend lemma:

Lemma: Als M een LGA is met q verschillende toestanden en g symbolen in het tape alfabet, dan zijn er juist qng^n verschllende configuraties van M op een tape van lengte n.

Bewijs: Een configuratie bestaat uit een controle toestand, een positie voor de kop, en de inhoud van de tape. M heeft in dit geval q toestanden. De lengte van de tape is n, dus de kop kan zich op één van de n posities bevinden en er kunnen g^n verschillende strings op de tape bevinden. Dus het aantal verschillende mogelijke configuraties is de vermenigvuldiging van deze drie hoeveelheden en is dus qng^n.

Stelling: A_{LBA} is beslisbaar

Intuïtief bewijs: De reden waarom A_{TM} niet beslisbaar is, is omdat we niet kunnen beslissen wanneer een Turing Machine in een loop terechtkomt. Dit probleem kunnen we hier oplossen omdat we weten dat er maar een beperkt aantal mogelijke configuraties zijn. Als een bepaalde configuratie herhaalt wordt, kunnen we er zeker van zijn dat de Turing Machine zich in een loop bevindt. En als er meer stappen worden uitgevoerd dan dat er verschillende configuraties zijn, dan weten we ook dat er een bepaalde configuratie herhaalt is, dus dat de Turing Machine zich in een loop bevindt.

We laten de turing Machine dus voor die beperkte tijd lopen op de input. Als deze binnen de gegeven tijd stopt, aanvaarden we als de Turing Machine dat doet, en anders niet. Als de Turing Machine niet binnen de gegeven tijd stopt, weten we dat het in een loop zit, en dat het de input dus niet aanvaardt.

Bewijs: We maken een Turing Machine L die A_{LGA} beslist als volgt:

$L =$ Op input $\langle M, w\rangle$:
1. Simuleer M op w totdat het stopt of totdat er meer dan qng^n stappen zijn voorbij gegaan.
2. *Aanvaard* als M stopt en aanvaardt. Anders *aanvaard niet*.

Als M nog niet is gestopt binnen qng^n stappen, moet het in een loop zitten en wordt er niet aanvaard.

Op dit moment zouden we misschien kunnen denken dat omdat het aanvaardingsprobleem beslisbaar is, ook het leegheidsprobleem beslisbaar zal zijn. Toch is dat niet het geval, zoals we hieronder zullen zien.

$$E_{LGA} = \{\langle M \rangle | M \text{ is een LGA met } L(M) = \emptyset\}$$

Stelling: E_{LGA} is onbeslisbaar.

Intuïtief bewijs: We gaan aantonen dat indien E_{LGA} wel beslisbaar is, we dan ook A_{TM} kunnen beslissen. Vanuit een TM M en een string w maken we een LGA B die alle aanvaardende computatie histories van M op w beslist. Duidelijk geldt dan dat als M aanvaardt op w, dat $L(B)$ niet leeg is. En als M niet aanvaardt op w, is $L(B)$ leeg. Dus als we B kunnen construeren en testen of zijn taal leeg is of niet, kunnen we A_{TM} beslissen.

Bewijs: Stel dat E_{LGA} wel beslisbaar is door de TM R. We maken dan een LGA B die beslist of een bepaalde input een aanvaardende computatie historie is van M op w. Om dit te doen veronderstellen we (zonder verlies van algemeenheid) dat de aanvaardende computatie historie is voorgesteld als een string, waarbij de configuraties zijn onderscheiden door het symbool #.

B = Op input x:

1. Breek x in stukken $C_1, C_2, \ldots, C_l$ op, waarbij C_i het stukje string is dat zich tussen het $i-1^e$ en het i^e symbool # bevindt.

2. Ga na of aan de volgende condities is voldaan:

  1. C_1 is de start configuratie.

  2. Elke C_{i+1} is een legale opvolpging op C_i.

  3. C_l is een aanvaardende configuratie van M.

De startconfiguratie ziet er als volgt uit: $q_0 w_1 w_2 \ldots w_n$, waarbij $x = w_1 w_2 \ldots w_n$. Dus dit is gemakkelijk na te gaan. Om na te gaan of C_l een aanvaardende configuratie is, moet er gewoon gekeken worden of $q_{aanvaardt}$ zich ergens in C_l voordoet. Om te checken of C_{i+1} legaal volgt op C_i, moet er gewoon nagegaan worden of de twee string identiek zijn op de positie onder de kop van de TM en de posities ervoor en erna, na. Deze moeten aan een transitie functie van M voldoen.

We zijn nu klaar om een TM S te maken die A_{TM} beslist:

S = Op input $\langle M, w \rangle$:

1. Construeer de LGA B.

2. Laat R op $\langle B \rangle$ lopen.

3. Als R aanvaardt, *aanvaard niet*. Als R niet aanvaardt, *aanvaard*.

Als R aanvaardt op B, wilt het zeggen dat $L(B) = \emptyset$, en dat er dus geen aanvaardende computatie historie van M op w bestaat. Dit wilt zeggen dat M niet aanvaardt op w.

Als R niet aanvaardt op B, wilt dat zggen dat $L(B) \neq \emptyset$, en dat er dus een aanvaardende computatie historie bestaat van M op w. Hieruit volgt dat M aanvaardt op w.

3.3 Mapping reduceerbaarheid

Eigenlijk hebben we de techniek van reduceerbaarheid al gebruikt bij het veronderstellen dat als het ene probleem beslisbaar is een ander onbeslisbaar probleem ook beslisbaar zou zijn. In deze sectie zal de notie van reduceerbaarheid geformaliseerd worden. Ruw gesproken komt het reduceren van een probleem A tot een probleem B overeen met een functie die instanties van probleem A afbeelt op instanties van probleem B. Als er zo een functie bestaat, kunnen we A oplossen als we een een algoritme R hebben dat B oplost. De reden hiervoor is omdat we elke instantie van A kunnen oplossen door ze eerst om te vormen tot een instantie van B en dan daar R op toe kunnen passen om tot het juiste resultaat te komen.

Om tot de definitie van mapping reduceerbaarheid te komen hebben we eerst nog volgend begrip nodig:

Definitie: Een functie $f : \Sigma^* \rightarrow \Sigma^*$ heet een berekenbare functie, als er een Turing Machine M bestaat die op elke input w stopt met $f(w)$ op zijn tape.

Alle rekenkundige operaties op integers zijn berekenbare functies. Een berekenbare functie kan ook een transformatie van Turing Machines zijn.

Definitie: Een taal A is mapping reduceerbaar tot een taal B, genoteerd als $A \leq_m B$, als er een berekenbare functie $f : \Sigma^* \rightarrow \Sigma^*$ bestaat, zodat:

$$\forall w : w \in A \Leftrightarrow w \in B$$

Een mapping reductie van A naar B biedt de mogelijkheid om het bepalen van lidschap van A om te vormen tot het bepalen van lidschap van B. om na te gaan of $w \in A$, gebruiken we de reductie f om w af te beelden op $f(w)$ en testen of $f(w) \in B$.

Als het ene probleem mapping reduceerbaar is tot het andere, dat al een oplossing bevat, dan hebben we ook een oplossing voor het ene. Dit idee wordt in de volgende stelling geformailiseerd.

Stelling: Als $A \leq_m B$ en B is beslisbaar, dan is ook A beslisbaar.

Bewijs: Stel dat M de Turing Machine is die B beslist en dat f de reductie van A naar B is. Dan maken we een Turing Machine N die A beslist als volgt:

N = Op input w:
1. Bereken $f(w)$.
2. Laat M op $f(w)$ lopen en *aanvaard* als M aanvaardt, anders niet.

Het is duidelijk dat als $w \in A$, dan moet $f(w) \in B$ en zal N aanvaarden. Als $w \notin A$, dan zal ook $f(w) \notin B$ en dus N ook niet aanvaarden.
Het volgende gevolg is de hoofdzaak om onbeslisbaarheid aan te tonen.

Gevolg: Als $A \leq_m B$ en A is onbeslisbaar, dan is ook B onbeslisbaar.

Bewijs: Stel dat B wel beslisbaar is, dan is volgens vorige stelling ook A beslisbaar, wat tot een contradictie leidt. Dus moet B wel onbeslisbaar zijn.

Eerder in dit hoofdstuk hebben we via een contradictie aangetoond dat $HALT_{TM}$ niet beslisbaar is. We kunnen dat ook aantonen met behulp van een mapping reductie op de volgende manier: we moeten een functie f vinden die een input voor A_{TM}, die van de vorm $\langle M, w\rangle$ is, afbeeldt op een input voor $HALT_{TM}$, die van de vorm $\langle M', w'\rangle$ is, zodat: $\langle M, w\rangle \in A_{TM} \Leftrightarrow \langle M', w'\rangle \in HALT_{TM}$.
We weten dat als M aanvaardt, dat hij ook moet stoppen, dus in dat geval mag M' ook accepten. We weten ook dat als M niet aanvaardt, het ook stopt en dus voor dezelfde input geen element van $HALT_{TM}$ is. Dus we moeten ervoor zorgen dat als M niet aanvaardt, M' loopt. Als aan deze voorwaarden is voldaan geldt het volgende: $\langle M, w\rangle \in A_{TM} \Leftrightarrow \langle M', w\rangle \in HALT_{TM}$.

We stappen nu over van beslisbaarheid naar herkenbaarheid.

Stelling: Als $A \leq_m B$ en B Turing-herkenbaar is, dan is ook A Turing-herkenbaar.

Bewijs: Stel dat M de Turing Machine is die B herkent en dat f de reductie van A naar B is. We maken dan een Turing Machine N die A herkent als volgt:

N = Op input w:
1. Bereken $f(w)$.
2. Laat M op $f(w)$ lopen en *aanvaard* als M aanvaardt, anders niet.

Het is duidelijk dat als $w \in A$, dan moet $f(w) \in B$ en zal N aanvaarden, want M moet alle elementen van B herkennen.

Gevolg: Als $A \leq_m B$ en A niet Turing-herkenbaar is, dan is ook B niet Turing-herkenbaar.

Bewijs: Stel dat B wel Turing-herkenbaar is, dan zou wegens vorige stelling ook A Turing-herkenbaar moeten zijn. Vermits dit niet het geval is, kan B niet Turing-herkenbaar zijn.

Een typisch gebruik van vorig gevolg is het volgende: we nemen voor A het complement van A_{TM}, genoteerd als $\overline{A_{TM}}$, waarvan ik uit mijn opleiding al weet dat het niet Turing-herkenbaar is. Om te bewijzen dat B niet herkenabaar is zouden we kunnen aantonen dat $\overline{A_{TM}} \leq_m B$. Maar omdat uit de definitie van mapping reduceerbaarheid duidelijk geldt dat $\overline{A} \leq_m \overline{B}$ hetzelfde betekent als $A \leq_m B$, kunnen we aantonen dat $A_{TM} \leq_m \overline{B}$.

3.4 Voorbeelden en oefeningen

Stelling: A_{TM} is niet mapping reduceerbaar tot E_{TM}

Bewijs: Stel dat $A_{TM} \leq_m E_{TM}$. Hieruit volgt dat $\overline{E_{TM}}$ niet Turing herkenbaar is. Maar het is wel Turing herkenbaar volgens volgende Turing Machine T, die E_{TM} duidelijk herkent.

T = Op input $\langle M \rangle$:
1. Loop alle strins s op één of andere natuurlijke manier af.
2. Laat M op s lopen.
3. Als M aanvaardt, *aanvaard*, anders ga verder met de volgende string.

Stelling: $\leq_m$ is een transitieve relatie

Bewijs: Uit $A \leq_m B$ weten we dat $\exists f : w \in A \Leftrightarrow f(w) \in B$.
Uit $B \leq_m C$ weten we dat $\exists g : s \in B \Leftrightarrow g(s) \in C$.
Dus $w \in A \Leftrightarrow g(f(w)) \in C$. dit wilt zeggen dat $A \leq_m C$.

Stelling: Als A Turing-herkenbaar is en $A \leq_m \overline{A}$, dan is A beslisbaar.

Bewijs: We weten dat als A en $\overline{A}$ beiden Turing herkenbaar zijn, dat A beslisbaar is. Dus rest ons nog aan te tonen dat $\overline{A}$ Turing herkenbaar is.

Uit $A \leq_m \overline{A}$ volgt rechtstreeks uit de definitie van mapping reduceerbaarheid dat ook $\overline{A} \leq_m \overline{\overline{A}}$, dus $\overline{A} \leq_m A$. Omdat A Turing herkenbaar is volgt dat ook $\overline{A}$ Turing herkenbaar is.

Stelling: Elk Turing herkenbaar probleem is mapping reduceerbaar naar A_{TM}

Bewijs: $\exists$ TM T die B herkent. $\forall$ input w voor $T : f(w) = \langle T, w \rangle$ is een input voor A_{TM}. Dan geldt er: $w \in B \Leftrightarrow T$ aanvaardt $w \Leftrightarrow \langle T, w \rangle \in A_{TM} \Leftrightarrow f(w) \in A_{TM}$.

Oefening: Stel $J = \{w|w = 0x \text{ met } x \in A_{TM} \text{ of } w = 1y \text{ met } y \in \overline{A_{TM}}\}$. Toon aan dat J noch $\overline{J}$ Turing herkenbaar is.

Oplossing:

- J is niet Turing herkenbaar: Stel dat J wel Turing herkenbaar is. Dan weten we dat er een TM M bestaat die J gedeeltelijk beslist. Maak dan volgende TM B die $\overline{A_{TM}}$ herkent:

  $B =$ Op input $\langle T, w\rangle$:
  1. Maak $1\langle T, w\rangle$.
  2. Laat M op $1\langle T, w\rangle$ lopen en output wat M output.

  Als $1\langle T, w\rangle \in M$, dan $\langle T, w\rangle \in B$. Dus B herkent A_{TM}

- $\overline{J}$ is niet Turing herkenbaar: Voor elke input w voor A_{TM}, is $f(w) = 0w$ een input voor J. Dus $w \in A_{TM} \Leftrightarrow 0w \in J \Leftrightarrow f(w) \in J$. Hieruit volgt dat $A_{TM} \leq_m J$ via f, zodat we kunnen besluiten dat $\overline{J}$ niet Turing herkenbaar is.

Stelling: $S = \{\langle M\rangle | M \text{ is een TM die } w^{R2} \text{ aanvaardt zodra deze } w \text{ aanvaardt}\}$ is onbeslisbaar.

Bewijs: We gaan een functie f beschrijven die een TM M omzet in een TM M', op een manier zodat $M \in E_{TM} \Leftrightarrow M' \in S$.

We breiden het alfabet van M uit met twee nieuwe symbolen die we hier $\diamond$ en # zullen noteren. We willen ervoor zorgen dat $w \in L(M) \Leftrightarrow \diamond w\# \in L(M')$. We kunne hiervoor M' als volgt maken:

$M' =$ Op input w
1. Ga na of w van de vorm $\diamond w_0\#$ is.
2. Simuleer M op w_0.

Als $M \notin E_{TM}$, dan bestaat er een $w \in L(M)$. Dus $w' = \diamond w\# \in L(M')$ en daarmee $w'^R \notin L(M')$. Hieruit volgt dat $M' \notin S$.

Als $M \in E_{TM}$, dan bestaat er geen $w \in L(M)$. Dus er bestaat ook geen $w \in L(M')$. Hieruit volgt triviaal gezien dat $M' \in S$.

[2] met w^R bedoelen we de string die we bekomen door w van rechts naar links te lezen i.p.v. links naar rechts

Hoofdstuk 4

Geavanceerde onderwerpen

In dit hoofdstuk zal ik een iets diepere kijk nemen op bepaalde aspecten in de berekenbaarheidstheorie. Daarmee zal elke sectie op zijn eigen staan. Ik zal het recursie theorema in de eerste sectie behandelen, de tweede sectie zal over logische theoriën gaan en in het derde deel bekijken we een andere vorm van reduceerbaarheid, nl. Turing reduceerbaarheid. Het laatste onderdeel zal een manier om informatie voor te stellen behandelen.

