Games en Puzzels en hun Complexiteit

Thesis voorgedragen tot het behalen van de graad van licentiaat in de Informatica / doctorandus in de Kennistechnologie, afstudeervariant Databases

Filip Smets

Promotor: Prof. dr. Bart Kuijpers
Begeleider: Dr. Sofie Haesevoets

Academiejaar 2004-2005

Samenvatting

Een algoritme rijkt een oplossing aan voor een bepaald probleem. Het heeft een complexiteit die bepaalt hoeveel tijd en/of geheugen het algoritme nodig heeft, naargelang de grootte van de input, om een oplossing voor het probleem te vinden. De complexiteit van algoritmen horende bij puzzels en spellen doet pas begin jaren '80 zijn intrede, met het EXPTIME-compleetheids bewijs voor schaken als voorloper. Hierna is de interesse om spellen en complexiteit te laten samensmelten steeds gegroeid. De analyse van puzzels en spellen bleek een leuke invalshoek om complexiteit te bekijken. Vandaag de dag zijn reeds complexiteitsresultaten voor tientallen, zo niet honderde puzzels en spellen bekend. Het blijkt dat deze in verschillende klassen zijn onder te verdelen, en dit naargelang de moeilijkheid van het spel in kwestie. Deze thesis begint met een introductie tot de complexiteitstheorie, gevolgd door de fundamenten van de klasse NP en het begrip compleetheid. Hierna volgt een overzicht van een paar gekozen puzzels en spellen waarvoor complexiteitsresultaten bekend zijn, en dit naargelang de klasse waartoe ze behoren. Uiteindelijk wordt dieper ingegaan op Minesweeper dat NP compleet is. Aangaande dit spel zullen meerdere varianten bekeken worden, als ook de bijbehorende complexiteits resultaten. De complexiteitsresultaten van de varianten van Minesweeper die in deze thesis besproken worden, zijn nieuw en vormen mijn belangrijkste bijdrage aan dit werk.

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

Hoofdstuk 1

Inleiding

Reeds vanaf de oudheid is de mens gefascineerd door puzzels en spellen. Geweten is dat in het oude Egypte en China reeds vele, nu nog bekende spellen, werden gespeeld. Ook de Romeinen en de Arabieren kende het spel als het ontspanningsmiddel tegen verveling. Voorbeelden van bekende spellen die reeds lange tijd gespeeld worden zijn schaken, dammen, Backgammon en Go. Maar ook vandaag de dag verschijnen er nog enorm veel nieuwe puzzels en spellen op de markt, te veel om op te noemen. Het spreekt voor zich dat zowel puzzels als spellen in vele domeinen zijn terug te vinden, zo ook sinds twee decennia in de complexiteitstheorie.

In de complexiteitstheorie bestudeert men onder andere de complexiteit van bepaalde algoritmen. In deze thesis kijken we vooral naar hoeveel tijd en/of geheugen een algoritme nodig heeft om een bepaalde spellen en puzzels op te lossen. Om te kunnen praten over tijd en geheugen hebben we nood aan een speciaal model voor berekening. We zullen hiervoor de Turing Machine, bedacht door Alan Turing in de jaren 1930, in detail bespreken. De Turing Machine is als het ware de basis om over complexiteit te denken. De complexiteit van een algoritme wordt gemeten in functie van de grote van een bepaalde voorstelling van de input. Zoals zal blijken, bestaan er zeer grote verschillen in de looptijd van algoritmen. De uitkomst van deze looptijd is zeer bepalend voor de moeilijkheid van het probleem waarvoor het algoritme een oplossing biedt. Wanneer een te lange tijd of te veel ruimte benodigd is, zijn deze problemen zo goed als niet oplosbaar (bij grote input) met de huidige middelen.

Na de bespreking van Turing Machines zal een indeling gemaakt worden voor de verschillende *complexiteitsklassen* die er zijn, en dit aan de hand van hun looptijd/benodigde geheugen. De belangrijkste klassen komen aan bod. La-

ter doen verscheidene problemen aangaande puzzels en spellen hun intreden en deze zullen op hun beurt ingedeeld worden in de verschillende complexiteits klassen, naargelang de moeilijkheid van het probleem. Zo verkrijgen we een overzicht van de verschillende complexiteitsklassen en kan men zich een beeld vormen van waar bepaalde puzzels/spellen zich hierin bevinden qua complexiteit.

Uiteindelijk bespreken we Minesweeper in detail, tesamen met zijn belangrijkste varianten. Het probleem aangaande Minesweeper is: gegeven een Minesweeper configuratie, is deze consistent. We zullen het begrip consistentie duidelijk inkaderen en de problemen hieromtrent aankaarten. Het Minesweeper consistentie probleem zal NP-compleet blijken te zijn en dit heeft als gevolg dat er, met de huidige kennis, geen polynomiaal algoritme bestaat om dit probleem op te lossen. Dit wil zeggen dat er geen 'snelle' winnende strategie bestaat. De enige gekende winnende strategie is om alle mogelijke oplossingen voor het probleem af te gaan.
Het bewijs voor de NP-compleetheid van het Minesweeper consistentie probleem zal steunen op SAT en Circuit SAT, twee reeds gekende NP-complete problemen. Deze komen op hun beurt ook uitvoerig aan bod. De reductie (zoals dit genoemd wordt) van Circuit SAT naar Minesweeper zal gebruik maken van welbepaalde constructies die de simulatie verwezenlijken. De originele versie zal Minesweeper[4] genoemd worden, en dit omdat de grid in regelmatige vierhoeken wordt verdeeld. Richard Kaye bewees de eerder genoemde resultaten [Kay00a] en verwachtte dat varianten van Minesweeper een zelfde resultaat zouden opleveren. Ik nam de proef op de som en toonde van verscheidene varianten aan dat ze ook NP-compleet zijn. De grid van de varianten is verdeeld in respectievelijk regelmatige zes-, drie- en achthoeken. Ook wordt een klein uitstapje gemaakt naar 3-dimensionale-Minesweeper en ook hier blijkt hetzelfde resultaat te gelden. Er zal dus aangetoond worden dat wat R. Kaye intuïtief aanvoelde juist was.

Hoofdstuk 2

Complexiteit

Wat is een computer? Met deze vraag zou men de straat op kunnen gaan. Ongetwijfeld zal men een hele resem aan verschillende antwoorden verkrijgen, want wat voor de ene een computer is, is dat zeker niet voor de andere. Vele mensen zullen antwoorden dat een computer dat ding van om en bij de 1000 euro is dat ze thuis hebben staan en waarmee ze op Internet kunnen surfen en hun mails nalezen (PC-gebruiker). Een computer kan echter nog een heleboel andere dingen representeren. Zo zou het een eindige automaat kunnen zijn, alsook een mainframe. Daarom hebben we een conventie nodig van wat we precies met 'computer' bedoelen. Als we de term in de *grote van Dale* opzoeken krijgen we de volgende, algemene definitie: elektronische informatieverwerkende machine die door een reeks gecodeerde instructies wordt bestuurd.

Wanneer men echter over tijd- en ruimtecomplexiteit wil gaan spreken schiet deze definitie ruimschoots te kort. Er is namelijk geen precieze beschrijving van hoe die machine eruit moet zien, hoe ze werkt en welke instructies ze gebruikt. Bij complexiteit wil men namelijk bekijken hoeveel berekeningen[1] er gebeuren binnen een bepaalde tijd (ruimte). Voor het verdere verloop zullen we met een computer hetgene bedoelen dat Alan Turing rond 1936 definieerde als zijn model voor berekening. Hiernaar wordt vandaag vernoemd als de *Turing Machine* (TM). Hoe deze werken en wat ze kunnen zal in het volgende deel beschouwd worden. Voorlopig doen we het met: er kan aangetoond worden dat TMs qua rekenkracht equivalent zijn met (huidige) PC's, degene dus bij de mensen thuis in de woonkamer. Equivalent wil zeggen dat ze juist dezelfde berekeningen kunnen uitvoeren, en dezelfde dingen niet kunnen uitvoeren.

[1] Hier zal later duidelijk worden wat er met berekeningen bedoeld wordt.

2.1 Turing Machine

Om berekeningen te kunnen doen moet een Turing Machine (computer) kunnen *praten*. Niet letterlijk praten zoals mensen, maar ze hebben ook nood aan een computeralfabet waarmee computerwoorden, en zo ook een computertaal opgebouwd kunnen worden. Dit alfabet, dat veelal met de Griekse hoofdletter Sigma, Σ, genoteerd wordt, is het volgende:

Definitie 2.1.1 (Alfabet). *Een computer alfabet Σ is een niet-lege, eindige set (verzameling) van symbolen.*

Over welke symbolen dit zijn en hoe ze eruit zien spreken we ons niet uit, het is enkel van belang dat de set eindig en niet leeg is. Net zoals dit in de Nederlandse taal het geval is, kunnen symbolen van het alfabet samengesteld worden tot woorden. In het computerjargon noemt men dit *strings*.

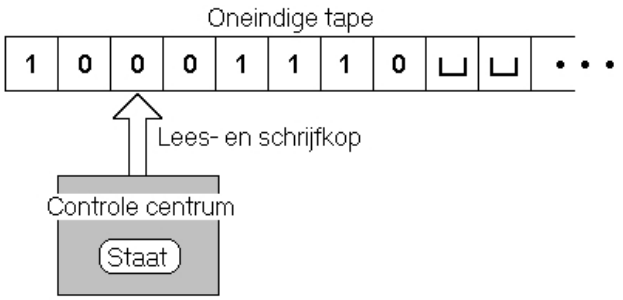
Definitie 2.1.2 (String). *Een string w over een alfabet Σ is een eindige sequentie symbolen uit Σ, waarbij het speciale symbool ϵ gereserveerd wordt voor de voorstelling van de lege string.*

In de Nederlandse taal is zo'n lege string niet aanwezig. Voor de leek is het daarom misschien moeilijk om zich hier iets bij voor te stellen. De beste voorstelling is een woord zonder letters, waarbij de lengte dus nul is. Vaak spreekt men over de *lengte* van een string. Dit zijn het aantal symbolen die in de string voorkomen en wordt vaak genoteerd door de string tussen twee verticale lijnen te plaatsen. Wanneer we dus over string w spreken, is de lengte van w gelijk aan $|w|$, en wordt meestal genoteerd met de letter n. Als laatste noteren we de *set van alle mogelijke strings* over een alfabet Σ, met Σ^* (de notatie heeft een nauwe band met reguliere expressies). Merk op dat deze set oneindig is, vermits herhalingen van symbolen uit Σ mogen voorkomen. Ook wordt, net zoals in het Nederlands het geval is, de taal zelf gevormd door een deelverzameling (subset) van Σ^*.

Definitie 2.1.3 (Taal). *Een taal L (van het Engelse 'language') is* een *subset van Σ^*.*

Er zijn meerdere mogelijkheden voor een taal per alfabet! Om te eindigen wordt nog een voorbeeld gegeven om alles verder te verduidelijken.

Voorbeeld 2.1.1 (Termen). Het alfabet der Nederlandse taal bestaat uit de volgende verzameling van symbolen {a,b,...,z,{leestekens}}, waarbij {leestekens} de set van alle mogelijke leestekens is; punt, komma, spatie, enzoverder. De mogelijke woorden zijn degene die voorkomen in het *groene boekje* en



Figuur 2.1: Turing Machine.

de taal wordt gevormd door de woorden volgens de grammaticale regels tot een zin te vormen. Bij een Turing Machine werkt dit net hetzelfde. We zouden bijvoorbeeld ook de Engelse taal kunnen nemen, die ook gebruik maakt van het eerder gedefiniëerde alfabet.

Zoals eerder vermeld kiezen we voor ons berekeningsmodel de Turing Machine. Met deze machines kunnen we ons interessante vragen gaan stellen over complexiteit van problemen en ze door middel van die TMs analyseren. Veelal handelt het rond de vraag of een bepaalde string w tot een taal L behoort. Zodoende begon men rond de helft van de vorige eeuw met het bouwen van machines die deze berekening deden. Deze machines waren in staat om bepaalde problemen sneller op te lossen dan mensen. In de tweede wereldoorlog waren ze bijvoorbeeld reeds belangrijk als middel om codes te kraken. Echter bleken TMs beperkingen te hebben, er waren namelijk problemen die ze niet konden oplossen[2]. Vanaf 1936, toen Alan Turing zijn *model of computation* (model voor berekening) introduceerde, begon de theoretische benadering van wat mogelijk was om te berekenen. Verdere resultaten zullen steeds gebaseerd zijn op dit model. Meerdere resultaten zullen geleend worden uit Sipser [Sip96], de resultaten in deze sectie komen uit [Sip96] en uit [McP03]. Een Turing Machine kan gezien worden als een eindige automaat, maar dan met een onbeperkt en oneindig geheugen.

Laten we beginnen met de opbouw van een TM. Deze machine heeft verschillende onderdelen. Ten eerste is er een soort controle centrum dat alles met elkaar verbindt. Laten we zeggen het koetswerk van de machine. Aan het controle centrum zit een arm met een lees- en schrijfkop bevestigd. Ten tweede is er de oneindige tape, die verdeeld is in (even grote) cellen. Deze tape is langs de linkse kant eindig, en langs de rechtse kant loopt hij onein-

[2]Eigenlijk is de reden waarom een TM uitgevonden is net om problemen die niet te berekenen vielen te analyseren.

dig door. In het begin van elke berekening bevindt de kop zich op de meest linkse cel van de tape. In de meest linkse cellen van de tape bevindt zich de input die aan de TM wordt meegegeven. Achter de input bevindt zich in elke cel het blanco symbool, weergegeven door $\sqcup$, tot in het oneindige. Voor een visuele representatie van een Turing Machine zie figuur 2.1[3].

2.1.1 Deterministische Turing Machine

Nu we een beeld van een TM voor ogen hebben, kunnen we verder gaan met de werking ervan. Daarvoor hebben we nood aan een formele definitie.

Definitie 2.1.4 (Turing Machine). *Een Turing Machine is een 7 tuppel* $(Q, \Sigma, \Gamma, \delta, q_s, q_a, q_r)$ *zodat:*

Symbool	Betekenis
Q	Set van alle staten
Σ	Input alfabet ($\sqcup \notin \Sigma$)
Γ	Tape alfabet ($\sqcup \in \Gamma$ en $\Sigma \subseteq \Gamma$)
$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$	Transitie functie
$q_s \in Q$	Start staat
$q_a \in Q$	Accept staten
$q_v \in Q$	Reject staten

Q, Σ *en* Γ *zijn steeds eindige verzamelingen (sets). Er kunnen meerdere Accept en Reject staten zijn, maar er is slechts één Start staat.*

De kern voor de werking van een TM is de transitie functie. Deze beschrijft hoe een TM een configuratie leest en eventueel aanpast om zo in een nieuwe configuratie te komen. Een configuratie van een TM is een setting van zowel de huidige staat, de huidige tape inhoud en de huidige positie van de kop en wordt vaak op zijn eigen manier voorgesteld door uqv. Hierbij is q de huidige staat en zijn u en $v \in \Gamma^*$ zodat uv de (huidige) inhoud van de tape is (zodat 'na' v enkel $\sqcup$'s voorkomen) én waarbij de (huidige) kop zich boven het eerste symbool van v bevindt. Er zijn twee speciale soorten configuraties:

- Start configuratie:
  Voor elke Turing Machine M is dit de configuratie waarmee begonnen wordt. In de definitie van een TM wordt deze voorgesteld door q_s. Voor M op input w is deze configuratie gelijk aan $q_s w$ (de string voor q_s is leeg).

[3]Bron: http://www.cs.pomona.edu/~marshall/courses/2003/spring/cs10/lectures/week13/TuringMachines/TuringMachines.htm

- Halt configuraties:
  Een halt configuratie doet een Turing Machine M stoppen. Hierdoor zijn er geen verdere berekeningen meer en wordt output gegeven. Deze output kan twee dingen zijn, naargelang de staat die de configuratie op dat moment heeft. Dus er zijn twee mogelijkheden:

  1. Aanvaard configuratie:
     Wanneer de huidige staat van een configuratie gelijk is aan q_a spreken we over een aanvaard configuratie. In dit geval zijn er geen verdere berekeningen meer en wordt output *true* teruggegeven. Dit betekend dat de berekening het gewenste resultaat heeft opgeleverd.
  2. Verwerp configuratie:
     Wanneer de huidige staat van een configuratie gelijk is aan q_v spreken we over een verwerp configuratie. In dit geval stopt M ook met berekenen, maar wordt output $false$ teruggegeven. In dit geval werd het te zoeken resultaat niet berekend door M.

Nu zal een beschrijving volgen van wat de transitie functie 'doet'. Stel dat we in een bepaalde configuratie uqv zijn aangekomen. We vertrekken steeds bij de start configuratie. Dan leest de kop van de TM M het eerste symbool van v, noem dit s. Daarna gaan we opzoek naar de mogelijke transitie functie waarbij q en s een paar vormen links van de pijl in de transitie functie. Deze bestaat omdat we van een deterministische TM uitgaan. Noem q', s' en r respectievelijk de staat, het symbool en de richting die zich rechts van de pijl in de transitie functie bevinden. Als laatste gebeuren er dan drie belangrijke dingen tijdens de berekining van een TM:

1. De lees- en schrijfkop schrijft s' op de plaats waar s staat, waardoor de inhoud van de huidige cel (eventueel) wordt veranderd.

2. De lees- en schrijfkop beweegt zich na de eerse stap een positie in richting r (waarbij $r = \mathrm{L} = $ links; $r = \mathrm{R} = $ rechts).

3. De staat van de configuratie verandert van q in q'.

Wanneer we de werking van een TM zo definiëren treedt er een probleem op als $r = L$ en de lees- of schrijfkop zich reeds boven de meest linkse cel bevindt. In dit geval kunnen we niet verder naar links bewegen. Dit wordt opgelost door te stellen dat in dit geval de kop zich boven de meest linkse cel blijft bevinden in de volgende configuratie en is hiermee een uitzondering op de regel.

Zo blijft de TM M de transitie toepassen tot het eventueel in een halt configuratie terecht komt. Dan stopt M onmiddellijk met berekenen en wordt *true* weergegeven indien de halt configuratie gelijk is aan de aanvaard configuratie. Anders wordt $false$ teruggegeven. Wanneer M niet in een halt configuratie terecht komt, komt M in een oneindige loop terecht en blijft M eeuwig doorlopen.

In tabel 2.1 worden drie voorbeelden gegeven van berekiningen in een TM aan de hand van een transitie functie. In de eerste kolom vinden we de originele configuratie. De tweede kolom bevat de transitie functie en in de derde kolom vinden we het resultaat van de berekening. Merk op dat er geen volledige formele invulling van de definitie van de TM gegeven wordt. De tabel dient enkel ter illustratie. De eerste regel geeft een verplaatsing naar rechts, de tweede en de derde een verplaatsing naar links, waarbij de kop zich in het derde voorbeeld reeds links bevindt. Er wordt ook gesteld dat de letters hier geen strings voorstellen, maar wel degelijk elementen van Γ.

start configuratie	toegepaste transitie	eind configuratie
$aaqbb$	$\delta(q,b) = (p,c,R)$	$aacpb$
$aaqbb$	$\delta(q,b) = (p,c,L)$	$apacb$
qbb	$\delta(q,b) = (p,c,L)$	pcb

Tabel 2.1: Toepassing van transitie functie op configuratie.

Een Turing Machine zoals net beschreven zal deterministisch genoemd worden. Dit om de reden dat de transitie functie steeds een gedeterminiseerd resultaat oplevert. Dit wil zeggen dat er maar een mogelijke transitie mogelijk is voor elk (Q, Γ)-paar. Verder zal ook de niet-determinisme TM aan bod komen.

Er zijn vele definities van TMs die allemaal equivalent (zouden moeten) zijn. Hier maken we gebruik van de gegeven definitie, die gebaseerd is op die van Sipser [Sip96].

Opmerking 2.1.1 (TM stopt niet altijd). Vermits een Turing Machine M pas stopt met berekenen wanneer hij een halt configuratie tegenkomt, kan het zijn dat M niet stopt en dus oneindig blijft doorlopen wanneer er geen halt configuratie wordt tegen gekomen.

Hieruit bekomen we de volgende definities:

$\delta(q,0)$	=	$(q,0,\mathrm{R})$	$\delta(p,1)$	=	$(p,0,\mathrm{L})$
$\delta(q,1)$	=	$(q,1,\mathrm{R})$	$\delta(p,0)$	=	$(p_a,1,\mathrm{R})$
$\delta(q,\sqcup)$	=	$(p,\sqcup,\mathrm{L})$	$\delta(p,\sqcup)$	=	$(p_a,1,\mathrm{R})$

Tabel 2.2: Transitie functie.

Definitie 2.1.5 (Aanvaardende TM). *Een Turing Machine M aanvaardt input w als er een seqentie van opeenvolgende configuraties C_1, C_2, ... , C_k bestaat waarbij:*

- *$C_1 = q_s$ van M op input w.*
- *De transitie functie toegepast op C_i brengt ons in C_{i+1}.*
- *$C_k = q_a$.*

Definitie 2.1.6 (Taal van TM M). *De taal van M is de verzameling van strings die M aanvaardt; en wordt genoteerd door $L(M)$.*

Voor we een concreet voorbeeld van een TM beschouwen, geven we nog twee definities waarvan in het vervolg nog gebruik gemaakt zal worden:

Definitie 2.1.7 (Herkenbaarheid). *Een taal A is* Turing herkenbaar *als er een TM bestaat die A herkent.*

Definitie 2.1.8 (Beslisbaarheid). *Een taal A is* Turing beslisbaar *(beslisbaar) als er een TM is die A beslist.*

Herkenbaar betekent dat de TM stopt in een accept staat voor elementen die tot de taal behoren en ofwel stopt ofwel eeuwig blijft lopen voor elementen die niet tot de taal behoren. Bij beslisbaarheid geldt hetzelfde als bij herkenbaarheid, behalve dat de TM steeds stopt (en dus niet eeuwig kan blijven loopen). Dus alle beslisbare talen zijn ook Turing herkenbaar zijn, maar de andere richting geldt niet. Om deze paragraaf af te sluiten wordt er nog een voorbeeld gegeven van een bepaalde TM, M:

$M = (Q, \Sigma, \Gamma, q_s, q_a, q_v, \delta)$ waarbij $Q = \{p, q, r, p_a\}$, $\Sigma = \{0,1\}$, $\Gamma = \{0,1,\sqcup\}$, $q_s = q$, $q_a = p_a$, $q_v = r$ en transitie functie δ zoals gegeven in tabel 2.2: Als input nemen we de binaire voorstelling van het getal 5, 101. Dan ziet de berekining van M eruit als volgt: M voert een som uit. Namelijk die van de input, noem w, en 1. Het resultaat (dat op de tape blijft staan) is dus gelijk aan *input* + 1. Zoals men kan zien in tabel 2.3 is het resultaat gelijk aan 6 in binaire notatie = 110.

q	1	0	1	$\sqcup$
1	q	0	1	$\sqcup$
1	0	q	1	$\sqcup$
1	0	1	q	$\sqcup$
1	0	p	1	$\sqcup$
1	p	0	0	$\sqcup$
1	1	p_a	0	$\sqcup$

Tabel 2.3: Berekening van M volgens de transitie functie uit tabel 2.2.

2.1.2 Niet-deterministische Turing Machine

Bij niet-determinisme gaat de TM M in configuraties *gokken* wat de volgende configuratie zal zijn. Het is dus mogelijk om vanuit een bepaalde configuratie door de transitie functie meerdere configuraties te bereiken. Bij determinisme was er slechts één mogelijke 'volgende' configuratie. Bij niet-determinisme hebben we vanuit een configuratie C meerdere morgelijke uitvoeringspaden en deze worden allen in één enkele stap uitgevoerd. Merk op dat dit een eerder onrealistische voorstelling is van een machine. Met andere woorden: er bestaat geen tastbare niet-deterministische Turing Machine, het is enkel een theoretisch model. Wel heeft dit model een grote impact op de theorie in verband met berekenbaarheid die in het vervolg aan bod zal komen. De formele definitie van een niet-deterministische TM is analoog aan die van een deterministische, behalve dan dat er nu nood is aan een andere transitie functie. We zullen deze definiëren als volgt:

δ: $Q \times \Gamma \rightarrow P(Q \times \Gamma \times \{\text{L,R}\})$

waarbij P het symbool is dat staat voor de mogelijke deelverzamelingen.

Sipser [Sip96] toont aan dat elke niet-deterministische TM een equivalente deterministische TM (DTM) heeft. Dit wil zeggen dat ze op elke input dezelfde output genereren alsook in dezelfde staat stoppen (of helemaal niet stoppen). Over de complexiteit spreken we ons nu nog niet uit. Ook vermeld hij andere soorten van Turing Machines waar we niet dieper op in zullen gaan dan ze hier even te vermelden. Hij vermeldt Multitape TMs, waarbij een TM meerdere tapes kan hebben, alsook enumerators, die strings als output kunnen genereren. Deze twee modellen zijn ook equivalent qua uitdrukkingskracht met DTMs, en dus ook met niet-deterministiche TMs.

2.2 Complexiteit

In deze sectie zullen de basisbegrippen van complexiteit aan bod komen. Eerst komen enkele termen en notaties voor die van belang zullen zijn. Daarna zal vertrokken worden vanuit de complexiteitsklasse P om zo verder te gaan naar de klasse NP. Nadat er wat uitleg over het begrip reduceerbaarheid wordt verschaft, zal NP-compleetheid nog aan bod komen, alsook het P *vs.* NP vraagstuk. Al deze begrippen zijn de basis om over complexiteit te spreken. Vermits het begrip complexiteit een groot domein omvat bespreken we hier enkel de zaken die nodig zijn voor het verdere verloop.

2.2.1 Termen

Wanneer men spreekt over een probleem, noemt men de oplossingsmethode ervoor vaak een algoritme. In feite is een algoritme een oplossing voor een probleem. Het aantal stappen, in functie van de grootte van de input w voor T, dat het algoritme in zijn uitvoering doet, bepaalt de complexiteit van de oplossingsmethode.

We bekijken nu verder de notatie in verband met complexiteit. We noteren de grootte $|w|$ van de input w, als n. Met $\mathbf{N}$ noteren we de verzameling van de natuurlijke getallen en met $\mathbf{R}$ noteren we de verzameling van de reële getallen. De *looptijd* van T is dan een functie $f\colon N \to N$, waarbij $f(n)$ het maximale aantal stappen is, gebruikt door T wanneer de input grootte n heeft.

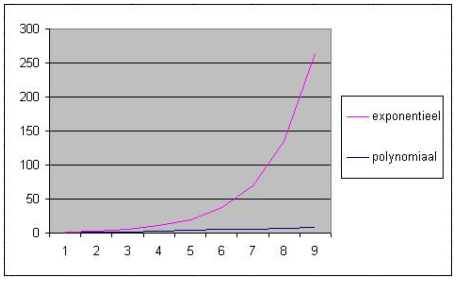
Men gebruikt de grote-O notatie om een bovengrens aan te geven. Deze grens is asymptotisch omdat we constante factors wegwerken.

Definitie 2.2.1 (Grote O). *Zei f en g twee functies $f, g\colon \mathbf{N} \to \mathbf{R}^+$. Dan is $f(n) = O(g(n))$ als en slechts als er een c, een $n_0 \in N$ bestaan zodat voor elke $n \geq n_0$ geldt:*

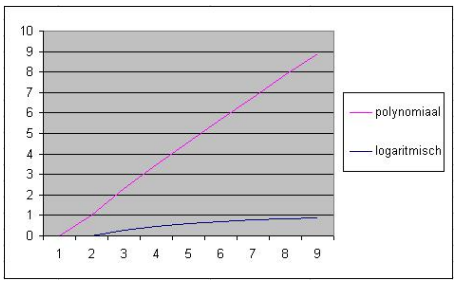
$$f(n) \leq c.g(n)$$

Voorbeeld 2.2.1. Zei $f(n) = 3n^2 + n + 8$. Wanneer men de hoogste term neemt ($= 3n^2$) neemt en de coëfficiënt wegneemt krijgt men $f(n) = O(n^3)$. Men kan dit resultaat nagaan. Neem $c = 4$ en $n_0 = 4$.

Als de grote-O van de vorm $O(n^k)$ is, met $k \geq 1$, dan noemen we dit een polynomiale grens. Is deze echter van de vorm $O(2^{(n^\delta)})$ met δ een reëel getal groter dan 0 dan hebben we te maken met een exponentiële grens. Wanneer de bovengrens van de vorm $O({}^m\log(n))$ is, met m een natuurlijk getal (groter



Figuur 2.2: Polynomiaal vs. exponentieel.



Figuur 2.3: Polynomiaal vs. logaritmisch.

dan 0), noemen we de bovengrens logaritmisch.

Er is een enorm verschil tussen de looptijd van een TM die in logaritmische, polynomiale of exponentiële tijd loopt. Figuren 2.2 en 2.3 maken dit verschil duidelijk. Hierin zijn de grafieken van de respectievelijke functie weergegeven. Het feit dat exponentiële algoritme een *lange* looptijd hebben wil niet zeggen dat ze niet bruikbaar zijn. Heel vaak zijn ze voor een kleine waarde van n zelfs heel nuttig. Wanneer de lengte van de input echter groot wordt, worden deze algoritmen onbruikbaar. De looptijd ervan is te groot en het zou dan ook voor vele problemen langer dan een eeuw (= 3.153.600.000 seconden) duren eer er een oplossing gevonden wordt voor een grote inputwaarden. Neem bijvoorbeeld de looptijd van de functie $f : n \mapsto 2^n$ voor de input grootte $n = 45$, dan zou het al tien eeuwen[4] duren om alle berekeningen te doen met 1000 berekeningen per seconde.

2.2.2 De klasse P

Vermits algoritmes waarvan de looptijd polynomiaal is, *berekend*[5] kunnen worden, zijn ze ook nuttig om te gebruiken. Dit in tegenstelling tot exponentiële algoritmes, waarbij vaak alle mogelijke oplossingen nagegaan moeten worden. Om de klasse P te bepalen zal polynomiale tijd belangrijk zijn: het zal niet verbazen dat de klasse P genoemd is naar het woord **P**olynomiaal.

Definitie 2.2.2 (Klasse P). *De klasse P bevat alle talen die in polynomiale tijd, in functie van de grootte van de input, beslisbaar zijn door een*

[4] 10 eeuwen aan 1000 berekeningen per seconde = 31.536.000.000.000 (schrikkeljaren niet meegerekend) in vergelijking met $2^{45} = 35.184.372.088.832$

[5] Dit is niet altijd het geval; omdat zowel w als de macht groot kunnen zijn, worden polynomiale algoritmes veelal beschouwd als 'snel' berekenbaar. Hierbij definiëren we echter "snel" niet nader.

deterministische Turing Machine.

Opmerking 2.2.1 (Andere TMs). Men zou "deterministische" kunnen vervangen door alle modellen die polynomiaal equivalent hieraan zijn, omdat dit geen invloed heeft op de polynomiale looptijd. Sipser [Sip96] geeft de 'multitape' TM als voorbeeld.

Voorbeeld 2.2.2 (Game of NIM). Een voorbeeld van een spel in P is *Game of NIM*. Met een spel in P bedoelen we dat er een polynomiaal algoritme bestaat om het spel *optimaal* te spelen. Voor de verdere uitwerking verwijs ik naar sectie 4.2.

2.2.3 De klasse NP

Tegen de verwachtingen in is de naam van deze klasse niet afgeleid van de woorden **N**iet **P**olynomiaal, waardoor veel verwarring omtrent deze klasse mogelijk is. De naam NP betekent echter **N**iet-deterministisch **P**olynomiaal. Dit leidt dan ook tot de voor de hand liggende definitie:

Definitie 2.2.3 (Klasse NP). *De klasse NP bevat alle talen die in polynomiale tijd, in functie van de grootte van de input, beslisbaar zijn door een niet-deterministische Turing Machine.*

Merk op dat er een fundamenteel verschil is met definitie 2.2.2. Een niet-deterministische TM kan immers 'gokken'. Fundamenteel hier is echter dat de looptijd ook polynomiaal is. Hierdoor bevat NP alle talen waarvoor we lidmaatschap in polynomiale tijd kunnen *gokken*.