4.1 Het recursie theorema

Het recursie theorema stelt dat er computer-programm's bestaan die zichzelf kunnen reproduceren. Ik zal hier bespreken hoe dit werkt voor Turing Machines, en dus als we weten dat het voor Turing Machines geldt, dan moet het ook voor alle andere volledige programmeertalen gelden. Eerst zal ik behandelen hoe we een Turing Machine, $SELF$ kunnen maken die zijn input negeert en gewoon een kopie van zijn eigen uitprint. Hiervoor hebben we eerst volgend lemma nodig:

Lemma: Er bestaat een berekenbare functie $q : \Sigma^* \to \Sigma^*$, die elke string, w, afbeeldt op $q(w)$, dat een beschrijving is ven een Turing Machine P_w, die w uitprint en dan stopt.

Bewijs: We moeten gewoon van een string w een Turing Machine maken die w op zijn tape schrijft.

Q = Op input w:
1. Maak de volgende TM P_w:
   P_w = Op input x
   1. Wis de input.
   2. Schrijf w op de tape.
   3. Stop.
2. Geef P_w terug.

We zijn nu klaar om de Turing Machine $SELF$ te construeren. $SELF$ bestaat uit twee delen, A en B. Dit zijn twee aparte procedures die samen $SELF$ vormen. De bedoeling van $SELF$ is dan $\langle SELF\rangle = \langle AB\rangle$ uit te printen.

Deel A loopt eerst en moet de beschrijving van B uitprinten. Hierna loopt B en moet de beschrijving van A uitprinten.

Om A te maken gebruiken we de Turing Machine $P_{\langle B\rangle}$, gegeven door $q(\langle B\rangle)$. Dus A kan pas bestaan zodra we B hebben.

We zouden geneigd zijn om B met behulp van $q(\langle A\rangle)$ te definiëren. Dit werkt niet omdat A dan in functie van B gedefinieerd zou zijn en B op zijn beurt in functie van A. Het is wel mogelijk voor B om de beschrijving van A te berekenen, door de output van A te gebruiken: A is als $q(\langle B\rangle)$ gedefiniëerd. Als B nu zijn eigen beschrijving kan bekomen, dan kan het daar q op toe passen om $\langle A\rangle$ te bekomen. En A laat juist $\langle B\rangle$ op de tape achter, dus B kan gewoon de tape lezen om zo de beschrijving van zijn eigen te vinden en daar dan q op toepassen. B berekent dus $q(\langle B\rangle) = \langle A\rangle$ en zet dit vooraan op de tape. Het resultaat van beiden is dus $\langle AB\rangle = \langle SELF\rangle$.

Samengevat hebben we dus volgende twee Turing Machines tot één Turing Machine gecombineer:
$A = P_{\langle B\rangle}$

$B =$ Op input $\langle M\rangle$
1. Bereken $q(\langle M\rangle)$.
2. Combineer het resultaat van de berekening met $\langle M\rangle$ om de volledige beschrijving te bekomen.
3. Print de beschrijving uit en stop.

We zijn tot het punt gekomen dat we kunnen aantonen dat we een Turing Machine kunnen maken die zijn eigen beschrijving kan verkrijgen en daarmee verder kan berekenen.

Recursie theorema: Stel dat T een Turing Machine is die de functie $t : \Sigma^* \times \Sigma^* \to \Sigma^*$. Dan is er een Turing Machine R die de functie $r : \Sigma^* \to \Sigma^*$ berekent, zodat:

$$\forall w : r(w) = t(\langle R\rangle, w)$$

Bewijs: Het bewijs hiervan verloopt ongeveer hetzelfde als de constructie van $SELF$. Hier moeten we een TM R maken die uit drie delen bestaat, we noemen ze A, B en T. Zoals bij de constructie van $SELF$ zal ook hier A de beschrijving van rest van M, nl. $\langle BT\rangle$, op de tape schrijven. Dit gebeurt natuurlijk weer door A als $P_{\langle BT\rangle}$ te definiëren. $P_{\langle BT\rangle}$ bekomt men weer door q op $\langle BT\rangle$ toe te passen. Weeral zal B de tape checken en daar ook q op toe passen zodat we $\langle A\rangle$ verkrijgen. B moet dan nog wel de twee delen aan elkaar plakken om $\langle ABT\rangle$ te verkrijgen. Nu wordt de controle aan T gegeven.

Op het eerste zicht kan het misschien lijken dat er met het recursie theorema enkel minder interessante dingen kunnen gedaan worden, zoals een Turing Machine maken die zijn eigen beschrijving afprint. Maar zoals we verder zullen zien kunnen er ook interessantere dingen mee aangetoond worden. Zo kunnen we er bijvoorbeeld op een elegante manier mee aantonen dat A_{TM} niet beslisbaar is.

Stelling: A_{TM} is onbeslisbaar.

Bewijs: Stel dat er wel een Turing Machine H bestaat die A_{TM} beslist. We kunnen dan de volgende Turing Machine B contrueren:

B = Op input w:
1. Verkrijg via het recursie theorema je eigen beschrijving $\langle B \rangle$.
2. Laat H op $\langle B, w \rangle$ lopen.
3. Doe het omgekeerde van H, dat is: *aanvaard* als H dat niet doet, en *aanvaard niet* als H wel aanvaardt.

Dit leidt tot een contradictie: als H ons vertelt dat B aanvaardt op input w, dan aanvaardt B niet. Dit is dus onmogelijk en daarmee kan A_{TM} onmogelijk beslisbaar zijn.

Om nog een ander mooi voorbeeld van een toepassing van het recursie theorema te geven hebben we eerst volgende definitie nodig:

Definitie: Als M een Turing Machine is, dan zeggen we dat de lengte van de beschrijving $\langle M \rangle$ gelijk is aan het aantal symbolen in de string die M beschrijft. We noemen een Turing Machine M minimaal als er geen equivalente[1] Turing Machine bestaat met een kortere beschrijving.

$$MIN_{TM} = \{\langle M \rangle | M \text{ is een minimale TM}\}$$

Stelling: MIN_{TM} is onbeslisbaar.

Bewijs: Veronderstel dat er wel een Turing Machine E bestaat die de elementen van MIN_{TM} opsomt. Dan zouden we de volgende Turing Machine C kunnen maken:

C = Op input w:
1. Bekom eigen beschrijving $\langle C \rangle$ via het recursie theorema.
2. Laat E lopen totdat we TM D tegenkomen met een grotere beschrijving dan die van C.
3. Simuleer D op w.

Omdat MIN_{TM} oneindig is, moet D bestaan. In dit geval zou C equivalent zijn aan D, en toch een kleinere beschrijving hebben. Dit kan niet want $D \in MIN_{TM}$.

[1] We noemen twee Turing Machines equivalent als ze voor iedere input, beiden dezelfde output produceren.

4.2 Turing reduceerbaarheid

Ik heb al een vorm van reduceerbaarheid, nl. mapping reduceerbaarheid, besproken in hoofdstuk 3. In deze sectie zal ik een andere vorm van reduceerbaarheid introduceren, die meer aan de intuïtie van reduceerbaarheid voldoet. Intuïtief willen we dat problemen reduceerbaar zijn naar elkaar als een oplossing voor het ene ook een oplossing voor het andere beidt. Bijvoorbeeld, A_{TM} is niet mapping reduceerbaar tot $\overline{A_{TM}}$, maar een oplossing voor één van beiden zou een oplossing voor het andere geven. Turing reduceerbaarheid vangt deze intuïtie op.

Definitie: Een orakel voor een taal B is een externe machine die kan bepalen of een string w al dan niet in B zit. Een orakel Turing Machine is een Turing machine die de bijkomende mogelijkheid heeft om een oracle te bevragen. We noteren M^B om een Turing Machine M met een orakel voor B te beschrijven.

Een Turing Machine $T^{A_{TM}}$ met een orakel voor A_{TM} kan duidelijk meer problemen oplossen dan een gewone Turing Machine. Om dit in te zien is het voldoende om op te merken dat $T^{A_{TM}}$ duidelijk A_{TM} zelf kan oplossen. Maar het kan ook bijvoorbeeld E_{TM} oplossen:

$C^{A_{TM}}$ = Op input $\langle M \rangle$:
1. Construeer de volgende TM N:
N = Op elke input:
1. Laat M op alle strings in Σ^* in parallel lopen.
2. Als M op minstens één van deze strings aanvaardt, *aanvaard*.
2. Vraag het orakel of $\langle N, 0 \rangle \in A_{TM}$.
3. Als het orakel NEE antwoordt, *aanvaard*, anders *aanvaard niet*.

Als $L(M) \neq \emptyset$, dan zal N op elke input aanvaarden, dus zeker op 0. Dus het orakel zal JA antwoorden, en $L(M)$ zal niet leeg bevonden worden door $C^{A_{TM}}$. Als $L(M)$ wel leeg is, dan zal $\langle N, 0 \rangle$ niet in A_{TM} zitten. Het orakel zal dus NEE antwoorden, en daardoor zal $L(M)$ als element van E_{TM} bevonden worden door $C^{A_{TM}}$.

Dus $C^{A_{TM}}$ beslist E_{TM}. We zeggen dat E_{TM} relatief beslisbaar is ten opzichte van A_{TM}.

Definitie: Een taal A is Turing reduceerbaar tot taal B, genoteerd als $A \leq_T B$, als A beslisbaar relatief is aan B.

Stelling: Als $A \leq_T B$ en B beslisbaar is, dan is ook A beslisbaar.

Bewijs: $A \leq_T B$ wilt zeggen dat we met een orakel voor B, A kunnen beslissen. Vermist B beslisbaar is, is het orakel overbodig, want we kunnen het gewoon vervangen door de Turing Machine die B beslist.

4.3 Een definitie voor informatie

Er bestaat geen veelomvattende, universele definitie voor informatie. Daarom wordt er voor ongeveer iedere toepassing een ander definitie gebruikt. In dit onderdeel wil ik een manier voorstellen die gebaseerd is op berekenbaarheidstheorie.

Vooraleer een definitie te geven wordt er aangehaald dat we het enkel over binaire informatie hebben. Dit is geen restrictie want alle informatie kan op de één of andere manier wel gecodeerd worden door binaire strings. Het volgende punt is dat de beschrijvingen van de informatie zelf ook in binair gebeurt. Zo kan de lengte van de originele string gemakkelijk vergeleken worden met zijn beschrijving.

Definitie: De minimale beschrijving van een binaire string x, genoteerd als $d(x)$ is de korste string $\langle M, w\rangle$[2], waarvoor geldt dat de TM M stopt op input w met x o pzijn tape. Als er verschillende van die strings bestaan met dezelfde kortste lengte, neem dan deze die lexicografisch gezien het eerste komt.

Definitie: De beschrijvingscomplexiteit van x, genoteerd als $K(x)$, is de lengte van $d(x)$, genoteerd als $|d(x)|$.

Stelling: $\exists c \forall x \, [K(x) \leq |x| + c]$.

Bewijs: Stel dat de TM M stopt zodra hij start. Deze TM berekent dus de identiteits-functie (de input is hetzelfde als de output). Dus een beschrijving voor x is simpelweg $\langle M, x\rangle$. Dus we kunnen c gelijk aan de langte van $\langle M\rangle$ stellen.

Deze stelling vertelt ons dat de beschrijvingscompexiteit op zijn meest een constante groter is als de lengte van de string zelf. Deze constante is universeel, onafhankeleijk van de string zelf.

De volgende stelling verifieert onze intuïtie dat de string xx niet veel meer informatie kan bevatten dan x zelf.

Stelling: $\exists c \forall x \, [K(xx) \leq K(x) + c]$.

Bewijs: Beschouw de volgende TM M:

M = Op input $\langle N, w\rangle$:
1. Laat N op w lopen totdat het stopt en een string s produceert.
2. Output ss.

Een beschrijving voor xx is $\langle M, d(x)\rangle$. De lengte van deze string is $c + K(x)$, waarbij c de lengte is van $\langle M\rangle$.

[2] $\langle M, w\rangle$ is eigenlijk $\langle M\rangle w$, waar we x gewoon achter de beschrijving van M plakken

We zullen nu onderzoeken hoe de beschrijvingscomlexiteit van xy gerelateerd is aan de individuele complexiteiten van x en y. Eigenaardig genoeg komen we tot een grotere complexieteit dan de som van de twee aparte complexiteiten, willen we een algemene limiet op de complexiteit leggen.

Stelling: $\exists c \forall x, y\, [K(xy) \leq 2K(x) + K(y) + c]$

Bewijs: We construeren een TM M die zijn input w in twee beschrijvingen kapt. De bits van de eerste beschrijving $d(x)$ worden verdubbeld en beëindigd met de string 01. Dit om nadien nog te weten waar de eerste beschrijving stopt. Hierachter wordt het tweede beschrijving $d(y)$ geplakt. Zodra beide beschrijvingen verkregen zijn, kan men ze laten lopen om de strings x en y verkrijgen en zo kan de output xy geproduceerd worden.

Het is duidelijk dat deze beschrijving van xy twee keer de complexiteit van x is plus die van y plus een vaste constante om M te beschrijven. Dit is dus $2K(x) + K(y) + c$.

Oncompresseerbare strings

We hebben reeds gezien dat de minimale beschrijving van een string nooit veel langer is dan de lengte van de string zelf. Natuurlijk is dit maar een bovengrens en zijn er vele strings waarvan de minimale beschrijving korter is dan de string zelf. DE vraag die hier behandeld wordt, is 'bestaan er strings waarvan de minimale beschrijving niet korter is dan de string zelf?'

Definitie: Als x een string is, zeggen we dat hij c-compresseerbaar is als

$$K(x) \leq |x| - c$$

Als x niet c-compresseerbaar is, zeggen we dat x oncompresseerbaar is door c.

Als x oncompresseerbaar is door 1, zeggen we dat x oncompresseerbaar is.

Stelling: Er bestaan oncompresseerbare strings van iedere lengte.

Intuïtief bewijs: Dit is intuïtief eigenlijk vrij gemakkelijk in te zien. Het aantal strings van lengte n is groter dan het aantal strings van lengte kleiner dan n. Elke beschrijving kan maar de voorstelling van één string zijn, dus moeten er strings van lengte n zijn die niet beschreven kunnen worden door een kortere string.

Bewijs: Het aantal binaire strings van lengte 2 is gelijk aan 2^n. Elke beschrijving is een niet lege binaire string, dus het aantal beschrijvingen van lengte minder dan n is gelijk aan de som van het aantal strings van lengte 1 tot lengte $n - 1$,

$$\sum_{i=0}^{n-1} 2^i = 1 + 2 + 4 + 8 + \ldots + 2^{n-1} = 2^n - 1$$

Dus het aantal korte beschrijvingen is kleiner dan het aantal strings van lengte n. Daarom moet er minsten één string va lente n zijn die oncompresseerbaar is.

Gevolg: Er zijn minstens $2^n - 2^{n-c+1} + 1$ strings van lengte n die oncompresseerbaar zijn door c.

Bewijs: Uit vorige stelling volgt rechtstreeks dat er maximaal $2^{n-c+1} - 1$ strings van lengte n zijn die c-compresseerbaar zijn, omdat er maximaal zoveel strings van lengte $n - c$ of kleiner bestaan. Dus de overige $2^n - 2^{n-c+1} + 1$ moeten oncompresseerbaar zijn door c.

Stelling: $\exists b \forall x$: $d(x)$ is oncompresseerbaar door b.

Bewijs: Beschouw de volgende TM M:

M =Op input $\langle R, y \rangle$:
1. Laat R op y lopen en als de output er van niet van de vorm $\langle S, z \rangle$ is, *aanvaard niet*.
2. Laat S op z lopen en stop met de output ervan op de tape.

We tonen aan dat voor $b = |\langle M \rangle| + 1$ de stelling geldt. Stel dat $d(x)$ wel b-compresseerbaar is voor een bepaalde string x. Dan is

$$|d\,(d\,(x))\,| \leq |d(x)| - b.$$

Maar dan zou $\langle M \rangle d\,(d\,(x))$ een beschrijving van x zijn waarvan de lengte maximaal gelijk is aan

$$|\langle M \rangle| + |d\,(d\,(x))\,| \leq (b-1) + (|d\,(x)\,| - b) = d\,(x) - 1.$$

Deze beschrijving van x zou korter zijn als $d(x)$, wat niet kan want $d(x)$ wordt als minimaal gedefinieerd.

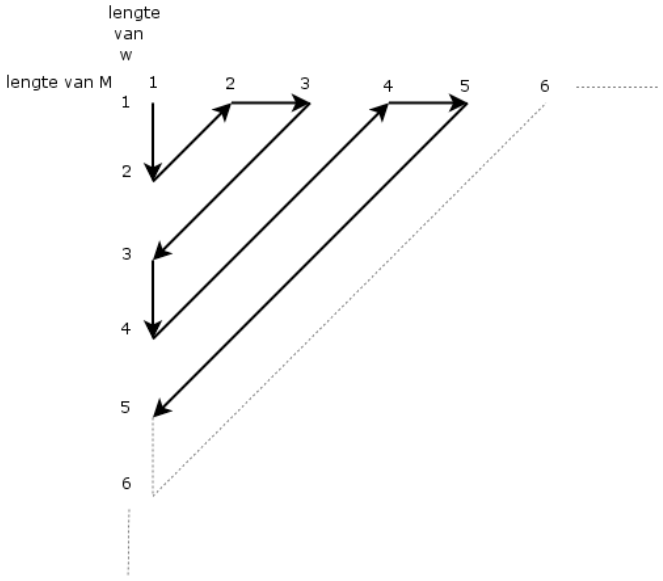
4.4 Voorbeelden en oefeningen

Stelling: Als $A \leq_T B$ en $B \leq_T C$ dan $A \leq_T C$.

Bewijs: We moeten aantonen dat A te beslissen valt met behulp van C. Dus dat als we een orakel of een beslissingsalgoritme voor C hebben, dat we dan A ermee kunnen beslissen. Maar we weten alreeds dat als we een orakel of beslissingsalgoritme voor B hebben, we daaarmee A kunnen beslissen. Maar zodra we een orakel voor of beslissingsalgoritme voor C hebben, kunnen we daarmee B beslissen. Dus uiteindelijk hebben we enkel iets voor C nodig, want daarmee hebben we iets voor B en dan hebben we iets voor A.

Stelling: Met een orakel voor A_{TM} kunnen we de de beschrijvingscomplexiteit $K(x)$ van een string x berekenen.

Bewijs: We gaan alle TM af samen met alle strings op een speciale manier zoals aangegeven in figuur 4.1



Figuur 4.1: speciale manier om TM's samen met inputs af te gaan

We bezoeken eerst alle TM's M van lengte 1 met inputs w van lengte 1, vervolgens inputs van lengte 2, samen met TM's van lengte 1. Hierna TM's van lengte 2 met inputs van lengte 1. Dan TM's van lengte 3 met inputs van lengte 1. Vervolgens nemen we TM's van lengte 2 en inputs van lengte 2, enzo verder. Dit is van oneindige duur, maar laat wel blijken dat de verzameling

$$\{\langle M, w\rangle | M \text{ is een TM en } w \text{ is een inputstring}\}$$

opsombaar is.

We zijn dan nu klaar om de TM $S^{A_{TM}}$, die $K(x)$ berekent, te construeren:

$S^{A_{TM}}$ = Op input x:

1. Loop alle TM's M en inputstrings w af, zoals hierboven beschreven.
2. Laat A_{TM} lopen op $\langle M, w\rangle$.
3. Als A_{TM} niet aanvaardt, ga verder met de volgende TM en inputstring.
4. Als A_{TM} wel aanvaardt, laat M op w lopen.
5. Als de uitkomst hiervan verschillend is van x, ga verder met de volgende TM en inputstring.
6. Als de uitkomst wel x is, ouput dan $|\langle M, w\rangle|$.

We weten zeker dat er een TM M en inputstring w bestaan die een beschrijving zijn van x. Hierdoor zal het bovestaand algoritme deze zeker eens moeten tegenkomen en dus altijd stoppen.

Gevolg: Met een orakel voor A_{TM} bestaat er een berekenbare functie f die iedere n afbeeldt op een oncompresseerbare string van lengte n.

Bewijs: Vermist er maar 2^n verschillende binaire strings x van lengte n zijn, kunnen we deze allemaal afgaan. We kunnen dan, dankzij voorgaande stelling, $K(x)$ berekenen, en als $K(x) \geq n$ is deze outputten.

We weten dat er voor iedere n minstens één string van lengte n bestaat die oncompresseerbaar is, dus ooit zullen we deze tegenkomen waardoor het bovenstaande algoritme zeker stopt.

Hoofdstuk 5

PSPACE-compleetheid

Definitie: Een taal B is PSPACE-compleet als het aan de volgende twee condities voldoet:

1. $B \in PSPACE$
2. $\forall A \in PSPACE : A$ is polynomiaal tijd reduceerbaar tot B

Als B enkel aan de tweede conditie voldoet noemen we B PSPACE-hard.

5.1 Het TQBF probleem

Ik zal dadelijk beschrijven hoe men kan aantonen dat het $TQBF$ (true quantified boolean formula) probleem. Maar vooraleer ik daaraan kan beginnen, moet het probleem gedefiniëerd worden. Eerst en vooral moet er duidelijk gemaakt worden wat een booleaanse formule is.

Een booleaanse formule is een expressie met booleaanse variabelen en operaties. Een booleaanse variabele is een variabele die de waarde 0 (VALS) of 1 (WAAR) kan aannemen. De booleaanse operatoren EN, OF en NIET, respectievelijk genoteerd als $\vee, \wedge$ en $\neg$ worden als volgt gedefinëerd:

$$x \vee y = 0 \Leftrightarrow x = 0 \text{ en } y = 0$$
$$x \wedge y = 1 \Leftrightarrow x = 1 \text{ en } y = 1$$
$$\bar{x} = 0 \Leftrightarrow x = 1$$

Eigenlijk is deze definitie voldoende om alle gevallen te bedekken, maar voor alle duidelijkheid zal ik het volgende er nog bij vermelden:

$$x \vee y = 1 \Leftrightarrow x = 1 \text{ of } y = 1$$
$$x \wedge y = 0 \Leftrightarrow x = 0 \text{ of } y = 0$$
$$\bar{x} = 1 \Leftrightarrow x = 0$$

Het volgende is een voorbeeld van een booleaanse formule:

$$\phi = (\bar{x} \wedge y) \vee (x \wedge \bar{z})$$

We noemen een booleaanse formule bevredigbaar als er een invullig, voor de variabelen in de formule, van 0'en en 1'en bestaat zodat de hele formule tot 1 evalueert.

Nu gaan we de quantifiers $\forall$ (voor alle) en $\exists$ (er bestaat) erbij voegen. De meesten zullen deze symbolen wel kennen vermits ze ook vaak gebruikt worden in de wiskunde. Als men $\forall x[\phi]$ schrijft, bedoelt me dat voor elke waarde die x kan aannemen de uitdrukking ϕ moet gelden. Gelijkaardig is de betekenis van $\exists x\phi$ dat er een waarde voor x moet bestaan zodat de uitdrukking ϕ geldt.

Er mogen meerdere quantifiers in een formule voorkomen. Ze mogen ook op eender welke plaats in die formule voorkomen, en hebben dan enkel betrekking op het geen erna komt wat tussen haakjes staat. Dit deel wordt het bereik van de quantifier genoemd.

Als alle quantifiers vooraan in de formule staan zeggen we dat de de formule in prenex normaal vorm is. Zonder bewijs wordt er vermeld dat elke formule naar een equivalente formule in prenex normaal vorm kan omgezet worden, en we beschouwen enkel deze formules.

Booleaanse formules met quantifiers noemen we gequantificiëerde booleaanse formules. Als elke variable die in de formule voorkomt in het bereik zit van een quantifier, zeggen we dat de formule volledig gequantificiëerd is. Deze worden soms ook zinnen genoemd en worden altijd tot WAAR of VALS geëvalueerd.

We zeggen dat een variable x gebonden is, als hij onmiddelijk na een quantifier komt.

Nu zijn we to het punt gekomen dat we het probleem kunnen definiëren.

$$TQBF = \{\langle\phi\rangle | \phi \text{ is een volledig gequantificiëerde booleaanse formule die tot WAAR evalueert}\}$$

Stelling: TQBF is PSPACE-compleet.

Bewijs: Eerst moet er aangetoond worden dat $TQBF \in PSPACE$.

T = Op input $\langle\phi\rangle$

1.Als ϕ geen quantifier vanvoor heeft staan, dan bestaat hij enkel uit constanten. Dus evalueer dit stuk en *aanvaard* als hij tot WAAR evalueert en als hij tot VALS evalueert, *aanvaard niet*.

2.Als ϕ gelijk is aan $\forall x\ \psi$, roep dan recursief T op eerst door x te vervangen door 0 in ψ, en daarna x te vervangen door 1. *Aanvaard* enkel als beiden tot WAAR evalueren. In het andere geval, *aanvaard niet*.

3.Als ϕ gelijk is aan $\exists x\ \psi$, roep dan T recersief op met ψ door x eerst door 0 te vervangen en daarna door 1 te vervangen. *Aanvaard* zodra één van beiden tot WAAR evalueert. Als beiden tot VALS evalueren, *aanvaard niet*.

Het is duidelijk dat $TQBF$ door T wordt beslist. Om de space complexiteit te berekenen, dat de diepte van de recursie maximaal gelijk aan het aantal variabelen kan zijn. In iedere aanroep moeten we enkel de waarde van één variabele bijhouden. Dus er wordt $O(m)$ space gebruikt, met m gelijk aan het aantal variabelen die in de formule ϕ voorkomen. Daarom werkt T zelfs in lineaire space.

Nu moeten we nog aantonen dat $TQBF$ PSPACE-hard is. Neem daarvoor een willekeurige taal A die beslist wordt door een TM M die in n^k werkt, voor een bepaalde constante k.

We moeten een string w, een input voor M, mappen op een gequantificiëerde booleaanse formule ϕ die tot waar evalueert als en slechts als $w \in A$. Om aan te tonen hoe ϕ wordt geconstrueerd, lossen we een iets algemener probleem op. Gebruik makend van twee variabelen c_1 en c_2 en een getal $t > 0$, construeren we een formule $\phi_{c_1,c_2c,t}$. Als c_1 en c_2 twee echte configuraties van M voorstellen, willen we dat de formule ϕ_{c_1,c_2ct} WAAR is als en slecht als M van configuratie c_1 in configuratie c_2 kan geraken in maximaal t tijdstappen. Als we dit bereikt hebben, kunnen we voor ϕ de formule $\phi_{c_start,c_accept,h}$ nemen, met $h = 2^{df(n)}$ voor een constante d, die zo gekozen is dat M niet meer dan $2^d f(n)$ mogelijke configuraties heeft op een input van lengte n.

Voor elke positie op de tape zijn er meerdere variablen geassocieerd, één voor ieder tape symbool en toestand. Iedere configuratie heeft n^k posities, dus wordt gecodeerd door $O(n^k)$ variabelen.

Als $t = 1$, maken we een formule die zegt dat ofwel $c_1 = c_2$, ofwel dat c_1 in één stap in c_2 kan geraken. De gelijkheid kunnen we als een booleaanse formule formuleren door alle variabelen die c_1 voorstellen, dezelfde booleaanse waarde moet hebben als de overeenkomstige variabelen, die c_2 voorstellen. de tweede mogelijkheid kunnen we formuleren door er voor te zorgen dat de inhoud van ieder drie-tuppel van opeenvolgende posities in c_1 ervoor moet zorgen dat de inhoud van het overeenkomstige drie-tuppel in c_2 overeenkomstig is met de transitie functie.

Als $t > 1$ kunnen we $\phi_{c_1,C_2,t}$ recursief construeren:

$$\phi_{c_1,C_2,t} = \exists m_1 \forall (c_3, c_4) \in \{(c_1, m_1), (m_1, c_2)\} \left[\phi_{c_3,c_4,\frac{t}{2}}\right].$$

$\exists m_1$ is een afkorting voor $\exists x1, \ldots, x_l$, met $l = O(n^k)$ en $x_1, \ldots, x_l$ zijn de variabelen die m_1 encoderen. Het stukje $\forall (c_3, c_4) \in \{(c_1, m_1), (m_1, c_2)\} \left[\phi_{c_3,c_4,\frac{t}{2}}\right]$ wilt hetzelfde zeggen als $\phi_{c_1,m_1,\frac{t}{2}} \wedge \phi_{m_1,c_2,\frac{t}{2}}$. Maar deze laatste zou te groot worden. Elk level van de recursie halveert t, maar verdubbelt de grootte van de formule. Dus we zouden een exponentiëel grote formule krijgen.

We gaan nu de grootte van de formule $\phi_{q_aanvaard,q_verwerp,h}$ berekenen met $h = 2^{df(n)}$. We merken op dat elke recursie stap een deel toevoegd dat lineair is in de grootte van de configuraties, dus van de grootte $O(f(n))$. De diepte van de recursie is $log(2^{df(n)}) = O(f(n))$. Dus de resulterende grootte is gelijk aan $O(f^2(n))$.

5.2 Andere voorbeelden van PSPACE-complete problemen

In deze sectie ga ik nog 2 voorbeelden geven van PASPACE-complete problemen, het formule spel en het gegeneraliseerd geografie spel. Deze problemen worden op hun beurt gedefinieerd. Voor het formule spel wordt er een reductie van TQBF naar formule spel besproken, zodat we dit kunnen gebruiken om een reductie van het formule spel naar gegeneraliseerd geografie kunnen geven. hierdoor wordt aangetoond dat ze beiden PSPACE-compleet zijn.

Het formule spel

Om het formule spel te spelen zijn er twee spelers, speler E en speler V, nodig. Stel dat $\phi = \exists x_1 \forall x_2, \exists x_3 \ldots Q x_k[\psi]$, waarbij Q ofwel $\exists$ ofwel $\forall$ voorstelt, een gequantificiëerde booleaanse formule in prenex vorm is. Er wordt een spel met ϕ geassocieerd op de volgende manier. De twee spelers krijgen om de beurt de kans om een waarde te kiezen voor de variablen $x1, \ldots, xk$. Speler A kiest een waardes voor de variabelen die gebonden zijn aan de $\forall$ quantifiers en speler E kiest waarden voor de variabelen die gebonden zijn de $\exists$ quantifiers. Om de volgorde van de spelers te bepalen, loopt men de formule af, en als men een $\forall$ tegenkomt is het de beurt aan speler V, die dan een waarde voor de variabele die aan die $\forall$ gebonden is mag kiezen, als men een $\exists$ tegenkomt, is het de beurt aan speler E om een waarde voor de variabele, die aan die $\exists$ gebonden is, te kiezen. Op het einde van het spel gebruiken we de waarden van de gekozen variabelen om ψ te evalueren. Speler E wint als ψ tot WAAR evalueert, Speler V wint als ψ tot VALS evalueert.

Om meer duidelijkheid te scheppen zal ik er ook een voorbeeld bij geven: neem de formule

$$\phi_1 = \exists x_1 \forall x_2 \exists x_3 \left[(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})\right].$$

In het formule spel voor ϕ_1 kiest eerst speler E een waarde voor x_1, vervolgens mag speler V een waarde voor x_2 kiezen, waarna speler E als laatste zet een waarde voor x_3 mag kiezen.