Er is echter nog een andere zeer bekende definitie die de klasse NP karakteriseert.

Definitie 2.2.4 (Klasse NP). *NP is de klasse van talen waarvoor er een polynomiale* verificateur *is.*

Definitie 2.2.5 (Verificateur V). *Een verificateur voor een taal A is een algoritme V, zodat:*

$A = \{w : V$ *aanvaardt de input* w *en* c*, waarbij* c een *string is, die de oplossing voor het probleem encodeert*$\}$

Hierbij wordt de looptijd van een verificateur enkel gemeten in termen van de lengte van de input w. Men spreekt dus van een polynomiale verificateur *indien hij loopt in tijd die polynomiaal is in functie van $|w|$. Een taal A is in dit geval polynomiaal verifiëerbaar als het een polynomiale verificateur heeft.*

Dit is hetzelfde dan zeggen dat de klasse NP alle talen bevat waarvan lidmaatschap in polynomiale tijd nagegaan kan worden.

Opmerking 2.2.2. Een verificateur gebruikt dus extra informatie om na te gaan of een string $w \in A$, namelijk de string c. De c staat voor *c*ertificaat[6] voor lidmaatschap van A. Belangrijk hierbij is dat $|c|$ polynomiaal is in functie van $|w|$.

Als voorbeeld neem ik graag de metafoor die mijn professor theoretische informatica, Prof. Dr. J. Van den Bussche, tijdens een van zijn hoorcolleges gebruikte.

> Stel u de verificateur voor als de portier van de Snorrenclub. Hij kijkt na of iemand toegelaten mag worden tot de elite van de club. Hiervoor heeft men echter een snor nodig. De input w is dan de persoon die lid wil worden van de Snorrenclub. De snor is het certificaat c waarmee men toe kan treden tot de club. De woorden die tot de taal A behoren, zijn in dit geval de leden. Vermits de portier in een oogopslag kan zien of iemand een snor heeft[7], versterkt de link met het woord 'polynomiaal' alleen maar.

De twee gegeven definities zijn equivalent: dit wordt aangetoond in [Sip96]. Men kan dus kiezen tussen de twee mogelijke definities om aan te tonen dat een probleem in de klasse NP zit.

2.2.4 P *vs.* NP

Sipser [Sip96] vat het verband tussen deze twee klassen in kwestie mooi samen door het volgende te stellen:

- P = $\{L : L$ is een taal waarvan lidmaatschap snel *beslist* kan worden$\}$;
- NP = $\{L : L$ is een taal waarvan lidmaatschap snel *geverifiëerd* kan worden$\}$.

Belangrijk hierbij op te merken is dat elke taal waarvan lidmaatschap beslist kan worden in polynomiale tijd, dat lidmaatschap daarvan ook in polynomiale tijd kan geverifiëerd worden. De levensgrote vraag is nu: geldt de andere richting ook. Met name: kan een taal waarvan lidmaatschap in polynomiale tijd geverifiëerd kan worden, ook in polynomiale tijd beslist worden? En dit is nu juist één van de hamvragen van de informatica. Ze is zelfs zo belangrijk

[6] ook vaak bewijs genoemd

[7] valse snorren worden buiten beschouwing gelaten

dat het Clay Institute[8] 1 miljoen dollar uitlooft aan degene die het antwoord op de volgende vraag weet (en kan aantonen):

$$\text{Is P} = \text{NP of niet?}$$

Deze vraag kan op drie manieren opgelost worden:

1. Vind een polynomiaal algoritme voor *alle* problemen in NP. Zodoende is er geen enkel probleem meer in NP dat niet met een polynomiaal algoritme op te lossen is; daaruit zou meteen volgen dat P = NP.

   Anders gezegd: vind een polynomiaal algoritme voor *een* NP-compleet probleem. Dit is hetzelfde vermits alle problemen in NP polynomiaal reduceerbaar zijn tot elk NP-compleet probleem (zie verder).

2. Toon aan dat er een probleem is dat in NP zit, maar dat niet in P zit. Met andere woorden: vind een probleem dat polynomiaal verifiëerbaar is, maar niet polynomiaal beslisbaar. Hieruit volgt meteen dat P $\neq$ NP.

3. Misschien een van de moeilijkste dingen om aan te tonen is dit: bewijs dat het vraagstuk P $\stackrel{?}{=}$ NP niet bewezen kan worden. Bewijs dus dat we niet kunnen aantonen dat ofwel P = NP ofwel P $\neq$ NP.

Helaas heeft nog nooit iemand een van deze drie punten aangetoond. Het P *vs.* NP probleem is dus nog steeds niet opgelost. Sterker nog, het is één van de beroemdste open problemen. De relatie tussen de verschillende complexiteitsklassen is gegeven in het Venndiagram in figuur 2.4: waarbij we niet weten of er een probleem is in de deelverzameling aangeduid door de vraagtekens.

Opmerking 2.2.3. Men kan een niet-deterministische TM simuleren door een deterministische TM. De simulatie gebeurt echter niet in polynomiale tijd.

2.2.5 PSPACE en NPSPACE

Analoog zoals voor tijd kan men ook een complexiteitsklasse definiëren gebaseerd op het gebruik van geheugenruimte. De zogenaamde *space* die een Turing Machine nodig heeft om een berekening uit te voeren is de lengte van de tape die beschreven wordt. Net zoals bij tijd het geval is, onderscheiden we hier ook twee indelingen, namelijk die van de deterministische space en van de niet-deterministische space.

[8] http://www.claymath.org/

Definitie 2.2.6 (PSPACE). *De klasse PSPACE bevat alle talen die in polynomiale ruimte, in functie van de grootte van de input, beslisbaar zijn door een deterministische Turing Machine.*

Definitie 2.2.7 (NSPACE). *De klasse NSPACE bevat alle talen die in polynomiale ruimte, in functie van de grootte van de input, beslisbaar zijn door een niet-deterministische Turing Machine.*

Echter is er hier een zeer groot verschil met de tijdsklasse. In de volgende stelling staat NSPACE(f(n)) voor alle problemen in NSPACE die beslist kunnen worden in ruimte met grootte n (waarbij n de inputgrootte is). Analoog staat PSPACE(f^2(n)) voor alle problemen in PSPACE die beslist kunnen worden in ruimte n^2 (waarbij n de inputgrootte is). Savitch toonde namelijk het volgende aan:

Stelling 2.2.1 (Savitch). *Voor een functie $f\colon \mathbf{N} \to \mathbf{N}$, met $f(n) \geq n$ geldt:*

$$NSPACE(f(n)) \subseteq PSPACE(f^2(n))$$

Het bewijs voor Savitch's theorem laten we achterwege. Dit steunt op het zogenaamde *yieldability probleem*, waarbij er nagegaan wordt of een bepaalde configuratie een andere kan bereiken binnen een opgelegd aantal stappen.

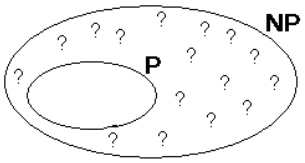
Merk op dat dit, juist door 'Savitch's theorem', alle talen zijn die zich zowel in PSPACE als in NSPACE bevinden. Ten tweede is het ook zo dat alle talen in NP zich ook in PSPACE bevinden. De langste run voor die taal neemt immers slechts polynomiale tijd in beslag en kan dus logischerwijze ook niet meer dan polynomiale ruimte gebruiken.

2.2.6 Andere complexiteitsklassen

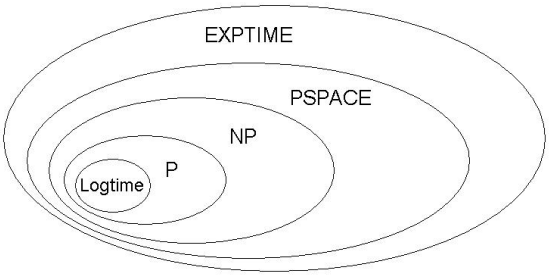
De looptijd van een TM hoeft vanzelfsprekend niet polynomiaal te zijn. Er zijn algoritmen die een andere complexiteit in functie van de inputgrootte hebben. Zo zijn daar exponentiële tijd en logaritmische tijd. Analoog aan de definitie gegeven voor de klasse P, geven we die nu ook voor deze twee. Buiten de definities te geven zullen we er niet veel langer bij stil blijven staan

Definitie 2.2.8 (LOGTIME). *De klasse LOGTIME bevat alle talen die in logaritmische tijd beslisbaar zijn door een deterministische Turing Machine.*

Definitie 2.2.9 (EXPTIME). *De klasse EXPTIME bevat alle talen die in exponentiële tijd beslisbaar zijn door een deterministische Turing Machine.*



Figuur 2.4: P vs. NP ([Sip96]).



Figuur 2.5: Algemeen aangenomen relatie tussen de belangrijkste complexiteitsklassen ([Sip96]).

De algemeen aangenomen relatie tussen de verschillende tijdscomplexiteits klassen zien we in figuur 2.5. Merk op dat we in werkelijkheid veelal niet weten of de inclusies strikt zijn. Zoals verwacht kunnen we deze definities ook uitbreiden naar de ruimte. We spreken in deze gevallen van logaritmische space en exponentiële space. De klasse logaritmische space heeft een voor de complexiteitstheorie belangrijke tegenhanger, nl. niet-deterministische logaritmische space. De definities zijn analoog en spreken voor zich.

Definitie 2.2.10 (L). *De klasse L bevat alle talen die in logaritmische space beslisbaar zijn door een deterministische Turing Machine.*

Definitie 2.2.11 (NL). *De klasse NL bevat alle talen die in logaritmische space beslisbaar zijn door een niet-deterministische Turing Machine.*

Definitie 2.2.12 (EXPSPACE). *De klasse EXPSPACE bevat alle talen die in exponentiële space beslisbaar zijn door een deterministische Turing Machine.*

Opmerking 2.2.4. Er zijn nog meerdere complexiteitsklassen zoals dubbel exponentiëel enzoverder. Deze zijn in ons opzicht van weinig belang en doen hier dan ook niet ter zake.

2.2.7 Reduceerbaarheid

In de voorbije sectie werden verschillende complexiteitsklassen besproken, tesamen met enkele onderliggende verbanden. Hier zullen we ons concentreren op de verbanden binnen een bepaalde klasse. De bekomen resultaten zijn representatief voor de verschillende complexiteitsklassen. De twee volgende definities zijn nodig om te kunnen werken met reduceerbaarheid:

Definitie 2.2.13 (Polynomiaal berekenbare functie). *Een functie $f : \Sigma^* \to \Sigma^*$ is een* polynomiaal berekenbare functie *als er een TM M, die op input een string w uit Σ^*, werkt in polynomiale tijd, en die stopt met $f(w)$ op zijn tape.*

Nu is het volgende waar:

Definitie 2.2.14 (Polynomiaal reduceerbare taal). *Een taal A over een alfabet Σ is polynomiaal reduceerbaar tot een taal B over een alfabet Σ, genoteerd $A \leq_P B$ als er een polynomiaal berekenbare functie $f : \Sigma^* \to \Sigma^*$ bestaat, zodat voor elke w geldt:*

$$w \in A \iff f(w) \in B$$

De functie f wordt de polynomiale tijd reductie van A naar B genoemd.

Met andere woorden: als we een oplossing hebben voor probleem B, hebben we die ook voor probleem A, de tijd die hiervoor nodig is, is afhankelijk van de tijd om een oplossing voor B te berekenen. Als de tijd nodig op een oplossing voor probleem B te vinden polynomiaal is in functie van de input $f(w)$, dan is de tijd (in dit geval) nodig om een oplossing voor A te vinden, ook polynomiaal in functie van de input w.

2.2.8 NP-compleetheid

We vallen onmiddellijk met de deur in huis door de definitie te geven:

Definitie 2.2.15 (NP-compleet). *Een taal B is NP-compleet als en slechts als:*

1. $B \in NP$
2. $\forall A \in NP$: $A \leq_P B$

Dus een taal B is pas NP-compleet als B eerst en vooral in NP zit. Dit wil zeggen dat er een NTM is die B herkent in polynomiale tijd. Anderzijds moet **elk** ander probleem in de klasse NP polynomiaal reduceerbaar zijn tot B. Dit is een zeer strenge conditie. Daarom is elk NP-compleet probleem *minstens* zo moeilijk op te lossen als een willekeurig ander probleem A in NP. De reden hiervoor is, dat als we een oplossing voor B hebben, we er ineens ook een hebben voor A, en dit door de reduceerbaarheid. Ter volledigheid vermelden we dat er ook NP-hardheid bestaat. In dit geval moet de taal B aan puntje twee van definitie 2.2.15 voldoen, maar niet noodzakelijk aan puntje één. We kunnen NP-harde problemen dus zien als de intersectie

tussen de problemen in de klasse NP en de NP-complete problemen.

Er zijn echter nog twee belangrijke stellingen in verband met NP-compleetheid.

Eerste tonen we aan hoe de NP-compleetheid van een nieuw probleem kunnen bewijzen, vertrekkende van een gekend NP-compleet probleem.

Stelling 2.2.2 (Afleiding NP-compleet). *Een taal C is NP-compleet als en slechts als aan de volgende drie punten voldaan is:*

- *B is NP-compleet,*
- *$C \in NP$,*
- *$B \leq_p C$.*

Bewijs. Vermits B NP-compleet is (puntje 1), zijn alle andere problemen (noem A) polynomiaal reduceerbaar tot B (def. NP-compleet). Nu is B ook polynomiaal reduceerbaar tot C (puntje 3). Dus alle problemen A zijn ook polynomiaal reduceerbaar tot C, vermits de samenstelling van twee polynomiale functies terug een polynomiale functie is. Het enige wat er volgens de definitie van NP-compleetheid nodig is, is dat $C \in$ NP. Vermits dit gegeven is in puntje 2, vervolledigt dit het bewijs. □

Wanneer men dus op zoek wil gaan naar nieuwe NP-complete problemen N, volstaat het op te vertrekken van een ander probleem A waarvan men reeds weet dat het NP-compleet is, en een polynomiale reductie vinden van A naar N. De enige restrictie die nog moet gelden is dat $N \in$ NP.

Uit stelling 2.2.2 volgt ook volgende stelling:

Stelling 2.2.3. *B is NP-compleet en $B \in P \Rightarrow P = NP$.*

Bewijs. Net zoals in het vorige bewijs kunnen alle problemen A in NP polynomiaal gereduceerd worden tot B om wille van de definitie. Als B zich in P bevindt, wil dit zeggen dat B op te lossen is in polynomiale tijd op een deterministische TM. Zodoende zijn dan alle problemen A in polynomiale tijd op te lossen. Met name door ze eerst naar B te reduceren en dan B op te lossen. □

Helaas heeft nog niemand een NP-compleet probleem kunnen vinden dat zich in P bevindt. Dit zou onnoemelijk veel bekende vraagstukken die er zijn vereenvoudigen. Misschien bestaat er zo ook geen probleem. Wie weet?

Hoofdstuk 3

SAT en Circuit SAT

Dit hoofdstuk vormt de overgang van de theorie in verband met complexiteit naar het meer praktische probleem aangaande Minesweeper, waar we in een later hoofdstuk verder op ingaan. Nu volgt een schets van de situatie om de notie van Circuit SAT in zijn context te plaatsen. In deel 3.2 zien we dat SAT NP-compleet is. In de volgende hoofdstukken die over Minesweeper handelen zal telkens een polynomiale reductie gegeven worden van Circuit SAT naar het Minesweeper probleem in kwestie. Ook tonen we in die delen aan dat de respectievelijke Minesweeper problemen zich in de complexiteitsklasse NP bevinden. Het gevolg is dat als er een polynomiale reductie van SAT naar Circuit SAT bestaat, door definitie 2.2.15 geldt dat de Minesweeper problemen dan NP-compleet zijn. Eerst moet het begrip *Circuit SAT* echter worden verduidelijkt.

Om tot een formele definitie van Circuit SAT te komen bekijken we eerst de *booleaanse logica*, vermits Circuit SAT hier nauw verband mee is. Booleaanse logica bevat een aantal operatoren die naargelang de input een bepaalde, welgedefiniëerde output opleveren. Daarna gaan we over naar *logische circuits*. De werking ervan is vrij analoog aan die van de booleaanse logica en zal er dus op gebaseerd zijn. Door middel van die logische circuits komen we uiteindelijk tot het probleem **Circuit SAT**. Als laatste vermelden we nog dat logische circuits ook gebruikt worden in de complexiteitstheorie om lidmaatschap van talen aan te tonen, analoog aan Turing Machines. Op dit aspect van logische (booleaanse) circuits zullen we echter niet ingaan.

3.1 Booleaanse logica

George Boole legde de basis voor de booleaanse logica rond het midden van de negentiende eeuw. Deze logica diende vooral om te redeneren over de wiskunde door middel van (booleaanse) regels [Kui01]. De booleaanse logica bleek zeer geschikt te zijn om te redeneren over wiskundige beweringen en is dan ook uitgegroeid tot een aparte tak binnen de wereld van de wetenschappen. Veelal wordt het ook de *booleaanse algebra* genoemd. We zullen hier zeker niet de hele logica bespreken noch de gehele werking van de booleaanse algebra. Wel zullen de nodige begrippen en definities aan bod komen die nuttig zijn voor het vervolg.

Zoals eerder aangehaald, werkt de booleaanse logica met operatoren. Voor het verdere verloop hebben we nood aan drie van deze operatoren. Er zijn meerdere operatoren, en we zullen er ook meerdere bekijken, het is enkel zo dat de andere operatoren kunnen opgebouwd worden uit een aaneenschakelijk van deze drie operatoren. De inputsignalen voor de functies zijn de zogenaamde *booleaanse variabelen* (meestal genoteerd met het symbool x, al dan niet voorzien van een index), ofwel een *constante*. Deze kunnen ofwel de waarde 1 hebben, ofwel de waarde 0. Dit is louter per definitie, we hadden even goed *true* en *false*; *op* en *af*; *wit* en *zwart* of zelfs *peper* en *zout* kunnen nemen. Verderop zullen we ook *true* (T) en *false* (F) gebruiken voor resp. 1 en 0. De drie operatoren zijn als volgt gedefiniëerd:

1. Conjunctie: symbool $\wedge$
   Een conjuctie heeft twee inputs (binaire operatie). Enkel wanneer beide inputs de waarde 1 hebben zal de output ook 1 zijn. In de drie andere gevallen is de output 0.

2. Disjunctie: symbool $\vee$
   Een disjuctie heeft ook twee inputs (binaire operatie). Wanneer beide 0 zijn zal de output ook de waarde 0 zijn. In alle drie andere gevallen is de output 1.

3. Negatie: symbool $\neg$
   De negatie heeft slechts een input (unaire operatie). Als deze 1 is zal de output 0 opleveren en vice versa.

De overeenkomstige tabellen zijn gegeven in tabellen 3.1, 3.2 en 3.3. We hebben nu genoeg materiaal om te komen tot de volgende recursieve definitie.

Definitie 3.1.1 (Booleaanse expressie). *Een booleaanse expressie over* $X \cup \{\wedge, \vee, \neg, (,), 0, 1\}$*, waarbij* X *de verzameling van booleaanse variabelen* $\{x_1, x_2, x_3, \ldots, x_n\}$ *is van de vorm:*

ϕ_1	ϕ_2	$\phi_1 \wedge \phi_2$
0	0	0
0	1	0
1	0	0
1	1	1

Tabel 3.1: Conjunctie.

ϕ_1	ϕ_2	$\phi_1 \vee \phi_2$
0	0	0
0	1	1
1	0	1
1	1	1

Tabel 3.2: Disjunctie.

- *de constante 0 of 1,*
- *een variabele $x_i \in X$ (met $i = 1, 2, 3, ...$),*
- *expressie van de vorm $\phi_1 \wedge \phi_2$,*
- *expressie van de vorm $\phi_1 \vee \phi_2$,*
- *expressie van de vorm $\neg\ \phi_1$,*

waarbij ϕ_1 en ϕ_2 zelf reeds booleaanse expressies zijn.

Opmerking 3.1.1 (Conjuctie vs. disjunctie). Een disjunctie van de vorm $\phi_1 \vee \phi_2$ kan steeds geschreven worden als een combinatie van een conjunctie en negatie. De equivalente booleaanse expressie is dan $\neg(\neg\phi_1 \wedge \neg\phi_2)$

Definitie 3.1.2 (Literal). *Een* literal *is een booleaanse expressie van de vorm x_i of $\neg x_i$, waarbij $x_i \in X$.*

Definitie 3.1.3 (Clause). *De conjunctie of disjunctie van literals noemen we een* clause *over X.*

De set van alle mogelijke booleaanse expressies over X noteren we door de Griekse hoofdletter Φ. Door de drie operatoren toe te passen op de waarden van de variabelen, komen we tot het resultaat van een booleaanse expressie, 1 of 0. Zo komen we tot volgende definitie.

ϕ_1	$\neg\ \phi_1$
0	1
1	0

Tabel 3.3: Negatie.

Definitie 3.1.4 (Waarheidstoekenning). *Een mapping van booleaanse variabelen uit X naar de set van waarheden {0, 1} noemen we een waarheidstoekenning, genoteerd door ω. Met andere woorden, een waarheidstoekenning is een functie f: {0,1}^{n} → {0,1}. Veelal wordt een waarheidstoekenning door middel van een tabel voorgesteld, dit noemen we dan de waarheidstabel.*

Als laatste geven we nog een booleaanse expressie die ook in verdere hoofdstukken gebruikt zal worden. In het vervolg zullen we hiervoor onder andere een equivalent logisch circuit geven, alsook een equivalente Minesweeper constructie.

$$x_1 \wedge \neg x_2 \tag{3.1}$$

De waarheidstabel overeenkomstig met formule 3.1 wordt gegeven in tabel 3.4.

x_1	x_2	$x_1 \wedge \neg x_2$
0	0	0
0	1	0
1	0	1
1	1	0

Tabel 3.4: Waarheidstabel voor formule 3.1.

3.2 SAT

SAT is het probleem aangaande de *sat*isfiability van booleaanse formules. We hebben reeds gezegd dat we vertrekken vanuit een NP-compleet probleem om NP-compleetheid van andere problemen aan te tonen, maar we kennen nog geen NP-compleet probleem. Hier zal het eerste NP-complete probleem aan bod komen. Ook in de geschiedenis was dit het eerste probleem waarvan men de NP-compleetheid heeft aangetoond. Dit gebeurde door zowel Cook als Levin, die onafhankelijk van elkaar tot hetzelfde resultaat kwamen. Vanaf het moment dat het eerste NP-complete probleem bekend was, zijn hieruit

honderden, zoniet duizenden andere NP-complete problemen gevonden[1] door middel van stelling 2.2.2.

Definitie 3.2.1 (SAT). *Gegeven een booleaanse expressie e; is er een waarheidstoekenning aan de variabelen uit e, die e waar maakt.*

De formule gegeven in 3.1 is satisfiable. Dit zien we aan de waarheidstoekenning gegeven in formule 3.4. Wanneer $x_1 = 1$ en $x_2 = 0$ dan is formule 3.1 waar.

3.2.1 Complexiteit

In dit deel zal aangetoond worden dat SAT NP-compleet is. Naar deze stelling wordt ook verwezen als de stelling van Cook en Levin.

Stelling 3.2.1 (Stelling van Cook en Levin). *SAT is NP-compleet.*

Bewijs. Volgens stelling 2.2.15 moeten we twee dingen aantonen:

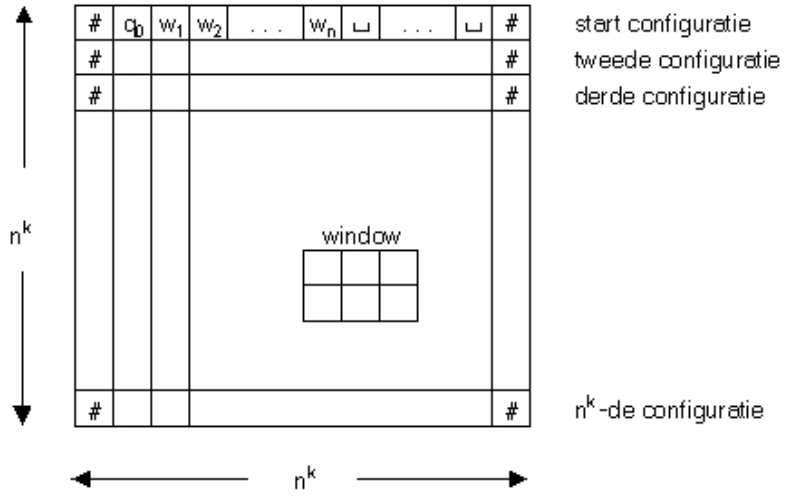
Stap 1: SAT $\in$ NP.

Door 2.2.5 moeten we een certificaat c hebben, waarvan de lengte polynomiaal in functie van de lengte van de input w is; en een verificateur v die in polynomiale tijd lidmaatschap voor SAT beslist. Als certificaat c nemen we een waarheidstoekenning aan alle variabelen uit de booleaanse expressie e. De verificateur vult e in met die waarden en kijkt wat de uiteindelijke waarde van e is. Wanneer deze true is aanvaardt v en anders verwerpt hij. De lengte van c is bij aanname polynomiaal in functie van de grootte van de input, omdat het aantal variabelen steeds kleiner of gelijk is aan de lengte van de input. Nu werkt v ook in polynomiale tijd, omdat het evalueren van een bepaalde waarheidstoekenning voor een gegeven booleaanse expressie steeds in polynomiale tijd kan gebeuren [McP03].

Stap 2: $\forall A \in$ NP: $A \leq_P$ SAT.

We nemen een willekeurig probleem $A \in$ NP. Laat nu TM de niet deterministische Turing Machine zijn die A beslist in tijd n^k, waarbij k een constante is. De polynomiale reductie van A naar SAT creëert een zogenaamd *tableau*, waarvan de breedte en de hoogte een lengte hebben van n^k. De eerste rij van het tableau zal overeenkomen met de startconfiguratie van TM. Als eender welke rij van het tableau overeenkomt met een aanvaard-configuratie,

[1] Voor een overzicht zie: http://www.nada.kth.se/~viggo/problemlist/compendium.html



Figuur 3.1: Visuele voorstelling van een tableau ([Sip96]).

noemen we het tableau *aanvaardend*. Elke overgang van een rij naar de volgende in het tableau komt overeen met de uitvoering van een stap van de transitiefunctie van TM, zodoende dat het probleem: aanvaard TM input w, overeenkomt met: bestaat er een tableau overeenkomstig met w uitvoeren op TM. Voor een visueel beeld van een tableau: zie figuur 3.1.

De polynomiale reductie functie f van A naar SAT poduceert nu een booleaanse expressie ϕ, die nakijkt of zulk een tableau bestaat. De tableau bestaat dus uit $(n^k)^2$ *cellen*. $\forall i, j$: $1 \leq i, j \leq n^k$ en $\forall s \in C = Q \cup \Gamma \cup \{\#\}$ is er een variabele $x_{i,j,s}$ in ϕ. Als $x_{i,j,s}$ *true* is wil dit zeggen dat cel$[i, j]$ symbool s bevat. De geconstrueerde expressie ϕ is een conjunctie van vier deelformules: $\phi_{cell} \wedge \phi_{start} \wedge \phi_{move} \wedge \phi_{aanvaard}$.

De formule ϕ_{cell} garandeert dat elke cel juist 1 symbool (niet meer en niet minder) bevat en ziet eruit als volgt:

$$\bigwedge_{1 \leq i,j \leq n^k} \Big[\Big(\bigvee_{s \in C} x_{i,j,s} \Big) \wedge \Big(\bigwedge_{s,t \in C \wedge s \neq t} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \Big) \Big].$$

Het eerste deel van de formule binnen de vierkante haken geeft aan dat er minstens 1 variable per cel moet zijn, waarbij het tweede deel dan aangeeft dat er maximaal 1 variabele per cel is. Laten we even de complexiteit voor ϕ_{cel} bekijken. Elk tableau heeft n^{2k} cellen, die op hun beurt elk van de symbolen uit C kunnen bevatten. Vermist het aantal symbolen in C enkel afhangt van TM en niet van n, is het totale aantal variabelen in ϕ_{cel} begrensd door $O(n^{2k})$.

De volgende deelformule is ϕ_{start}. Deze subformule kijkt simpelweg of de eerste rij van het tableau overeenkomt met de startconfiguratie van TM. Dit kunnen we dan ook eenvoudig nagaan door volgende booleaanse expressie:

$$x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}.$$

De formule ϕ_{start} heeft een fragment voor elke cel in de bovenste rij van het tableau, waardoor de complexiteit ervan begrensd is door $O(n^k)$.

Als derde formule beschouwen we $\phi_{aanvaard}$. Dit zorgt ervoor dat er zich een aanvaardende toestand in het tableau bevindt. De formule is eenvoudig:

$$\bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{aanvaard}}.$$

De formule $\phi_{aanvaard}$ heeft een vaste fragmentgrootte voor elke cel in het tableau. De complexiteit ervan is begrensd door $O(n^{2k})$.

Als laatste formule hebben we nog ϕ_{move}. Deze formule is de moeilijkste en garandeert dat elke configuratie (rij) in het tableau legaal volgt uit de vorige. Met legaal bedoelen we volgens de regels van de Turing Machine. Men zou hier kunnen stellen dat de transitie functie gevolgd moet worden. Dit doen we door een scope te definiëren die dit nakijkt. We zullen deze scope een *window noemen*. Een window is een rechthoek die 2 cellen hoog is en 3 cellen breed. Omdat we alle mogelijke overgangen kennen, kunnen we alle legale windows gaan opstellen. De inhoud van een formule noteren we als $a_1 \ldots a_6$. Alles is nu aanwezig voor ϕ_{move}:

$$\bigwedge_{1 \leq i < n^k, 1 < j < n^k} \Big(\bigvee_{a_1 \ldots a_6 - legaal - window} \phi \Big),$$

waarbij:

$$\phi = (x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}).$$

Net zoals $\phi_{aanvaard}$ heeft ϕ_{move} een vaste fragmentgrootte voor elke cel in het tableau. De complexiteit ervan is hier dus ook begrensd door $O(n^{2k})$.

We kunnen concluderen dat de totale grootte van $\phi_{cell} \wedge \phi_{start} \wedge \phi_{move} \wedge \phi_{aanvaard}$ begrensd is door $O(n^{2k})$, en dit resultaat is polynomiaal in functie van n wat aangetoond moest worden. Het resultaat is dat we een oplossing hebben slechts als de formule satisfiable is. □

3.3 Circuit SAT

Definitie 3.3.1 (Circuit SAT). *Gegeven is een circuit van AND(x,y) en NOT(x) poorten met een enkele output en zonder lussen. Is er een waarheidstoekenning aan de inputs die ervoor zorgt dat het circuit '1' output?*

De werking van een circuit is analoog aan die van de booleaanse algebra. Het signaal in een circuit wordt doorgegeven door middel van *draad*. Deze hoeft niet recht te lopen en *propageert* het signaal alzo in eender welke richting. Het signaal aan het begin van de draad (= *inputsignaal*) is hetzelfde als het signaal aan het einde ervan (= *outputsignaal*). Bij logiche circuits zijn deze signalen respectievelijk 1 en 0. Hier is de analogie met booleaanse logica duidelijk. In een circuit kan een signaal opsplitsen en twee verschillende wegen volgen. Deze laatste mogelijkheid van een circuit noemt men een *splitter* en is bijvoorbeeld handig wanneer een bepaalde variabele twee of meer keer nodig is. In principe zouden twee verschillende draden elkaar ook kunnen kruisen (zonder een lus te vormen), deze constructie is een *crossover*.