Stel dat speler E eerst $x_1 = 1$ kiest, waarna speler V aan de beurt is, en $x_2 = 0$ kiest, uiteindelijk kiest speler E $x_3 = 1$. Met deze waarden voor x_1, x_2 en x_3 wordt de subformule

$$(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})$$

tot WAAR geëvalueerd, en dus heeft speler E gewonnen. Als speler E een beetje slim is, kan hij eigenlijk altijd winnen, eender welke zet speler V doet. Als speler E $x_1 = 1$ kiest en voor x_3 de negatie van wat speler V voor x_2 kiest, wint speler E altijd. Door $x_1 = 1$ te kiezen is de subformule

$$(x_1 \vee x_2)$$

al zeker WAAR. Als speler V dan $x_2 = 0$ kiest, zal speler E $x_3 = 1$ kiezen waardoor

$$(x_2 \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})$$

ook WAAR wordt.

Als speler V $x_2 = 1$ had gekozen, zou speler E $x_3 = 0$ gekozen hebben. Ook hierdoor wordt

$$(x_2 \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})$$

tot WAAR geëvalueerd.

We zeggen dat speler E een winnende strategie heeft voor dit spel. Een speler heeft een winnende strategie voor een spel als die speler wint wanneer beiden spelers het spel optimaal spelen.

We kunnen de formule ϕ_1 een klein beetje veranderen zodat speler V een winnende strategie heeft. Stel

$$\phi_2 = \exists x_1 \forall x_2 \exists x_3 \left[(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3})\right]$$

Speler V heeft nu een winnende strategie. Namelijk, door $x_2 = 0$ te kiezen evalueert het deel

$$(x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3})$$

altijd tot VALS. Wat er ook voor x_3 wordt gekozen, ofwel is $x_3 = 0$ ofwel is $\overline{x_3} = 0$. Samen met het feit dat $x_2 = 0$, moet één van de twee subformules

$$(x_2 \vee x_3)$$

of

$$(x_2 \vee \overline{x_3})$$

tot VALS evalueren.

We zijn nu klaar om het formule spel te definiëren:

$$FORMULE - SPEL = \{\langle\phi\rangle | \text{ Speler E heeft een winnende strategie voor het spel geassocieerd met } \phi\}$$

Stelling: FORMULE-SPEL is PSPACE-compleet.

Intuïtief bewijs: Om in te zien dat FORMULE-SPEL PSPACE-compleet is, moet je enkel inzien dat het juist hetzelfde is als TQBF. Als een formule WAAR is, dan heeft speler E een winnende strategie in het geassocieerde formule spel.

Bewijs: De formule $\phi = \exists x_1 \forall x_2 \exists x_3 \ldots [\psi]$ is WAAR, als er een bepaalde waarde voor x_1 bestaat zodat voor elke waarde voor x_2, er een waarde voor x_3 bestaat,..., zodat de formule ψ, met die waardes voor $x_1, x_2, x_3, \ldots$ ingevuld, tot WAAR evalueert. Gelijkaardig heeft speler E een winnende strategie als hij een waarde voor x_1 vindt, zodat voor eender welke waarde speler V voor x_2 kiest, speler E weer een waarde voor x_3 kan kiezen,..., zodat met deze waardes de formule ψ WAAR is.

Dezelfde redenering geldt ook wanneer de formule niet alterneert tussen de $\exists$ en $\forall$ quantifiers. Als ϕ van de vorm

$$\forall x_1, x_2, \ldots, x_k \exists x_{k+1}, x_{k+2}, \ldots, x_l \forall x_{l+1}, x_{l+2}, \ldots, x_m, \ldots [\psi]$$

is, kan speler V de eerste k zetten doen, speler E de volgende $k-l$ zetten, dan weer speler V $m-l$ zetten,..., waarbij dezelfde redenering geldt als hierboven.

Uit deze redenering volgt onmiddelijk dat

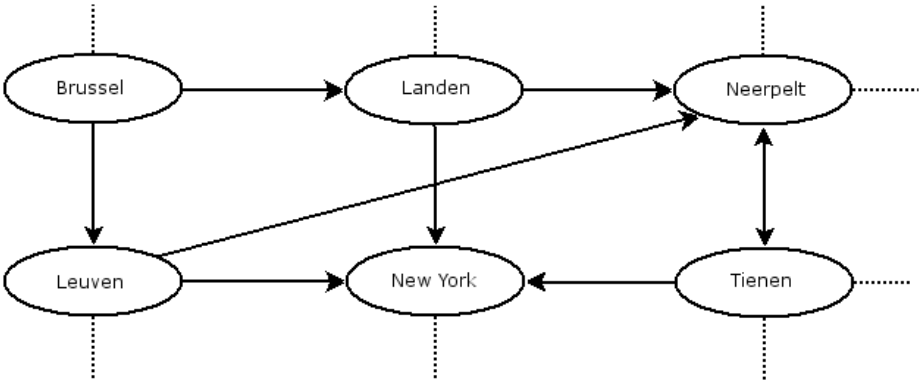
$$\phi \in TQBF \Leftrightarrow \phi \in FORMULE - SPEL$$

Het gegeneraliseerd geografie spel

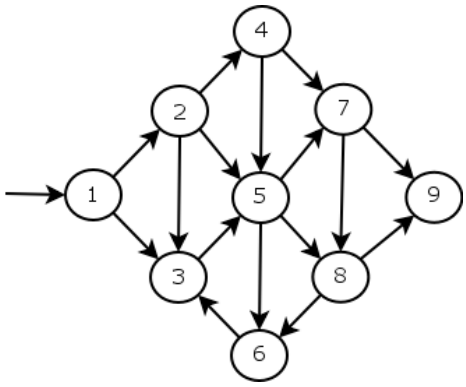
Geografie is een spel dat veel gespeeld wordt door kinderen. De spelers moeten om de beurt een stad uit eender waar in de wereld opnoemen. De stad die vermeld wordt moet met de letter beginnen waarmee de vorig genoemde stad eindigde. Als bijvoorbeeld de eerst genoemde stad Brussel is dan is Landen een mogelijke kandidaat als volgende stad. New York kan in dat geval als derde gezegd worden, vervolgens is Knokke een mogelijkheid, enzovoort totdat er een speler aan de beurt is die geen stad meer weet te bedenken, en daardoor verliest.

Dit spel kan gemodelleerd worden met een gerichte graaf waarvan de knopen steden van de wereld zijn. Er bestaat een pijl van de ene stad naar een andere, als de ene lijdt tot de andere volgens de regels van het spel. Dus de graaf bevat een boog van stad X naar stad Y als de laatste letter van X dezelfde letter is als de eerste van Y. Een klein deel van deze geografie graaf is in figuur 5.1 opgenomen.

De regels van het geografie spel kunnen een klein beetje aangepast worden zodat ze gelden voor eender welke gerichte graaf. De eerste speler kiest een begin-knoop en vervolgens mogen de spelers om de beurt een knoop kiezen zodat ze een simpel pad vormen in de graaf. Een simpel pad houdt in dat een gekozen knoop x een opvolger moet zijn van de laatste daarvoor gekozen knoop y (er moet een boog van de y knoop naar de x knoop zijn), en een knoop mag niet herhaald worden. De eerste speler die het pad niet kan uitbreiden is verloren. Als we nu nog toelaten dat eender welke gerichte graaf met aangeduide startknoop een speelbare graaf mag zijn, komen we tot de definitie van de spelregels van het gegeneraliseerd geografie spel. In figuur 5.2 zien we een voorbeeld van zo een gegeneraliseerde geografie graaf.



Figuur 5.1: Een deel van de geografie graaf



Figuur 5.2: Een voorbeeld van een gegeneraliseerde geografie graaf

Als het spel in knoop 1 begint en speler 1 van daaruit moet vertrekken, dan heeft hij een winnende strategie. Hij heeft de keuze om knoop 2 of knoop 3 te kiezen. Om zijn winnende strategie uit te buiten moet hij knoop 3 kiezen. Vanuit knoop 3 gaat er enkel een boog naar knoop 5, dus speler 2 moet wel knoop 5 kiezen. Nu zal speler 1 knoop 6 kiezen. Vanuit knoop 6 is er enkel een boog naar knoop 3. Maar knoop 3 is al bezocht, dus speler 2 heeft geen mogelijkheid meer om het pad uit te breiden. Beslissen welke speler een winnende strategie heeft, in het gegeneraliseerd geografie spel, is PSPACE-compleet. Stel

$$GG = \{\langle G, b\rangle | \text{ Speler 1 heeft een winnende strategie voor het gegeneraliseerd geografie spel op graaf } G \text{ startend in knoop } b\}$$

Stelling: GG is PSPACE-compleet.

Bewijs: Eerst en vooral moet er aangetoond worden dat $GG \in PSPACE$. Hiervoor bestaat het volgende algoritme:

M =Op input $\langle G, b\rangle$:
1. Als b geen bogen heeft die er vertrekken, *aanvaard niet*.
2. Verwijder knoop b en alle aanrakende bogen om een nieuwe graaf G te krijgen.
3. Voor elke knoop b_i waar er origineel een boog van b uit naartoe bestond, roep M recursief op met $\langle M, b_i\rangle$.
4. Als al deze gevallen aanvaarden, *aanvaard niet*, anders *aanvardt* wel.

In iedere recursie stap moet enkel één knoop op de stack toegevoegd worden. Vermits er maximaal m recursiestappen zijn, met m het aantal knopen, loopt het algoritme in lineaire SPACE.

we moeten nog aantonen dat GG ook PSPACE-hard is. Om dit in te zien wordt hier een polynomiale tijd reductie van $TQBF$ naar GG besproken. Deze reductie beeldt een formule

$$\phi = \exists x_1 \forall x_2 \exists x_3 \ldots Q x_k[\psi]$$

af op een instantie $\langle G, b\rangle$ van het gegeneraliseerd geografie spel, zodat

$$\phi \in TQBF \Leftrightarrow \langle G, b\rangle \in GG$$

We gaan zonder beperking van de algemeenheid ervan uit dat de quantiefiers van ϕ beginnen en eindigen met $\exists$ en strikt alterneren tussen $\exists$ en $\forall$. Dit is geen beperking van de algemeenheid omdat elke formule in prenex normaal vorm kan gezet worden en we quantifiers met ongebruikte variabelen kunnen toevoegen totdat de formule aan de eisen voldoet. We gaan er ook van uit dat ψ in cunjunctieve normaal vorm[1] is.

Speler 1 uit het geografie spel neemt de rol van speler E, uit het formule spel en speler 2 neemt de rol van Speler V.

We zullen de reductie in twee stappen doen. De eerste stap construeerst de graaf G_{deel} die we in figuur 5.3 zien. De startknoop is b. Onder b komt een sequentie van diamant structuren, ene voor elke variabele x_i van ϕ.

Het spel begint in b. Speler 1 moet één van de twee bogen die in b vertrekken volgen. Het kiezen van de linkse pijl komt overeen met de waarde WAAR te kiezen in het formule spel. Nadat speler 1 een richting heeft gekozen, is het de beurt aan speler 2, maar zijn keuze is geforceerd, want er vertrekt maar één boog vanuit de knoop gekozen door speler 1. Ook hierna is er voor speler 1 maar één mogelijkheid. Vervolgens zijn er terug twee mogelijkheden, maar nu komt de keuze bij speler 2 te liggen. Deze keuze komt overeen met de eerste

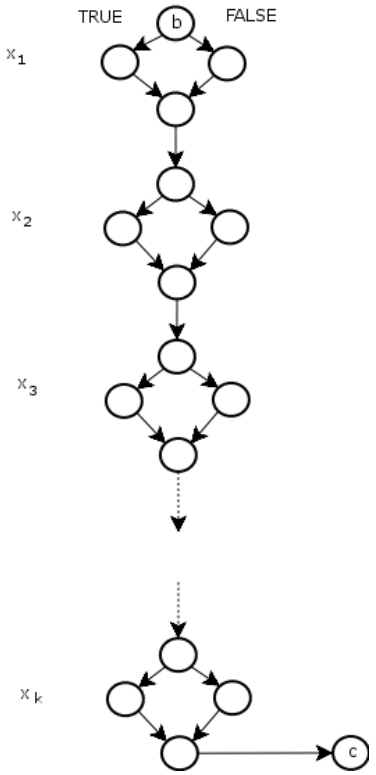
[1] Conjunctieve normaal vorm (cnv) is al reeds besproken in mijn opleiding. Een booleaanse formule is in cnv als het een conjuntie is van clausules. Een clausule is een disjunctie van literalen, en een literaal is een variabele of de negatie van een variabele. Een disjunctie is een aaneenschakeling door '$\wedge$' en een conjuntie een aaneenschakeling door '$\vee$'. We hebben ook geleerd dat elke booleaanse formule naar een equivalente formule in cnv kan omgezet worden

zet van speler V in het formule spel. Het spel gaat verder doordat speler 1 en speler 2 afwisselend voor het linker of het rechter pad kiezen door elke diamant.

Als het spel zo ver geraakt is dat de spelers door alle diamanten zijn gegaan, is het hoofd van het pad in knoop c geraakt, en de beurt is aan speler 2 (dit komt omdat de laatste quantifier een $\exists$ is).

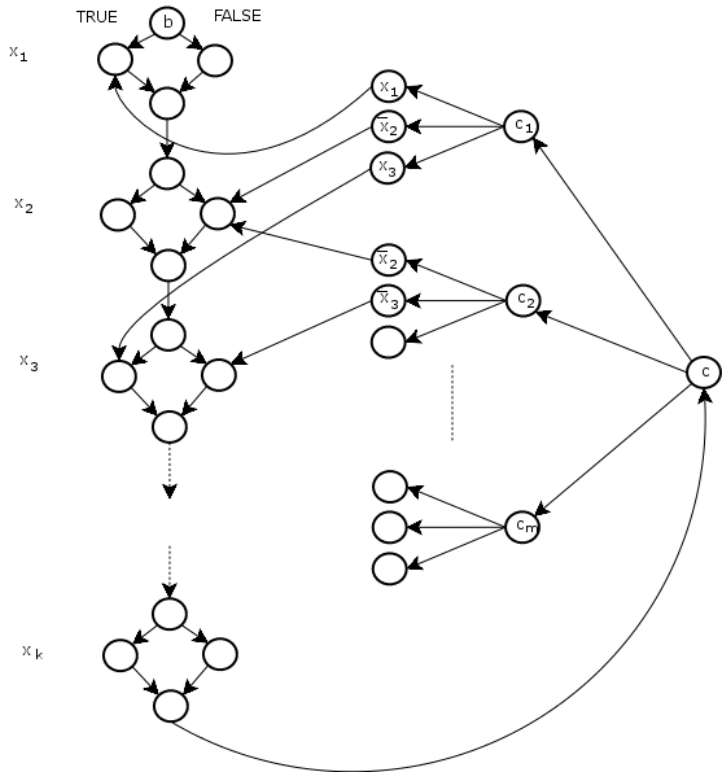
Dit punt correspondeert met het einde van het formule spel. Het gekozen pad stemt overeen met een invulling van de variabelen in ψ. Als ψ hierdoor tot WAAR evalueert, wint speler E, in het andere geval wint speler V.

De structuur van de uitbreiding van de graaf G_{deel}, getoond in figuur 5.4, zorgt ervoor dat wanneer speler E wint, speler 1 een winnende strategie heeft voor de rest van het spel in de graaf en dat wanneer speler V wint, speler 2 zeker kan winnen.



Figuur 5.3: Deel van het geografie spel dat het formule spel simuleert

In knoop c mag speler 2 een knoop kiezen die overeenkomt met een van de clausules van ψ. Deze clauseleknopen worden geconnecteerd met alle literalen die in die clausule voorkomen. De knopen die overeenkomen met een negatie van een variable, worden geconnecteerd met de rechterkant (VALS-kant) van de diamand, die met dezelfde variable overeenkomt. De gewone variabelen worden



Figuur 5.4: Volledige graaf van het geografie spel dat het formule spel simuleert, waarbij $\psi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3} \vee \ldots) \wedge \ldots \wedge (\ldots)$

geconnecteerd met de linkerkant (WAAR-kant) van de diamand die met dezelfde variable overeenkomt.

Als ϕ VALS is, kan speler 2 op de volgende manier winnen. Hij kiest eerst een clausule die VALS is. Zo een clausule moet bestaan omdat ϕ VALS is en ϕ uit conjuncties van clausules bestaat. Daarom moet er minstens één clausule zijn die tot VALS evalueert. Iedere literaal die speler 1 dan kan kiezen, is VALS. Speler 2 kan dan de knoop van de diamand kiezen, en het spel loopt nadien vast, want de volgende knoop is al eerder bezocht.

Als ϕ WAAR is bevat elke clausule een literaal die WAAR is. Als speler 1 deze literaal kiest, verliest speler 2 want de door speler 1 gekozen literaal is geconnecteerd met de kant van de diamant die al reeds gespeeld is.

We kunnen hieruit besluiten dat er geen algoritme, dat in polynomiale tijd werkt, bestaat om optimaal te spelen, tenzij $P = PSPACE$.

Om tot hetzelfde besluit te komen voor spellen zoals schaken en dammen,

komt er een obstakel te voorschijn. Op deze spelborden bestaan er maar een eindig aantal spelposities, omdat er een standaard formaat bestaat van deze borden. Men zou al deze spelpopsities in een tabel kunnen opslaan samen met de beste zet die erop kan volgen. Deze tabel zou wel zo groot zijn dat ze niet in onze galaxy zou passen, maar omdat ze eindig is, kan ze in bewaard worden in een Turing Machine. Deze machine zou dan in lineaire tijd de beste zet kunnen opzoeken in die tabel.

We zullen er niet dieper op ingaan maar er is aangetoond dat het berekenen van optimaal spelen voor dergelijke spellen, gegeneraliseerd tot $n \times n$ spelborden, PSPACE-hard is.

5.3 Voorbeelden en oefeningen

Stelling: PSACE is gesloten onder de unie operatie.

Bewijs: Als we twee talen $A, B \in$ PSPACE hebben, weten we ook dat ze beslist worden door TM T_1 en T_2 respectievelijk, die in polynomiale space werken. We moeten aantonen dat ook $A \cup B$ in PSPACE zit. We doen dit door een polynomiaal space algoritme ervoor te geven:

T_3 = Op input w:
1. Laat T_1 op w lopen.
2. Als T_1 aanvaardt, *aanvaard*.
3. Als T_1 niet aanvaardt, simuleer T_2 op w.

T_3 werkt duidelijk ook in polynomiale space, want T_1 doet dat en dat geheugen kunnen we opnieuw gebruiken om T_2 te simuleren, en ook deze werkt in polynomiale space.

Stelling: PSPACE is gesloten onder intersectie.

Bewijs: Als we twee talen $A, B \in$ PSPACE hebben, dan weten we ook dat ze beslist worden door TM T_1 en T_2 respectievelijk, die in polynomiale space werken. We moeten aantonen dat ook A/B in PSPACE zit. We doen dit door een polynomiaal space algoritme ervoor te geven:

T_3 = Op input w:
1. Laat T_1 op w lopen.
2. Als T_1 niet aanvaardt, *aanvaard niet*.
3. Als T_1 aanvaardt, simuleer T_2 op w.

T_3 werkt duidelijk ook in polynomiale space, want T_1 doet dat en dat geheugen kunnen we opnieuw gebruiken om T_2 te simuleren, dat ook in polynomiale space werkt.

Stelling: PSPACE is gesloten onder de star operatie.

Bewijs: We hebben een taal $A \in$ PSPACE en zijn beslisser M gegeven. Om te beslissen of een iput $w \in A^*$ doen we het volgende. We gaan alle mogelijke kappingen van w af. Met een kapping bedoelen we een aantal aanduidingen van verschillende posities van w. Voor elke stukje, dat zich tussen twee aanduidingen bevindt, gaan we na of dat stukje tot A behoort daar er M op te laten lopen. Indien elk stukje ertoe behoort, aanvaarden we. Indien voor geen enkele kapping elk stukje tot A behoort, aanvaarden we niet.

We hebben telkens maar geheugen nodig om één stukje te beslissen, want zodra het niet in A zit, weten we dat we naar de volgende fase moeten. Deze beslissing kan in polynomiale space gebeuren, dus het geheel ook.

Stelling: Elke PSPACE-harde taal is ook NP-hard.

Bewijs: Als A PSPACE-hard is weten we:

$$\forall B \in \text{PSPACE} : B \leq_P A.$$

Maar we weten ook dat NP $\subseteq$ PSPACE, dus:

$$\forall C \in \text{NP} : C \in \text{PSPACE}.$$

Dus $C \leq_P A \Rightarrow A$ is NP-hard.

Hoofdstuk 6

Ontraceerbaarheid

Er bestaan verschillende problemen waarvan de beste oplossingen nog altijd zo veel tijd of geheugen gebruiken dat ze niet gebruikt kunnen worden in de praktijk. Deze problemen noem ik in het nederlands ontraceerbaar, vermits er geen perfecte vertaling bestaat voor het Engelse 'intractable'.

De meeste mensen zijn ervan overtuigd dat alle NP-complete problemen ontraceerbaar zijn, maar dit is nooit bewezen kunnen worden. In dit hoofdstuk ga ik een aantal problemen bespreken waarvan we kunnen bewijzen dat ze ontraceerbaar zijn.

6.1 Hiërarchie stellingen

Het gezonde verstand vertelt ons dat als we een Turing machine meer tijd en geheugen gunnen, deze ook meer problemen kan oplossen. Anders gezegd, dat Turing machines meer problemen kan oplossen in n^3 tijd als in n^2 tijd. Deze redenering is correct en we zullen dit formaliseren en bewijzen in de rest van deze sectie.

Definitie: Een functie $f : N \to N$, met $f(n) \geq log(n)$, noemt men space constructible als de functie die 1^n afbeeldt op $f(n)$ berekenbaar is in $O(f(n))$ space[1].

Space hiërarchie stelling: Voor elke space constructible $f : N \to N$, bestaat er een taal A die beslisbaar is in $O(f(n))$, maar niet in $o(f(n))$.

Intuïtief bewijs: We kunnen A beschrijven met een algoritme B dat A beslist. B zal in $O(f(n))$ space werken, maar garanderen dat A verschillend is van elke taal die in $o(f(n))$ space beslist kan worden. B neemt als input een Turing Machine M. B simuleert vervolgens M op input M, in de space beperking van $f(n)$. Als M stopt binnen deze beperking, dan aanvaardt B als M niet aanvaardt. Als M niet stopt, aanvaardt B niet. Dus als M

[1] Met 1^n bedoelen we de string die uit de opeenvolging van n 1'tjes bestaat.

in $f(n)$ space loopt, heeft B genoeg geheugen om te verzekeren dat zijn taal verschillend is dan die van M. Als M meer geheugen nodig heeft, dan heeft B niet genoeg geheugen om uit te zoeken wat M zou doen. Gelukkig genoeg moet B ook helemaal niet verschillen van machines die in niet in $o(f(n))$ lopen.

Er komen wel nog een paar technische problemen kijken bij deze intuïtie, maar die worden in het echte bewijs wel opgelost.

Bewijs: Het volgende algoritme B bepaalt een taal A in $O(f(n))$ space, die niet in $o(f(n)$ kan beslist worden:

B = Op input w:
1. Bereken n, de lengte van w.
2. Bereken $f(n)$, en markeer zoveel tape af. Als er in de berekening ooit meer tape wordt gebruikt , *aanvaard niet.*
3. Als w niet van de vorm $\langle\rangle 10^*$, *aanvaard niet.*
4. Simuleer M op w en tel het aantal stappen. Als dit aantal ooit $2^{f(n)}$ overschrijdt, *accepteer niet.*
5. Als M aanvaardt, *aanvaard niet.* Als M niet aanvaardt, *aanvaard.*

Omdat het mogelijk is dat M in een oneindige loop terechtkomt door maar $f(n)$ space te gebruiken, hebben we in stap 4 ervoor gezorgd dat er maar $2^{f(n)}$ tijd gebruikt mag worden. Een machine dat in $o(f(n))$ space loopt, gebruikt immers maar $2^{o(f(n))}$ tijd.

Een ander probleem is dat als M in $o(f(n))$ spake werkt, het misschien wel meer dan $f(n)$ space gebruikt, voor kleine n (als het assymptotisch gedrag zich nog niet vertoont). Dus als we M gewoon op M simuleren, zou B misschien niet genoeg geheugen hebben om M tot een einde te brengen, terwijl M toch in $o(f(n))$ space werkt. Dit probleem is opgelost door ervoor te zorgen dat de input van de vorm $\langle M\rangle 10^*$. Dus als M in $o(f(n))$ space werkt, zal B genoeg geheugen hebben om M tot een einde te laten lopen op input $\langle M\rangle 10^k$, als k groot genoeg is om te zorgen dat het assymptotisch gedrag zich al vertoont.

Het is duidelijk dat A kan beslist worden in $O(f(n))$. Om aan te tonen dat A niet beslisbaar is in $o(f(n))$, veronderstellen we uit het ongerijmde dat er een Turing Machine M bestaat die A beslist in $g(n)$ space, met $g(n) = o(f(n))$. Dan bestaat er een n_0 waarvoor $g(n) < f(n)$, voor alle $n > n_0$. Neem dan nu $\langle M\rangle 10^{n_0}$ als input voor B. Deze input is langer als n_0, dus de simulatie van M in algoritme =b, zal tot zijn einde lopen. Maar B zal het omgekeerde van M antwoorden op dezelfde input. Daarom kan M geen beslisser zijn voor A. Waaruit volgt dat A niet beslisbaar is in $o(f(n))$.

Gevolg: Als $f_1, f_2 : N \to N$ twee functies zijn met $f_1(n) = o(f_2(n))$ en f_2 is space constructable, dan is $\text{SPACE}(f_1(n)) \subset \text{SPACE}(f_2(n))$[2].

Gevolg: Voor elke twee reële getalllen $0 \leq r_1 < r_2$ geldt: $\text{SPACE}(n^{r_1}) \subset \text{SPACE}(n^{r_2})$.

[2]met de expressie $A \subset B$ bedoel ik dat A een deelverzameling is van B en dat $A \neq B$

Bewijs: n_c is space constructible voor elk rationeel getal c. Dus Voor alle rationele getallen $0 \leq c_1 < c_2$ geldt: SPACE(n^{r_1}) $\subset$ SPACE(n^{r_2}). Uit de observatie dat elk paar rationele getallen c_1 en c_2 zich tussen twee reële gatellen bevinden, volgt dadelijk het gevraagde.

Gevolg: NL $\subset$ PSPACE

Bewijs: Savitch stelling stelt dat NL $\subset$ SPACE($log^2(n)$), en de space hiërarchie stelling zegt ons dat SPACE($log^2(n)$) $\subset$ SPACE(n). Hieruit volgt het bewijs.

Nu zijn we klaar om de hoofdzaak van dit hoofdstuk te verwezenlijken.

Gevolg: PSPACE $\subset$ EXPSPACE $= \bigcup_k$ SPACE$\left(2^{n^k}\right)$.

Bewijs: SPACE(n^k) $\subset$ SPACE$\left(n^{log(n)}\right)$ $\subset$ SPACE(2^n).

Dit gevolg toont aan dat er beslisbare problemen bestaan die ontraceerbaar zijn, in de zin dat hun beslissingsprocedures meer dan polynomiaal veel geheugen gebruiken. Voordat ik een voorbeeld van zo een ontraceerbaar probleem ga bespreken, gaan we nu de tijd hiërarchie stelling bekijken.

Definitie: Een functie $t : N \rightarrow N$, met $t(n) \geq nlog(n)$, heet time constructable als de functie die 1^n op de binaire representatie van $t(n)$ afbeeldt, berekenbaar is in $O(t(n))$ tijd.

tijd hiërarchie stelling: Voor elke time constructable functie $f : N \rightarrow N$, bestaat er een taal A die beslisbaar is in $O(t(n))$ tijd, maar niet in $o(t(n)/log(t(n)))$ tijd.

Intuitief bewijs: Zoals bij de space hiërarchie stelling zullen we hier ook een machine B construeren die een taal A aanvaardt in $O(t(n))$ tijd, maar die niet kan beslist worden in $O(t(n))$ tijd. B neemt een input w van de vorm $\langle M \rangle 10^*$ en simuleert M op w. B doet dit terwijl het ervoor zorgt dat er niet meer dan $t(n)$ tijd wordt gebruikt. Als M zijn berekening stopt binnen die tijd, zal B het tegengestelde doen aan wat M doet.

Bewijs: We construeren een Turing machine B die in $O(t(n))$ tijd loopt. B beslist een taal A die niet in $o(t(n))$ tijd beslist kan worden.

B =Op input w:

1. Stel n gelijk aand e lengte van w.

2. Bereken $t(n)$ en bewaar de waarde $v = t(n)/log(t(n))$ in eenbinaire teller. Verminder deze teller met 1 voor elke stap die wordt uitgevoerd in puntjes 3, 4 en 5. Als deze teller ooit 0 wordt, *aanvaard niet*.

3. Als w niet van de vorm $\langle M \rangle 10^*$ is, *aanvaard niet*.

4. Simuleer M op w.

5. Als M aanvaardt, *aanvaard niet*, anders *aanvaard*.

Het is duidelijk dat de eerste drie puntjes al zeker in $O(t(n))$ tijd berekend kunnen worden. We zullen zonder uitgebreide uitleg aannemen dat als M in $g(n)$ tijd werkt, B dit in $O(g(n))$ tijd kan simuleren. Het komt erop neer dat B voor elke simulatiestap de toestand, tape symbool en transitiefunctie moet bewaren op de tape. Als deze op een verfijnde manier worden bewaard kan de simulatie gebeuren door enkel een constante overhead. Dit gebeurt door de enkele tape te verdelen in tracks. Men kan bijvoorbeeld twee tracks op een enkele tape krijgen door de eerste track op de even posities te bewaren en de tweede track op de oneven posities. We besluiten dat B dus in $O(t(n))$ tijd werkt.

We gaan weer op ongeveer dezelfde manier te werk als in de space hiërarchie stelling om aan te tonen dat A niet kan beslist worden door een Turing machinen dat in $o(t(n)\log(t(n)))$ tijd loopt, kan beslist worden. Veronderstel dus dat een Turing Machine M in $g(n)$ tijd A kan beslissen, met $g(n) = o(t(n)\log(t(n)))$. Er bestaat dus een constante $n_0 > 0$ zodat voor iedere $n > n_0$ geldt dat $g(n) < t(n)/log(t(n))$. Dus als de input voor B groter is als n_0, zal de simulatie van M eindigen. Als men nu $\langle M\rangle 10^{n_0}$ aan B geven als input, zal B het omgekeerde doen van M. Hierdoor kunnen M en B onmogelijk dezelfde taal beslissen. Dus A is niet beslisbaar in $o(t(n)/log(t(n)))$ tijd.

Gevolg: Voor elk paar functies $t_1, t_2 : N \to N$, met $t_1(n) = o(t_2(n))/log(t_2(n)))$ en t_2 time constructable geldt: TIME$(t_1(n) \subset$ TIME$(t_2(n))$.

Gevolg: Voor elk paar reële getallen $1 \leq r_1 < r_2$ geldt: TIME$n^{r_1} \subset$ TIME(n^{r_2}).

Gevolg: P $\subset$ EXPTIME

Exponentiële space compleetheid

We kunnen voorgaande stellingen en gevolgen gebruiken om aan te tonen dat er een bepaalde taal ontraceerbaar is. Eerst en vooral vertellen de hiërarchie stellingen ons dat EXPSPACE meer talen bevat als PSPACE. Als we dan van een taal, die met een generalisering van reguliere expressies te maken heeft, kunnen aantonen dat deze EXPSPACE-compleet is, dan weten we met zekerheid dat hij niet in polynomiale tijd of polynomiale space kan beslist worden.