De draad kan aangesloten worden op twee zogenaamde *basispoorten*. Deze zijn analoog aan twee van de drie operatoren bij de booleaanse algebra. We hebben dus een *AND-poort* en een *NOT-poort*. De overeenkomstige waarheidstoekenningen zijn gegeven in de tabellen 3.1 en 3.3. Met andere woorden de werking ervan is dezelfde zoals gezien in het deel over de booleaanse algebra. In werkelijkheid bestaan hardware implementaties van deze poorten. Met behulp van geleidende draad is het dus mogelijk om logische circuits te construeren en de output van verschillende inputsignalen op een aantal poorten te testen.

Opmerking 3.3.1. Er is nog een derde mogelijke poort, namelijk de *OF-poort*, waarvan de werking is zoals deze bij de disjunctie (tabel 3.2). Deze kan gevormd worden door een NOT-poort achter een AND-poort te plaatsen én een NOT-poort achter de twee input-signalen, net voor deze de AND-poort bereiken. De OF-poort kan gezien worden als een basispoort, maar dit hoeft niet. Het resultaat van de omzetting van een AND-poort naar een OF-poort door middel van de negatie is te zien in tabel 3.5.

Buiten de basispoorten en de OR-poort zijn er ook nog de *afgeleide poorten*. We noemen ze afgeleide poorten omdat ze gesimuleerd kunnen worden door een aaneenschakeling van een of meerdere basispoorten. Deze zijn onder andere de XOR-poort, de NAND-poort en de NOR-poort. Elk van de ze poorten heeft twee inputsignalen en een outputsignaal. Wat het outputsignaal is bij de verschillende mogelijke inputsignalen voor deze poorten, is

ϕ_1	ϕ_2	$\neg(\neg\phi_1 \wedge \neg\phi_2)$	$\phi_1 \vee \phi_2$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

Tabel 3.5: Omzetting van AND-poort naar OF-poort.

gegeven in tabellen 3.6, 3.7 en 3.8. Hoe de constructie van de afgeleide poorten verloopt, gebaseerd op de twee basispoorten, laten we buiten beschouwing. Ook zijn er meer complexe constructies mogelijk. Deze bevatten meestal meerdere poorten tegelijkertijd. In het volgende deel zullen we de

ϕ_1	ϕ_2	XOR van ϕ_1 en ϕ_2
0	0	0
0	1	1
1	0	1
1	1	0

Tabel 3.6: Waarheidstabel XOR-poort.

ϕ_1	ϕ_2	$NAND$ van ϕ_1 en ϕ_2
0	0	1
0	1	1
1	0	1
1	1	0

Tabel 3.7: Waarheidstabel NAND-poort.

complexiteit van Circuit-SAT nader bekijken en tot de conclusie komen dat het probleem NP-compleet is.

3.3.1 Complexiteit

Eerst tonen we aan dat Circuit SAT in NP zit, waarna een polynomiale reductie vanuit SAT naar Circuit SAT geven. Gegeven deze twee dingen kunnen we dan wegens theorem 2.2.2 concluderen dat Circuit SAT NP-compleet is.

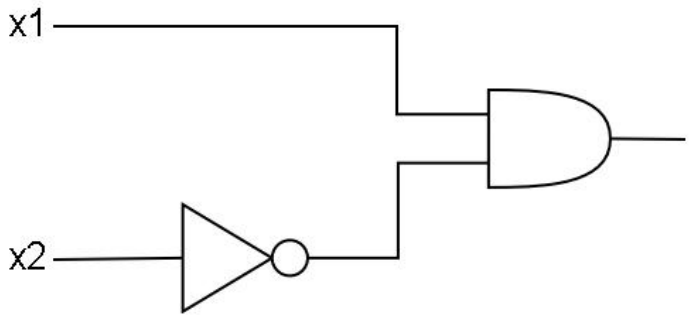
Door 2.2.5 moeten we een certificaat c hebben en een verificateur v die in polynomiale tijd lidmaatschap voor Circuit SAT beslist. Zij nu $c =$ een toekennig van waarden uit $0, 1$ aan $x_1, ..., x_n$. We hebben voor elke variabele één

ϕ_1	ϕ_2	NOR van ϕ_1 en ϕ_2
0	0	1
0	1	0
1	0	0
1	1	0

Tabel 3.8: Waarheidstabel NOR-poort.

toekenning, dus de lengte van c is polynomiaal in functie van de input w. Nu moet v lidmaadschap van Circuit SAT in polynomiale tijd beslissen. Dit kan v door het circuit te evalueren door invulling van elke variabele door de waarde van zijn toekenning. Als het circuit na de toekenning tot *true* leidt, is lidmaatschap een feit. Zowel de toekenning als de evaluatie van het circuit erna kunnen in polynomiale tijd gebeuren.

Nu hebben we nog een polynomiale reductie van SAT naar Circuit SAT nodig. Gelukkig hebben we voor elk van de drie operatoren bij SAT een overeenkomstige poort in Circuit SAT (waarbij de OR operator gesimuleerd kan worden door een AND en drie NOT poorten). We moeten het circuit dus alleen nog maar opbouwen naargelang de booleaanse expressie, waarbij we de conjuctie vervangen door de AND poort, de disjunctie door de OR poort en de negatie door de NOT poort. De verschillende booleaanse variabelen kunnen we voorstellen door een begindraad te nemen voor elke variabele. Wanneer een variabele meerdere malen voorkomt kunnen we zijn overeenkomstige draad laten splitsen (= splitter). Vermits draden over elkaar kunnen lopen, door middel van een crossover, hebben we alle nodige constructies om de simulatie te voltooien. Als een waarheidstoekenning voor het SAT probleem satisfiabel is, dan is het overeenkomstige Circuit SAT probleem ook satisfiabel, en vice versa. Een voorbeeld van zo'n omzetting is gegeven in fig. 3.2. Het is de Circuit SAT voorstelling van de booleaanse formule 3.1. De waarheidstabel voor het SAT probleem is gegeven in 3.4. Zoals je kan zien geeft het overeenkomstige Circuit SAT probleem, dat te zien is in figuur 3.2 dezelfde waarheidstabel.



Figuur 3.2: Circuit SAT voorstelling van $x_1 \wedge \neg x_2$.

Hoofdstuk 4

Puzzels en spellen

Tot nog toe hebben we alleen maar de pure theoretische kant bekeken. Namelijk wat is complexiteit en hoe hiermee om te gaan. Uiteraard vormt dit de basis om met deze begrippen te werken, maar nu gaan we onze kennis toepassen op bepaalde, meer praktische problemen. Gedurende mijn opzoekwerk heb ik de complexiteit van vele puzzels en spellen bekeken. Enkele van de interessantste komen in dit hoofdstuk aan bod. De gegeven games komen uit diverse complexiteitsklassen. Er zijn reeds vele spellen bestudeerd, waarvan een klein overzicht is te vinden in het artikel van E. Demaine [Dem01]. Het spreekt dan voor zich dat een compleet overzicht geven zo goed als onmogelijk is. Gedurende de literatuurstudie verplaatste mijn aandacht zich steeds meer naar Minesweeper. Op Minesweeper ga ik dan ook veel dieper in en worden meer aspecten bekeken. Mede dankzij een quote van R. Kaye [Kay00a] ben ik andere aspecten van Minesweeper gaan bestuderen. Zo probeer ik in latere hoofdstukken zelf de complexiteit van Minesweeper varianten te bestuderen. In sectie 6 bestuderen we Minesweeper dat gespeeld wordt op een grid met hexagonen. Later, in sectie 7 gaan we dieper in op de Minesweeper variant met regelmatige driehoeken als gridcellen. Minesweeper dat gespeeld wordt op een grid gevuld met regelmatige achthoeken komt in sectie 8.1 aan bod. Ook maken we een uitstap naar de wereld van 3D-Minesweeper (sectie 8.2). Het spreekt voor zich dat, omdat de varianten van elkaar verschillen, er misschien verschillen zijn in het eindresultaat. We zullen dieper ingaan op deze verschillen in het volgende hoofdstuk. Al de gegeven resultaten voor de varianten zijn persoonlijk werk, welliswaar gebaseerd op het onderzoek door R. Kaye voor de bekende Minesweeper variant.

Echter in dit hoofdstuk zullen verscheidene, reeds gekende, complexiteitsresultaten voor een aantal puzzels en spellen aan bod komen. Beginnen doen we met de klasse P, waarna we overgaan naar achtereenvolgens NP, PSPACE

en EXPTIME. In dit deel wordt de precieze bewijsvoering niet gegeven, er zullen enkel ideeën gegeven worden omtrent "de moeilijkheid" van het spel of de puzzel. Ook bespreken we tilings, puzzels die naargelang de variant onder beschouwing qua complexiteit tot elk van deze klassen kunnen behoren. Zo ziet men dat een puzzel/spel, door aanpassingen eraan te maken, tot een andere complexiteitsklasse kan behoren. Voor meer puzzels en spellen verwijs ik graag naar [Dem01], een klein overzicht kan gevonden worden in tabel 4.1[1]. Merk op dat het hier om problemen omtrent het spel of de puzzel handelt, en niet om het spel of de puzzel op zich. Uiteraard zijn er ook veel problemen waarvoor (nog) geen complexiteitsresultaten bekend zijn.

Zoals eerder aangehaald beginnen we met de klasse P. Voor deze klasse zullen we *Game-of-NIM* bespreken. Alvorens hiermee te beginnen, wordt er nog een algemene inleiding gegeven op Tilings zodat we met dit begrip kunnen werken wanneer nodig bij andere complexiteitsresultaten. Na P komt uiteraard NP, waar we even stilstaan bij Tetris. Vervolgens represeert Soukoban de klasse PSPACE. Afsluiten doen we met schaken, dat representatief zal zijn voor de klasse EXPTIME.

4.1 Tiling: Inleiding

Tiling, tesamen met zijn verschillende varianten, is wijd verspreid en heeft takken die rijken tot verscheidene onderzoeksdomeinen, niet enkel binnen de informatica. Degene die hier aan bod komen zijn logischerwijs de complexiteitsstudie van puzzel problemen alsook het domein van de combinatorische spellen. Het laatste situeert zich in de jaren '70 en kende vooral een doorbraak na de ontdekking van het *alternerend computatie model* (1977-1981) (zoals beschreven in [CKS81]). Andere domeinen waar Tilings hun intrede hebben gemaakt zijn onder andere de propositielogica en reducties. Veelal worden Tilings gebruikt om simpelere en meer elegante bewijzen te leveren voor reeds gekende problemen. Een voorbeeld hiervan voor de propositielogica zijn de bewijzen voor enkele compleetheidsresultaten, vermeld in [Chl86]. Enkele reducties worden gegeven in [vEB96], waaronder een reductie van *Square Tiling* (zie 4.4.1) naar *Exact Cover*.

Tilings zijn interessant om te bekijken in verband met complexiteit om wille van hun diversiteit aan compleetheidsresultaten. Naargelang de Tiling variant verkrijgt men vaak een compleetheid voor een bepaalde, andere, com-

[1] Enkele resultaten uit tabel 4.1 zijn afkomstig van http://en.wikipedia.org/wiki/Game-tree_complexity.

Complexiteit	Puzzel of spel
Klasse P	Game of NIM
	Kayles (normal play)
NP-compleet	Phutball (mate in 1)
	Tetris (leegmaken)
	Instant Insanity
	Minesweeper (consistency)
	Shanghai
SPACE-compleet	Soukoban
	HEX (generalized)
	Geography
	Othello
	Rush Hour
	Tic-Tac-Toe
EXPTIME-compleet	Dammen
	Schaken
	Capture
	Shogi
	Go
? (onbekend)	Awari
	Pentominoes
	Connect four
	Backgammon

Tabel 4.1: Overzicht van enkele puzzels en spellen en hun complexiteit.

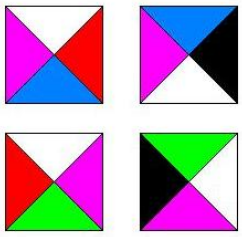
plexiteitsklasse. Op deze manier kan men enig inzicht verkrijgen in de toenemende complexiteit van problemen, in dit geval van spellen en puzzels, naargelang de aard van het probleem verandert. De theorie die in dit hoofstuk besproken wordt is grotendeels afkomstig van [vEB96] en [Chl86], andere bronnen zijn vermeld in de tekst.

4.1.1 Algemeen

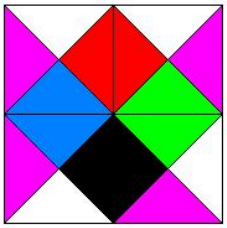
Bij Tiling gaat men een vlak vullen met tiles. In dit deel komen twee soorten vlakken aan bod, namelijk ten eerste het *vierkant* (= square) en ten tweede

de *rechthoek* (= rectangle).

Een *tile* is een vierkant dat uit vier delen bestaat. Deze delen zijn de respectievelijke driehoeken gevormd, in het vierkant, door de twee diagonalen. In Tiling heeft elk van de vier vakken van elke tile een bepaalde kleur, gekozen uit een eindige set kleuren. Dit alles samen geeft meteen ook aan tot welk *tile-type* deze tile behoort. Belangrijk hierbij op te merken is dat tiles niet mogen gereflecteerd of geroteerd worden. Als een tile tesamen met zijn beeld onder een rotatie kunnen voorkomen dan is er voor zowel de oorspronkelijke tile als voor de geroteerde tile een apart tile-type aanwezig. Fig.4.1 geeft vier voorbeelden van verschillende tile-types. In het algemeen kunnen bij tilings ook tiles gebruikt worden met andere vormen, deze komen hier niet aan bod.



Figuur 4.1: Voorbeelden van tile-types.



Figuur 4.2: Legaal getiled vlak (square).

Definitie 4.1.1 (Tile). *Een tile is van de volgende vorm <l, b, r, o> waarbij l, b, r en o respectievelijk de kleuren van de links, de bovenste, de rechtse en de onderste driehoek voorstelt. Hierbij noteren we de witte kleur door het symbool ·.*

De vlakken worden opgevuld met tiles op een *bepaalde* manier. De manier waarop de tiles moeten getiled worden is ook vast bepaald volgens regels. Die zeggen dat tiles enkel tegen elkaar gelegd mogen worden als de gemeenschappelijke zijde daardoor dezelfde kleur representeert. Dit wil zeggen de respectievelijke driehoeken van de twee tiles die de zijde in kwestie bevatten, hebben dezelfde kleur. De tiles moeten uiteraard het gehele vlak opvullen. In deze context hebben we de volgende definitie.

Definitie 4.1.2 (X-Tiling). *Zei X een oneindige set van tile-types, dan is een X-Tiling een opvulling van een vlak met tiles uit X. De tiling moet hierbij voldoen aan de tiling regels.*

Nu zijn er genoeg ingrediënten aanwezig om te komen tot de definitie van Tiling.

Definitie 4.1.3 (Tiling probleem). *Gegeven een vlak V en set van tile-types X; bestaat er een X-Tiling van V.*

Veelal worden er nog extra beperkingen gelegd op het algemene Tiling probleem. Zo bekomt men verschillende Tiling varianten, waarvan er in de volgende secties enkele besproken worden.

Zie figuur 4.2 voor een Tiling van een vierkant, gebruik makende van de 4 tile-types in figuur 4.1. Later zullen we gebruik maken van deze algemene inleiding tot Tilings om verschillende Tiling varianten en hun complexiteitsresultaten te bespreken.

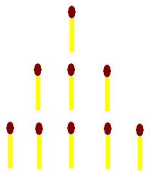
4.2 Game of NIM

NIM is een oud spel waarvan het onstaan (nog) niet met zekerheid is achterhaald. Volgens [TUN] zijn er twee belangrijke mogelijke geschiedenissen. Sommige claimen dat een zekere C.L. Bouton van de Harvard University het spel zo'n 100 jaar geleden heeft uitgevonden. Hij zou ook het algoritme om het spel optimaal te spelen hebben ontdekt. Volgens andere bronnen is NIM afgeleid van een oeroud Chinees spel waarbij twee spelers om de beurt 1, 2 of 3 knikkers nemen uit een voorafbepaald aantal knikkers. De naam zou ofkomstig zijn van het Duitse woord *nehmen*, wat *iets nemen* betekent.

Hoe het ook zij, NIM heeft een niet te verwaarlozen impact gehad op de spellenwereld. Zo vormt het onder andere de basis voor de Sprague-Grundy theorie [Dem01] voor impartial games. Het heeft ook vele varianten, waarvan er twee zullen besproken worden. Naar deze spellen wordt ook gerefereerd als NIM in vermomming omdat ze met behulp van een analoge techniek kunnen worden opgelost. Wat volgt is naar analogie met [NIM]. Zoals eerder aangehaald zullen de problemen aangaande het Game-of-NIM zich in de klasse P bevinden.

4.2.1 Spelregels

NIM wordt gespeeld met een vooraf bepaald aantal rijen, zij $n > 0$. Elk van deze n rijen bestaan op hun beurt uit een aantal voorwerpen en zij v_i het aantal voorwerpen behorende bij rij i, $0 < i \leq n$. Deze voorwerpen kunnen eender wat zijn. Veelal wordt er met lucifers gespeeld, maar het kunnen ook



Figuur 4.3: NIM-spel = {3; 1, 3, 5}.

knikkers of munten zijn. Laten we voor het verdere verloop lucifers nemen. Nu mag n een willekeurige positieve waarde aannemen, maar meestal wordt er gespeeld met $3 \leq n \leq 5$. Elke v_i is ook arbitrair, en hoeft niet dezelfde te zijn voor elke rij, hoewel dit mag. Twee spelers, noem ze E en T (van eerste en tweede), nemen om de beurt een aantal lucifers weg. De enige restrictie hierbij is dat de lucifers die door de speler aan zet weggenomen worden tot dezelfde rij moeten behoren. De winnaar is degene die de laatste lucifer weet weg te nemen. Merk op dat E steeds een winnende strategie heeft wanneer $n = 1$. Om te winnen hoeft hij enkel alle lucifers weg te nemen, vanzelfsprekend neemt hij dan ook de laatste weg. Wanneer $n > 1$ ligt het anders en is het afhankelijk van het aantal lucifers per rij, nl. de verschillende v_i, wie de winnende strategie heeft.

Definitie 4.2.1 (NIM-Spel). *Een NIM-spel is een tupple* $\{n; v_1, v_2, \ldots, v_n\}$ *met* n *het aantal rijen en* v_1 *tot en met* v_n *het aantal lucifers voor de overeenkomstige rij.*

4.2.2 Wie wint?

Dit is afhankelijk van de verschillende v_i. Om het antwoord op die vraag te weten moeten we volgend algoritme uitvoeren:

1. Bereken voor elke v_i de overeenkomstige binaire waarde.
2. Bereken de bitsgewijze XOR operatie (genoteerd door $\oplus$) van de bekomen binaire getallen uit 1. In voorbeeld 4.2.1 bekijken we deze operatie voor de decimale getallen 2, 18 en 29.
3. Als het resultaat van stap 2 overeenkomt met de decimale waarde 0, dan heeft T een winnende strategie, anders E.

Voorbeeld 4.2.1 (XOR van meerdere getallen). De bitsgewijze XOR van meerdere binaire getallen berekenen we door het aantal enen per positie in elk van deze binaire getallen op te tellen. Wanneer het bekomen resultaat even is, heeft het resultaat op de bekeken positie een 0 staan, en anders

een 1. Voor de decimale getallen 2, 18 en 29 hebben we de respectievelijke binaire waarden 00010, 10010 en 11011. In totaal zijn er zes posities. We bekijken ze van rechts naar links. Op positie één vinden we slechts één 1 terug, namelijk degene afkomstig van de het getal 6. In het resultaat hebben we dus op positie één een 1 staan. Wanneer we verder gaan naar positie twee, heeft elk decimaal getal een 1 staan in zijn binaire voorstelling. Het totaal aantal enen op positie twee is dus drie, waardoor we in het resultaat een 1 terug vinden op positie twee. Op positie drie staat in geen enkele binaire voorstelling de waarde 1, zodoende verkrijgen we een 0 op de derde positie in ons resultaat. Zo werken we verder tot en met positie zes. Het uiteindelijke binaire resultaat is dan 01011, ofwel het decimale getal elf.

Voorbeeld 4.2.2 (Bepaal wie de winnende strategie heeft voor het NIM-spel). Stel $n = 3$, $v_1 = 1$, $v_2 = 3$ en $v_3 = 5$ zoals in fig. 4.3. Dan zijn de corresponderende binaire waarden (00)1, (0)11 en 101. De XOR hiervan levert ons het binaire getal 111, wat overeenkomt met het decimale getal 7 $\neq 0$. Hieruit mogen we concluderen dat E de winnende strategie heeft.

Stelling 4.2.1. *Bepalen of E of T de winnende strategie heeft voor een bepaald NIM-spel $\in P$.*

Het algoritme is reeds gegeven. Elke stap erin is duidelijk polynomiaal in functie van de input, cfr. 4.2.1.

4.2.3 Winnende strategie

Door naar het voorgaande algoritme te kijken, zien we dat ervoor gezorgd moet worden dat je tegenspeler niet in het bezit komt van de winnende strategie. De speler aan de beurt moet er dus voor zorgen dat de XOR van alle $v_i = 0$ wordt.

Dit kan hij doen als volgt:

1. Zij m de Most Significant Bit (MSB) van het resultaat van de XOR-operatie, toegepast op alle rijen, die op 1 staat. Kies een rij r waarbij m ook op 1 staat (merk op dat dit steeds mogelijk is omdat deze MSB gevormd is door minstens één v_i). Wanneer er meerdere rijen in aanmerking komen staat het vrij te kiezen.

2. Bereken de bitsgewijze XOR voor alle v_i, $0 < i \leq n$, $i \neq r$ en noem het resultaat x_{or}.

3. Verwijder r - x_{or} lucifers uit rij r.

Voorbeeld 4.2.3 (Bepaal de winnende strategie). Zij het spel gedefinieerd zoals in 4.2.2. Dan zijn de corresponderende binaire waarden (00)1, (0)11 en 101. De XOR hiervan leverde binaire getal 111. De MSB die op 1 staat is de eerste 1. De rij verantwoordelijk voor deze bit is de rij met 5 lucifers (binair: 101). Nu is x_{or} gelijk aan 001 (=1) $\oplus$ 011 (=3) = 010 (=2). De winnende strategie is dus om 5 - 2 = 3 lucifers weg te nemen in de rij waarin er oorspronkelijk 5 lagen. Zie dat er nu nog respectievelijk 1, 3 en 2 lucifers per rij liggen en dat $1 \oplus 3 \oplus 2 = 0$.

Stelling 4.2.2. *De winnende strategie voor een NIM-spel bepalen $\in P$.*

Het algoritme is reeds gegeven. Elke stap erin is duidelijk polynomiaal in functie van de input, cfr. 4.2.1.

4.2.4 Correctheid

Het bewijs zal gegeven worden voor het NIM-spel waarbij $n = 3$ om de uitleg bondig te houden. Een veralgemening is rechttoe rechtaan en kan worden herkend in het bewijs. Om al het voorgaande aan te tonen moeten we twee dingen aantonen:

1. Als de XOR toegepast op alle rijen gelijk is aan 0 zal de speler aan zet het spel in zo'n staat laten dat de totale XOR verschillend is van 0.

2. Als de XOR toegepast op alle rijen niet gelijk is aan 0 dan kan de speler aan zet het spel in zo'n staat laten dat de totale XOR gelijk is aan 0.

Merk op dat deze twee regels voldoende zijn om de winnende strategie in je bezit te houden wanneer je hem eenmaal hebt.

Bewijs. 1. Veronderstel dat $a \oplus b \oplus c = 0$ en dat de speler aan zet c' lucifers wegneemt in rij c (analoog voor andere rijen). Als hij door een zet te doen het spel in de staat kan laten zodat de totale XOR nog steeds 0 is dan is $a \oplus b \oplus (c - c') = 0$. Nu is het eerste gelijkwaardig met zeggen dan $a \oplus b = c$. Op dezelfde manier hebben we voor het tweede dat $a \oplus b = c - c'$. Hieruit volgt dat $c = c - c'$, wat overeenkomt met zeggen dat $c' = 0$. Dit houdt in dat de speler aan de beurt geen lucifer weggenomen zou hebben. Dit is uiteraard geen legale zet.

2. Veronderstel dat $a \oplus b \oplus c = x$, $x \neq 0$. Neem nu een van de rijen die voor de MSB van x zorgde (deze bestaat altijd vermits $x \neq 0$). Laten we het getal dat overeenkomt met deze bit y noemen. Stel zonder verlies van

veralgemening dat de rij die voor de MSB van x zorgt c is. De vraag is hoeveel lucifers we nu uit c moeten verwijderen? Neem er $c - (a \oplus b)$. Dan zal rij c nog $c - (c - (a \oplus b)) = a \oplus b$ lucifers bevatten. Hierdoor wordt de totale $XOR = a \oplus b \oplus (a \oplus b) = 0$.

Er resten enkel nog twee kleinigheidjes aan te tonen, nl. dat $c-(a \oplus b) \leq c$ en dat $c - (a \oplus b) > 0$. Het eerste geldt omdat er anders een negatief aantal lucifers zouden overblijven, het bewijs is echter triviaal. Het tweede omdat er wel degelijk een lucifer moet weggedaan worden in de beurt. Zij z gelijk aan het binaire getal gevormd door de bits van c die wegvallen t.o.v. $a \oplus b$ door de XOR-operatie. Dan is $c - z = y$. Ook is $y > (a \oplus b) - z$ waardoor $c - z > (a \oplus b) - z$ ofwel $c - (a \oplus b) > 0$. □

We kunnen concluderen dat het probleem omtrent het NIM-spel, namelijk bepalen wie de winnende strategie heeft, zich in de complexiteitsklasse P bevindt.

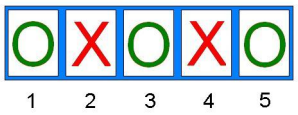
4.2.5 Varianten

Er zijn vele spellen die hun oorsprong vinden bij het NIM-spel. Vaak hebben ze een analoge strategie om het spel te winnen en om te bepalen wie het spel kan winnen. Het spreekt van zelf dat deze spellen ook bestudeerd zijn en hun functie hebben in de spellenwereld. In deze paragraaf zullen de varianten Turning Turtles en Bogus NIM besproken worden.

Turning Turtles

Bij Turning Turtles (TT) is het de bedoeling om een rij plaatjes, waarop zich aan de ene kant een O bevindt en aan de andere kant een X, om te draaien tot er alleen X'en overblijven (zie fig. 4.4). Degene die het laatste plaatje weet om te draaien wint. In zijn beurt mag een speler twee dingen doen. Eerst mag hij elke X omvormen tot een O. Hierna heeft hij in dezelfde beurt de keuze om één of geen plaatje, links van het gekozen plaatje, om te draaien. Hierbij maakt niet uit of het gaat om een plaatje met O of met een X op.

Dit is een NIM-spel in vermomming. Vanuit TT kunnen we een NIM-spel opbouwen door voor elke O een rij te creeëren met lengte gelijk aan de positie van het plaatje met de O op, gezien van links naar rechts, te beginnen bij 1. Nu kan men bepalen hoeveel lucifers er moeten verdwijnen in welke rij in het NIM-spel, maar hoe simuleren we dat in het oorspronkelijke TT-spel? Dit



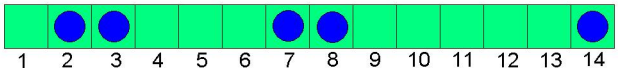
Figuur 4.4: TT-spel overeenkomstig met het NIM-spel uit fig 4.3.

blijkt eenvoudig te zijn. Stel dat we in rij n, l lucifers moeten wegnemen. Dan imiteren we dit in TT door het plaatje op positie n om te draaien (dit is zeker een O) gevolgd door het draaien van plaatje op positie $n-l$ ($n-l < n$). Het maakt niet uit of er op positie $n-l$ een O of een X staat.

Bogus NIM

Bij Bogus NIM (BN) ligt de opzet van het spel iets anders dan bij TT. Deze keer is het de bedoeling om dam stukken (munten en dergelijken kunnen ook) te verleggen naar links totdat er geen zet meer gedaan kan worden. De speler die als laatste een damstuk kan verplaatsen is de winnaar. De damstukken liggen op een lijn met posities genummerd van 1 tot en met de positie van het laatste damstuk. Een stuk mag naar links bewogen worden zover men wil, het mag niet op of over een ander damstuk geplaatst worden. Vanzelfsprekend mag het ook niet verder dan positie 1 gelegd worden, omdat het dan uit het spel verdwijnt.

Hoe kan men het NIM-spel in BN herkennen? Hiervoor beginnen we met het meest rechtse stuk. Het aantal open posities tot aan het volgende (linker) stuk is dan de lengte van de eerste rij in het oorspronkelijke NIM-spel. We slaan vanaf dan om de beurt een opening tussen twee opeenvolgende stukken over, afgewisseld met het aanmaken van een rij in NIM. Voor elke oneven opening met grootte g, beginnende van rechts, is er dus een rij van lengte g. Wanneer er een oneven aantal stukken zijn, heeft de laatste rij dus een lengte gelijk aan de positie van het meest linkse stuk. Er zijn twee belangrijke opmerking die men hierbij moet maken. Ten eerste vormen de even openingen in BN geen rijen in het oorspronkelijke NIM spel. Hierdoor heeft het verplaatsen van een stuk in BN slechts invloed op één rij in het overeenkomstige NIM spel, en niet op twee zoals men zou kunnen denken. Ten tweede is het zo dat een speler een rij in het oorspronkelijke NIM-spel kan vergroten door het juiste stuk s te verplaatsen in het Bogus spel. Dit echter kan gecounterd worden door de andere speler door het stuk rechts van s over dezelfde afstand te plaatsen!



Figuur 4.5: Bogus NIM-spel overeenkomstig met het NIM-spel uit fig 4.3.

Het Bogus NIM-spel in fig. 4.5 bootst het NIM-spel = {3; 1, 3, 5} na. Hier zijn drie oneven openingen, namelijk de posities {1}, {4,5,6} en {9,10,11,12} met lengte respectievelijk 1, 3 en 5. De twee even posities hebben lengte = 0 (ze bevinden zich respectievelijk tussen de stukken 2 en 3 en tussen de stukken 7 en 8).

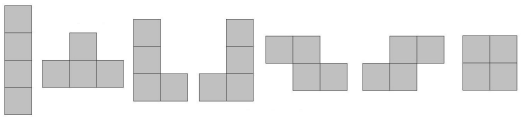
Uiteindelijk blijken zowel het probleem omtrent NIM als de twee besproken varianten, Turning Turtles en Bogus NIM, in de klasse P te zitten.

4.3 Tetris

4.3.1 Inleiding

Tetris dankt zijn naam aan het feit dat elk speelblokje uit vier (in het Latijn tetra) deelblokjes (pixels) bestaat. Het spel is rond 1985 onstaan in Rusland en het duurde niet lang voor het ver buiten de grenzen van Rusland bekend raakte. Doorheen de tijd werd het voor bijna elke spelconsole geïmplementeerd en Tetris wordt momenteel door miljoenen mensen gespeeld. Vele varianten zagen ondertussen het levenslicht, maar we focussen ons hier enkel op het originele spel.

Tetris wordt gespeeld met zeven basisblokken, elk bestaande uit vier vierkanten van steeds dezelfde grootte. De 7 basisblokken zijn respectievelijk de I-vorm ('I'), de T-vorm ('T') , het rechter geweer ('L'), het linker geweer ('J'), de linker slang ('Z'), de rechter slang ('S') en het vierkant ('O'). Zie figuur 4.6



Figuur 4.6: De 7 mogelijke tetris blokken.

Beginnende bij een al dan niet gedeeltelijk gevuld 2-dimensionaal bord (verdeeld in vakjes zo groot als de pixels van de blokken) vallen de blokken één voor één naar beneden in een willekeurige volgorde. Hierbij kan elk stuk geroteerd en getransleerd worden tot het op een reeds ingevulde pixel of de bodem stuit. In de meeste implementaties is daarna nog een laatste horizontale translatie mogelijk (indien er vanzelfsprekend geen ingevulde pixels deze translatie verhinderen). Wanneer men een rij van het bord volledig met pixels heeft gevuld, scoort men wat men noemt een *lijn*. Deze lijn verdwijnt dan, en alle pixels boven deze lijn zakken een rij. Merk op dat men meerdere, maximum vier, lijnen tegelijk kan vormen; deze verdwijnen dan ook gelijktijdig. Door het verdwijnen van lijnen kan het ook voorkomen dat bepaalde pixels lijken te zweven in het 2-dimensionale grid. Het spel is verloren wanneer een blok niet meer kan vallen vanuit zijn startpositie (=steeds midden bovenaan het bord) omdat het geblokkeerd wordt door pixels afkomstig van eerder geplaatste blokken. Deze versie van Tetris wordt ook de 'offline' versie van Tetris genoemd.

4.3.2 Het probleem

Gegeven Een initieel leeg spelbord en een eindige sequentie van Tetris blokken.

Gevraagde Kan men de Tetris-blokken, voorzien in de gegeven volgorde, transleren en roteren zodat het speelbord leeg wordt achtergelaten, en waarbij de laatste blok de laatste holte opvult.

Om dit probleem aan te pakken, moeten we enkele aanpassingen doen aan de originele regels. Zo zal er geen beperking liggen op de hoogte h en zal de breedte $b \geq 4$ zijn. Ook geldt dat elk blok begint te vallen midden bovenaan het bord. Voor de rest gelden de regels zoals eerder beschreven. De onbeperkte hoogte is nodig, zodat we genoeg tijd hebben om elk Tetrisblok op de juiste manier te transleren en roteren alvorens het de constructie bereikt. Een breedte van ten minste vier in nodig om eender welke pixelrij te kunnen construeren (zie verder).

Uit definitie 2.2.15 in hoofdstuk 2 weten we dat, om aan te tonen dat het Tetris-probleem NP-compleet is, er aan 2 voorwaarden moet voldaan zijn:

1. Tetris zit in de complexiteitsklasse NP,
2. Er is een NP-compleet probleem B zodat B $\leq_P$ Tetris.

Definitie 4.3.1 (3-Partition). ***Gegeven**: Een sequentie $a_1, a_2, \ldots, a_{3s}$ van positieve natuurlijke getallen en een positief natuurlijk getal T, en dit zodat: $T/4 < a_i < T/2$ $\forall i : 1 \leq i \leq 3s$ en zodat: $\sum_{i=1}^{3s} a_i = sT$. **Gevraagd**: Kan $\{1,2,\ldots,3s\}$ verdeeld worden in s disjointe subsets $A_1, A_2, \ldots, A_s$ zodat $\forall j : 1 \leq j \leq s$ er geldt dat $\sum_{a_i \in A_j} a_i = T$.*

We hebben dus met andere woorden een bepaald natuurlijke getal T en een drievoud aan natuurlijke getallen (noem dit aantal i en noem $s = i \ / \ 3$), allen gelegen tussen $T/4$ en $T/2$. De voorwaarde die moet gelden is dat de som van alle i getallen gelijk is aan s$\times T$. Het 3-Partition probleem vraagt of al deze i natuurlijke getallen in s subsets kunnen worden verdeeld, waarbij geen enkel getal i twee keer voorkomt én waarbij de som van de getallen in de verschillende subsets steeds gelijk aan T is.

Voorbeeld 4.3.1 (3-Partition). Neem het triviale geval waarbij $a_1 = 9, a_2 = 9, a_3 = 10, a_4 = 11, a_5 = 12, a_6 = 13$ en $T = 32$. We moeten nu $6 \ / \ 3 = 2$ subsets vinden, met allemaal verschillende a_i, $1 \leq i \leq 6$, waarvoor

de som telkens 32 is. Dit kunnen we doen: neem de subsets $A_1 = \{9, 10, 13\}$ en $A_2 = \{9, 11, 12\}$.

Stelling 4.3.1. *(Carey & Johnson [3, p99]) 3-Partition is NP-compleet.*

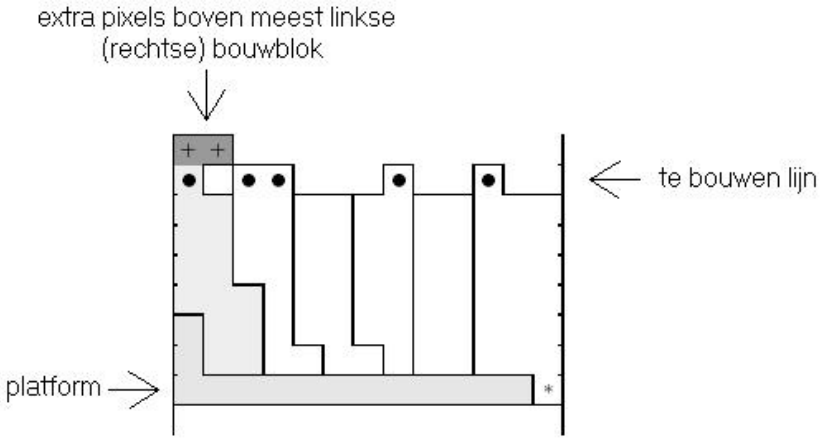
Men toont aan dat Tetris (leegmaakprobleem) NP-Compleet is door middel van een reductie van 3-Partition naar Tetris [BHK03]. Hiervoor gebruikt men een Tetris-constructie die het 3-Partition probleem simuleert. Hoogeboom en Kosters [HK04] toonden namelijk aan dat het mogelijk is om elke willekeurige Tetris configuratie te bouwen, vertrekkende van een leeg grid. De enige beperking die hier geldt is dat er geen lege of volle lijn kan voorkomen (logischerwijze). Het wordt nog mooier als ze ook daadwerkelijk het algoritme geven om een willekeurige constructie te construeren. We zullen hier de hoofdideeën voor deze constructie geven.

Constructie

De constructie werkt rij per rij, dit wil zeggen dat men de nodige *pixels* per rij opvult. Om dit te kunnen doen vertrekt men van een platform. Dit platform is een volle lijn die een pixel mist langs één kant, anders zou de lijn 'verdwijnen'. Merk op dat de kant er niet toe doet omdat we voor elk Tetris-blok ook zijn spiegelbeeld hebben en de constructie dus ook in spiegelbeeld uitgevoerd kan worden. Ook heeft het langs de andere kant een bepaalde hoogte < 4 die de *state* genoemd wordt. Deze is nodig omdat het aantal ingevulde pixels modulo 4 maximum vier kan zijn. De reden hiervoor is dat elk Tetris blok vier pixels toevoegt. Het algoritme beschouwt elke mogelijkheid om een platform te bouwen op eender welke *ondergrond*. Een ondergrond is een rij van een of meerdere ingevulde pixels. In het begin is dit aantal gelijk aan nul, omdat er nog geen blokken gevallen zijn en we starten met een leeg grid. Later kan dit nooit meer nul worden omdat de rij (lijn) dan zou verdwijnen. Met andere woorden, we hebben altijd een ondergrond om op te bouwen! Voor voor elke mogelijke ondergrond, en elke mogelijke breedte ≥ 4 kan een platform geconstrueerd worden.

Vanop het platform wordt de volgende pixelreeks (rij) gebouwd. Dit doormiddel van bouwblokken met breedte twee, met eventueel tot drie pixels onderaan in de derde kolom, omdat anders niet alle mogelijkheden geconstrueerd kunnen worden. Ook kunnen er bij het meest linkse (of de meest rechtse naargelang het platform) tot drie extra pixels bovenop het bouwblok komen. De nood hieraan is er omdat het aantal aanwezige pixels steeds een veelvoud van vier moet zijn, modulo de breedte. De totale hoogte van de constructie om de volgende pixelrij te bereiden is zes. Ook hier worden alle

mogelijkheden uitgepluisd en wordt voor elke mogelijke pixelrij een constructie gegeven. Zodoende kan men op het platform, door middel van zes extra rijen, elke pixelrij genereren (buiten de lege en de volle, zoals reeds eerder aangehaald). Voor een visueel beeld zie figuur 4.7.



Figuur 4.7: Visueel beeld Tetris constructie ([HK04]).

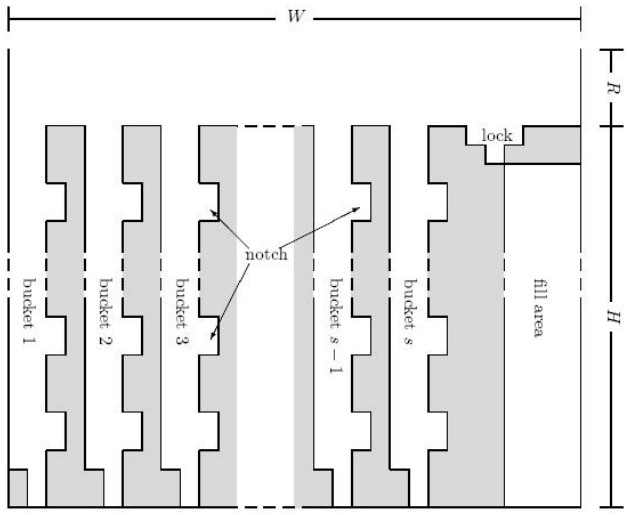
Natuurlijk zijn er nog enkele kleine puntjes waar aandacht aan besteed moet worden, zoals het wissen van de laatste overflow blokken, maar dit gaat ook zonder problemen. We weten nu dat elke Tetris configuratie kan geconstrueerd worden, we gebruiken dit om de reductie te realiseren. Merk op dat het in de geschiedenis anders is verlopen. Toen ging men ervan uit dat de constructie die ze gaven voor de reductie ook daadwerkelijk mogelijk was, zonder dit aan te tonen. Hier geven we ook weer enkel de hoofdideeën. De gebruikte variabelen s en T zijn afkomstig van definitie 4.3.1.

Constructie

Het is de bedoeling om 3-Partition te simuleren door een Tetris configuratie. Deze ziet eruit zoals te zien is in figuur 4.8. In de figuur is R de ruimte nodig om een blok te roteren en transleren als het naar beneden valt. R moet groot genoeg zijn. W is de breedt van het Tetris grid en is gelijk aan $4s + 6$. Als laatste is er nog de H die de hoogte van de buckets weergeeft. Deze moet gelijk zijn aan $5T + 18$.

Voor elke subset uit 3-Partition hebben we een bucket in Tetris, dus in totaal zijn er s buckets. Elke bucket heeft $T + 3$ *nissen* aan de rechterkant. Ook is er een *slot* (T gebied rechts boven) aanwezig dat zal verzekeren dat alle

blokken die de buckets moeten vullen, gevallen zijn voordat de lijnen kunnen gewist worden. De reden dat er hiervoor geen lijnen kunnen gewist worden is dat het *opvulgebied* (rechts) leeg is, waardoor elke rij tot boven minstens een pixel oningevuld heeft. Let wel op dat het opvulgebied een hoogte van H - 2 moet hebben en dat het opvulbaar moet zijn bij elke reductie, zijn vorm heeft echter weinig belang.



Figuur 4.8: Visueel beeld Tetris reductie ([BHK03]).

Volgorde van de blokken

De sequentie van blokken die het 3-Partition probleem simuleren heeft de volgende volgorde:

1. $\forall a_i \in A$ hebben we:
   (a) 'L' (=beginopvulling),
   (b) a_i keer ['O', 'J', 'O'],
   (c) 'O' gevolgd door 'I' (=eindopvulling).
2. s keer 'L' om de bucket alle buckets af te sluiten (dit mag niet eerder gebeuren!),
3. een 'T' blok om het slot te openen. Merk op dat de eerste lijnen kunnen nu pas verdwijnen,

4. $5T + 16$ keer 'I' om het opvulgebied te vullen. Dit is de sequentie voor ons opvulgebied, als we een ander opvulgebied hadden gekozen had deze sequentie anders moeten zijn en moeten voldoen aan de eerder opgegeven voorwaarden.

Het bewijs toont aan dat enkel de gegeven sequentie de constructie 'leeg' kan maken. Ook toont het aan dat een *ja* (*nee*) antwoord op 3-Partition ook een *ja* (*nee*) antwoord geeft bij de gereduceerde Tetris construcite en vice versa. Dit alles is wat aangetoond moet worden. Ook kan de constructie in polynomiale tijd gebeuren. Conclusie: Tetris is NP-compleet.

Andere problemen

De reductie hoeft niet alle Tetris blokken te gebruiken. Men heeft dus enkel voor bepaalde subsets van Tetris blokken de reductie gedaan. Zodoende is Tetris met een welbepaalde subset van blokken uit de origenele Tetris puzzel ook NP-compleet.

4.4 Square Tiling Puzzel

Definitie 4.4.1 (Square Tiling puzzel probleem). *Gegeven een set van tile-types X en een vierkant V, waarvan de zijden een gegeven kleuring hebben; bestaat er een X-Tiling van V: en dit zodat de kleur van elke zijde op de plaats waar hij aan een tile t grenst, dezelfde is als de kleur van de kant van t die tegen die zijde ligt.*

Eerst zal er aangetoond worden dat het Square Tiling puzzel probleem tot de complexiteitsklasse NP behoort. Daarna zal er, via een constructie, aangetoond worden dat het probleem compleet is voor deze klasse.

Stelling 4.4.1. *Square Tiling puzzel probleem $\in$ NP.*

Bewijs. Als input voor de verifier geven we het vierkant V tesamen met zijn kleuring, alsook een set X van tile-types. De lidkaart is dan de X-Tiling van V. De verifier kan nu in polynomiale tijd checken of de gegeven tiling een legale X-Tiling van V is, door te controleren dat aan de Tiling regels voldaan is. Hierdoor weten we dat Square Tiling puzzel probleem $\in$ NP. □

De compleetheid van het Tiling probleem in kwestie wordt aangetoond door middel van een reductie. Men kan de loop van niet-deterministische single tape Turing Machines simuleren door varianten (zie ook verder) van Tiling problemen.

4.4.1 Reductie

Het handige aan Tilings en TM's, of toch de ruimte-tijd diagrammen die ermee overeenkomen, is dat er een rechtstreekse vertaling mogelijk is van de laatste naar de eerste. Bepaalde klassen van TM's (deterministisch, niet-deterministisch, alternerend,...) kunnen gesimuleerd worden door verschillende versies van het Tiling-probleem. Laten we deze simulatie noteren door $X(M)$, deze wordt volledig bepaald door de Turing Machine die het simuleert. Hierdoor bekomt men de compleetheidsresultaten.

Opmerking 4.4.1 (Vertaling). In het verdere verloop zal de ruimte van het ruimte-tijdsdiagram, overeenkomstig met de TM, horizontaal lopen: in het bijzonder van links naar rechts. Analoog zal de tijd verticaal lopen, van boven naar beneden toe. Dit zoals aangegeven in figuur 4.11.

De vertaling zal gegeven worden aan de hand van een welbepaalde constructie. Peter van Emde Boas [vEB96], op wie de gegeven constructie gebaseerd is, verwijst hiervoor naar [Rob71] en [Wan60] voor de originele ideeën. Het gaat erom om opeenvolgende configuraties van een TM T de coderen door opeenvolgende horizontale lijnen bij Tiling. Door te stellen dat de initiële configuratie van T met input w gecodeerd wordt door de bovenste zijde van vlak V; en analoog de onderste zijde van V de accept-configuratie codeert, bekomt men de reductie. Er moet nog wel vermeld worden hoe deze reductie nu precies in zijn werk gaat. De grote lijnen zijn reeds gekend. Zij K en Σ het alfabet voor respectievelijk de 'state' en de 'tape' symbolen, zoals gedefinieerd in 2.1.4; dan worden de kleuren voor de bovenste en onderste zijde van elke tile gekozen uit de set $\Sigma \cup (K \times \Sigma)$. De links en rechtse kleur van de elke tile worden gekozen uit $K \cup \{\cdot\}$. Verder zullen staten worden voorgesteld door de letters p en q en een symbool door s. Ook zal een tile niet steeds met kleuren worden voorgesteld, maar zullen de kleuren vervangen worden door een combinatie van staten en symbolen. Dus voor elk van de verschillende mogelijkheden komt in het werkelijke Tiling spel een kleur uit de eindige set van mogelijke kleuren overeen.

Nu zijn er drie grote categoriën van tile-types in $X(M)$, die moeten geconstrueerd worden om de T te simuleren:

1. Voor elk van de symbolen $s \in \Sigma$, en voor het blank symbool $\sqcup$, hebben we een tile in $X(M)$ van het type $<\cdot, s, \cdot, s>$.

   Merk op dat er bij de overgang van de ene configuratie in T naar een andere, het gros van de tijd niets gebeurd. Een tape symbool blijft meestal gewoon behouden naar de volgende configuratie toe.

2. In puntje 3 worden de tile-types gegeven voor de werkelijke instructie van T. Omdat deze reeds aangeeft of de head in de volgende configuratie op dezelde positie blijft of naar links/rechts beweegt, moet er ook een tile-type aanwezig zijn die reeds aangeeft of het overeenkomstige symbool s in de volgende stap gelezen en/of veranderd gaat worden. Zodoende kan een tile uit puntje 3 de positie van de head in de volgende configuratie vast bepalen. De tile-types die hiervoor aanwezig zijn in $X(M)$ worden gegeven door $<\cdot, s, q, (q, s)>$ en $<q, s, \cdot, (q, s)>$ voor de beweging van de head naar respectievelijk rechts en links. Merk op dat dit voor problemen kan zorgen. Wanneer deze twee tiles naast elkaar worden geplaatst, is het zo dat waar eerst twee tape symbolen aanwezig zijn, er in de volgende configuratie plots twee leeskoppen aanwezig kunnen zijn. Dit gedrag willen we echter niet simuleren en zal verholpen worden door de restrictie dat met elke state p in T een richting geassocieerd wordt. Het is dus zo dat een programma niet tegelijk het tuppel $(\_, \_, q', \_, R)$ en het tuppel $(\_, \_, q', \_, L)$ kan bevatten ($\_$ is een willekeurig symbool van het overeenkomstige type). Men kan dus niet door tegelijk naar links en naar rechts te gaan vanuit een verschillend symbool, in eenzelfde staat q' terechtkomen.

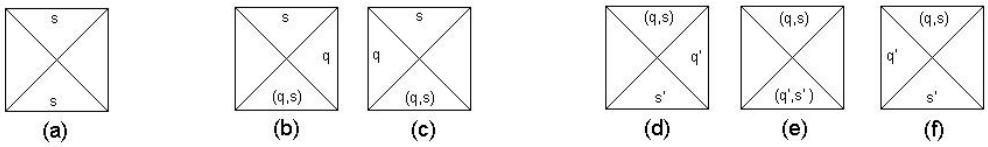
   Voor elk paar $(q, s) \in K \times \Sigma$ is er een van de twee bovenstaande tile-types in $X(M)$. Deze zijn uiteraard afhankelijk van de richting.

3. Zoals eerder aangegeven stellen de laatste soort tile-types de instructies van T voor. De instructies (q, s, q', s', R), (q, s, q', s', L) en $(q, s, q', s', \perp)$ worden gecodeerd door respectievelijk tiles van het type $<\cdot, (q, s), q', s'>$, $<\cdot, (q, s), q', s'>$ en $<\cdot, (q, s), \cdot, (q', s')>$. Uit de restrictie in puntje 2 weten we dat deze eerste twee niet tegelijk kunnen voorkomen. Hierdoor kan het dus niet voorkomen dat twee leeskoppen beiden verdwijnen en in de volgende configuratie uit twee tape symbolen bestaan.

   Als laatste worden dergelijke tile-types voor elke instructie in P toegevoegd aan $X(M)$.

De gehele conversie van het ruimte-tijd diagram van een TM T naar Tiling is nu gegeven, alle mogelijkheden worden in de vorige drie puntjes gevat. Alle overeenkomstige tile-types zijn gegeven in figuur 4.9. Het lege quadrant steld de witte kleur '$\cdot$' voor. De tile-types uit puntje 1, 2 en 3 zijn respectievelijk van de vorm (a); (b) en (c); (d),(e) en (f).

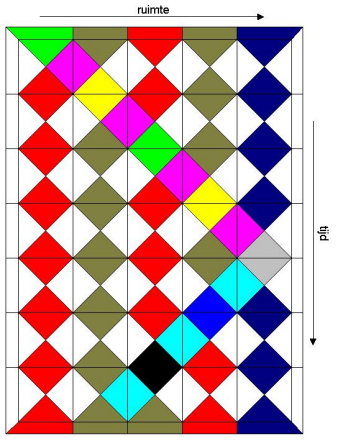
Voorbeeld 4.4.1 ($5 + 1 = 6$). Hoe een ruimte-tijd diagram van een TM kan omgezet worden naar een Tiling, zie figuur 4.10 voor de TM-computatie



Figuur 4.9: De soorten tile-types om TM te simuleren ([vEB96]).

(q,0)	1	0	1	␣
0	(q,1)	0	1	␣
0	1	(q,0)	1	␣
0	1	0	(q,1)	␣
0	1	0	1	(q,␣)
0	1	0	(p,1)	␣
0	1	(p,0)	0	␣
0	1	1	0	␣

Figuur 4.10: TM-diagram: $5 + 1 = 6$.



Figuur 4.11: Tiling: $5 + 1 = 6$.

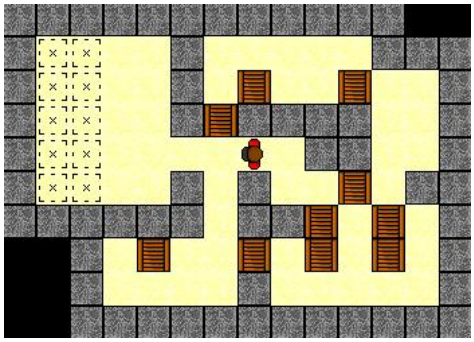
en figuur 4.11 voor de overeenkomstige Tiling. Hierin wordt de computatie van *5 + 1 = 6* gegeven. Dit is een variant op het bekende *11 + 1 = 12* voorbeeld, dat als leerinstrument gebruikt wordt aan de Universiteit van Amsterdam.

Stel nu dat we een taal $T \in$ NP hebben, dan is er ook een overeenkomstige niet-determinischtiche single tape Turing Machine M die T herkent in tijd n^k, $k \in N$. Dan verloopt de constructie als volgt. Gegeven een woord w in het alfabet van M, construeer dan een set van tile-types X_w zoals net beschreven, en wel zo dat $w \in T$ als en slechts als er een X_w-tiled vierkant is met aantal rijen en kolommen gelijk aan $|w|^k$.

We kunnen dus over het Square Tiling puzzel probleem denken als dat de tijd en ruimte beperkt zijn door de looptijd van de DTM die het simuleert. Als we daarbij denken dat die polynomiaal is in functie van de input, begrijpen we het verband met NP-compleetheid.

4.5 Soukoban

Soukoban, het Japanse woord voor *magazijnier*, is een spel dat vele varianten en implementaties kent. Bij deze puzzel, die overigens op een grid gespeeld wordt, is het de bedoeling om de magazijnier paketten naar het magazijn te laten verplaatsen. De magazijnier kan enkel de paketten verplaatsen en muren zijn opstakels waar niets door kan. Naargelang de variant kan de magazijnier een aantal paketten duwen of trekken. Dit aantal mag ook gelijk zijn aan nul. Ook naargelang de variant hebben paketten een bepaalde grootte en moeten de ze al dan niet op een bepaalde positie in het grid achtergelaten worden. Bij de gewone variant kan de magazijnier een paket duwen (niet meerdere), maar aan geen enkel paket trekken, deze variant wordt vaak Soukoban(1,0) genoemd. Een andere beperking kan zijn dat de magazijnier na afloop ook op een welbepaalde positie terecht moet komen, dit is echter meestal niet het geval. Voor een Soukoban voorbeeld zie figuur 4.12[2]. Wanneer we een '+' als superscript plaatsen bij Soukoban, handelt het over een variant waarbij paketten van de vorm 1×2 verplaatst dienen te worden.



Figuur 4.12: Soukoban voorbeeld puzzel.

Eerst en vooral enkele definities om de verschillende varianten van Soukoban van elkaar te onderscheiden.

Definitie 4.5.1 (Soukoban$^+$(∞,1)). *Het speciale aan Soukoban$^+$(∞,1) is dat het met 1×2 paketten gespeeld wordt. De magazijnier heeft de mogelijkheid om 2 (of meer als we willen) paketten tegelijk te duwen, maar hij/zij kan slechts aan 1 tegelijk trekken.*

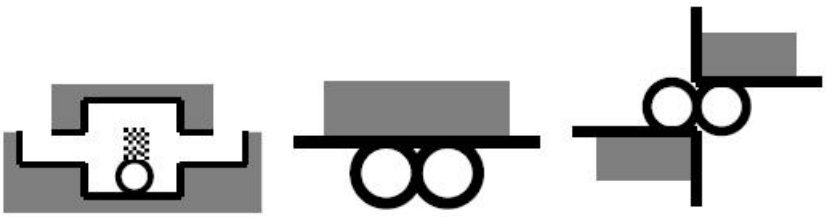
Definitie 4.5.2 (Soukoban(∞,1)). *De magazijnier heeft de mogelijkheid om alle paketten tegelijk vooruit te duwen, en aan 1 te trekken. De paketten zijn van de vorm 1×1.*

[2]Level 2 van url: http://www.pimpernel.com/sokoban/sokoban.html

Definitie 4.5.3 (Soukoban(1,0)). *Dit is de originele Soukoban versie waarbij er slechts 1 paket tegelijkertijd kan geduwd worden en waarbij aan geen enkel paket getrokken kan worden. De paketten hebben grootte 1 × 1.*

Opmerking 4.5.1 (Duwen van paketten). Als de mogelijkheid geboden wordt om 2 of meerdere paketten tegelijk te duwen/trekken dan wil dit zeggen dat ze ook in mekaars verlengde moeten liggen.

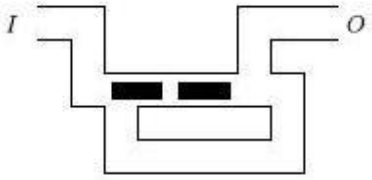
Voor de verschillende varianten van Soukoban zijn reeds complexiteits resultaten gekend. Een van de eerste bekende resultaten worden gegeven in [DZ99], en worden vermeld in de volgende paragrafen. De regels die in de verschillende bronnen gebruikt worden, verschillen lichtjes. Het belang hiervan is echter ondermaats (kan gemakkelijk aangepast worden) en hier zal dan ook niet naar gekeken worden. Veelal maakt men gebruik van onherstelbare posities. Dit zijn posities van waaruit de paketten niet meer kunnen verplaatst worden. Enkele van deze onherstelbare posities voor Soukoban(1,0) worden gegeven in figuur 4.13. Hierin, en in figuren 4.16 en 4.17, worden paketten die reeds op hun eindpositie staan, voorgesteld door volle bollen. Lichte bollen stellen paketten voor die nog niet in hun eindpositie staan. In dat geval wordt de eventuele eindpositie weergegeven door een finish-vlag patroon. De zwarte lijnen representeren muren en alleen in het witte gebied kan de magazijnier komen.



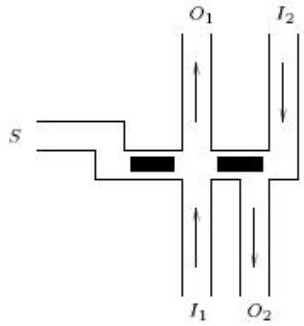
Figuur 4.13: Drie onherstelbare configuraties voor Soukoban(1,0) ([Cul97]).

In [DZ99] toont men eerst de PSPACE-compleetheid van Soukoban$^+(\infty,1)$ aan, en dit door een TM met een vaste tape lengte te simuleren door Soukoban$^+(\infty,1)$. Een analoge techniek, maar dan voor de simulatie van SAT zal gegeven worden in het hoofdstuk aangaande Minesweeper. In figuren 4.14 en 4.15 en zien we twee basis gadgets die nodig zijn voor de simulatie. We leggen ze kort even uit:

- Figuur 4.14 representeert een *eenrichtingsstraat*. Hier kan de magazijnier slechts in 1 richting door, niet in de andere, en dit zo vaak hij



Figuur 4.14: Soukoban$^+(\infty,1)$: Eenrichtingsstraat ([DZ99]).



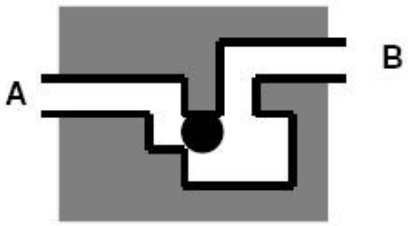
Figuur 4.15: Soukoban$^+(\infty,1)$: Schuivende deur ([DZ99]).

maar wil. Als hij binnenkomt moet hij de twee paketten een positie naar rechts schuiven (als hij twee opschuift kan de positie niet herstelt worden). Vervolgens gaat hij langs onder onder de paketten door om ze terug 1 positie naar links te schuiven om ze zo terug in originele positie te brengen. Wanneer de paketten twee posities vooruit geduwd worden, komen we in een onherstelbare configuratie terecht. Als de magazijnier probeert vanuit de uitgang binnen te treden, kan hij niet meer verder en is hij daar voor niets aanbeland.

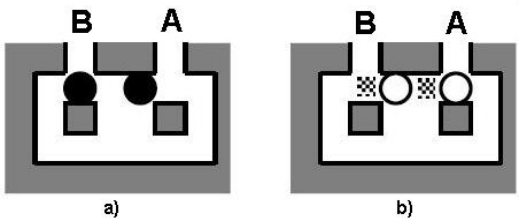
- Een *schuivende deur* is weergegeven in figuur 4.15. Hier kan de magazijnier oneindig veel keer van I_1 naar O_1 wandelen. Wanneer hij echter een keer van I_2 naar O_2 loopt, kan hij oneindig deze weg nemen, maar niet meer van I_1 naar O_1 gaan. Als hij dit wil doen moet hij eerst via s de deur herstellen en dan zijn we terug aanbeland in de beginsituatie. Schuivende deuren blijken een groot nut te hebben voor de simulatie. Wanneer we uitgang O_1 verbinden met ingang I_2 noemen we het gadget een *poort* en ook deze is van belang voor de simulatie. In figuur 4.15 stellen de pijlen een eenrichtingsstraat voor.

Twee jaar later toonde Colberson [Cul97] aan dat Soukoban(1,0), de originele versie, PSPACE-compleet is. Hiervoor toont hij aan dat een Turing Machine kan gesimuleerd worden, en dit in lineaire tijd, door een oneindige versie van Soukoban met enkel een eindig aantal paketten nog niet op zijn plaats. Als de lengte van de tape dan (eindig) beperkt wordt, is de puzzel PSPACE-hard (zie sectie 2.2.8 voor een analoge definitie). Vermist in [DZ99] reeds aangetoond werd dat Soukoban(1,0) $\in$ PSPACE, volgt hieruit dat de puzzel

ook PSPACE-compleet is. Ook hier worden enkel twee basisgadgets getoond (figuren 4.16 en 4.17):



Figuur 4.16: Soukoban(1,0): Eenrichtingsstraat ([Cul97]).



Figuur 4.17: Soukoban(1,0): Reverser ([Cul97]).

- Figuur 4.16 heeft dezelfde werking als de eenrichtingsstraat als in figuur 4.14.
- De reverser is getoond in figuur 4.17. De magazijnier kan enkel via A door de reverser gaan en langs B uitgaan. Zo bekomen we het gadget getoond in figuur b). Van hieruit moet de magazijnier terugkomen via B en uitgaan via A om het gadget terug in zijn oorspronkelijke goede positie te brengen. Met goed bedoelen we: waar de paketten op zijn plaats staan (=vol bolletje).

Zoals we zagen is het originele Soukoban spel PSPACE-compleet [Cul97], net zoals vele van zijn varianten (zie hiervoor naar [DZ99] en [HD02]).