Reguliere expressies werden ook al in mijn opleiding behandeld, maar ik ga er hier even een korte herhaling van geven. Reguliere expessies worden gebouwd uit de atomaire expressies $\emptyset, \epsilon$ en elementen van het alfabet, door ze te verbinden met de reguliere operaties unie, concatenatie en ster, respectievelijk genoteerd als $\cup, \circ$ en $*$. We weten dat de equivalentie tussen twee reguliere in polynomiale space kan getest worden.

Ik zal nu een uitbreiding van de operaties op reguliere expressies behandelen, waardoor de complexiteit van de analyse ervan dramatisch groeit. We bekijken de exponentiële operatie, genoteerd als $\uparrow$. Als R een reguliere expressie is en k een natuurlijk getal, dan is $R \uparrow k$ equivalent aan k keer de concatenatie van R met zichzelf. R^k is een afkorting voor $R \uparrow k$.

We zullen de reguliere expressies, die bovenop de standaard operaties ook de exponentiële operatie toelaten, gegeneraliseerde expressies noemen. Het is duidelijk dat deze dezelfde klasse van reguliere talen voortbrengen. De reden hiervoor is dat we de exponentiële operatie kunnen elimineren, door de expressie waarop hij van toepassing is, het juist aantal keer tye herhalen.

$$EQ_{REX\uparrow} = \{\langle Q, R\rangle | Q \text{ en } R \text{ zijn equivalente gegeneraliseerde reguliere expressies } \}$$

Definitie: Een taal B is EXPSPACE-compleet als het aan de volgende twee condities voldoet:

1. $B \in EXPSPACE$
2. $\forall A \in EXPSPACE : A$ is polynomiaal tijd reduceerbaar tot B

Om aan te tonen dat $EQ_{REX\uparrow}$ ontraceerbaar is, tonen we aan dat het compleet is voor de EXPSPACE klasse. Een EXPSPACE-compleet probleem kan niet in PSAPCE zitten, anders zouden EXPSPACE en PSPACE dezelfde klasse van talen bevatten, in contradictie met de space hiërarchie stelling.

Stelling: $EQ_{REX\uparrow}$ is EXPSPACE-compleet.

Intuïtief bewijs: We veronderstellen dat de exponenten in het binair zijn geschreven. De lengte van een expressie is dan gelijk aan het totaal aantal symbolen dat die expressie bevat.

Om te testen of twee expressies equivalent zijn, elimineren we eerst de exponenten met behulp van repetitie. Vervolgens converteren we de bekomen reguliere expressies naar ENDA's[3]. Als laatste moeten we dan nog een procedure toepassen die nagaat of twee ENDA's equivalent zijn.

Om aan te tonen dat eender welke taal $A \in$ EXPSPACE polynomiaal tijd reduceerbaar is tot $EQ_{REX\uparrow}$ gebruiken we computatie histories, besproken in sectie 3.2. Als we een TM M voor A hebben, moeten we een input w voor M in polynomiale tijd kunnen mappen op twee expressies, Q en R die equivalent zijn als en slechts als w aanvaardt wordt door M. Expressie Q genereert alle strings over het alfabet bestaande uit de symbolen die kunnen voorkomen in computatie histories. We zorgen ervoor dat R alle strings, behalve de niet aanvaardende computatie histories, genereert. Dus als M aanvaardt op w dan bestaat er geen niet aanvaardende computatie historie, waardoor Q en R dezelfde taal zullen genereren.

Bewijs: Zoals in het intuïtief bewijs vermeld, moeten we kunnen nagaan of twee ENDA's equivalent zijn. We zullen hiertoe eerst een niet deterministische Turing Machine N bespreken die test of twee ENDA's niet equivalent zijn, en dit in lineaire space. Hierdoor weten we dat de deterministische versie in kwadratische space werkt.

[3]Een ENDA is een eindig niet-deterministische automaat. Automatenleer was het eerste onderwerp dat ik gestudeerd heb binnen de theoretische informatica tijdens mijn opleiding. De klasse van talen bepaald door ENDA's zijn juist die klasse van talen die reguliere expressies kunnen voortbrengen.

N = Op input $\langle N_1, N_2 \rangle$:

1. Plaats een marker op elk van de begintoestanden van N_1 en N_2.

2. Herhaal het volgende $2^{q_1+q_2}$ keer, met q_1 en q_2 het aantal toestanden van N_1 en N_2 respectievelijk.

3. Kies, niet deterministisch, een input symbool en plaats de markers op de toestanden van N_1 en N_2 waar men in terecht kan komen door dat symbool te lezen.

4. Als er op eender welk moment een marker geplaatst wordt op een eindtoestand van een automaat en niet dezelfder tijd een marker op een eindtoestand van de andere automaat, *aanvaard*. Anders *aanvaard niet*.

De correctheid is niet moeilijk na te gaan. Als N_1 en N_2 equivalent zijn, zal N niet aanvaarden, want het aanvaardt enkel als de ene automaat een string aanvaardt die de andere automaat niet aanvaardt. Als N_1 en N_2 niet equivalent zijn, dan bestaat er een string die door de ene automaat aanvaardt wordt en door de andere niet, en er bestaat zelfs altijd zo een string die maximaal lengte $2^{q_1+q_2}$ heeft. Stel dat er een langere string is. Er bestaan maar $2^{q_1+q_2}$ verschillende mogelijkheden om de markers te zetten. Dus voor een langere string moet er een deel herhaalt worden. Door al deze herhalende delen weg te laten bekomt men een string die maximaal $2^{q_1+q_2}$ groot is. Deze string zal door ooit wel eens geraden worden door N, dus N werkt correct.

We zijn nu klaar om het algoritme E te beschrijven, dat $EQ_{REX\uparrow}$ beslist in exponentiële tijd.

E = Op input $\langle R_1, R_2 \rangle$:

1. Converteer R_1 en R_2 naar equivalente reguliere expressies Q_1 en Q_2 die herhaling gebruiken in plaats van exponenten.

2. Converteer R_1 en R_2 naar equivalente ENDA's, N_1 en N_2.[4]

3. Laat de deterministische versie van N op input $\langle N_1, N_2 \rangle$ lopen.

We moeten nu wel nog de space complexiteit van dit algoritme analyseren. Herhaling gebruiken in plaats van exponenten kan de lengte van de expressie doen toenemen met een faktor van 2^l met l gelijk aan de som van alle voorkomende exponenten. Dus de expressies Q_1 en Q_2 hebben maximaal een lengte van $n2^n$. De conversie in puntje 2 vergroot de grootte lineair, waardoor de ENDA's N_1 en N_2 maximaal $O(n2^n)$ toestanden hebben. Dus de deterministische versie van N gebruikt op inputgrootte $O(n2^n)$ een geheugen van $O((n2^n)^2) = O(n^2 2^{2n})$. We kunnen hieruit besluiten dat $EQ_{REX\uparrow} \in EXPSPACE$.

Nu moeten we nog aantonen dat $EQ_{REX\uparrow}$ EXPSPACE-hard is. Neem daartoe een taal $A \in EXPSPACE$ die beslist wordt door een TM M die in 2^{n^k} loopt. We moeten een input w mappen op twee expressies Q en R. Stel daarvoor eerst het tape alfabet en de toestanden van M gelijk aan Γ en $\hat{Q}$, respectievelijk.

[4] In mijn opleiding heb ik ook bestudeerd hoe men van reguliere expressies een equivalente ENDA kan maken. Deze conversie doet de grootte lineair groeien.

Q is gewoon Δ^*, met $\Delta = \Gamma \cup \hat{Q} \cup \#$, het alfabet bestaande uit alle symbolen die kunnen voorkomen in een computatie historie. We zullen R alle strings, behalve de strings die niet aanvaardende computatie histories zijn, laten genereren. Als M aanvaardt op w, dan zijn er geen niet aanvaardende computatie histories, dus genereert R dan alle strings, $R = \Delta^*$. Op die manier zijn de twee expressies equivalent als en slechts als M aanvaardt op w.

Een niet aanvaardende computatie historie is een sequentie van configuraties, gescheiden door #. Alle configuraties die korter zijn als 2^{n^k}, worden rechts bijgevuld met blanco's totdat de lengte juist gelijk is aan 2^{n^k}. Zo zijn alle configuraties juist even lang met een lengte van 2^{n^k}. De eerste configuratie in zo een historie is de startconfiguratie en de laatste is een niet aanvaardende configuratie. Iedere tussenliggende configuratie moet op de vorige volgen overeenstemmend met de transitie functie.

Hieruit kunnen we afleiden wanneer een string geen niet aanvaardende computatie historie is. ofwel is de startconfiguratie niet juist, ofwel de einconfiguratie niet, ofwel is er ergens een fout in een tussenliggende configuratie. Dus we kunnen in grote lijnen R al gelijk stellen aan:

$$R = R_{slecht-begin} \cup R_{slecht-venster} \cup R_{slecht-einde}.$$

We moeten $R_{slecht-begin}$ zo construeren dat het alle strings genereert die niet met de startconfiguratie C_1 beginnen. C_1 heeft de vorm $q_0 w_1 w_2 \ldots w_n \sqcup\sqcup \ldots \sqcup \#$, waarbij het symbool $\sqcup$ een blanco voorstelt. Voor $R_{slecht-begin}$ moeten we er dus voor zorgen dat hij alle strings genereert die niet met C_1 beginnen. De gemakkelijkste manier om dit te verwezenlijken is door een unie van verschillende subexpressies te nemen die ieder één symbool van C_1 behandelen:

$$R_{slecht-begin} = S_0 \cup S_1 \cup \ldots \cup S_n \cup S_b \cup S_\#$$

Expressie S_0 moet alle strings genereren die niet met q_0 beginnen, $S_0 = \Delta_{-q_0}{}^5\Delta^*$. S_1 genereert alle strings die geen w_1 op de tweede plaats hebben, $S_1 = \Delta\Delta_{-w1}\Delta^*$. In het algemeen ziet S_i er als volgt uit: $S_i = \Delta^i\Delta_{-w_i}\Delta^*$.

Expressie S_b genereert alle strings die geen blanco symbool op een positie tussen $n+2$ en 2^{n^k} bevat. We zouden hiervoor op dezelfde manier als voor S_0 tot S_n te werk kunnen gaan door nog een deel subexpressies S_{n+2} tot $S_{2^{n^k}}$ te introduceren, maar hierdoor zou de lengte van de expressie $R_{slecht-begin}$ exponentieel groot worden. Daarom moeten we S_b in één klap kunnen opschrijven als:

$$S_b = \Delta^{n+1} \left(\Delta \cup \epsilon\right)^{2^{n^k}-n-2} \Delta_{-\sqcup}\Delta^*.$$

Dus S_b genereert alle strings die eender welk symbool bevatten op de eerste $n+1$ posities, elk symbool op de volgende t posities, waarbij t kan variëren tussen 0 en $2^{n^k} - n - 2$ en elk symbool op de volgende positie behalve een blanco.

[5] De notatie Δ_{-s} is een afkorting voor de unie van alle symbolen in Δ behalve s.

Als laatste stukje voor $R_{slecht-begin}$ moeten we nog $S_{\#}$ definiëren. $S_{\#}$ moet alle strings genereren die geen # symbool bevatten op positie $2^{n^k} + 1$. Dus $S_{\#} = \Delta^{2n^k}\Delta_{-\#}\Delta^*$.

Vervolgens kunnen de constructie van $R_{slecht-einde}$ bekijken. Deze genereert alle strings die geen niet aanvaardende configuratie bevatten, of anders gezegd, strings die de toestand $q_{nietaanvaard}$ niet bevatten. We kunnen dus $R_{slecht-einde}$ gelijk stellen aan:

$$R_{slecht-einde} = \Delta^*_{-q_{nietaanvaard}}.$$

Nu rest ons nog het laatste deeltje, namelijk $R_{slecht-venster}$. Deze expressie genereert alle strings waarbij een configuratie niet volgens de transitie functie tot de volgende leidt. Een configuratie leidt tot een andere configuratie met respect voor de transitie funtie, als elke drie op elkaar volgende symbolen in de tweede configuratie, volgens de transitie functie, volgen op de overeenkomstige drie symbolen uit de eerste configuratie. Dus we moeten enkel de juiste zes symbolen onderzoeken om na te gaan of er een fout optreedt. Hiermee kunnen we $R_{slecht-venster}$ construeren:

$$R_{slecht-venster} = \bigcup_{slechte(abc,def)} \Delta^* abc \Delta^{2n^k-2} def \Delta^*$$

Hier betekent $slecht(abc, def)$ dat def niet volgt op abc volgens de transitie functie. De unie is dan ook enkel over die symbolen a, b, c, d, e, f uit Δ genomen.

Er komen een aantal exponenten van grootte 2^{n^k} voor in R, en hun totale lengte in binair is dus $O(n^k)$. Hieruit kunnen we besluiten dat de lengte van R polynomiaal is in n.

6.2 Relativisatie

Het bewijs dat $EQ_{REX\uparrow}$ ontraceerbaar is, rust op de diagonalisatie methode. In deze sectie behandel ik de methode van relativisatie om een heel sterk argument te geven dat het P versus NP probleem niet op te lossen valt met behulp van deze diagonalisatie methode.

In de relativisatie methode maken we weer gebruik van orakel Turing Machines zoals besproken in Hoofdstuk 4. Herinner ons nog dat een Turing Machine met een orakel voor SAT eender welk probleem in NP kan oplossen, en dat zo een Turing machine berekenbaar relatief ten opzichte van SAT wordt genoemd. Vandaar ook de term relativisatie.

Zoals ik reeds vermeld had is NP $\subseteq$ P^{SAT}. Verder is ook coNP $\subseteq$ P^{SAT}, omdat P^{SAT} gesloten is onder complemantatie, vermits het deterministisch is.

De volgende stelling geeft twee orakels A en B waarvoor P^A en NP^A verschillend zijn, en P^B en NP^B gelijk zijn. Dit is een sterk argument om ervan overtuigd

te geraken dat het P versus NP probleem heel waarschijnlijk niet kan opgelost worden met behulp van diagonalisatie.

De diagonalisatie methode is een simulatie van een Turing Machine door een ander. De simulerende machine kan het gedrag van de de andere machine achterhalen en zich dan anders gedragen. Stel dat beide machines hetzelfde orakel krijgen. Dan kan de simulerende machine nog altijd het gedrag van de gesimuleerde na bootsen, want hij kan zich ook bevragen aan het orakel zodra de gesimuleerde dat doet. Dus elke stelling over Turing Machines die bewezen is, enkel met behulp van de diagonalisatie methode, is nog steeds geldig als beide machines hetzelde orakel hebben.

Als we zouden kunnen bewijzen dat P en NP verschillend zijn, gebruik makend van de diagonalisatie methode, dan zouden we kunnen conluderen dat ze ook verschillend zijn met elk orakel. Maar we zullen aantonen dat P^B en NP^B gelijk zijn aan elkaar, dus vorige conclusie is niet geldig. Op dezelfde manier kan me dan ook aantonen dat de diagonalisatie methode ook niet werkt om aan te tonen dat P en PB gelijk zijn aan elkaar, want dan zouden ze beiden gelijk zijn met elk orakel, maar P^A is verschillend van NP^A. Dus de diagonalisatie methode is niet krachtig genoeg om het P versus NP probleem op te lossen.

Stelling: Er bestaat een orakel A waarvoor $\mathrm{P}^A \neq \mathrm{NP}^A$.

Bewijs: Eerst definiëren we voor elk orakel A, de taal L_A die alle strings bevat waarvoor er een string in A bestaat met dezelfde lengte.

$$L_A = \{w | \exists x \in A[|x| = |w|]\}$$

Duidelijk geldt voor iedere A dat $L_A \in \mathrm{NP}^A$.