Als laatste punt vermelden we dat 10 jaar later Hearn en Demaine [HD02] een heel andere techniek gebruiken, waarbij de bewijzen lopen aan de hand van Nondeterministic Constraint Logic Model of Computation (NCL). Door middel vand deze techniek, die op zich de nodige uitleg vereist, kan men voor meerdere problemen, op een analoge manier, aantonen dat ze PSPACE-compleet zijn. In [HD02] tonen ze dit resultaat (buiten voor Soukoban) ook

aan voor Rush Hour. We vermelden alleen dat NCL een mogelijkheid biedt om meerdere (overeenkomstige) complexiteitsresultaten te verkrijgen door middel van dezelfde techniek, en dit voor verschillende problemen.

4.6 Rectangle Tiling puzzel

Definitie 4.6.1 (Rectangle Tiling puzzel probleem). *Gegeven een set van tile-types X en een rechthoek R met breedte n en hoogte k (voor een eindige k, met n vast en k variabel), waarvan de zijden een gegeven kleuring hebben; bestaat er een X-Tiling van R: en dit zodat de kleur van elke zijde op de plaats waar hij aan een tile t grenst, dezelfde is als de kleur van de kant van t die tegen die zijde ligt.*

Rectangle tiling puzzels hebben veel weg van square tiling puzzels, met het verschil dat de hoogte nu ongedefinieerd is, maar wel eindig! Door deze aanpassing ligt rectangle tiling puzzel in een andere complexiteitsklasse. Ook zullen we bij het analyseren ervan het begrip 'tijd', uit rectangle tiling puzzel, door 'space' moeten vervangen.

Stelling 4.6.1. *Rectangle Tiling puzzel probleem $\in$ PSPACE.*

Hoe kunnen we over rectangle tiling puzzel denken? De hoogte is onbepaald (wel eindig), maar dit maakt niet uit voor onze space, vermits hoogte overeenkomt met tijd (zie figuur 4.11). De breedte echter heeft wel een vaste waarde, afhankelijk van de NTM die het nabootst. We kunnen dus zeggen dat de space beperkt is. Als we daarbij denken dat de ruimte van die NTM polynomiaal is in functie van de input, begrijpen we het verband met SPACE-compleetheid.

Ter volledigheid vermelden we nog dat *Square Tiling Game*, wat overeenkomt met Square Tiling gespeeld door twee personen (respectievelijk *constructeur* en *saboteur*), ook PSPACE-compleet is. Voor de reductie zijn in dit geval twee sets met tilings nodig, een voor elke speler. Het bewijs wordt gegeven in [Chl86] en maakt gebruik van alternerende Turing Machines [CKS81]. De constructeur wint als de overeenkomstige tiling daadwerkelijk gevormd wordt, de saboteur in het andere geval. Merk op dat de tilings zo gekozen worden dat de constructeur het spel veelal kan manipuleren, zodat de saboteur verplicht wordt bepaalde zetten te doen, en de constructeur uiteindelijk wint.

4.7 Schaken

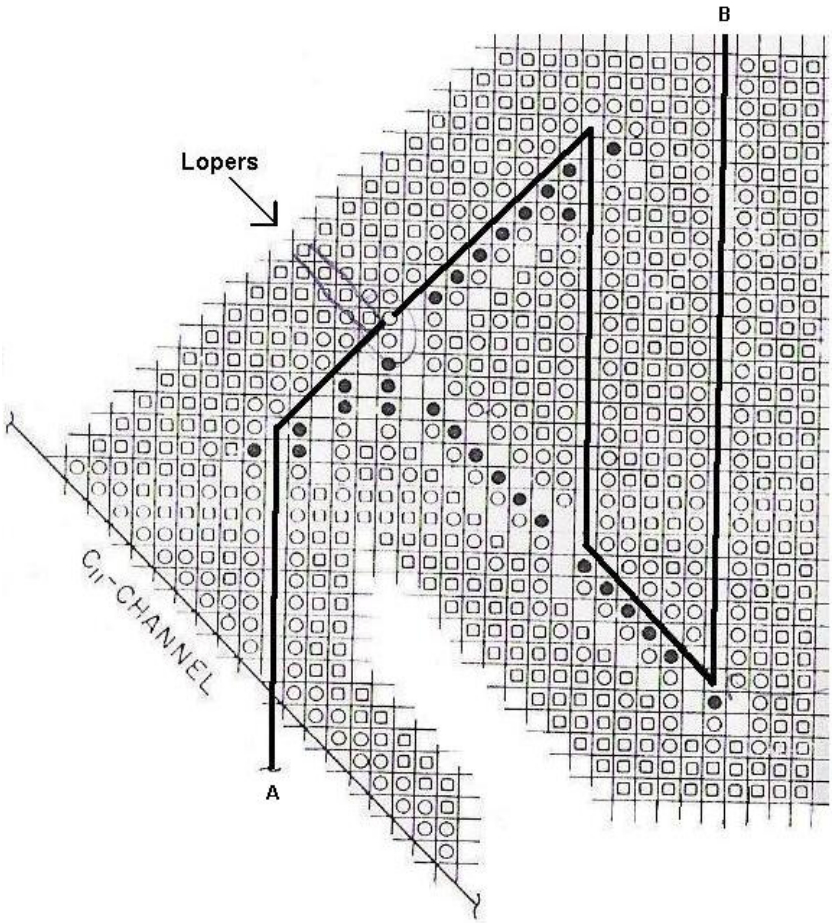
Men vermoedt dat Schaken afkomstig is uit het Oosten en al minstens 1500 jaar oud is. Het is een uiterst complex spel en een van de eerste spellen waarvoor een complexiteitsresutaat bekend werd. De gegeven resultaten zijn afkomstig van een reeds in 1981 gepubliceerd artikel van Fraenkel en Lichtenstein [FL81]. Merk op dat het hier wel handelt over een $n \times n$ schaakbord waarop de aanwezige pionnen en stukken ongeveer evenredig zijn met die op een 8×8 bord, met uitzondering van de koning, waarvan er slechts 1 per speler aanwezig mag zijn. Buiten deze voorwaarde is de initiële positie niet van belang vermits er gekeken wordt naar arbitraire schaakposities.

Zoals gezien hebben we een polynomiale reductie nodig van een reeds gekend EXPTIME-compleet probleem naar schaken. Dit probleem zal een aangepaste versie van het G3 probleem zijn, zoals beschreven in [SC79]. Het schaakprobleem is: heeft wit (zwart) een winnende strategie vanuit een willekeurige gegeven positie.

Definitie 4.7.1 (G3). *Elke positie in G3 is een vier-tuppel (τ, W-LOSE(X, Y), B-LOSE(X, Y), a) waarbij $\tau \in \{W, B\}$ de speler aan beurt voorstelt, W-LOSE $= C_{1,1}, C_{1,2}, \ldots, C_{1,p}$ en B-LOSE $= C_{2,1}, C_{2,2}, \ldots, C_{2,q}$ booleaanse formules zijn, en a is een toekenning van waardes aan de set van variabelen $X \cup Y$. De spelers wisselen van beurt, een beurt overslaan is niet toegelaten. In een beurt **moet** een speler juist 1 variabele in X (respectievelijk Y) van waarde veranderen. Het spel eindigt als W-LOSE (respectievelijk B-LOSE) waar is nadat W (respectievelijk B) een zet heeft gedaan, de speler in kwestie verliest het spel dan. Als iemand de regels niet volgt, verliest hij ook onmiddellijk.*

In [FL81] simuleert men G3 op een $n \times n$ schaakbord. De doelstelling is om per variabele in het G3 probleem slechts drie bewegende stukken op het schaakbord te hebben, namelijk twee dames en één toren. De andere stukken (pionnen, lopers,...) zijn 'vastgepind' (ze kunnen niet bewegen). De toren kan slechts twee 'legale' posities innemen, namelijk degene overeenkomstig met $true$ en $false$ per bijbehorende variabele. Het is de bedoeling dat de dames zorgen voor een doorbraak in de constructie van de tegenspeler, wanneer de overeenkomstige formule *false* is, om zo de koning van de tegenspeler schaakmat te zetten. De gehele constructie is vrij complex, we geven hier enkel de eenrichtingsstraat (figuur 4.18) met bijhorende uitleg.

In de figuur stellen de circels en de vierkanten respectievelijk pionnen en



Figuur 4.18: Schaakconstructie: eenrichtingsstraat ([FL81]).

lopers voor. Wit speelt naar boven en zwart naar beneden. We zien onmiddellijk dat de dame het enige stuk is dat vlot door de constructie kan. De weg die ze moet afleggen is in de figuur te herkennen door de volle lijn. De mogelijkheden dat ze het pad niet volgt, is gegeven in [FL81] en men toont aan dat dit geval tot verlies leidt (door de extra zet die nodig is). Nu zijn er twee mogelijkheden om het pad te volgen, ofwel komt de dame de constructie binnen via A, ofwel via B.

- **Via B:** De koningin kan vrij lopen tot ze de pion op haar pad slaat. Daarna kan ze terugkeren met tempoverlies (en uiteindelijk spelverlies) of verder gaan en de constructie langs A verlaten.
- **Via A:**
  - Pion op lange diagonaal staat er nog: Deze moet genomen worden. Hierna kan de dame enkel terugkeren. Als ze verder gaat kan wit de pion onder de net geslagen pion vooruit zetten. Door deze zet valt de loper (aangeduidt door lopers en pijl) de korte diagonaal aan, waar door de koningen hier niet meer kan passeren zonder geslagen te worden. Als ze geslagen wordt komt dit overeen met verlies. In de constructie zijn genoeg lopers aanwezig om elke dame op deze wijze te kunnen slaan, zodat er geen doorkomen aan is in de tegengestelde richting.
  - Pion op lange diagonaal reeds genomen: Eens de dame langs onder komt en reeds voorbij de eerder geslagen pion is, gebruiken we dezelfde taktiek als hiervoor, waardoor men de dame verliest.

Wanneer de dame de constructie langs A binnentreedt, komt ze de pion op de lange diagonaal tegen, die ze moet nemen om niet zelf genomen te kunnen worden. Als ze via B gaat, zal ze de pion op de lange diagonaal (ongeveer in het midden ervan) moeten nemen, om vervolgens de weg te vervolgen.

Stelling 4.7.1. *Schaken is EXPTIME-compleet ([FL81]).*

Naast schaken blijkt ook dammen EXPTIME-compleet te zijn. Het probleem aangaande dammen heeft ook betrekking op de posities van waaruit 'wit' (respectievelijk 'zwart') kan winnen. Het resultaat is gegeven in [Rob84] en is, net zoals het resultaat bij schaken, gebaseerd op een reductie vanuit het spel genaamd G3, wat reeds gekend EXPTIME-compleet is. Voor de constructie wordt ver genoeg buiten een centraal bord een groot genoege spiraal opgetrokken bestaande uit zowel een witte als een zwarte spiraal. De bewegingen hierin zullen uiteindelijk bepalend zijn voor het gehele spel.

4.8 Rectangle Tiling spel

We halen hier een versie van Tilings aan die EXPTIME-compleet is. Het resultaat is afkomstig uit [Chl86] en [vEB96].

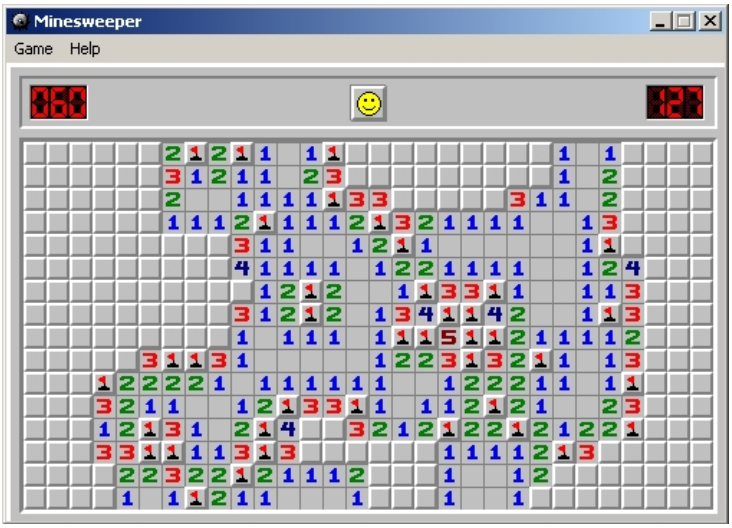
Rectangle Tiling spel Net zoals bij Square tiling spel worden er twee tiling sets aangemaakt, een voor de zogenaamde *constructeur* en een voor de *saboteur*. Dan door deze twee om beurt een tiling te laten kiezen kan via alternerende Turing Machines aangetoond worden dat het rectangle tiling spel zowel in EXPTIME ligt alsook compleet is voor deze klasse. Hoofdzakelijk komt het erop neer dat de saboteur niet veel keuze gelaten wordt, de constructeur mag immers eerst een tile leggen, de saboteur moet vervolgens de spelregels volgen. De tiles worden dus zo verdeeld dat de saboteur na de beurt van de constructeur (bijna) geen keus meer heeft.

Uiteindelijk kunnen we afleiden dat een spelvariant van een puzzel een complexiteitsklasse 'hoger' ligt. Hoewel we dit niet in het algemeen kunnen stellen, wordt toch vaak opgemerkt dat wanneer een puzzel variant omgezet wordt in een spel (van een speler naar twee spelers), de complexiteit toeneemt.

Hoofdstuk 5

Minesweeper[4]

Bijna iedere PC-gebruiker kent Minesweeper wel. Het is een zeer populair spel en wordt bijna bij elke Windows[1] versie standaard meegeleverd en dankt zijn populariteit hier dan ook aan. Minesweeper werd voor het eerst uitgebracht in 1992 als onderdeel van het toenmalige Windows 3.1. Het spel werd geprogrammeerd door Robert Donner en Curt Johnson. Een voorbeeld van hoe het huidige spel er onder Windows uitziet, is gegeven in figuur 5.1.



Figuur 5.1: Voorbeeld van het huidige Windows Minesweeper (expert level).

Het originele Minesweeper spel wordt gespeeld op een rechthoekig vlak, verdeeld in vierkante vakjes, waarin zich een aantal mijnen bevinden. De bedoeling bestaat erin om de plaatsen waar een mijn ligt te ontdekken, zonder een

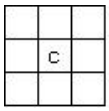
[1]Microsoft Windows (http://www.microsoft.com/)

mijn tot ontploffing te brengen. Wanneer men een vakje in het vlak *onthult* krijgt men te zien wat er zich onder bevond. In het geval die een mijn is 'ontploft hij' en verliest men het spel. Een van de acht andere mogelijkheden is een getal van 0 tot en met 8. Dit getal representeert het aantal mijnen dat er zich in de aangrenzende vakjes van het onthulde vakje liggen. Dit zijn indicaties voor waar eventuele mijnen zich zouden kunnen bevinden. Zie figuur 5.2 voor een vakje en zijn 8 buren. Bij het originele spel kan er bij de eerste onthulling geen mijn verschijnen. Wanneer er toch een mijn onder het onthulde vakje zou liggen wordt deze dan verplaatst naar een vakje waar zich nog geen mijn bevond. Om het Minesweeper spel te winnen moet de speler alle vakjes onthullen waar zich geen mijn bevindt, dit met behulp van de cijfers in de reeds onthulde vakjes. Als extra heeft het originele spel ook nog een teller die aangeeft hoeveel mijnen er zich nog in het vlak bevinden.

5.1 Notatie

We zullen in het vervolg het originele Minesweeper spel weergeven als Minesweeper^4. Dit omdat het originele spel gespeeld wordt in een vlak dat verdeeld is in *regelmatige* vierhoeken (= vierkant) (vandaar het superscript vier). Deze notatie wordt gebruikt omdat we in verdere hoofdstukken zullen spreken over varianten van Minesweeper waarbij het vlak verdeeld wordt in regelmatige n-hoeken. De net genoemde varianten zullen overeenkomstig de waarde van n de benaming Minesweeper^n toegewezen krijgen.

Voorbeeld 5.1.1 (Minesweeper). Veronderstel het 6*6 vlak zoals gegeven in figuur 5.3. Hierbij zijn de letters toegevoegd ter identificatie van de overeenkomstige vakken. We kunnen met zekerheid concluderen dat er op de plaats van een A zich een bom bevindt. Dit om wille van het de cel met cijfer twee onder de linkse A, die slechts 2 lege buren heeft (namelijk de vakken gelabeld met een A), waardoor deze automatisch een bom moeten bevatten. Op dezelfde manier kan men ook concluderen dat de vakken die overeenkomen met een B en een C ook een mijn moeten bevatten, en dit door de cijfers 4 en de 5 links van de vakken met een B in, en het cijfer 1 links van het vak gelabeld met C. Nu we dit weten kunnen we met zekerheid zeggen dat de vakken met D en E geen mijn bevatten en dus zonder gevaar onthuld kunnen worden. Het aantal mijnen aangegeven door de buren van de cellen met D en E worden reeds ingenomen (ofwel door een 'zichtbare' mijn, ofwel door een vak A, B of C). Alleen het vak met F rest ons nu nog. Door onze teller weten we hoevel mijnen er aanwezig zijn en weten we of deze cel al dan niet een mijn bevat (het laatste geval als de teller nog op 1 staat).



Figuur 5.2: Een grid cel c met zijn acht buren.

F	D	2	1	2	1
A	A	3	*	4	B
2	2	3	*	5	B
0	0	1	1	4	B
0	1	1	1	2	B
0	1	C	E	E	E

Figuur 5.3: zes op zes Minesweeper vlak ([Kay00a]).

2	3	*	2	2	*	2	1
*	*	5			4	*	2
	?	*	*	*		4	*
*	6		6	*	*		2
2	*	*		5	5		2
1	3	4		*	*	4	*
0	1	*	4			*	3
0	1	2	*	2	3	*	2

Figuur 5.4: Bevat het vraagteken een mijn? ([Kay00b]).

Het is dus mogelijk om vertrekkende van de 16 verhulde vakken in dit 6*6 vlak, te concluderen waar zich alle mijnen bevinden. Op deze manier kunnen alle vakjes zonder mijn geïdentificeerd worden en kan met het spel zodoende winnen.

5.2 Algemeen Minesweeper Probleem

Het probleem dat we hier zullen behandelen is lichtjes anders dan het originele Minesweeper probleem, maar de basisideeën zijn dezelfde, alsook de regels omtrent het spel. We zullen refereren naar dit probleem als zijnde het *algemene Minesweeper* probleem. Brandon P. McPhail [McP03] verwoordt dit probleem op een mooie, bijna metaforische manier, als volgt:

> We zijn meer geïnteresseerd in een soort *paper-and-pencil* variant van het computerspel Minesweeper. Als we op een vast punt gedurende een Minesweeper spel de 'Print Screen'-knop in zouden drukken en naar de printer lopen, is wat er uit zou komen de paper-and-pencil versie van het spel, ook wel *offline* Minesweeper genoemd. In plaats van nu een cel te zoeken die geen mijn bevat en die te onthullen om zo extra informatie te verkrijgen, is

> ons doel nu een plaatsing van mijnen te vinden op het afgedrukte blad, overeenkomstig met de genummerde cellen. Zo zou Minesweeper gezien kunnen worden als een hoop opeenvolgingen van 'offline' spellen, die slecht één cel van elkaar verschillen.

Dit brengt ons bij de volgende definitie. Deze is dezelfde als gegeven door Richard Kaye [Kay00a].

Definitie 5.2.1 (Algemeen Minesweeper probleem). *Gegeven een rechthoekig grid dat gedeeltelijk onthuld is; bestaat er* een *patroon van mijnen in de niet onthulde vakken, die aanleiding geven tot de reeds onthulde nummers.*

Het gaat hier om een booleaanse vraag, we zoeken dus niet naar de oplossing, enkel naar het bestaan van *een* oplossing. Met andere woorden, het Minesweeper probleem handelt over de vraag of de gegeven data *consistent* is. Hiernaar wordt ook vaak verwezen als het *consistentie probleem*. Zoals R. Kaye aangeeft is dit een typisch ja/nee probleem dat, wanneer we er een algoritme a voor zouden hebben, we ook een uitstekende manier hebben om het spel te spelen. Om te bepalen of een vakje v *veilig* is, zouden we de configuratie die we momenteel zien kunnen opschrijven, aangepast op die manier dat we v markeren met een mijn. Als de uitkomst van a is dat het ingegeven patroon inconsistent is, zal er zich **geen** mijn op v bevinden en kunnen we v met gegarandeerd succes onthullen. In het andere geval zou het kunnen dat er zich een mijn op v bevindt, al kunnen we dan niet met zekerheid zeggen dat v een mijn bevat. De grootte van de input is het aantal gridcellen.

Stelling 5.2.1. *Het Algemeen Minesweeper probleem is NP-compleet.*

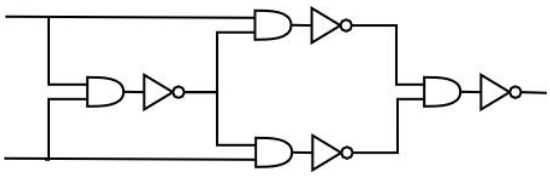
Dat het probleem zich in NP bevindt is duidelijk. We kunnen namelijk niet-determinischtisch gaan gokken waar de mijnen zich bevinden om dan in polynomiale tijd na te gaan of de mijnen legaal geplaatst waren (overeenkomstig met de reeds onthulde cijfers). Het precieze algoritme wordt gegeven in [BPM]. In hoofdstuk 6 geven we dit algoritme voor Minesweeper[6]. Om de NP-compleetheid ervan aan te tonen zullen we echter de reductie van Circuit SAT naar Minesweeper gebruiken zoals eerst geintroduceerd door R.Kaye [Kay00a]. In dit artikel worden de nodige Minesweeper constructies gegeven om Circuit SAT te simuleren. Enkele vereenvoudigingen worden gegeven in een later artikel van dezelfde persoon [Kay00b] en zullen ook kort besproken worden.

We zullen nu kort nagaan welke constructies er nodig zijn. Op het eerste zich lijkt het alsof we enkel een "stroomdraad" nodig hebben, tesamen met de NOT en de AND poort. Een OR poort kan namelijk gesimuleerd worden

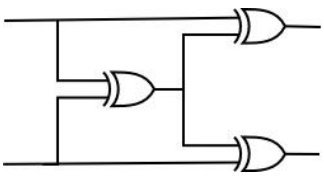
door een AND en drie NOT poorten. Een nadere kijk leert ons dat we daarbij tenminste een manier moeten hebben om de draad buigen en op te splitsen. Men zou kunnen denken dat er een *crossover* nodig is om twee signalen elkaar te laten kruisen. Dit is echter niet het geval: de crossover kan namelijk gesimuleerd worden door gebruik te maken van drie splitters en drie XOR poorten zoals aangegeven in figuur 5.5. De nodige XOR poorten kunnen samengesteld worden door vier AND en vier NOT poorten (figuur 5.6). In de booleaanse logica wordt de overeenkomstige XOR-formule gegeven door:

$$\neg(\neg(x_1 \wedge \neg(x_1 \wedge x_2)) \wedge \neg(x_2 \wedge \neg(x_1 \wedge x_2))). \tag{5.1}$$

Merk op dat in [Kay00b] wel degelijk een Minesweeper constructie voor een crossover gegeven wordt en deze dan ook gebruikt zou kunnen worden. Wat nog wel een vereiste is, is iets dat de draad een andere lengte kan geven, zodat de verschilllende onderdelen juist op elkaar aangesloten kunnen worden. Dit zal gebeuren door de *phase-changer*. Een draad moet vanzelfsprekend ook beëindigd kunnen worden. Als we al deze onderdelen kunnen simuleren



Figuur 5.5: XOR-poort gemaakt met vier AND- en vier NOT-poorten ([Kay00a]).



Figuur 5.6: Crossover door middel van drie XOR-poorten ([Kay00a]).

door een bepaalde Minesweeper configuratie, dan kunnen we Circuit SAT reduceren naar Minesweeper. Wanneer de reductie in polynomiale tijd kan gebeuren, is het Minesweeper probleem NP-Compleet.

5.3 Constructies

Voor de constructies moeten we een bepaalde richting aangeven in dewelke het signaal zich propageert. Laten we voor de duidelijkheid aannemen dat het signaal zich van links naar rechts voortbeweegt. In elk van de constructies stellen de cellen gelabeld met x' of x nog niet onthulde cellen voor, die al dan niet een mijn bevatten. We hebben nu enkel nog een manier nodig om te zeggen hoe een *true* signaal en een *false* signaal voorgesteld worden. Vanaf dan kunnen we beginnen met de simulatie. Laat ons aannemen dat we *true* laten overeenkomen met - er ligt een mijn onder x - en analoog *false* laten

$\cdots$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\cdots$
$\cdots$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	$\cdots$
$\cdots$	x	1	x'	x	1	x'	x	1	x'	x	1	x'	x	1	x'	x	$\cdots$
$\cdots$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	$\cdots$
$\cdots$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\cdots$

Figuur 5.7: Minesweeper[4]: Draad ([Kay00a]).

overeenkomen met - er ligt geen mijn onder x. Dit laatste zal steeds hetzelfde zijn als zeggen dat er een mijn ligt onder x', omdat we steeds veronderstellen dat x' false is wanneer x true is. Vaak zal true voorgesteld worden met T en false met F.

5.3.1 Nodige constructies

Draad (figuur 5.7)

De draad is nodig om een signaal te kunnen propageren, in ons geval van links naar rechts. Wanneer er een mijn onder de eerst x ligt, zal er ook een mijn onder de laatste x zitten. De waarde wordt dus vanaf het begin doorgegeven naar het einde. Men kan zich nu afvragen wat onze constructie werkelijk tot imitatie van een draad maakt, waarbij die draad een signaal kan propageren. Dit alles doet de draad-constructie door gebruik te maken van drie *onderdelen*.

1. Signaal: Het signaal wordt doorgegeven door middel van cellen waarvoor we (nog) niet weten of er zich al dan niet een mijn onder bevindt. De volgende twee onderdelen zorgen ervoor dat er telkens twee cellen zijn, laten we ze x' en x noemen, die een mijn kunnen bevatten. Ze zorgen ook dat x' en x nooit beiden een mijn kunnen hebben en dat een van de twee steeds een mijn bevat. We kunnen dus concluderen dat als x' geen mijn bevat, x dat wel doet, en vice versa. In de constructies nemen we aan dat als het (de) input signaal (signalen) van het originele booleaanse circuit *true* is (zijn) (of 1 in de logica), dat we dit dan laten overeenkomen in onze draad-constructie met *er ligt een mijn onder de cellen gelabeld met* x (of welk symbool we ook gebruiken voor het input signaal). Analoog komt een signaal dat $false$ voorstelt overeen met: er liggen mijnen onder cellen gelabeld met x'. Merk drie dingen op. Ten eerste is er feitelijk geen *echt* verschil is tussen de signalen, we hebben de keuze immers arbitrair gemaakt. Men zou evengoed de andere keuze kunnen maken. Ten tweede kiezen we (wederom arbitrair) dat de oriëntatie loopt van links naar rechts. Als we het hebben over een

Figuur 5.8: Minesweeper[4]: Basis draad.

cel die zich achter *achter* een andere cel bevindt, bedoelen we dus gezien vanuit deze oriëntatie. Ten derde kan $true$ en $false$ nooit tesamen voorkomen in eenzelfde draad.

2. Keuzecellen: Dit zijn de cellen die grenzen aan x' en x (zoals steeds gelezen van links naar rechts) en waarbij de x' zelf ook aan de x grenst. We plaatsen in een keuzecel nu het getal overeenkomend met het aantal reeds gekende buurmijnen plus een. Zodoende bevat ofwel x' een mijn, ofwel x, maar nooit beiden. Een keuzecel zorgt er dus voor dat x' en x steeds het tegengestelde verbergen (een mijn of niets). Merk op dat er slechts een keuzecel nodig is om een keuze vast te leggen voor de gehele draad, zie figuur 5.8. De verbindingscellen (zie puntje 3) zorgen er dan voor dat alle andere x'en (respectievelijk $x$en) dezelfde waarde hebben als de x' (respectievelijk x) bij de keuzecel. Er kunnen meerdere keuzecellen aanwezig zijn, en in de constructies is dit ook het geval. Om echter tot een oplossing te komen die een consistente positie achter laat, moeten deze steeds dezelfde keuze vastleggen voor x' en bijgevolg ook voor x'.

3. Verbindingscellen: Deze cellen grenzen (van links naar rechts) aan x en x'. Verbindigscellen zijn nodig bij elke overgang van x naar x'. Op deze manier zorgt de cel in kwestie voor de propagatie van signaal dat bepaald wordt door de keuze van de keuzecel. Wanneer deze niet aanwezig zouden zijn, is het mogelijk dat twee selecteercellen die gelegen zijn in eenzelfde draad, een verschillende keuze maken. Dit leidt uiteraard tot een foute constructie. In figuur 5.9 kan men niet meer garanderen dat x hetzelfde verbergt als een cel gelabeld met y.

Opmerking 5.3.1. We zouden de oorspronkelijke draadconstructie zo kunnen aanpassen dat ze slechts één selecteercel heeft en een minimum aan verbindingscellen. Vanzelfsprekend introduceren we dan een grote hoeveelheid aan mijnen. Deze niet zo mooie constructie is gegeven in figuur 5.8.

	0	0	0	0	0	1	1	1	0	0	0	0	0	
	1	1	1	1	1	3	*	3	1	1	1	1	1	
	1	x'	x	1	x'	x	*	y'	y	1	y'	y	1	
	1	1	1	1	1	3	*	3	1	1	1	1	1	
	0	0	0	0	0	1	1	1	0	0	0	0	0	

Figuur 5.9: Minesweeper[4]: Draad met bindingscel te kort.

Bocht (figuur 5.10)

Een bocht is nodig om een signaal te laten afbuigen. Het signaal beweegt hier idem als in de draad, behalve dat het hier een draai van 90^o maakt. Dit gebeurt door de aangeduidde 2 in de centrale benedenhoek. Deze heeft reeds een mijn als buur, waardoor x' en x in de bocht een andere waarde moeten hebben. De bocht wordt vervoledigd door de vijf mijnen rechtsboven.

Einde (figuur 5.11)

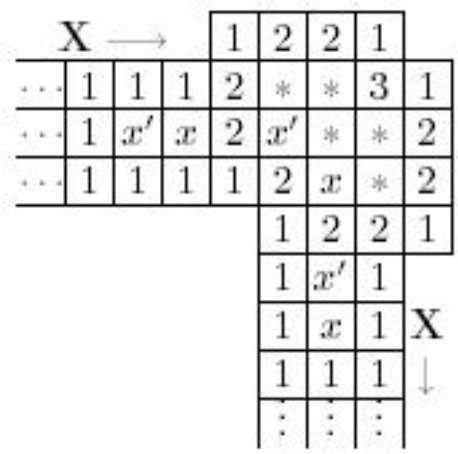
Het einde zorgt ervoor dat een signaal ophoudt te bestaan. Dit kan nodig zijn wanneer we bijvoorbeeld een splitter (zie verder) willen met slechts twee vertakkingen, dan moeten we er een van beëindigen. Door de twee gemerkte 3s, die al twee buren hebben, is het zo dat ofwel x' ofwel x zorgt voor de derde mijn. Hierdoor stopt het signaal dan ook.

Splitter (3-way) (figuur 5.12)

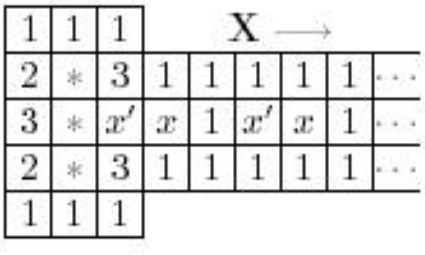
Met de splitter kunnen we een signaal opsplitsen. Als we bijvoorbeeld een 'true' waarde verder willen gaan propageren over meerdere draden, gebruiken we de splitter. De opsplitsing kan weer verder opgesplitst worden enzoverder. Heel belangrijk hierbij op te merken is dat het oorspronkelijke signaal slechts in twee richtingen verder gaat, en dit door de cel gelabeld met 2 in het midden. In de andere richting zal een tegengesteld signaal gepropageerd worden. Door een van de drie richtingen af te sluiten met een *einde* voorkomen we dat een ongewenst signaal verder gaat.