Om aan te tonen dat $L_A \notin \mathrm{P}^A$, gaan we alle polynomiale orakel Turing Machines af, en verzekeren we dat ze er niet in slagen om L_A te beslissen. Laat daartoe $M_1, M_2, \ldots$ de lijst van alle polynomiale tijd Turing Machines zijn. Om het gemakkelijker te houden veronderstellen we dat M_i in tijd n_i werkt. We construeren A in verschillende stadia, waarbij stadium i ervoor zorgt dat M_i^A L_A niet kan beslissen. Hiertoe zullen we bepaalde strings wel tot A voegen en bepaalde strings niet. Elk stadium bepaald enkel de status van een eindig aantal strings, en initieel hebben we geen informatie over a en bevat het ook nog geen strings.

Stadium i. Tot nu toe zijn er een deel strings aan A toegevoegd en een deel strings niet. We kiezen n groter dan de lengte van al die strings en groot genoeg zodat 2^n groter is dan n^i. Ik zal nu bespreken hoe we de informatie van A moeten aanpassen zodat M_i^A de string 1^n aanvaardt als die string niet in L_A zit.

We laten M_i lopen op de input 1^n en reageren op de orakel vragen als volgt. ALs M_i het orakel raadpleegt over een string y waarvan de status al bepaald is, reageren we consistent. Als de status van y nog niet bepaald

is declareren we dat y niet in A zit en antwoorden dus ook NEE op de vraag aan het orakel. We gaan verder met de simulatie van M_i totdat deze stopt. Als M_i aanvaardt op 1^n, dan declareren we dat alle strings van lengte n niet tot A behoren. Als M_i niet aanvaardt op 1^n, dan kunnen we een string van lengte n vinden die nog niet aan het orakel bevraagd is, en declareren dat deze string in A zit, zodat 1^n zeker in L_A zit. We kunnen zo een string zeker vinden omdat er 2^n verschillende strings van lengte n zijn, en 2^n is groter dan n^i, het aantal stappen dat M_i doet.

Nadat alle stages zijn afgelopen, declareren we dat alle strings, waarvan hun status nog ongedefiniëerd is, niet tot A behoren. Zo kan geen enkel polynomiale tijd Turing Machine L_A beslissen met een orakel voor A.

Stelling: Er bestaat een orakel B waarvoor $\mathrm{P}^B = \mathrm{NP}^B$.

Bewijs: We kunnen voor B eender welk PSPACE-compleet probleem nemen, en neem bijvoorbeeld $TQBF$. We weten dat $\mathrm{NP}^{TQBF} \subseteq$ NPSPACE, want we kunnen ieder niet-deteministsche polynomiale tijd machine met orakel voor $TQBF$ omzetten in een niet-deterministische machine die de vragen naar het orakel $TQBF$ echt berekent, en dit in PSPACE. NPSPACE is op zijn beurt vervat in PSPACE, ze zijn zelfs gelijk aan elkaar volgens de stelling van Savitch. Ten laatste is dan PSPACE nog een deel van P^{TQBF}. Dit geldt omdat $TQBF$ PSPACE-compleet is.

Samenvattend hebben we dan de volgende drie inclusies:

$$\mathrm{NP}^{TQBF} \subseteq \mathrm{NPSPACE} \subseteq \mathrm{PSPACE} \subseteq \mathrm{P}^{TQBF}$$

Omdat $\mathrm{P}^{TQBF} \subseteq \mathrm{NP}^{TQBF}$, mogen we hieruit besluiten dat $\mathrm{P}^{TQBF} = \mathrm{NP}^{TQBF}$.

6.3 Circuit complexiteit

Definitie: Een booleaans circuit is een collectie van poorten en inputs die geconnecteerd zijn door draden. het is niet toegelaten om een cyclus te maken. De standaard poorten nemen drie vormen: AND, OR en NOT poorten. De grafische voorstelling van deze poorten kan men zien in figuur 6.1.

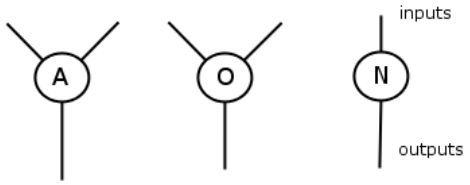
De draden in het circuit kunnen de waarden 0 of 1 dragen. De poorten berekenen simpelweg de booleaanse funties: AND, OR en NOT. Dus de AND-funtie heeft als output de waarde 1 als en slechts als beide inputs de waarde 1 hebben, de OR-functie heeft als output de waarde 0 als en slechts als beide inputs de waarde 0 hebben. De NOT-functie heeft als ouput de waarde 0 als en slechts als de input de waarde 1 heeft.

Eén poort in een circuit wordt benoemd tot de output poort, en de inputs worden gewoonlijk genoteerd met $x_1, x_2, x_3 \ldots x_n$.

Een booleaans circuit berekent de waarde die uit de output poort komt, door de inputs te volgen en zodra men een poort tegenkomt, daarvan de output waarde te berekenen en zo verder te gaan, totdat men de waarde van de output poort berekend heeft.

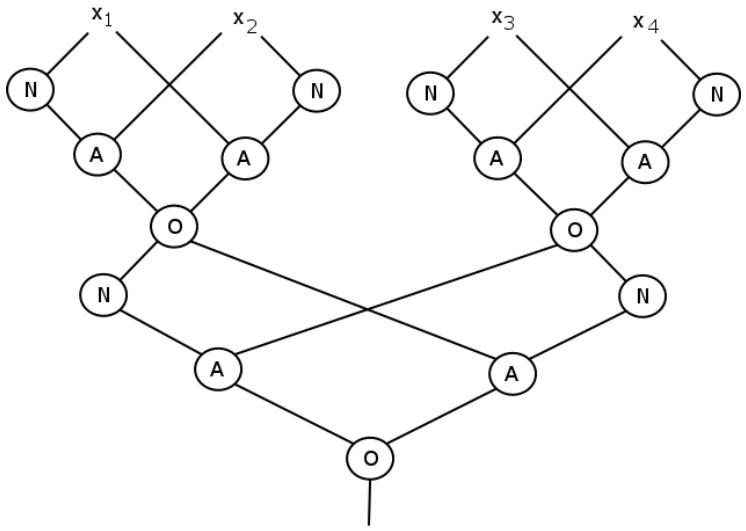
We kunnen een functie $f_C : \{0,1\}^n \to \{0,1\}$ associeren met een booleaans circuit C, op de volgende manier. Als de output van C gelijk is aan b, wanneer de inputs $x_1, X_2, x_3, \ldots, x_n$ op $a_1, a_2, a_3, \ldots, a_n$ gezet zijn, dan is $f_C(a_1, a_2, a_3, \ldots, a_n) = b$. We zeggen dat C de functie f_C berekent.

Er kunnen ook booleaanse circuits beschouwd worden die meerdere output poorten hebben. De hiermee geassociëerde functies mappen dan waarden uit $\{0,1\}^n$ op waarden uit $\{0,1\}^k$, als C k output poorten heeft.



Figuur 6.1: Links een AND-poort, in het midden een OR-poort en rechts een NOT-poort

Als een voorbeeld is de output van de n-pariteitsfunctie $pariteit_n : \{0,1\}^n \rightarrow \{0,1\}$ gelijk aan 1 als er een oneven aantal 1'tjes in de input voorkomen. In figuur 6.2 zien we een circuit dat $pariteit_4$ berekent.



Figuur 6.2: Een booleaans circuit dat de parieteitsfunctie berekent voor vier input variabelen.

We willen deze circuits gaan gebruiken om lidmaatschap tot bepaalde talen, gecodeerd in binair, te testen. Een probleem dat hierbij komt kijken is het feit dat één bepaald circuit enkel verschillende inputs kan behandelen die allen dezelfde lengte hebben. De 4-pariteitsfunctie kan bijvoorbeeld enkel inputs aan van de lengte vier. We kunnen dit probleem omzeilen door een familie van circuits te definiëren, waarbij elk circuit uit de familie verantwoordelijk is voor één welbepaalde inputgrootte.

Definitie: Een circuit familie C is een oneindige lijst van circuits

$$(C_1, C_2, C_3, \ldots),$$

waarbij C_n ook n input variabelen heeft. We zeggen dat C een taal A beslist als voor elke string w geldt:

$$w \in A \Leftrightarrow C_n(w) = 1$$

waarbij n de lengte is van w.

De grootte van een circuit is gelijk aan het aantal poorten dat het bevat. Twee circuits zijn equivalent als ze voor elke input dezelfde output

genereren. Een circuit heet minimaal als er geen equivalent cicuit bestaat met een kleinere grootte. Een circuit familie heet minimaal als elke C_i in de lijst minimaal is. De grootte complexiteit van een circuit familie $(C_1, C_2, C_3, \ldots)$ is de functie $f : N \to N$ met $f(n)$ gelijk aan de grootte van C_n.

De diepte van een crcuit is de lengte van het langste pad tussen een input variabele en de output poort. we definiëren minimaal diepte circuits en diepte complexiteit op dezelfde manier als voor circuit grootte.

De circuit grootte complexiteit van een taal is de grootte complexiteit van een miminmale circuit familie die de taal beslist. De circuit diepte complexieteit wordt analoog gedefiniëerd.

Stelling: Stel dat $t : N \to N$ een functie is met $t(n) \geq n$. Als $A \in \text{TIME}(t(n))$, dan heeft A circuit complexiteit $O(t^2(n))$.

Bewijs: Stel $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{aanvaard}, q_{verwerp})$ een beslisser zijn voor een taal A, en dit in tijd $t(n)$. Laat w een input zijn voor M van grootte n. We definiëren dan een tableau T voor M op w als volgt. T is een $t(n) \times t(n)$ matrix waarvan de rijen configuraties van M zijn. De bovenste rij bevat de startconfiguratie. De i-de rij bevat de configuratie die M bevat na de $(i-1)^e$ stap.

We zullen het formaat van de configuraties iets anders representeren als gewoonlijk. Het enigste verschil is dat we hier het toestandsymbool samen met het karakter waar het bij hoort (dat zich dus rechts van het toestandsymbool bevindt) als één karakter beschouwen. Dus iedere entry van de matrix kan een tape symbool (element van Γ) bevatten of een combinatie van een toestand en een tape symbool (element van $Q \times \Gamma$). We noemen de entry op de i-de rij en j-de kolom, zoals gewoonlijk bij een matrix, $T(i, j)$.

We maken wel nog twee veronderstellingen in het definiëren van de notie van een tableau. Ten eerste gaan we ervan uit dat M enkel anvaardt als het hoofd van de tape zich op de uiterst linkse positie bevindt en daar moet een blaco symbool, $\sqcup$, staan. Als tweede puntje gaan we ervan uit dat zodra M stopt, het ook in dezelfde configuratie blijft voor alle volgende tijd stappen. Dus zodat als we naar $T(t(n), 1)$ kijken we kunnen beslissen of M aanvaard heeft of niet.

De waardes van $T(i-1, j-1), T(i-1, j)$ en $T(i-1, j+1)$ bepalen de waarde van $T(i, j)$ volledig.

We zijn nu klaar om het circuit C_n te construeren. C_n heeft verschillende poorten voor elke cell in het tableau. Deze poorten berekenen de waarde van een cell door de waardes van de drie cellen die deze bepalen. Als k het aantal elementen is in $\Gamma \cup (\Gamma \times Q)$, dan maken we k lichtjes voor iedere entry in het tableau T, een lichtje voor ieder element uit Γ en een lichtje voor ieder element uit $\Gamma \times Q$. In het totaal geeft dit $kt^2(n)$ lichtjes. We noemen deze lichtjes $licht(i, j, s)$, met $1 \leq i, j \leq t(n)$ en $s \in \Gamma \cup (\Gamma \times Q)$.

Als het $licht(i, j, s)$ aan is wilt dat zeggen dat $T(i, j)$ het symbool s bevat. Natuurlijk is het de bedoeling dat er maar één lichtje per entry in T aan mag staan, anders zou de constructie verkeerd zijn.

De lichtjes zijn enkel een hulpmiddel om de output van een deel poorten duidelijk te maken, de constructie zou evengoed werken zonder deze lichtjes.

We weten al door welke drie waardes de waarde van $T(i, j)$ bepalen, dus $licht(i, j, s)$ moet branden als de lichtjes $licht(i-1, j-1, a), licht(i-1, j, b)$ en $licht(i-1, j+1, c)$ branden en s volgt op a, b en c volgens de transitie functie δ. We moeten de drie lichtjes op het $(i-1)^e$ niveau met een AND-poort verbinden, waarvan de ouput moet verbonden worden met $licht(i, j, s)$. In het algemeen kunnen er wel meerdere invullingen $(a_1, b_1, c_1), (a_2, b_2, c_2) \ldots, (a_l, b_l, c_l)$ van de drie entries in T bestaan zodat $T(i, j)$ het symbool s bevat. We verbinden dan voor iedere invullig ieder van deze overeenkomstige lichtjes met een AND-poort en alle AND-poorten met een OR-poort.

Deze constructie moeten we dan voor ieder lichtje herhalen, met een paar uitzonderingen op de grenzen. Elke entry op de linker grens van het tableau, $T(i, 1)$ voor $1 \leq i \leq t(n)$, heeft enkel twee voorgaande entries die zijn inhoud bepalen. De grenzen aan de rechterkant zijn gelijkaardig. Dus hiervoor moeten we maar twee lichtjes met een AND-poort verbinden.

De entries in de eerste rij hebben geen voorgangers, dus deze moeten op een andere manier behandeld worden. Deze entries bevatten de startconfiguratie en hun lichtjes worden verbonden met de inputvariabelen. Dus $licht(1, 1, q_0 1)$ is verbonden met de input w_1, en $licht(1, 1, q_0 0)$ is via een NOT-poort verbonden met de input w_1. Op dezelfde manier zijn de lichtjes $licht(1, 2, 1), \ldots, licht(1, n, 1)$ verbonden met $w_2, \ldots, w_n$ en de lichtjes $licht(1, 2, 0), \ldots, licht(1, 2, 0)$ verbonden met $w_2, \ldots, w_n$ via een NOT-poort. We moeten ook nog wel $light(1, n+1, \sqcup), \ldots, licht(1, t(n), \sqcup)$ initieel aanzetten, want de overige entries van de eerste rij in T bevatten het blanco symbool, $\sqcup$.

Tot nu toe hebben we een circuit gemaakt dat M simuleert tijdens de eerste $t(n)$ stappen. Het enigste dat nog resteert is één van de poorten tot outpout poort te benoemen. We weten dat M aanvaardt op w als de meest linkse entry in het tableau een symbool is dat met $q_a anvaard$ begint. Dus de ouput poort zal de poort zijn die verbonden is me $licht(t(n), 1, q_a ccept)$.

De vorige stelling geeft een manier op welke we zouden kunnen bewijzen dat NP≠P. Als we een bepaalde taal in NP zouden vinden, waarvoor de circuit complexiteit meer als polynomiaal is, dan zijn we zeker dat ook de tijdcomplexiteit groter is als polynomiaal. Hieruit volgt dan dadelijk dat we een probleem in NP hebben gevonden dat niet in P zit, dat dan ook de twee klassen van elkaar scheidt.

6.4 Voorbeelden en oefeningen

Stelling: TIME(2^n) = TIME(2^{n+1}).

Bewijs: Dat TIME(2^n) $\subseteq$ TIME(2^{n+1}) geldt, is duidelijk. Nu is TIME2^{n+1} = TIME(2×2^n). En $2 \times 2^n = O(2^n)$, dus TIME(2^n) = TIME(2^{n+1}).

Stelling: NTIME(n) $\subset$ PSPACE.

Bewijs: NTIME(n) $\subseteq$ NSPACE(n) $\subseteq$ SPACE(n^2) $\subset$ SPACE(n^3). De laatste inclusie volgt uit de hiërarchie stelling.

Stelling: Als $A \in$ P, dan is P^A = P.

Bewijs: Het is duidelijk dat P $\subseteq \mathrm{P}^A$. Om in te zien dat ook $\mathrm{P}^A \subseteq$ P, nemen we een $B \in \mathrm{P}^A$ en laten zien dat ook $B \in$ P. Telkens als B een vraag aan het orakel stelt, kunnen we dit stuk vervangen door een polynomiale tijd TM T. Al de rest van B werkt ook in polynomiale tijd, dus de uiteindelijke simulatie door enkel de vragen naar het orakel door T te vervangen werkt ook in polynomiale tijd.