Phase changer (figuur 5.13)

Werkt idem als de draad, behalve dat in het midden het signaal twee maal van waarde verandert, waardoor het originele signaal gepropageerd wordt. Merk op dat de lengte van dit stuk net 1 cel korter is dan die van de draad. Hierdoor is de phase changer handig om verschillende constructies op mekaar



Figuur 5.10: Minesweeper[4]: Bocht ([Kay00a]).



Figuur 5.11: Minesweeper[4]: Einde ([Kay00a]).

aan te sluiten. Het zou immers kunnen dat de plaatsing van de $x$en in de ene constructie niet overeenkomen met de plaatsing van de $x$en in een andere. Om dit probleem op te lossen hebben we de phase changer.

NOT-poort (figuur 5.14)

Hierbij wordt het originele signaal veranderd en wordt het tegengestelde signaal gepropageerd. Zoals men kan nagaan verandert de middelste constructie (met de twee 3en in het midden) het signaal.

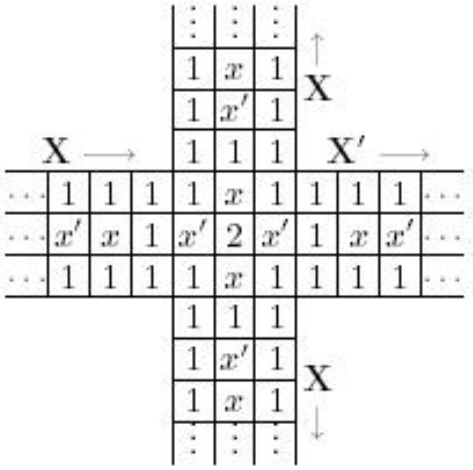
AND-poort (figuur 5.15)

Dit is ongetwijfeld de moeilijkste constructie van allemaal. In [Kay00a] wordt de constructie op een lichtjes andere manier uitgelegd dan hier. We zullen hier een tegenovergestelde denkwijze hanteren.

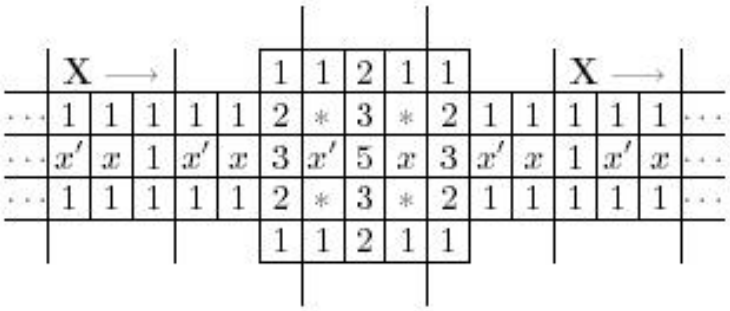
Laten we eerst van achter controleren of de uitkomst false is. Dit wil zeggen dat de cellen met symbool t' mijnen bevatten en die met t niet. Door het cijfer 3 boven a_3 weten we dat ofwel cel a_2 ofwel cel a_3 een mijn bevat, maar niet beide. Hieruit volgt (door de 2 boven a_2) dat a_1 een mijn bevat.

Stel a_1 bevat dus een mijn, dan zijn er twee mogelijkheden:

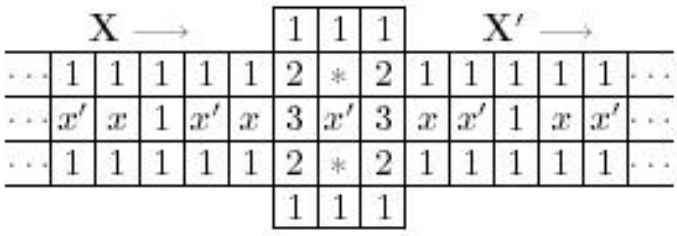
- a_2 bevat ook een mijn. Dan heeft s geen mijn, en dit door de 3 boven a_1.



Figuur 5.12: Minesweeper[4]: Splitter ([Kay00a]).



Figuur 5.13: Minesweeper[4]: Phase changer ([Kay00a]).



Figuur 5.14: Minesweeper[4]: NOT-poort ([Kay00a]).

- a_2 bevat geen mijn. Dan heeft s wel een mijn, om wille van dezelfde reden.

We kunnen hier dus concluderen dat (voorlopig) s zowel een mijn, als geen mijn kan bevatten. Op een analoge manier bekomen we dat r al dan niet een mijn kan bevatten. Dat t geen mijn bevat is reeds geweten. Vanaf nu zal de gemarkeerde 4 een belangrijke rol gaan spelen. Beschouw nu de volgende drie mogelijkheden, die tot het gewenste resultaat zullen leiden.

- s en r bevatten beiden een mijn.
  Dan volgt dat ofwel u' ofwel v' een mijn bevat en dit door de gemarkeerde 4. Hieruit kunnen we concluderen dat of input via u true is of input via v is true, maar niet beiden tegelijk.

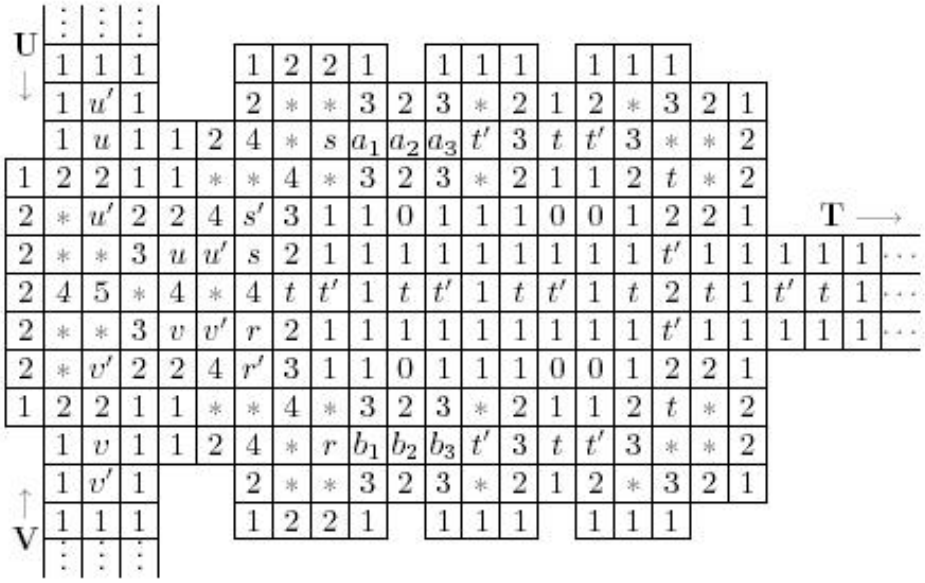
- ofwel bevat s een mijn, ofwel r.
  In dit geval is het zo dat zowel u' als v' een mijn bevatten, wat overeenkomt met zeggen dat onze beide inputs false zijn.

- s en r bevatten beiden geen mijn.
  Deze mogelijkheid kan niet voorkomen. Omdat t geen mijn bevat, kan de gemarkeerde 4 in dit geval geen vier mijnen als buur meer hebben. We zouden immers een inconsistente positie achter laten.

Hiermee zijn alle mogelijkheden beschouwd bij output $= F$. De output $= T$ werd namelijk nog niet beschouwd. Stel nu dat u en v een mijn bevatten, wat overeenkomt met twee inputs met waarde T. Dan bevatten u' en v' logischerwijze geen mijnen. Wederom door de gemarkeerde 4 weten we dat dan zowel r, s als t een mijn bevatten. Vermits t een mijn bevat is de output true, wat we moesten aantonen. Merk als laatste op dat wanneer zowel r en s een mijn bevatten, dat er dan een consistente positie achtergelaten wordt (vermits we dan in de deelconstructies met a_1, a_2 en a_3, resp. b_1, b_2 en b_3 kunnen kiezen of we volgen met ofwel t bevat een mijn, ofwel t' bevat een mijn. We kiezen dus t bevat een mijn om consistent te zijn.

Een leuke opmerking die Brandon P. McPhail [McP03] reeds maakte is dat wanneer we de waarde van $true$ en $false$ omdraaien [1], dat de AND-poort dan verandert in een OR-poort. Dit kan men heel gemakkelijk inzien door de waarden van de AND-tabel gegeven in de linker kolom van tabel 5.1 om te zetten (T naar F en F naar T). Het resultaat is weergegeven in de rechter kolom van tabel 5.1. De verificatie aan de hand van de constructie wanneer men de waarden omwisselt laat ik over aan de geïnteresseerde lezer.

[1] Dit is tevens de manier waarmee hij werkt.



Figuur 5.15: Minesweeper[4]: AND-poort ([Kay00a]).

AND	Omzetting van waarden = OR
T + T = T	F + F = F
T + F = F	F + T = T
F + T = F	T + F = T
F + F = F	T + T = T

Tabel 5.1: Omzetting van true naar false en vice versa.

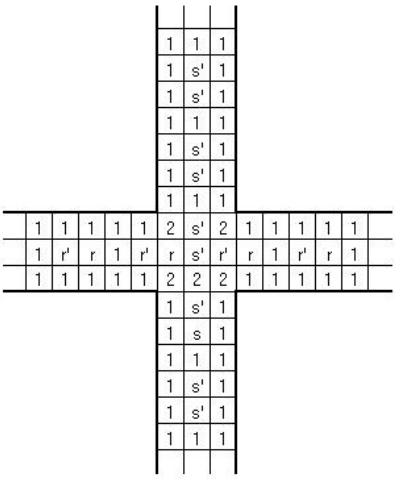
5.3.2 Overige constructies

Crossover (figuur 5.16)

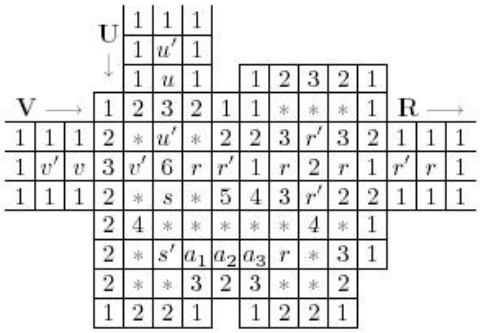
Een constructie die opmerkelijk kleiner is dan de samenstelling eerder gebruikt (met de 4 XOR-poorten) om twee signalen elkaar te laten kruisen. Ze is echter niet nodig om de simulatie te doen, maar komt wel erg van pas. De originele crossover constructie was vrij gecompliceerd en vergezocht. Later heeft Richard Kaye een grote vereenvoudiging doorgevoerd. Dit resultaat is te zien in de figuur.

OR (figuur 5.17)

De werking hiervan wordt volledig beschreven in [McP03]. Merk op dat hij een andere waardetoekenning gebruikt voor true en false. Hierdoor is onze OR-poort bij hem een AND-poort. De werking is echter geheel hetzelfde en wordt door hem dan ook nagegaan. De werking kan vergeleken worden met die van de AND-poort, al betreft het hier een heel andere constructie.

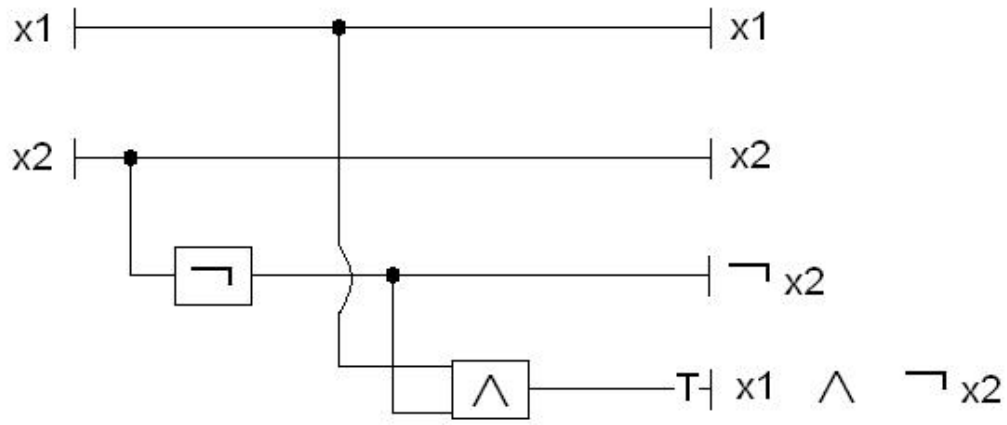


Figuur 5.16: Minesweeper[4]: Crossover ([Kay00b]).



Figuur 5.17: Minesweeper[4]: OR-poort ([Kay00b]).

Opmerking 5.3.2. Er zijn ook nog andere, meer gecompliceerde constructies bekend voor Minesweeper. Deze zijn echter van minder belang; voorbeelden hiervan zijn de XOR, NAND (beide handig voor de constructie), OR/XOR, AND/XOR en de NAND/XOR.

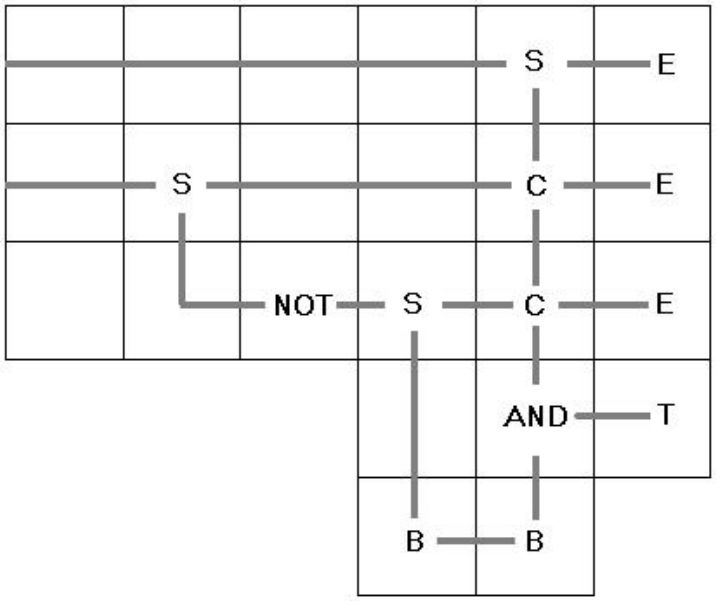


Figuur 5.18: Minesweeper (algemene) constructie voor formule 3.1.

5.3.3 Complexiteit van de omzetting

We moeten nu enkel nog aantonen dat de eerder genoemde constructies in polynomiale tijd kunnen gebeuren. Dat wil zeggen dat we, gegeven een Circuit, de overeenkomstige Minesweeper configuratie moeten kunnen creëeren in polynomiale tijd (in functie van de grootte van de input w). Nu past elke constructie in een zogenaamde *bounding box*. Elke constructie past namelijk in een vierkant met breedte en hoogte N, met N groot genoeg. In een N $\times$ N vierkant kunnen we dan uiteraard elke constructie plaatsen. De niet opgevulde cellen bevatten dan een 0, zodat de Minesweeper configuratie consistent blijft.

Op deze manier kan de Minesweeper configuratie eenvoudig geconstrueerd worden. Een voorbeeld op de formule 3.1 is gegeven in figuur 5.18. Hierbij stelt een rechthoek de bounding box van de overeenkomstige poort voor, een cirkel de splitter, twee snijdende draden een crossover en de verticale streep een terminale. Let wel op dat de uitkomst van het SAT probleem true moet zijn: we moeten de terminale T dus true maken. Dus voor elk symbool uit de oorspronkelijke SAT-formule hebben we zo een bounding box nodig. Vermits we elke variabele ook moeten kunnen verbinden met elke andere variabele (omdat deze beide inputs kunnen zijn voor een poort) is de uiteindelijke complexiteit van de constructie begrensd door N^2n^2, met $n = |w|$. In figuur 5.19 wordt een visuele representatie gegeven van hoe de bounding box constructie eruit kan zien voor formule 3.1. Hierbij stelt de dikke, grijze lijn draad voor, 'B' een bocht, 'C' een crossover, 'E' een einde, 'T' een *true* einde (het eindresultaat moet true zijn) en 'S' een splitter.



Figuur 5.19: Een bounding box voorstelling van formule 3.1 voor Minesweeper[4].

5.4 Andere problemen

Er zijn nog een heleboel andere problemen aangaande het Minesweeper spel. In deze paragraaf zullen er hier enkele van gegeven worden, tesamen met hun complexiteitsresultaten. Echter zal er geen bewijs geleverd worden voor de verschillende problemen, deze zijn te vinden in [McP03].

5.4.1 k-Minesweeper

Definitie 5.4.1. *k-Minesweeper is een Minesweeper versie waarbij elk vakje maximaal k mijnen als buur heeft.*

Merk op dat de originele Minesweeper dus een 8-Minesweeper versie is, vermits er maximaal acht buren zijn en deze allemaal mijnen mogen zijn. In [BPM] wordt aangetoond dat alle k-Minesweeper versies, waarbij $3 \leq k \leq 8$ NP-compleet zijn. Om dit aan te tonen worden analoge constructies gebruikt als in het bewijs van Minesweeper. De enige uitzondering hierbij is dat enkel vakjes met waarde maximaal 3 mogen voorkomen.

Opmerking 5.4.1. We vermelden hier *maximaal* 3. Het is niet moeilijk om in te zien dat wanneer we een reductie kunnen verzinnen waarbij elk vakje van eender welke constructie maximaal de waarde 3 heeft, dat we dan

dezelfde constructies kunnen maken, waarbij de vakjes maximaal een waarde a hebben, met a groter of gelijk aan drie. Om dit in te zien kunnen we voor alle waarden a groter dan drie, de constructies nemen die gelden wanneer elke cel maximaal drie buurcellen heeft die een mijn bevatten. Uiteraard heeft elke cel in dit laatste geval dan een waarde kleiner of gelijk aan a ($3 \leq a$).

Open probleem: We vermelden nog dat er voor 2-Minesweeper geen complexiteitsresultaat bekend is. Algemeen wordt aangenomen dat 2-Minesweeper $\in$ P, hoewel men hier geen bewijs voor heeft.

5.4.2 Een andere oplossing

Definitie 5.4.2. *Gegeven een oplossing o_1 voor Minesweeper; bestaat er een oplossing o_2 voor Minesweeper, waarbij $o_1 \neq o_2$.*

Het *vind een andere oplossing* probleem (voor Minesweeper) bestaat er dus in, vertrekkende vanuit een gegeven oplossing, een andere oplossing te vinden voor hetzelfde probleem. Dit zou handig kunnen zijn om Minesweeper spellen te creëren die slecht één enkele oplossing hebben. Voor vele Minesweeper spelers is het immers enorm vervelend om, wanneer het spel bijna gespeeld is, een keuze te moeten maken tussen een aantal vakjes, waar zich een mijn kan bevinden. Vaak kan men niet met zekerheid zeggen waar de mijn zich juist bevindt, omdat er meerdere oplossingen zijn. Dit probleem zou getackled kunnen worden indien de Minesweeper generator[2] kan checken of er meerdere oplossingen mogelijk zijn. Zodoende kan hij vertrekkende van een leeg Minesweeper vlak, een spel opstellen dat slechts één oplossing heeft. Dit door telkens wanneer hij een mijn plaatst, te kijken of er meerdere oplossingen mogelijk zijn, zo ja, plaats de mijn dan ergens anders. Helaas komt men in [McP03] tot de volgende conclusie.

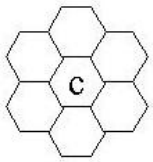
Stelling 5.4.1. *Vind een ander oplossing probleem voor Minesweeper is NP-compleet.*

[2]Met Minesweeper generator wordt hier bedoeld degene die het Minesweeper spel opstelt. Dit zou bijvoorbeeld een computer programma kunnen zijn.

Hoofdstuk 6

Minesweeper[6]

Zoals eerder aangegeven is Minesweeper[6] de variant van Minesweeper[4] waar de gridcellen regelmatige zeshoeken zijn. De zeshoeken vullen het vlak op, waardoor een soort honinggraad ontstaat[1]. Belangrijk op te merken hier is dat elke gridcel maximaal zes buren kan hebben (zie figuur 6.1). Dit zijn er minder dan de acht bij Minesweeper[4]. Dit heeft natuurlijk gevolgen voor het spelen van het spel. Er zijn minder buurcellen die je informatie kunnen geven aangaande de cel in kwestie. Het is dan ook moeilijker om te ontdekken of er zich nu al dan niet een mijn bevindt in de te onderzoeken cel, omdat de omgevingsinformatie geringer is. Ik ben op het idee gekomen om varianten



Figuur 6.1: De zes mogelijke buren van een gridcel c.

van het originele Minesweeper spel te bestuderen toen ik in een paper van Richard Kaye [Kay00a] naar het einde toe het volgende las:

> There is no reason why Minesweeper need be played on a square grid, and you will find plenty of other Minesweeper games on other grids available if you look on the web. I suspect that they are all NP-complete for much the same sorts of reasons.

[1]Minesweeper[6] is hier niet te verwarren met Hex, een spel dat ook in een vlak wordt gespeeld met regelmatige zeshoeken als gridcellen. Voor referentie zie http://www.krammer.nl/hex/hexrules.htm

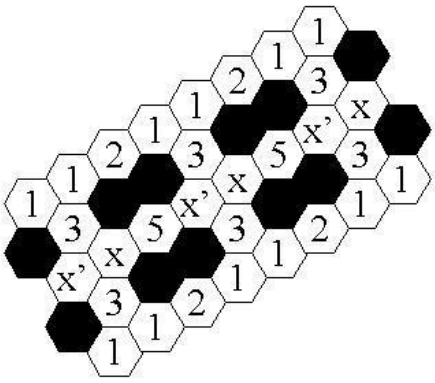
Desondanks het feit dat dit resultaat vrij aannemelijk klinkt, vond ik het interessant genoeg om het probleem aangaande verschillende varianten nader te bekijken. In dit en het volgende hoofdstuk ga ik dan ook dieper in op enkele varianten van Minesweeper die niet gespeeld worden op een grid met vierkante cellen. Alles wat van hieraf volgt in verband met Minesweeper is eigen werk (buiten de fundamenten waarop gesteund wordt; zie eerder).

6.1 Bedenkingen

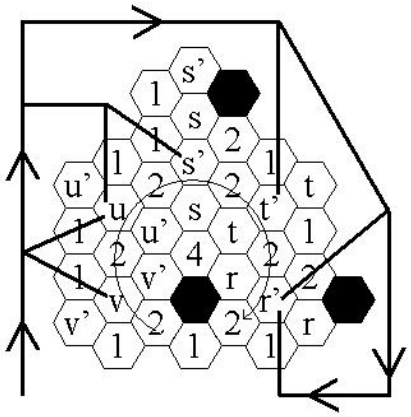
Vooraleer we tot de werkelijke constructies komen, filosoferen we eerst over mogelijke moeilijkheden die we kunnen tegenkomen bij het zoeken naar overeenkomstige constructies voor Minesweeper[6]. Het is namelijk zo dat we in dit geval twee buurcellen minder tot onzer beschikking hebben. Dit verschil is duidelijk, maar wat zijn de consequenties hiervan? Vermits nu met een andere indeling van het grid gewerkt wordt, kan dit eventueel ook extra complicaties opleveren. Enkele van de belangrijkste bemerkingen over het verschil tussen de bewijsvoering (constructiebouw) voor Minesweeper[4] en Minesweeper[6] komen nu aan bod. In de vorige sectie gebruikten we het *-symbool om gridcellen die een mijn bevatten voor te stellen, omdat dit de notatie is die gebruikt wordt in de originele artikels [Kay00a] en [Kay00b]. Vanaf nu echter zullen cellen die een mijn bevatten, voorgesteld worden door ze volledig zwart in te kleuren (eigen notatie).

Ten eerste is er het verschil voor de draad. Wanneer we dezelfde taktiek toepassen als eerder bekomen we een draad zoals gegeven in figuur 6.2 en het spreekt voor zich dat we van de overvloed aan mijnen afwillen. De reden waarom er zo veel mijnen hun intreden doen als we de draadconstructie letterlijk willen overnemen, is dat de gridcel boven de opening tussen twee opeenvolgende mijnen, verdwenen is. Hierdoor is er nog maar een enkele cel die twee opeenvolgende, van elkaar gelegen mijnen verbindt, en zo moeten de andere vier cellen wel van een mijn voorzien worden, omdat we niet op voorhand weten of de met x gelabelde cellen een mijn bevatten, of degene gelabeld met x'. Hoewel we deze constructie wel als draad kunnen gebruiken, willen we toch iets handiger om mee te werken tijdens de bouw van de andere constructies. Als conclusie kunnen we voorlopig stellen dat de afwezigheid van cellen kan leiden tot een gebrek aan verbindings- en/of keuzecellen. Dit leidt op zijn beurt tot onder andere de bovengenoemde complicatie.

Een tweede complicatie die kan optreden is wanneer we meer dan zes buren nodig hebben. Deze voorwaarde lijkt overwonnen te kunnen worden, omdat



Figuur 6.2: Bedenking 1 bij Minesweeper[6].



Figuur 6.3: Bedenking 2 bij Minesweeper[6].

elke constructie zo kan gevormd worden dat het maximum aantal buurcellen gelijk is aan drie (zie sectie over 3-Minesweeper[4]). Het is namelijk zo dat vaak vanuit de centrale cel meerdere draden aankomen en/of vertrekken en deze draden op zich hebben ook cellen nodig die ervoor zorgen dat het signaal gepropageerd wordt. Wat hiermee bedoeld wordt is dat een draad niet dikte één heeft, maar dikte drie. Zoals we bij de basisdraad constructie voor Minesweeper[4] bemerkten, hebben we niet altijd deze dikte nodig, omdat we ook bochten kunnen maken, alsook kunnen we door middel van een verbindingscel een cel overslaan. Desondanks moet er wel een onderscheid blijven tussen de signalen van verschillende draden tegelijkertijd. Bijvoorbeeld voor de AND-constructie bij Minesweeper[4] (zie figuur 5.15) kunnen de twee inkomende, de twee verbindende draden en de uitgaande draad, die samenkomen in de centrale vier links, netjes gescheiden worden. Wanneer we hetzelfde bij Minesweeper[6] proberen te doen, komen we al snel in de problemen. De uiteindelijke oplossing voor het probleem, die op een kleinere schaal ook toegepast wordt bij Minesweeper[4], is om de keuze voor een bepaald signaal bij een draad door te laten propageren in een andere draad. Deze techniek is waar te nemen door de binnenste circelvormige pijl van figuur 6.3 te volgen. Deze figuur representeert een deelconstructie van de gegeven AND-poort. Hierbij bepaalt de 2 onder de laatste v en v' dat er slechts één mijn ligt onder deze twee. Vervolgens bepaald de 2 tussen v, v', u en u' dat er zich slechts één mijn bevindt onder u en u', zoals het hoort. Dit proces gaat verder, helemaal rond tot bij de r. Merk op dat er nog een keuze moet gemaakt worden tussen ofwel u, ofwel u' (reps. s en s', t en t', r en r'). Een laatste, niet onbelangrijke bemerking dit kan gesteld worden is: mankeert er geen verbindingscel? We

zijn namelijk uitgegaan dat de keuze voor v de keuze voor u bepaalt, enzoverder, maar hoe weten we nu of u, dan wel u' een mijn bevat? Nu hebben we geluk, Minesweeper^6 biedt net genoeg ruimte om telkens waar nodig een keuzecel toe te voegen, net buiten de centrale constructie, die deze bepaling maakt. Deze cellen bevinden zich in figuur 6.3 op de aftakkingen van de pijl rondom de gehele constructie. Zodoende kan toch bepaald worden welke van de twee keuzes nu de mijn bevat.

Uiteindelijk zijn er zo een heel aantal problemen en akkefietjes die telkens opgelost moeten worden. De werkwijze is om eerst, hoe complex ook, een manier te vinden om de constructie te bouwen, waarbij gebruik gemaakt wordt van de kennis opgedaan bij de analyse van de originele constructie. Vaak kunnen deelconstructies van de ene constructie gebruikt worden als oplossing voor een probleem in een andere constructie. Ook is het handig om een bepaalde constructies zelf op te delen in deelconstructies en eerst de deelconstructies te simuleren om ze dan achteraf samen te voegen tot de complete constructie. Hierbij moet wel opgelet worden dat de verschillende deelconstructies wel degelijk op elkaar aangesloten kunnen worden. Uiteindelijk kan men dan de nodige optimalisaties uitvoeren om het geheel 'leesbaar' te houden. Dit is taktiek/techniek waar ik in deze en de volgende sectie gebruik van maak.

6.2 Constructies

Uiteraard zijn dezelfde constructies nodig als bij Minesweeper^4, omdat we tot eenzelfde resultaat willen komen, namelijk dat Minesweeper^6 NP-compleet is. In dit deel zal er vaak gesproken worden over een opening of een cel tussen x' en x. Dit zal steeds inhouden: gelezen van links naar rechts; omdat er in elke constructie na een x' wel ergens een x volgt en na de x een x'. We hebben dus ook hier nood aan een volgorde.

Draad (figuur 6.4)

Een eerste bemerking die onmiddellijk opvalt is dat de opening tussen x en x' (dus van links naar rechts) niet meer aanwezig is in Minesweeper^6. Deze constructie is mogelijk omdat de verbindingscellen in dit geval niet grenzen aan twee x-en, wat wel het geval zou zijn indien we bij Minesweeper^4 de opening zouden weglaten. De draadconstructie hier leunt dus meer aan bij een werkelijke, tastbare draad die een elektrisch signaal door kan laten. Het heeft echter geen gevolgen op de juistheid ervan. Voor de rest verloopt alles

analoog zoals bij Minesweeper[4]. Merk op dat dezelfde bemerkingen die hier gegeven zijn ook gelden voor de volgende constructies. Het grote verschil met Minesweeper[4] is wel dat de draad niet meer horizontaal loopt. Dit is echter geen probleem omdat alle constructies in dezelfde richting, onder dezelfde hoek (van 60^o) lopen. Hierdoor zouden we alle constructies kunnen roteren onder een hoek van 60^o, waardoor we alle constructies als horizontaal gelegen zouden kunnen beschouwd worden!

Bocht (figuur 6.5)

Het grote verschil hier ten opzichte van Minesweeper[4] is dat een bocht hier niet in een hoek van 90^o draait, maar in een van 120^o. Hierdoor is de buiging minder sterk. Gelukkig levert dit geen extra problemen op, omdat alle nieuwe constructies voor Minesweeper[6] juist de nood aan bochten van 120^o vereisen. De nieuwe grootte van de hoek is een logisch gevolg van het honinggraadachtige grid dat we gebruiken.

Splitter (figuur 6.7)

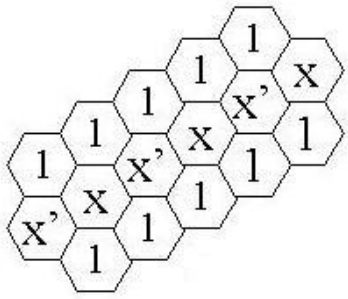
Zeer belangrijk op te merken is het verschil met de splitter bij Minesweeper[4] dat het oorspronkelijke signaal hier slechts in twee richtingen gepropageerd wordt. Zelfs wanneer het niet mogelijk zou zijn om een signaal in drie te splitsen in een stap (zoals bij Minesweeper[4]), is de constructie toch nog correct. Dit omdat we enkel hoeven te kunnen splitten in twee aparte signalen, waarvan er een van op zijn beurt weer gesplitst kan worden om zo een derde signaal te bekomen. Net zoals bij de draad lijkt de splitter bij Minesweeper[6] meer op een werkelijke splitter. De werking is dezelfde als eerder aangegeven, behalve dat de hoek er nu twee maal een van 120^o is, zodat de compabiliteit met de andere constructies gehandhaafd blijft.

Einde, phase changer en NOT-poort (Figuren 6.6, 6.8 en 6.9)

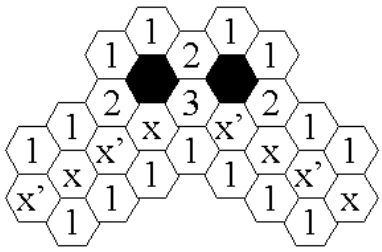
De principes van deze drie constructies zijn voor Minesweeper[6] gelijk aan die bij Minesweeper[4]. In dit geval is er geen verdere uitleg nodig en spreken de constructies voor zich. Er zal dan ook niet verder op ingegaan worden.

AND-poort (figuur 6.11)

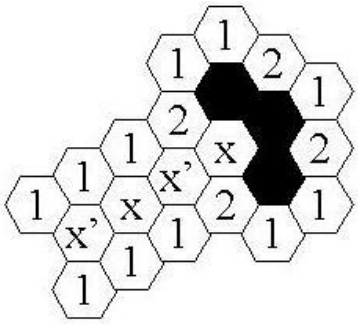
Bij Minesweeper[4] werd reeds de werking van de AND-poort uit de doeken gedaan. In dit deeltje zullen we de drie gebruikte deelconstructies nader bekijken. We doen echter geen uitspraken over de totale werking van de AND-poort, zie hiervoor naar de werking van de AND-poort in sectie 5.3.1.



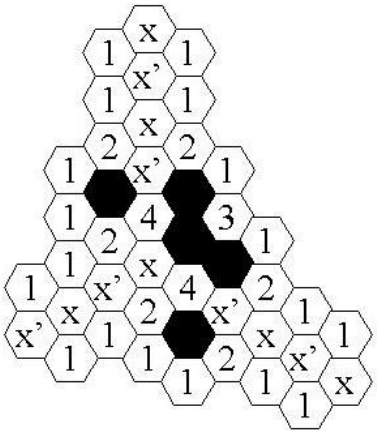
Figuur 6.4: Minesweeper[6]: Draad.



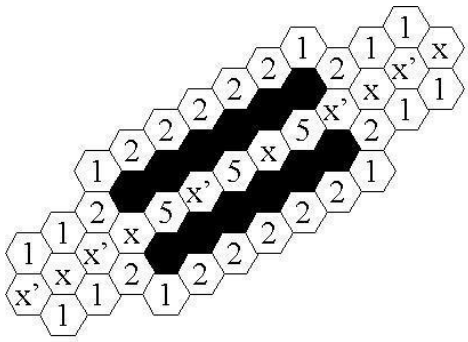
Figuur 6.5: Minesweeper[6]: Bocht.



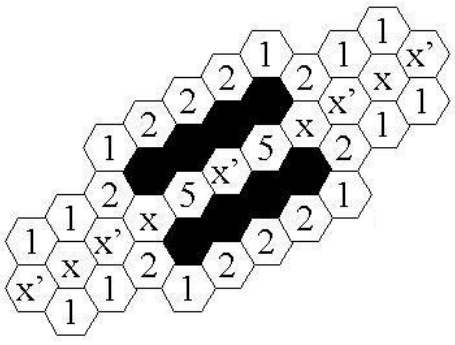
Figuur 6.6: Minesweeper[6]: Einde.



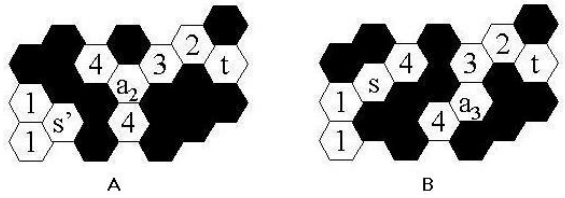
Figuur 6.7: Minesweeper[6]: Splitter.



Figuur 6.8: Minesweeper[6]: Phase changer.



Figuur 6.9: Minesweeper[6]: NOT-poort.



Figuur 6.10: Minesweeper[6]: AND-poort deelconstructie 1 (zie verder) bij t bevat geen mijn A) s bevat een mijn, B) s bevat geen mijn.

Er zijn drie van deze deelconstructies aanwezig in de constructie voor de AND-poort bij Minesweeper[6], deze zijn:

1. De eerste deelconstructie is de verbinding tussen de twee volgende.

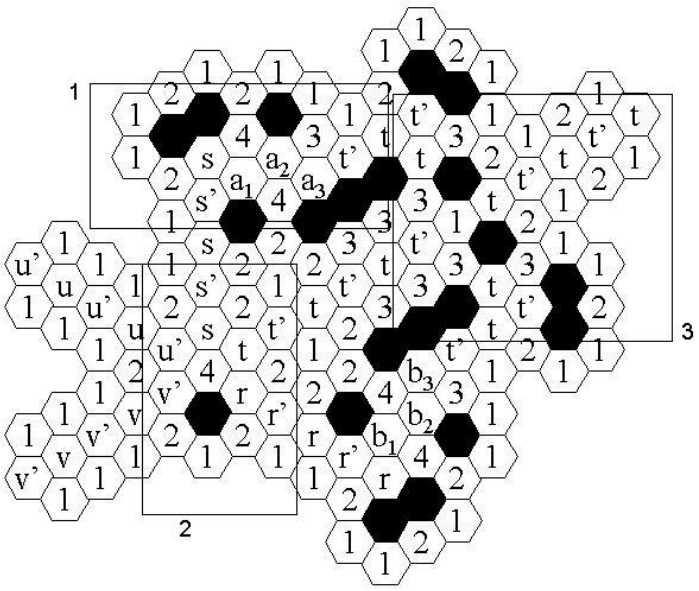
   (a) Als het uitgaande signaal $false$ is, dit wil zeggen t bevat geen mijn, dan moet de keuze voor het inputsignaal open gelaten worden. Dit komt in figuur 6.10 overeen met r en s al dan niet een mijn te laten bevatten. De twee mogelijkheden, r (s) bevat al dan niet een mijn zijn gegeven in deze figuur.

   (b) Als het uitgaande signaal $true$ is en t dus wel een mijn bevat, dan moeten de input steeds true zijn. Dit komt in figuur 6.10 overeen met r en s moeten in alle mogelijke gevallen een mijn bevatten.

2. Naar de tweede deelconstructie verwijs ik graag als zijnde het vijf-armen-kruispunt, omdat hier alle wegen (de draden u, v, r, s en t) samenkomen. Het spreekt van zich dat de hoofdbeslissingen in de AND-poort overeen komen met de beslissingen die hier gemaakt worden. Net zoals in 1 beschouwen we hier de twee mogelijkheden voor t. Ook zullen we gebruik maken van de aannames in 1.

   (a) Het simpelste geval is wanneer t een mijn bevat. Uit $1b$ weten we dan dat zowel r als s ook mijnen bevatten. De centraal gelegen cel met 4 buurmijnen heeft nu reeds vier gekende buurmijnen, er is dus geen ruimte meer voor extra buurmijnen. Het gevolg is dat u' en v' geen mijn meer kunnen bevatten. Dus bevatten logischerwijs u en v wel een mijn. De inputsignalen zijn dus beiden $true$ in dit geval.

   (b) De tweede case is iets gecompliceerder. We stellen dus dat t geen mijn bevat. Uit $1a$ weten we dat r en s zowel een mijn, als geen mijn kunnen bevatten. Nu moet onze centrale vier dus nog drie onbekende buurmijnen hebben, die gekozen moeten worden uit u',

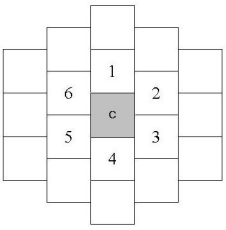


Figuur 6.11: Minesweeper[6]: AND-poort en de bijbehorende deelconstructies.

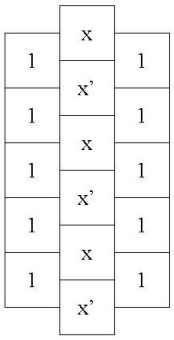
v', r en s (er is reeds een buurmijn zichtbaar en van t weten we dat die geen mijn bevat). Uit dit alles kunnen we niet anders dan besluiten dat minstens u' of $'v$ een mijn bevat. Dit om wille van de centrale vier die anders nooit meer aan vier buurmijnen kan geraken, waardoor we een inconsistente configuratie zouden achterlaten. Het zou ook kunnen dat ze beiden een mijn bevatten. Al het vorige komt overeen met stellen dat in dit geval minstens een van de inkomende signalen $false$ moet zijn.

3. De laatste deelconstructie is een simpel kruispunt. Het signaal dat van links naar rechts gaat, wordt geïnverteerd en loopt daarna verder naar rechts toe (waar het nog eens geïnverteerd wordt). Tegelijkertijd splitst het signaal naar boven en onder toe. Het gesplitste signaal is gelijk aan het geïnverteerde signaal van dat van het oorspronkelijke signaal dat van links kwam.

De eigenschappen van de deelconstructies zijn een noodzaak om de gehele AND-constructie te laten werken. Ze werken wonderwel samen en vormen zo de moeilijkste constructie.



Figuur 6.12: Geschraagde Minesweeper[4] en de buurcellen.



Figuur 6.13: Mogelijke draad constructie bij geschragde Minesweeper[4].

6.2.1 Verband met Minesweeper[4]

Wanneer we bij Minesweeper[4] de opeenvolgende vierkanten niet naast elkaar plaatsen, maar geschraagd, dan bekomen we hetzelfde probleem als bij Minesweeper[6]. Zie figuur 6.12 voor een geschraagde indeling van de vierkanten en het aantal buren van een cel in dat geval. De gelijkenis met Minesweeper[4] is overduidelijk. Een mogelijke voorstelling van de draad constructie gebruik makende van deze geschraagde indeling is gegeven in figuur 6.13. Net zoals bij Minesweeper[6] is de bocht in de bocht-constructie 120^o groot. Dit geeft echter om dezelfde reden geen problemen.

Stelling 6.2.1. *Minesweeper[6] is NP-compleet.*

Eens we de juiste constructies voor de verschillende componeneten gevonden hebben, kunnen we een redenering volgen, die analoog is aan die voor Minesweeper[4]. Dat Minesweeper[6] $\in$ NP leiden we op eenzelfde manier af, door niet-deterministisch te gokken waar de mijnen zich bevinden. Daarna kan in polynomiale tijd nagegaan worden of de nieuwe positie consistent is als volgt (op basis van [McP03]).

Gegeven een grid G en een toekenning van mijnen α. Deze α kent mijnen toe aan cellen. Doe dan het volgende:

1. Kijk na of elke mijn in α zich in een cel bevindt waar nog geen toekenning aan is gegeven. De mijnen moeten met andere woorden in cellen komen waar nog geen gegevens over bekend waren. Als er een mijn is in de toekenning, die geplaatst wordt in een cel die reeds toegekend was, VERWERP dan.

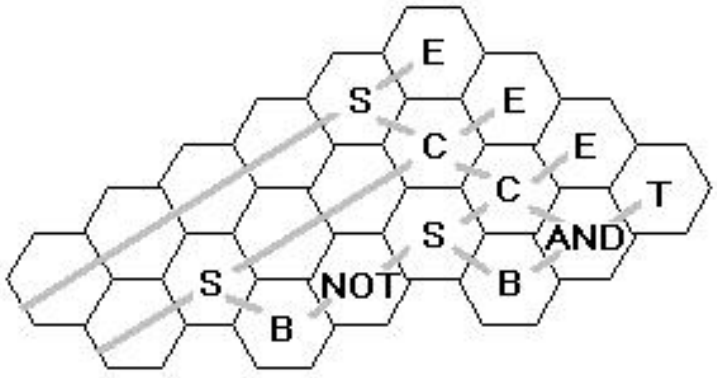
2. Voor elke cel c in G die als mijn aangeduid is, doe voor *alle* buren van c het volgende:
   (a) Als de buur het getal 0 bevat, VERWERP dan.
   (b) Als de buur een getal bevat verschillend van 0, doe dan dat getal - 1.
3. Voor elke cel c in G, als c een getal bevat, verschillend van 0, VERWERP dan.
4. AANVAARD.

De constructies kunnen in polynomiale tijd in functie van de grootte van de input worden opgesteld. Wederom stellen we de bounding boxen op en kunnen de verschillende constructies op dezelfde manier op mekaar aangesloten worden. De bounding box moet aan twee voorwaarden voldoen:

- De bounding box moet groot genoeg zijn. Elke constructie moet er vlot inpassen zodat ook aan de volgende voorwaarde voldaan kan worden. Op deze manier bekomen we een vaste grootte, die in de complexiteit wordt voorgesteld door de constante N.
- De bounding boxen moeten perfect op elkaar passen. Om hieraan te voldoen zijn wederom twee voorwaarden nodig:
  1. De ingang(en)/uitgang(en) moeten zich op dezelfde plaats bevinden. Anders zou het uitgaande signaal van de ene box niet het inputsignaal van de volgende kunnen zijn.
  2. De verbinding die door het vorige puntje ontstaat tussen twee bounding boxen moet ook het juiste signaal propageren. Hiermee wordt bedoeldt dat wanneer de uitgang van de ene box het $false$-signaal propageert, dan moet de ingang van de aangrenzende box logischerwijze ook het $false$-signaal verder propageren.

De opvulling van de bounding boxen gebeurt door gridcellen die gelabeld zijn met een 0. We beschouwen dus nergens anders bommen, waardoor we het vlak consistent kunnen vullen. Enkel aan de in- en uitgangen moet het signaal doorgegeven worden en hebben we dus een overgang waar wel cellen met een ander label dan 0 kunnen voorkomen, maar deze worden enkel gevormd door het door te geven signaal. Tussen de verschillende constructies liggen *lang genoege* draden, zodat we ook op dit punt geen problemen zouden tegen komen. Voor de vorm van de bounding boxen kiezen we voor een regelmatige zeshoek. Vermits we in dit geval werken met hoeken van 120^o,

zijn vierkantige bounding boxen, zoals bij Minesweeper[4], hier zeer onhandig. Door een zeshoekige vorm voor de bounding box te kiezen, kunnen de eerder gegeven constructies vlot op mekaar aangesloten worden, het signaal kan dan immers op een consistente manier doorgegeven worden (de overgang van een signaal verloopt in een rechte lijn, zoals we verwachten). Zie figuur 6.14 voor een mogelijke voorstelling van formule 3.1 met bounding boxen voor Minesweeper[6]. Hierbij stelt de dikke, grijze lijn draad voor, 'B' een bocht, 'C' een crossover, 'E' een einde, 'T' een *true* einde (de formule moet immers *waar* gemaakt worden) en 'S' een splitter. Zoals men in de figuur reeds kan zien, zijn er telkens twee zijden (dit zijn telkens dezelfde) van de hexagonale bounding box die we niet gebruiken om een signaal door te geven, waardoor er nog vier in gebruik zijn. Hierdoor zijn we consistent met de bounding box constructie uit Minesweeper[4].



Figuur 6.14: Bounding box voorstelling van formule 3.1 voor Minesweeper[6].

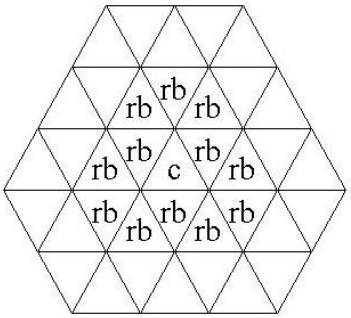
Stelling 6.2.2. *Men kan de eerder genoemde bounding boxen construeren voor eender welke constructie.*

We concluderen dat Minesweeper[6] NP-compleet is.

Hoofdstuk 7

Minesweeper^3

Deze Minesweepervariant heeft een eerder chaotisch uitzicht. Regelmatige driehoeken wisselen elkaar af in het vlak. Alle centraal gelegen driehoeken hebben twaalf buren (zie figuur 7.1). Merk op dat dit er dubbel zo veel zijn dan bij Minesweeper^6 en anderhalve keer meer dan bij Minesweeper^4. Dit kan grote gevolgen hebben op het spelen van zulk een spel. Men zou kunnen stellen dat het extraheren van de aanwezige informatie niet zo voor de hand liggend is als bij de twee eerder vernoemde varianten. Voor de menselijke gebruiker ligt het niet zo eenvoudig om hierbij het bos door de bomen te zien, vermits het opmerken van de buren niet voor de hand liggend is. Er zijn namelijk slechts drie directe buren (buren die een zijde gemeen hebben). Buiten deze drie zijn er nog negen andere buren die slechts een hoekpunt gemeen hebben. In totaal zijn er dus twaalf buren. Deze twaalf hebben op hun beurt vierentwintig andere buren (hiermee bedoelen we buren (cellen) die nog niet beschouwd zijn). Verduidelijking vindt men in figuur 7.1, waar de rechtstreekse buren van een cel met label 'c' voorgesteld zijn door het label 'rb'. Ter informatie: bij Minesweeper^4 en Minesweeper^6 zijn er respectievelijk zestien en twaalf onrechtstreekse buren. Het zal dan ook niet verbazen dat de nodige constructies eerder chaotisch zijn ten opzichte van de andere twee Minesweeper varianten. Wel is het zo dat ze op een analoge manier opgebouwd zijn en dat de principes achter elke constructie dezelfde blijven. Bij sommige constructies hoort echter toch nog een woordje uitleg om de verschillen te verduidelijken. De constructies horende bij Minesweeper^3 kunnen gevonden worden in figuren 7.3 tot en met 7.11.



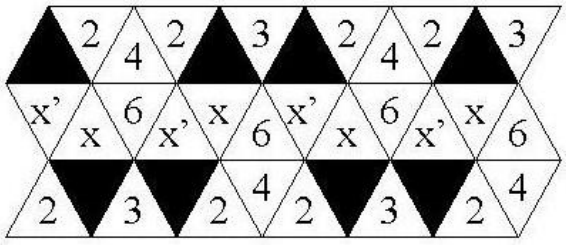
Figuur 7.1: De rechtstreekse buren (rb) van een gridcel (c).

7.1 Bedenkingen

Net zoals bij Minesweeper[6] kunnen we ook hier enkele bedenkingen maken met betrekking tot de verschillen tussen de constructies in kwestie. Zoals eerder aangehaald hebben we nu te maken me maar liefst twaalf buren. Men zou denken dat dit grote aantal buren een voordeel oplevert bij het construeren, omdat er dan meer ruimte aanwezig is om de delen te bouwen. De beperking die er was bij Minesweeper[6], namelijk dat er minder buren waren, is hier niet meer aanwezig. Echter de toevoeging van extra buren levert nieuwe problemen op. Zo rijkt een cel verder, waardoor bepaalde delen, die eerst niet aan alkaar grensden (en ook niet aan elkaar mogen grenzen), nu plots wel aanliggend worden. Door het probleem van de nabijheid zijn vaak warrige constructies nodig. Het feit dat een cel veel buren heeft, heeft ook een invloed op de complexiteit van de constructies. Een cel die aan een x grenst, maar niet aan een x', en die niet als bindings- of keuzecel dient, moet voorzien worden van een mijn omdat we niet weten of de een mijn onder de x of onder de x' ligt. Dit komt vaak voor bij Minesweeper[3].

Net zoals de conclusie bij de draad bij Minesweeper[6] kan men hier ook concluderen dat wanneer men de techniek van Minesweeper[4] overneemt, de draad log en onhandig wordt om mee te werken. De overeenkomstige constructie is gegeven in figuur 7.2. Wat meteen opvalt is de wederkerende 6 in het midden. Dit is een vrij groot getal, enkel voor de draad alleen al, zonder extra's. Wel is het een "mooie" constructie voor Minesweeper[3], echter om er mee te werken missen we iets. De uiteindelijk draad zal er dan ook iets anders uitzien, maar komt even mooi uit.

Wanneer we de constructies letterlijk zouden willen overnemen komen we al snel het eerder vernoemde probleem van de nabijheid tegen. Kijk maar naar



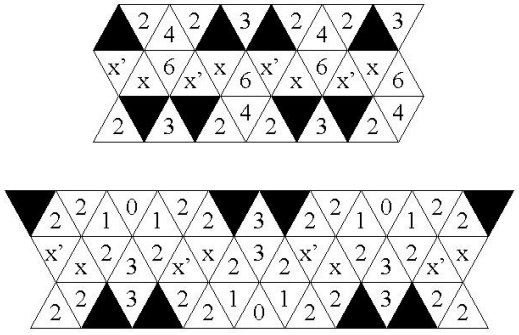
Figuur 7.2: Bedenking 1 bij Minesweeper[3].

de splitter constructie (figuur 7.6), waar de centrale 6 zo goed als volledig geëncapsuleerd is in mijnen. Wanneer deze mijnen niet aanwezig zouden zijn, verbond deze 6 zowel de x van de draad die naar boven toeloopt, als de draad die naar onder toeloopt, alsook zouden er dan verbindingen zijn tussen twee x'en en een x en dergelijke. Het spreekt voor zich dat voor de splitter constructie dit alles niet mag voorkomen. Het probleem dat zich hier duidelijk stelt is dat de cellen een te groot bereik hebben. Juist door hun grote bereik interfereren ze met andere deelmechanismen waar ze geen deel van uitmaken. Hier moet bij Minesweeper[3] wel degelijk mee rekening gehouden worden.

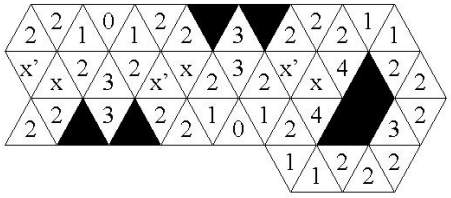
Door de vaak warrige en complexe constructies van Minesweeper[3] wordt voor de AND-constructie enkel de deelconstructies gegeven. Het aaneensluiten is voor de hand liggend, maar zou een grote figuur opleveren. Ook zijn er net zoals bij Minesweeper[6] nog andere problemen die overwonnen moeten worden en die niet aan bod kwamen.

7.2 Draad, einde, NOT- en AND-poort

Zoals eerder opgemerkt zullen er nu veel meer mijnen aanwezig zijn in de constructies dan bij Minesweeper[4] en Minesweeper[6]. Dit volgt uit het grote aantal buren van een gridcel, met als gevolg dat buren soms eerder *ver* van elkaar komen te liggen. Daarom raken enkele cellen niet aan zowel een x als een x', maar wel aan een van de twee. Omdat we echter de keuze willen laten of een x of een x' al dan niet een mijn bevat, kunnen we geen uitspraak doen over de inhoud van deze cellen en moeten we er een mijn in plaatsen. Ook kan het voorkomen dat een cel aan twee $x$en of aan twee x'en grenst. Door dezelfde, eerder gegeven reden, kunnen we ook over deze cellen niets zeggen en moeten we er wedeom een mijn in plaatsen. Als laatste opmerking kan men stellen dat het aantal buurmijnen groot kan worden (tot 12), dit is ook te zien in de constructie van de AND-poort, waar een 10 voorkomt.



Figuur 7.3: Minesweeper^3: Draad.



Figuur 7.4: Minesweeper^3: Einde.

De AND-constructie is wel opgedeeld in drie delen. Deel twee past boven en onder (twee maal dus) deel één, terwijl deel drie rechts van deel één hoort en de verbinding vormt tussen de twee AND-deel2 (figuur 7.9) subconstructies.

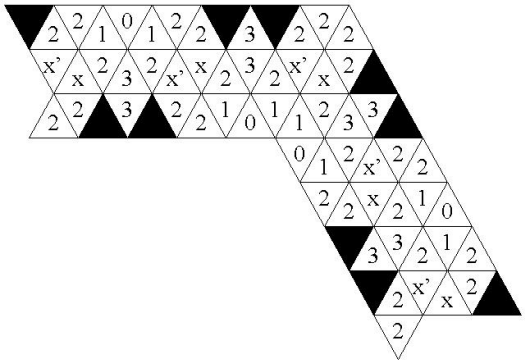
7.3 Splitter en bocht

De splitter-constructie bij Minesweeper^3 is analoog aan die bij Minesweeper^6. Ook hier wordt het signaal opgesplitst in twee dezelfde signalen en geldt hetzelfde resultaat als daar besproken (deel 6.2). Dit alles gebeurt hier ook onder een hoek van 120^o en net zoals bij Minesweeper^6 is dit wederom net wat we willen om compabiliteit met de andere constructies te behouden. Mede wordt dit gerealiseerd door de bocht-constructie die ook een hoek van 120^o aanhoudt. Het verschilt dus op dezelfde manier als bij Minesweeper^6 van de splitter-constructie bij Minesweeper^4.

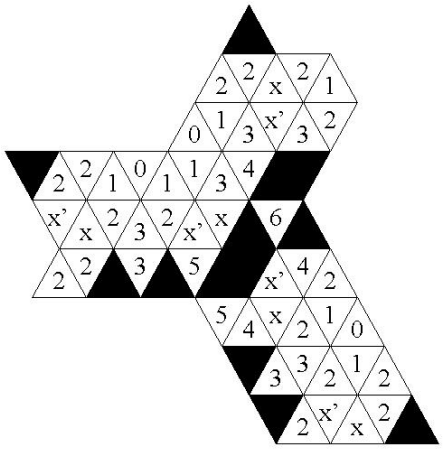
7.4 Opmerking

In de Minesweeper^3 versie hier besproken, wordt er enkel gebruik gemaakt van regelmatige driehoeken die ook regelmatig over het vlak verdeeld liggen. Er zijn ook andere indelingen van het vlak in driehoeken te bedenken. Als voorbeeld kan men gelijkbenige driehoeken nemen, of willekeurige driehoeken met verschillende grootttes. Natuurlijk zijn er nog meerdere mogelijk, de meeste ervan zullen ongetwijfeld weinig nut hebben. Ik durf met een grote zekerheid zeggen dat vele van deze opdelingen, waarin men het Minesweeper spel zou gaan spelen, ook NP-compleet zijn.

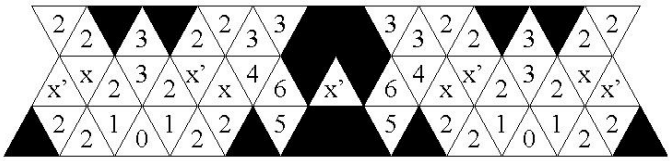
Stelling 7.4.1. *Minesweeper^3 is NP-compleet.*



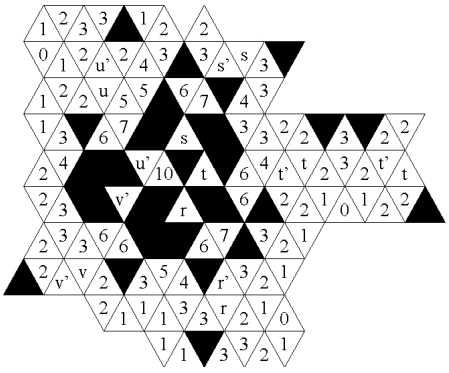
Figuur 7.5: Minesweeper^3: Bocht.



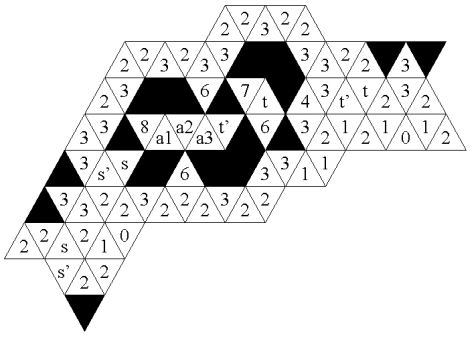
Figuur 7.6: Minesweeper^3: Splitter.



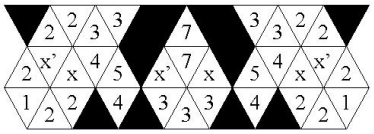
Figuur 7.7: Minesweeper^3: NOT-poort.



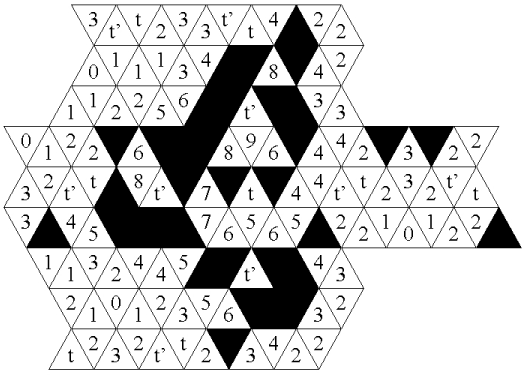
Figuur 7.8: Minesweeper^3: AND-poort deel1.



Figuur 7.9: Minesweeper^3: AND-poort deel2.



Figuur 7.10: Minesweeper^3: Phase changer.



Figuur 7.11: Minesweeper^3: AND-poort deel3.

Eens we inventief genoeg zijn geweest om alle constructies die nodig zijn voor de reductie te vinden, kunnen we de NP-compleetheid van ons probleem gaan aantonen, op een manier die analoog is aan degene gebruikt bij Minesweeper^4 en Minesweeper^6. Dat Minesweeper$^3 \in$ NP leiden we op een identieke manier af als voordien, door niet-deterministisch te gokken waar de mijnen zich bevinden. Daarna kan in polynomiale tijd nagegaan worden of de nieuwe positie consistent op een manier analoog aan degene gebruikt bij Minesweeper^6.

De constructies kunnen in polynomiale tijd in functie van de input worden opgesteld. Wederom stellen we de bounding boxen op en kunnen de verschillende constructies op dezelfde manier op mekaar aangesloten worden. De opvulling van deze bounding boxen, die ook in dit geval hexagonaal zijn zoals bij Minesweeper^6, gebeurt wederom door cellen die geen mijn bevatten. Zodoende zijn deze allemaal gelabeld met een 0. Enkel de input- en outputsignalen moeten in elke bounding box op eenzelfde, eenvormige manier voorkomen. De reden waarom we hier ook voor hexagonale bounding boxen kiezen, is dat de gebruikte constructies dezelfde hoek maken als bij Minesweeper^6, namelijk een van 120^o. Dit in tegenstelling tot Minesweeper^4 waar de constructies gebruik maken van hoeken van 90^o en waarvoor vierkanten dus meer in aanmerking komen.

Stelling 7.4.2. *Men kan de nodige bounding boxen voor Minesweeper^3 construeren voor eender welke constructie.*

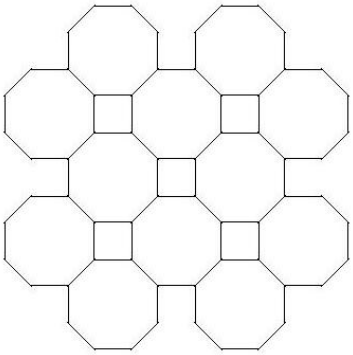
We concluderen dat Minesweeper^3 NP-compleet is.

Hoofdstuk 8

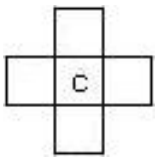
Andere versies

Het is niet voor niets denk ik dat de originele Minesweeper versie van Windows zo veel succes heeft en anderen veel minder succes kennen/kenden. Dit heeft ongetwijfeld te maken met een goede indeling van het vlak. Varianten zijn echter wel interessant om te bekijken, en sommige ongetwijfeld ook om te spelen. Kijk maar naar bijvoorbeeld Hex-Minesweeper (= Minesweeper[6]), waarvan er meerdere implementaties te vinden zijn op het World Wide Web[1].

8.1 Minesweeper[8]



Figuur 8.1: Een grid verdeeld met regelmatige achthoeken.



Figuur 8.2: De vier buren van een cel c als de diagonale buren niet meetellen.