Stelling: Als NP = P^{SAT}, dan is NP = coNP.

Bewijs: NP = P^{SAT} = coP^{SAT} = coNP.

Stelling: A is beslisbaar in tijd $O(t(n))$ als en slechts als het herkenbaar is in tijd $O(t(n))$, en $t(n)$ time constructible is.

Bewijs: Als A beslisbaar is, is het ook herkenbaar (triviaal). We moeten nu nog de andere richting aantonen. Beschouw daartoe het volgende algoritme:

D = Op input w:
1. Bereken n, de lengte van w.
2. Bereken $t(n)$.
3. Laat het herkenbaarheidsalgoritme H op w lopen, $O(t(n))$ stappen lang.
4. Als H stopt, geef dezelfde output, als H niet in die tijd stopt, *aanvaard niet*.

Stap 2 is mogelijk, en dit zelfs in tijd $t(n)$, omdat $t(n)$ time contructible is. We weten dat A herkenbaar is in $O(t(n))$, dus moet H na die tijd aanvaarden als $w \in A$. Als het na die tijd dus niet stopt, weten we zeker dat $w \notin A$.

Hoofdstuk 7

Alternatie

Definitie: Een alternerende Turing Machine is een nietdeterministische Turing Machine met een extra kenmerk. Zijn toestanden, behalve $q_{aanvaard}$ en $q_{verwerp}$, zijn verdeeld in universele en existentiële toestanden. Als we een alternerende Turing Machine op een input laten lopen, labelen we alle knopen van de niet deterministische boom met $\wedge$, als de overeenkomstige configuratie een universele toestand bevat, of $\vee$, als de overeenkomstige configuratie een existentiële toestand bevat. Een knoop is aanvaardend als het met $\wedge$ gelabeld is en al zijn kinderen aanvaardend zijn, of als hij met $\vee$ is gelabeld en er minstens één kind is dat aanvaardend is. De alternerende Turing Machine aanvaardt, als de wortel een aanvaardende knoop is.

7.1 Alternerende tijd en space

We definiëren de tijd en space klassen van deze machines op dezelfde manier als voor niet deterministische Turing Machines:

Definitie: $\text{ATIME}(t(n)) = \{L | L$ is een taal, beslist door een $O(t(n))$ tijd alternerende Turing Machine $\}$

$\text{ASPACE}(f(n)) = \{L | L$ is een taal, beslist door een $O(f(n))$ space alternerende Turing Machine $\}$

We definiëren AP, APSPACE en AL als zijnde de klassen van talen, beslist door alternerende polynomiale tijd, alternerende polynomiale space en alternerende logaritmische space Turing Machines, respectivelijk.

Voorbeeld: Een tautologie is een booleaanse formule die altijd tot 1 evalueert, onder eender welke toekenning van de variabelen:

$$TAUT = \{\langle\phi\rangle | \phi \text{ is een tautologie}\}$$

Het volgende alternerende algoritme laat zien dat $TAUT \in$ AP.

$T =$ op input $\langle\phi\rangle$

1. Selecteer universeel alle mogelijke toekenningen voor de variabelen van ϕ.

2. Evalueer ϕ voor een bepaalde toekenning.

3. Als ϕ tot 1 evalueert, *aanvaard*, anders *aanvaard niet*.

In de eerste stap worden op een universele manier alle mogelijke toekenningen geselecteerd. Opdat T zou aanvaarden, moeten alle kinderen van deze selectie aanvaarden. Stappen 2 en 3 checken deterministisch of een bepaalde toekenning tot 1 evalueert.

Dus T aanvaardt, als het beslist dat alle toekenningen bevredigbaar zijn.

Lemma: Voor $f(n) \geq n$ hebben we ATIME($f(n)$) $\subseteq$ SPACE($f(n)$).

Bewijs: We converteren een alternerende tijd $O(f(n))$ Turing Machine M in een deterministisch space $O(f(n))$ Turing Machine S die M simuleert, op de volgende manier. Op een input w zal S een depth first search doen op de boom van M om na te gaan welke knopen aanvaardend zijn. Dan kan S aanvaarden als het bepaald heeft dat de wortel een aanvaardende knoop is.

S heeft geheugen nodig om de recursie stack, die gebruikt wordt in de depth first search, op te slaan. ieder level van de recursie bewaart één configuratie. De recursie diepte is gelijk aan de tijd complexitiet van M. Dus elke configuratie gebruikt $O(f(n))$ space, en de tijd complexietiet van M is gelijk aan $O(f(n))$. S gebruikt dus $O(f^2(n))$ space.

Gelukkig kan deze space complexiteit verbeterd worden door te observeren dat S niet de volledige configuratie moet onthouden. In de plaats kan het gewoon de niet deterministische keuze die M maakt om tot die configuratie te komen, onthouden. S kan dan de configuraties opnieuw bekomen door van het begin te beginnen en de onthouden keuzes te volgen. Dus S moet enkel constant veel geheugen hebben in ieder stadium van de recursie. Wat de space comlexitiet doet verminderen tot $O(f(n))$.

Lemma: Voor $f(n) \geq n$ hebben we SPACE($f(n)$) $\subseteq$ ATIME($f^2(n)$).

Bewijs: We vertrekken van een deterministische space $O(f(n))$ machine M en we construeren een alternerende machine S dat $O(f^2(n))$ tijd gebruikt om M te simuleren.

Als er 2 configuraties c_1 en c_2 gegeven zijn van M en eengetal t, dan gaan we testen of M van configuratie C_1 in c_2 kan geraken in t stappen. Een alternerende procedure P hiervoor kiest eerst existentiëel een configuratie c_m. Vervolgens gaat het universeel checken of c_1 tot c_m kan geraken in $t/2$ stappen, en of c_m in $t/2$ stappen tot c_2 kan geraken, door recursief P op te roepen met de juiste parameters. Machine S gebruikt P om na te gaan of

een eindconfiguratie kan bereikt worden uit de beginconfiguratie in $2^{df(n)}$ stappen. Hier is d zo gekozen dat M niet meer als $2^{df(n)}$ configuraties kan hebben binnen zijn geheugen grenzen.

De maximale tijd die hiervoor gebruikt wordt is $O(f(n))$, om een configuratie te schrijven op elk level van de recursie, vermenigvuldigd met de diepte van de recursie, die gelijk is aan $log(2^{df(n)}) = O(f(n))$.

Samengevat krijgen we de volgende stelling:

Stelling: For $f(n) \geq n$ is ATIME($f(n)$) $\subseteq$ SPACE($f(n)$) $\subseteq$ ATIME($f^2(n)$).

Lemma: Voor $f(n) > log(n)$ hebben we ASPACE($f(n)$) $\subseteq$ TIME($2^{O(f(n))}$).

Bewijs: We construeren een detrministische tijd $2^{O(f(n))}$ machine S, die M, een alternerende space $O(f(n))$ machine, simuleert. Op een input w construeert S een graaf waarvan de knopen de configuraties van M op w, die maximaal $df(n)$ geheugen gebruiken, zijn. Er worden bogen toegevoegd van een configuratie c_1 naar een configuratie c_2 als c_2 in een enkele stap uit c_1 volgt. Na het construeren van deze graaf, gaat S er herhaldelijk door en markeert bepaade configuraties alszijnde aanvaardende. Initiëel zijn enkel de echte configuraties zo gemarkeerd. Een existentiële configuratie wordt gemarkeerd als minstens één van zijn kinderen gemarkeerd is. Een universele configuratie wordt gemarkeerd als al zijn kinderen al gelmarkeerd zijn. S blijft dit herhalen totdat er geen veranderingen meer optreden. Uiteindelijk zal S accepteren als de beginconfiguratie gemarkeerd is en anders niet.

Het aantal configuraties van M op w is gelijk aan $2^{O(f(n))}$, omdat $f(n) > log(n)$. Dus de grootte van de graaf is gelijk aan $2^{O(f(n))}$ en de constructie kan ook in deze tijd gebeuren. Het totaal aantal keer dat S door de graaf loopt is gelijk aan het aantal knopen en neemt $2^{O(f(n))}$ tijd in beslag. Dus de totale tijd die hiervoor gebruikt wordt is $2^{O(f(n))}$.

Lemma: Voor $f(n) \geq log(n)$ hebben we ASPACE($f(n) \supseteq$ TIME($2^{O(f(n))}$.

Bewijs: Ik laat zien hoe we een deterministische tijd $2^O(f(n))$ machine M kunnen simuleren door een alternerende space $O(f(n))$ Turing Machine S. In dit geval heeft S enkel genoeg geheugen om pointers te bewaren in een tableau T. We gebruiken nogmaals de representatie van configuraties waarbij één enkel symbool de toestand en de daarop volgende inhoud van de tape kan bevatten. Zoals we al eerder gezien hebben wordt de inhoud van een entry d in T bepaald door drie symbolen die op de rij erboven liggen in T, deze symbolen zullen we de ouders van d noemen(entries op grenzen van het tableau hebben maar twee ouders). S raadt en verifiëert recursief de inhoud van de entries. Om de inhoud van een entrie d, die zich

niet in de eerste rij bevindt, te verifiëren, raadt S existentiëel de inhoud van de ouders en maakt vervolgens gebruik van de transitie functie. Hij gaat dan universeel verder met alle andere entries. We gaan er nogmaals van uit dat de eindtoestand zich in de linker hoek van T bevindt, zodat S kan bepalen of M accepteert, door de linker onder hoek van T te bekijken. S moet nooit meer als één enkele pointer naar een entry in het tableau, bijhouden, dus het werkt in space $log(2^{O(f(n))}) = O(f(n))$.

Samengevat geven deze twee laatste lemma's de volgende stelling:

Stelling: Voor $f(n) \geq log(n)$ is ASPACE($f(n) =$ TIME($2^{O(f(n))}$.

Alternerende Turing Machines worde voornamelijk gebruikt omdat ze handig in gebruik zijn. Men kan soms eenvoudigere Turing Machines bedenken voor een problemen met behulp van alternatie. Als men dan zo een alterenerend algoritme heeft gevonden, kan men da complexiteit ervan meten en vervolgens de deterministische complexiteit er ook van weten door gebruik van vorige stellingen.

Hoofdstuk 8

Parallelle berekeningen

Parallelle computers kunnen meerdere berekeningen gelijktijdig doen. Ze kunnen bepaalde problemen sneller oplossen dan sequentiële computers, die maar één berekening per stap kunnen doen. In dit hoofdstuk houden we ons vooral bezig met uitgebreid parallellisme, waarbij een heel groot aantaal processor elementen samenwerken voor één enkele berekening.

8.1 Uniforme booleaanse circuits

In het booleaanse circuit model van parallelle computers, laten we elke poort een individuele processor zijn. Hierdoor kunnen we de processor compexiteit definiëren als zijnde de grootte van het circuit. We definiëren de parallele tijd complexieteit van een booleaans circuit als de diepte van dat circuit, wat gelijk is aan de lengte van het langste pad van een input variabele tot aan de output poort.

Definitie: Een familie circuits $(C_1, C_2, C_3, \ldots)$ heet uniform, als er een log space transducer T bestaat die $\langle C_n \rangle$ berekent op input 1^n.

Definitie: Een taal heeft gelijktijdige grootte-diepte circuit complexiteit van ten hoogste $(f(n), g(n))$ als er een uniforme circuit familie bestaat voor die taal met grootte complexiteit $f(n)$ en diepte complexiteit $g(n)$.

Voorbeeld: Laat A de taal zijn over het alfabet $\{0, 1\}$ bestaande uit alle strings met een oneven aantal 1tjes. We kunnen lidmaatschap testen met behulp van de pariteit functie. We kunnen de 2-pariteitsfunctie poort $x \oplus y$ implementeren met de standaard AND, OR en NOT poorten op de volgende manier: $(x \wedge \overline{y}) \vee (\overline{x} \wedge y)$. Laat $x_1, x_2, \ldots, x_n$ de inputs zijn voor het circuit. Eén manier om een circuit te construeren dat de pariteitsfunctie berekent is door poorten p_i te definiëren als volgt: $p_1 = x_1$ en $p_i = x_i \oplus g_{i-1}$ voor $i \leq n$. Deze constructie gebruikt $O(n)$ grootte en diepte.

Een andere constructie bestaat erin om een binaire boom te maken met poorten die de XOR-functie berekenen, waarbij de XOR-functie gewoon de 2-pariteitsfunctie is, hetzelfde als de $\oplus$ operator van hierboven. Hier is de diepte maar $log(n)$. We kunnen concluderen dat de grootte-diepte complexiteit van A gelijk is aan $(O(n), O(log(n)))$.

8.2 De klasse NC

Het is gebleken dat veel interessante problemen grootte-diepte complexiteit $(O(n^k), O(log(n^k)))$ hebben, voor een constante k. Deze problemen worden hoog parallellizeerbaar beschouwd. Dit geeft aanleiding tot de volgende definitie.

Definitie: NC^i is de klasse van talen die kunnen herkend worden door een uniforme familie van circuits met polynomiale grootte $O(n^i)$ en een diepte van $O(log^i(n)$, voor $i \geq i$. NC is de klasse van talen die in NC^i zitten voor een i. Functies die door zo een circuit familie kunnen berekend worden noemt men NC^i-berekenbaar of NC-berekenbaar.

We gaan nu onderzoeken wat de relatie van deze klassen zijn ten opzichte van reeds gekende klassen.

Stelling: $\mathrm{NC}^1 \subseteq \mathrm{L}$

Bewijs: We construeren een log space algoritme L om een taal $A \in \mathrm{NC}^1$ te berekenen. op een input w van lengte n, kan L de beschrijving van circuit C_n berekenen. Vervolgens kan het circuit evalueren door een depth-first search uit te voeren vanaf de output poort. Het enigste geheugen dat nodig is, is om het pad naar de poort die bezocht wordt bij te houden en de gedeeltelijke resultaten. De diepte van het circuit is logaritmisch, dus we hebben maar log space nodig voor de simulatie.

Stelling: $\mathrm{NL} \subseteq \mathrm{NC}^2$.

Bewijs: Neem een taal $A \in \mathrm{NL}$, gecodeerd in binair. We weten dat er een NL machine M voor bestaat. We construeren een circuit familie $(C_0, C_1, C_2, \ldots)$ voor A. Om C_i te construeren, maken we een graaf G, die erg gelijkaardig is aan de boom van M op input w van lengte n. Maar we kennen w niet wanneer we C_n contrueren, enkel de lengte n. De inputs voor het circuit zijn $w_1, w_2, \ldots, w_n$. We weten dat de collectie van configuraties van M niet van w afhangt, wel van n. Deze polynomiaal vele configuraties vormen de knopen voor G. De bogen in G zijn gelabeld met de input variablen w_i . Als c_1 en c_2 twee knopen zijn in G en c_1 heeft het hoofd op positie i, dan leggen we een boog van c_1 naar c_2, met label w_i (of $\overline{w_i}$) als C_2 op c_1 kan volgen in één enkele stap als het hoofd een 1

(of 0) leest. Als deze overgang mogelijk is, eender wat er wordt ingelezen, dan leggen we een ongelabelde boog.

Als we de bogen van G overeenkomstig met een string w van lengte n zetten, dan bestaat er een pad van de begintoestand, naar de eindtoestand als en slechts als er M aanvaardt op w. Dus een circuit dat de transitieve sluiting van G berekent en de aanwezeigheid van een dergelijk pad output, aanvaardt alle strings in A van lengte n. Dat circuit heeft polynomiale grootte en diepte $O(log^2(n))$.

Stelling: NC $\subseteq$ P

Bewijs: Een polynomiaal tijd algoritme kan, op input w van lengte n, de log space transducer laten lopen om het circuit C_n te genereren ne het vervolgens simuleren.