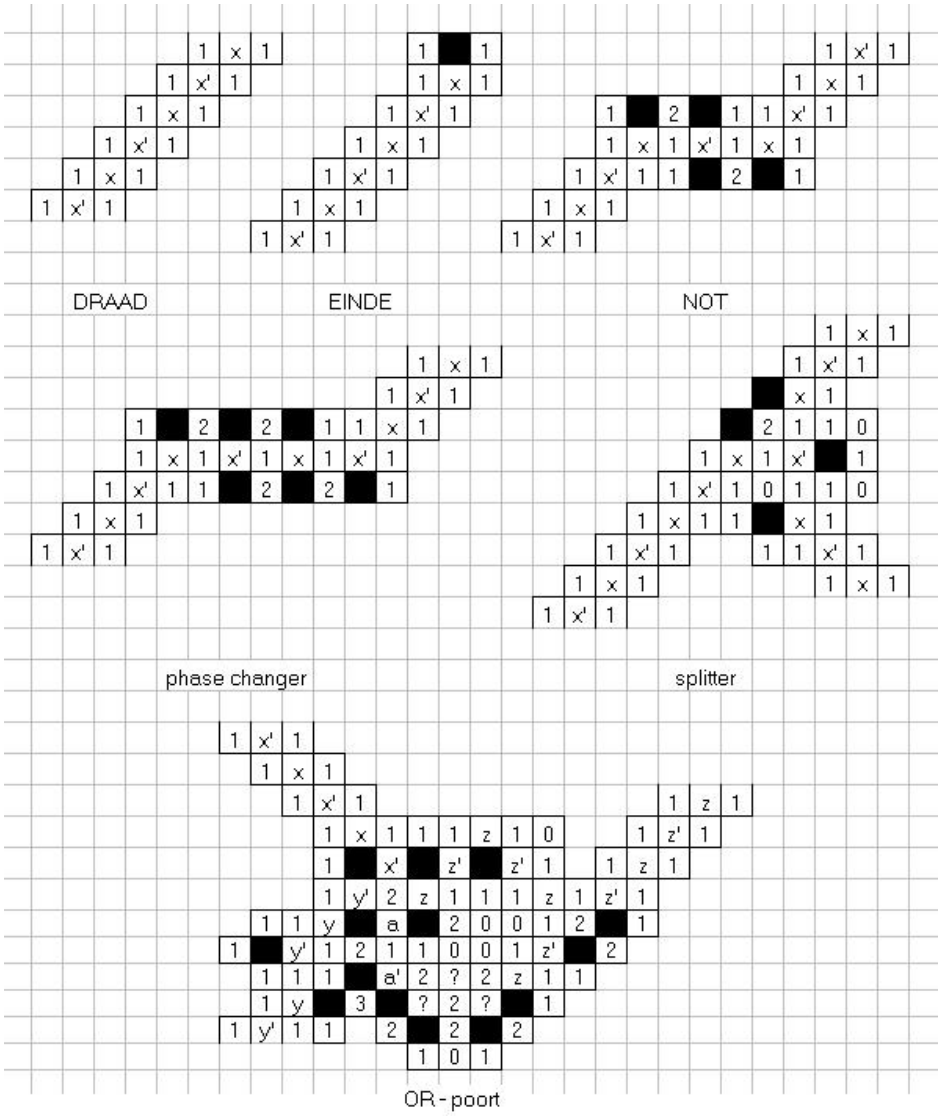
[1]Onder andere is er een versie te vinden op http://www.novelgames.com/flashgames/popup.php?id=9

Voor Minesweeper[8] ziet de opvulling van het vlak eruit als in figuur 8.1. Wat aan deze figuur meteen opvalt is dat regelmatige achthoeken niet het volledige vlak opvullen, wat voordien wel steeds mogelijk was. Om compatibel te blijven met de vorige paragrafen, laten we datgene van het vlak dat niet opgevuld wordt, buiten beschouwing. Wanneer de overblijvende vierkanten ook als gridcel bekeken worden, kunnen we vrijwel zeker dezelfde constructies maken als eerder[2], hier gaan we niet op in. Wanneer de overblijvende vierkanten niet als gridcellen gezien worden, kunnen we het volgende bemerken: Minesweeper[8] is equivalent met Minesweeper[4], waarbij de diagonaal gelegen buurcellen niet meetellen om het aantal buren van een bepaalde cel weer te geven (figuur 8.2). De vraag is of de beperking in aantal buren een ander complexiteitsresultaat tot gevolg heeft. Wel is het mogelijk om alle nodige constructies, met uitzondering van de AND-poort, te bouwen. Het bouwen van de AND-poort analoog zoals in Minesweeper[4] is onmogelijk, en dit omdat er daar een cel is die minstens vijf buren nodig heeft. Dit is namelijk de centrale spilcel in een van de deelconstructies. Desondanks dit is het probleem toch NP-compleet, want het is namelijk wel mogelijk om de OR-poort te bouwen. Alle constructies zijn gegeven in figuur 8.3.

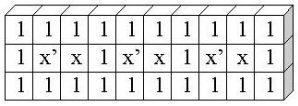
8.2 3D-Minesweeper

Iets wat bij velen tot de verbeelding spreekt is 3D-Minesweeper. Er zijn reeds enkele implementaties online te vinden van dit spel en het is zeer leuk om zelf eens te spelen. Echter is het nog niet in die mate bestudeerd dat er andere complexiteitsresultaten voor bekend zijn. Wel is het zo dat dezelfde resultaten zeker moeten gelden als voor de 2D versie. De gebruikte constructies kunnen namelijk in de hoogte "gelift" worden, zodat we meteen de nodige drie-dimensionale constructie bekomen. Dit is te zien in figuur 8.4 voor de draad-constructie. Omdat we in 3D een vrijheidsgraad meer hebben, zou het kunnen dat bijvoorbeeld 2-3D-Minesweeper (3D-Minesweeper waarbij elk vakje maximaal 2 mijnen als buur heeft) wel NP-compleet is. Hierbij moet wel opgemerkt worden dat we in 3D-Minesweeper met zesentwintig buren te maken hebben, in plaats van acht bij 2D-Minesweeper. Er zijn ook 3D-Minesweeper versies waarbij enkel de horizontale en verticale buren meetellen. Hierdoor worden slechts zes van de zesentwintig buren in rekening gebracht. Deze laatste versie komt echter niet overeen met het originele spel, waarbij de diagonaal gelegen buren wel meetellen. Een constructie die wel simpel te bouwen is bij 3D-Minesweeper, en die werkelijk de extra dimensie

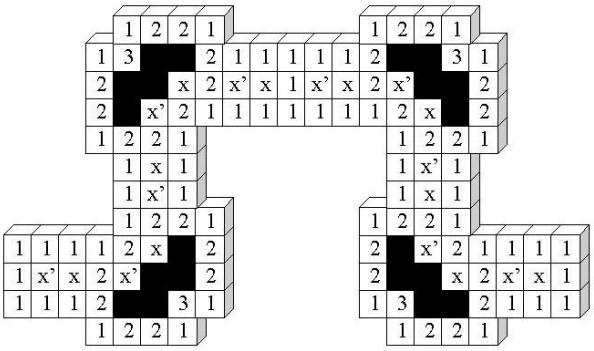
[2]Dit is echter een veronderstelling en niet aangetoond



Figuur 8.3: De constructies voor Minesweeper[8](hierbij wordt de voorstelling door middel van de vierkantjes uit figuur 8.2 gevolgd).



Figuur 8.4: 3D-Minesweeper: Draad.



Figuur 8.5: 3D-Minesweeper: Brug.

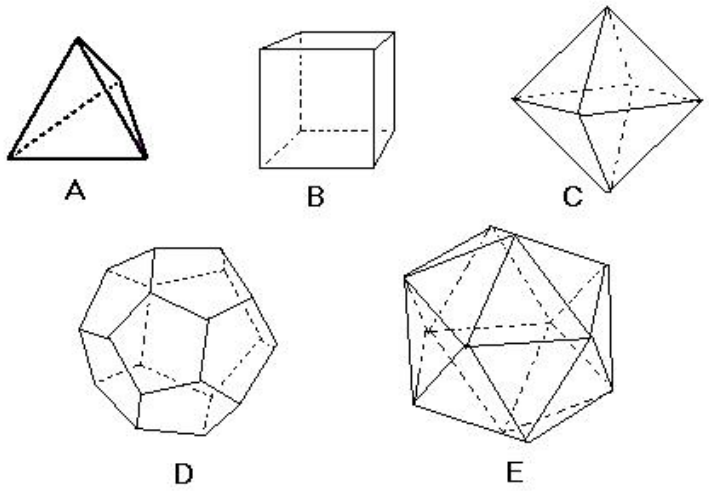
benut, is de crossover. Door de extra vrijheidsgraad kunnen we nu een constructie in de hoogte laten gaan. Het is dus perfect mogelijk om een draad door middel van een *bocht naar boven* over een andere draad heen te laten propageren, zonder dat die met de onderliggende draad interageert. Deze constructie, die lijkt op een brug, is te zien in figuur 8.5 en spreekt voor zich.

We zouden zelfs voorbij 3D-Minesweeper naar 4D-Minesweeper, 5D-Minesweeper enzoverder kunnen gaan kijken, en hoogstwaarschijnlijk op identieke wijze dezelfde resultaten aantonen.

8.3 3D-Minesweeper$^{n^m}$

Bij de 3D versie hoeven we ons uiteraard niet te beperken tot kubussen. Net zoals we in het vlak een andere indeling kozen door gridcellen te nemen die niet vierhoekig waren, kunnen we dit ook doen wanneer we overgaan naar een derde dimensie. In deze context kunnen we de vijf platonische figuren bespreken. Zo bekomen we 3D-Minesweeper$^{3^4}$, 3D-Minesweeper$^{4^6}$, 3D-Minesweeper$^{3^8}$, 3D-Minesweeper$^{5^{12}}$ en 3D-Minesweeper$^{3^{20}}$. We moeten dan wel specifieren wat we hiermee bedoelen. De eerste macht staat voor het aantal hoeken van elk regelmatig vlak. De tweede macht staat voor het aantal van die regelmatige vlakken in de figuur. Zo vormen de gegeven versies respectievelijk een tetraeder, een kubus, een oktaeder, een dodekaeder en een isocaeder (zie figuur 8.6[3]). Over hoe het opvullen van de ruimte in deze gevallen gebeurt, spreken we ons niet uit. Ongetwijfeld zullen we hier dezelfde constructies mee kunnen bouwen. Belangrijk hierbij is dat we nu

[3]Bron: http://www.enchantedlearning.com/math/geometry/solids



Figuur 8.6: Platonische figuren: A) Tetraeder, B) Kubus, C) Octaeder, D) Dodecaeder, E) Isocaeder.

de 2D constructies niet altijd zomaar in de hoogte kunnen liften. Ook zou men veel gecompliceerdere constructies in de ruimte kunnen bouwen dan degene met regelmatige vlakken. Het zou zelfs kunnen dat men een hele hoop verschillende vlakken gebruikt om zo een ruimtelijke figuur te vormen. De mogelijkheden zijn (bijna) eindeloos.

8.4 Tilings

Een begrip dat nauw verwant is met opvullingen van een vlak is *Tilings*, verwant met de eerder besproken tilings, maar niet hetzelfde. Hiervan is er ook een hele theorie, waarop we echter niet in willen gaan. Een grote hoeveelheid van Tilings zijn onder andere te vinden op http://www.scienceu.com/geometry/articles/tiling. Minesweeper kan immers gespeeld worden op alle mogelijke indelingen van het vlak. Vele onder hen zijn minder interessant.

Hoofdstuk 9

Heuristieken voor Minesweeper

Het feit dat het beste algoritme dat we kennen voor Minesweeper (welke besproken versie dan ook) alle mogelijkheden afgaat om tot de oplossing te komen, wil niet zeggen dat we bij de pakken moeten blijven zitten. Een techniek om (meestal) sneller tot een oplossing te komen, zijn heuristieken. Heuristieken geven niet (noodzakelijk) de beste oplossing, maar geven veelal wel een goede benadering. Met andere woorden geven ze een bepaalde garantie. Nu zijn er reeds vele technieken ontwikkeld om NP-complete problemen toch in polynomiale tijd, bij benadering op te lossen. Het voor de hand liggende voordeel is dat we niet meer al die mogelijkheden moeten afzoeken. Het nadeel dat we met heuristieken hebben is dat we niet (meestal toch niet) de juiste oplossing hebben, maar slechts één die de juiste oplossing benadert. Gelukkig kennen we in de meeste gevallen de fout die we maken ten opzichte van de perfecte oplossing die we zouden vinden door alle mogelijkheden af te gaan.

Er zijn reeds goede heuristieken bekend voor welbepaalde NP-complete problemen, en soms kunnen we die heuristiek aanpassen zodat die ook werkt voor andere problemen. Helaas is dit niet altijd het geval en moeten we op zoek naar een aparte heuristiek. Een voorbeeld van reeds gekende, goede heuristieken zijn die voor het *Traveling Sales Man* probleem, waarbij een vertegenwoordiger een aantal steden moet bezoeken, en dit zodat hij de kortst mogelijke afstand aflegt en terugkomt in de beginstad. Een gekende techniek hierbij is de *kortste buur* (in het Engels *nearest neighbour*). Bij de korste buur heuristiek kiezen we een beginstad en vanuit deze stad gaan we naar de stad die zich hier het dichtste bij bevindt en zo gaan we verder. Nadelig hierbij is dat naar het einde toe de mogelijkheden kleiner worden en vaak lange afstanden moeten afgelegd worden; bijvoorbeeld om terug in de beginstad te geraken. Een betere heuristiek die nog steeds eenvoudig te begrijpen

is, is die van de *grootste hoek*. Voor Minesweeper bekijken we ook twee eenvoudige methoden die kunnen helpen bij het zoeken naar een oplossing. Vermits (Circuit) SAT reduceert tot Minesweeper, bestaat de mogelijkheid dat heuristieken voor het eerste probleem afgeleid kan worden tot een voor het tweede, maar hier gaan we niet op in. Wel bekijken we twee methoden die in/als een heuristiek kunnen gebruikt worden. Terzijde vermelden we ook nog dat heuristieken welliswaar een afwijking kunnen garanderen, maar dit niet hoeven te doen. Ook is het zo dat het slechts een manier geeft om een oplossing voor Minesweeper te assisteren, niet te geven. Voor de bespreking van dit hoofdstuk baseer ik me deels op mijn eigen ervaringen bij het spelen van Minesweeper als ook op [Wes].

9.1 Pattern matching

Bij pattern matching gaan we op zoek naar bepaalde, bekende patronen waarin we reeds weten welke cellen als veilig beschouwd kunnen worden, en welke niet. In onze Minesweeper configuratie proberen we zulke reeds gekende patronen te vinden. Eens deze gevonden kijken we wat gedaan in dit geval. Men zou bijvoorbeeld een kleine database kunnen aanleggen met de meest voorkomende patronen om hierin dan te gaan opzoeken. Let wel op dat het dan geen heuristiek meer is, maar een perfecte oplossing, het wordt pas een heuristiek in de volgende paragraaf. De voorbeelden zijn simplistisch gehouden, het valt te voorspellen dat veel complexere gevallen mogelijk zijn, zeker als het zoekdomein groot is.

Zo zijn er reeds een heleboel *standaard* situaties bekend, die eenvoudig en correct opgelost kunnen worden. De echte Minesweeper fanaat kent deze configuraties uit het hoofd en speelt ze dan ook zonder nadenken. We baseren ons hier op eigen ondervindingen als ook op [Wes]. Eerst geven we nog even extra informatie met betrekking tot de figuren in dit hoofdstuk. De lijnen geven de grenzen van een cel weer en het label in de cel geeft meer informatie over de inhoud ervan, de symbolen met hun overeenkomstige betekenis zijn te vinden in tabel 9.1.

De reden waarom een cel gelabeld met '-' geen mijn mag bevatten, is omdat die mijn dan kan interfereren met het beoogde patroon, waardoor het aantal buurmijnen van reeds gekende, gelabelde cellen in het patroon verstoord kan worden.

Enkele simpele basispatronen die elke Minesweeper-fan wel herkent zijn ge-

Symbool	Inhoud van de cel
*	Cel bevat een mijn, maar is nog niet geopend
v	Cel kan veilig geopend worden
?	Men kan de inhoud niet voor 100 procent zeker weten
-	Cel bevat geen mijn, maar de inhoud is wel gekend
(leeg)	Ongeopende cel (inhoud kan gekend zijn, maar dit hoeft niet)
1	Cel in kwestie heeft juist 1 buurcel die een mijn bevat
2	Cel in kwestie heeft juist 2 buurcellen die een mijn bevatten
3	Cel in kwestie heeft juist 3 buurcellen die een mijn bevatten

Tabel 9.1: Symbolen gebruikt figuren 9.1, 9.2, 9.3 en 9.4.

geven in figuur 9.1. Deze zijn allen gebaseerd op hetzelfde principe, namelijk dat er nog een aantal ongeopende buurcellen zijn die allen een mijn moeten bevatten. We bespreken de drie gegeven configuraties kort.

- **A: Een cel in een hoek bevat het label '1'**. Vermits er nog maar van een buurgridcel de inhoud niet gekend is, en vermits alle andere buurcellen geen mijn bevatten, moet de ongeopende cel een mijn bevatten. Dit alles om de configuratie consistent te houden. Hierdoor weten we meer informatie in verband met de configuratie en kunnen we weer verder gaan en zo een of meerdere cellen veilig openen.

- **B: Een cel in een hoek bevat het label '2'**. In dit geval zijn er nog twee ongeopende buurcellen en heeft de gridcel in kwestie nog geen buurcellen die een mijn bevatten. Het gevolg is dat de twee ongeopende buurcellen van de cel gelabeld met 2 een mijn moeten bevatten. Hopelijk komen we na het invullen van deze cellen met een mijn tot nieuwe gekende patronen, om zo de configuratie verder te gaan analyseren.

- **C: Een cel aan de kant bevat het label '3'**. Vermits de cel in de kant ligt, zijn er drie cellen ongeopend. Nu is het ook zo dat de cel in kwestie nog drie buurcellen moet hebben die een mijn bevatten. Wederom moeten alle ongeopende buurcellen een mijn bevatten.

Bij elk van de voorgaande configuraties geldt dat wanneer we door middel van een bepaald patroon meerdere cellen veilig kunnen openen, er weer een andere configuratie ontstaat waarvoor we eventueel weer een (ander) patroon in herkennen. Het is de bedoeling dat er genoeg patronen gekend zijn om Minesweeper te kunnen spelen.

Als laatste merken we hier op dat een bepaald patroon zich in verschillende vormen kan voordoen. Neem als voorbeeld figuur 9.2. Hierin zijn twee mogelijke vormen van de zogenoemde 1-2-1 configuratie te vinden (er zijn er nog meerdere bekend, maar die worden hier niet gegeven). Hierbij zijn er drie opeenvolgende cellen gelabeld met respectievelijk 1, 2 en nog eens 1.

- **A:** De normale 1-2-1 configuratie. Zoals de regels van Minesweeper het voorschrijven moet de centrale 2 twee buurcellen hebben die een mijn bevatten. Deze twee cellen moeten gekozen worden uit de drie ongeopende cellen boven de 2-cel, en dit omdat de geopende cellen geen mijn bevatten. Nu mogen de cellen gelabeld met 1 slechts een buurcel hebben die een mijn bevatten. In totaal zijn er drie keuzes, maar door de twee constraints blijft er uiteindelijk maar één echte mogelijkheid over.

    1. Een van de linkse drie lege cellen bevat een mijn (respectievelijk de rechtse drie geeft hetzelfde resultaat om wille van symmetrieredenen). Hierdoor bevatten de twee meest rechtse cellen boven de cel gelabeld met een 2 een mijn. Dit kan niet omdat de cel rechts van de cel gelabeld met twee dan twee buurmijnen heeft, terwijl het gelabeld is met een 1. Dit is in strijd met de Minesweeper regels, dus deze keuze leidt niet tot een juiste configuratie.
    2. De cel precies boven de cel gelabeld met 2 bevat een mijn. Deze configuratie is ook niet juist omdat in dit geval ofwel de cel links, ofwel de cel rechts van de cel die een mijn bevat, ook een mijn moet bevatten (door de cel gelabeld met 2). Deze configuratie is fout om dezelfde reden als hiervoor.
    3. De keuze zoals gegeven in figuur 9.2 leidt tot de enige juiste configuratie. Dit is dan ook de keuze die gemaakt moet worden.

- **B:** Een verdoken 1-2-1 configuratie. Door de mijn die zich onder de cel gelabeld met 3 bevindt, doet zich aan de bovenkant van diezelfde cel de 1-2-1 configuratie voor, want de drie heeft dan reeds een buurcel die een mijn bevat (alsook de twee cellen gelabeld met 2). Vanzelfsprekend kunnen we hiermee op dezelfde manier als bij figuur 9.2A omgaan.

Pattern matching kan ook toegepast worden om te zoeken op kansen. Zo weten we als we het patroon zoals gegeven in figuur 9.3A tegenkomen, er een kans van 1 op 2 is dat er een mijn is in de ongelabelde gridcel bevindt, veel keuze is er in dit geval dus niet. Zodoende, door waarschijnlijkheids patronen

te gaan herkennen, kan men bijvoorbeeld bij de eerder gegeven waarschijnlijkheids techniek ook pattern matching gaan toepassen. De figuren gegeven in de volgende sectie kunnen zodoende als patroon gebruikt worden.

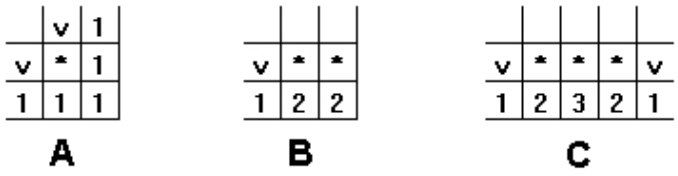
9.2 Waarschijnlijkheid

Zoals de titel al zegt, bekijken we de waarschijnlijkheid dat er zich een mijn bevindt in een bepaalde gridcel. Dit is simpelweg de kans op een mijn. Wanneer men twijfelt tussen het verhullen van twee cellen kan men bijvoorbeeld de kans (op welke manier dan ook) berekenen dat er zich een mijn bevindt op de respectievelijke posities, en dan degene kiezen met de kleinste kans op een mijn. Let op dat de kansberekening op eender welke methode gebaseerd kan zijn: bijvoorbeeld alleen de rechtstreekse buren in rekening brengen, of pattern matching toepassen (zie verder). Het klinkt misschien raar om met een bepaalde waarschijnlijkheid te winnen bij een puzzel/spel. Desondanks is het toch nuttig, omdat we zo 'snel' kunnen spelen en daar bovenop nog eens met een bepaalde waarschijnlijkheid kunnen winnen. Dit is typisch aan heurisieken. Hier baseren we ons enkel op [Wes].

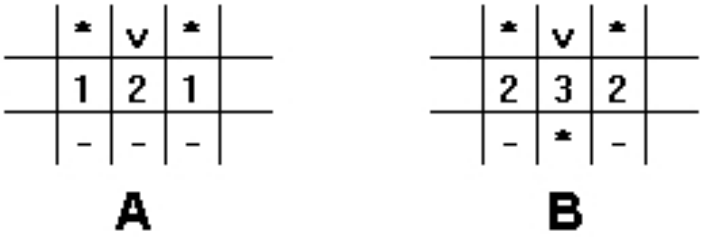
Voorbeeld 9.2.1 (TSP). Vermits het wat bizar klinkt om over heuristieken te denken bij puzzels/spellen, geven we een kleine vergelijking van heuristieken bij het eerder aangehaalde TSP probleem. Hierbij zijn we ook helemaal niet zeker dat onze vertegenwoordiger de korst mogelijke weg aflegt en in de meeste gevallen zal hij dit niet doen, maar veelal kunnen we wel stellen dat hij niet meer dan een bepaald percentage van de best mogelijke weg aflegt. Dit percentage is afhankelijk van de gebruikte heuristiek (en is niet altijd gekend).

Bij Minesweeper is de afwijking, in het geval dat hier besproken wordt, de kans dat er zich een mijn bevindt in de cel die we gaan blootleggen. Eenvoudige voorbeelden worden gegeven in figuur 9.3.

- **A:** Hier ziet men een voorbeeld van een geval waarin we toch de kans weten, maar alsnog de keuze niet uitmaakt, omdat alle kansen gelijk verdeeld zijn. In deze figuur is de kans dat er zich een mijn bevindt onder de cellen die geen label dragen voor elke cel even groot, namelijk 50 percent. Men kan in dit geval niet anders dan een van de mogelijkheden te kiezen. Het enige wat men in dit geval weet is dat als er een mijn onder de cel linksboven ligt (respectievelijk rechtsboven), ligt er ook een onder de cel rechtsbeneden (respectievelijk linksbeneden). Dus de kans op winnen is evengroot als de kans op verliezen, want men heeft 50 percent kans dat men een cel kiest waarin zich een mijn bevindt.



Figuur 9.1: Basispatronen voor Minesweeper.

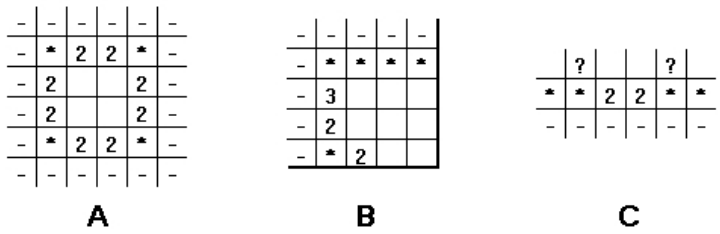


Figuur 9.2: 1-2-1 patronen voor Minesweeper ([Wes]).

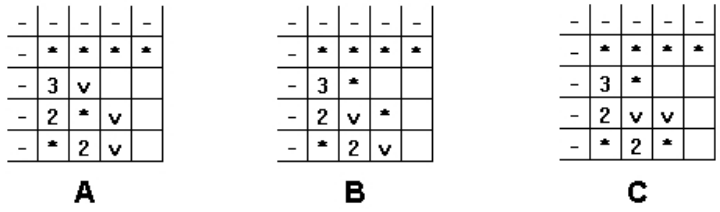
- **B:** Dit is een voorbeeld van een (vrij) complexe situatie waarin kansberekening kan toegepast worden. De drie correct mogelijke configuraties zijn gegeven in figuur 9.4. Van deze drie mogelijkheden is volgens Frank Wester [Wes] de kans op configuratie 9.4A groter dan die op 9.4B of 9.4C.

- **C:** De kans kan ook afhangen van het aantal mijnen dat nog in het spel is, wanneer dit gegeven is. Stel je namelijk voor dat er op honderd vakjes twintig mijnen liggen, dan ligt er gemiddeld een mijn in vijf vakjes. Dit is ook ongeveer de verhouding in de Microsoft Minesweeper expert level (hoogte = 16 $\times$ breedte = 30 met aantal bommen = 99 geeft als resultaat 1 mijn op de 5.8 gridcellen). Stel dat we in figuur 9.3C een gemiddelde kans hebben van een mijn op vijf gridcellen. Dan is het meest waarschijnlijke dat er zich de gridcellen gelabeld met een '?' geen mijn ligt. Dit omdat de kans groter is dat de twee cellen gelabeld met een 2 een mijn delen, wat wil zeggen dezelfde buurcellen hebben die de tweede nodige mijnen bevatten.

Het spreekt voor zich dat ook hele complexe situaties (met eventueel veel buurcellen die bekeken worden) kunnen ontstaan en geanalyseerd worden.



Figuur 9.3: Kanspatronen voor Minesweeper ([Wes]).



Figuur 9.4: Detail kanspatroon voor Minesweeper ([Wes]).

Hoofdstuk 10

Conclusie

In deze thesis gingen we op zoek naar de complexiteit van bepaalde spellen en puzzels, waarbij dieper ingegaan werd op Minesweeper en enkele varianten ervan. Om onze zoektocht te kunnen beginnen, hadden we nood aan een formele beschrijving van het begrip complexiteit. In het eerste hoofdstuk werd dan ook formeel beschreven wat dit begrip inhoud en werd met Turing Machines een formeel model gegeven om te werken met tijd- en ruimtecomplexiteit. Zo kwamen we te weten dat er, naargelang de omvang van een probleem, verschillende complexiteitsklassen zijn, representatief voor de soorten problemen die er zich in bevinden. In elk van deze klasse zijn er op hun beurt speciale problemen, die de moeilijkste zijn van heel de klasse, en die we definiëren als zijnde compleet voor de klasse in kwestie.

Hierna werden een aantal bekende puzzels en spellen aangehaald, en besproken tot welke complexiteits klasse ze behoren. Elk van de besproken puzzels/spellen bleek compleet te zijn voor de klasse waartoe het behoort. Dit ging van Tetris voor NP over Soukoban voor PSPACE tot schaken voor de klasse EXPTIME. Ook werd het spel van NIM besproken voor de klasse P. Zo kwamen vier belangrijke klassen uit de complexiteitstheorie door middel van puzzels en spellen aan bod. In alle gevallen (buiten voor NIM) maakt het compleetheidsbewijs gebruik van constructies, om zo een reeds gekend compleet probleem uit de respectievelijke klassen te simuleren. Telkens werd het idee van de reductie gegeven, tesamen met een of meerdere (deel)constructie uit de reductie.

Verder kwam het SAT en het Circuit SAT probleem aan bod, die nodig waren als basis voor het compleetheidsbewijs voor Minesweeper. Zowel SAT als Circuit SAT hebben de booleaanse logica als basis. Het probleem handelt telkens over het *waar* maken van een booleaanse formule. In de geschiedenis

was SAT het eerste NP-complete probleem en via een polynomiale reductie hieruit heeft men aangetoond dat Circuit SAT ook NP-compleet is. Vanuit dit nieuwe NP-complete probleem toonde R. Kaye [Kay00a] op zijn beurt aan dat de originele Minesweeper versie, die we Minesweeper[4] genoemd hebben, ook NP-compleet is.

In het artikel dat hij schreef [Kay00a], haalt R. Kaye aan dat verschillende varianten van Minesweeper, om dezelfde reden als in zijn artikel besproken, waarschijnlijk ook NP-compleet zouden zijn. Echter wordt dit enkel geïnsinueerd. In deze thesis ging ik op zoek naar de waarheid achter zijn vermoeden door een aantal varianten te bekijken. Zo kwamen Minesweeper versies met respectievelijk regelmatige zes-, drie- en achthoeken als gridcellen aan bod. Ook werd er een uitstap naar 3D-Minesweeper gemaakt. Door zelf de bewijzen op te stellen voor de NP-compleetheid van deze varianten kon geconcludeerd worden dat geen van al deze aanpassingen aan het originele spel invloed bleek te hebben op de complexiteit van Minesweeper. Als conclusie kunnen we dus stellen dat R. Kayes voorgevoel juist was en dat uiteindelijk alle beschouwde gevallen het door R. Kaye verwachtte resultaat op leveren.

Hoofdstuk 11

Woord van dank

Mijn dankwoord gaat uit naar mijn promotor prof. dr. Kuijpers voor zijn tips aangaande de thesisinhoud en -opbouw, alsmede naar de assistente dr. Haesevoets voor haar continue hulpvaardigheid en analyse. Ook wil ik Lode Vanacken danken voor het delen van zijn LaTeX kennis, alsook mijn mede leerlingen voor de voortdurende steun. Als laatste gaat mijn dank uit naar iedereen die een kritische noot aangaande mijn thesis plaatste en die zo mijn inzicht veruimden.

Bibliografie

[BHK03] R. Breukelaar, H.J. Hoogeboom, and W.A. Kosters. Tetris is hard, made easy, 2003. Technical Report, Leiden Institute of Advanced Computer Science Universiteit, the Netherlands. (document), 4.3.2, 4.8

[Chl86] Bogdan S. Chlebus. Domino-tiling games. *J. Comput. Syst. Sci.*, 32(3):374–392, 1986. 4.1, 4.6, 4.8

[CKS81] Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981. 4.1, 4.6

[Cul97] Joseph Culberson. Sokoban is PSPACE-complete, 1997. Technical Report, Dept. of Computing Science, University of Alberta. (document), 4.13, 4.5, 4.16, 4.17, 4.5

[Dem01] Erik D. Demaine. Playing games with algorithms: Algorithmic combinatorial game theory. In *MFCS*, pages 18–32, 2001. 4, 4.2

[DZ99] Dorit Dor and Uri Zwick. Sokoban and other motion planning problems. *Comput. Geom.*, 13(4):215–228, 1999. (document), 4.5, 4.5, 4.14, 4.15, 4.5, 4.5

[FL81] Aviezri S. Fraenkel and David Lichtenstein. Computing a perfect strategy for n x n chess requires time exponential in n. *J. Comb. Theory, Ser. A*, 31(2):199–214, 1981. (document), 4.7, 4.7, 4.18, 4.7, 4.7.1

[HD02] Robert A. Hearn and Erik D. Demaine. PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *CoRR*, cs.CC/0205005, 2002. 4.5

[HK04] Hendrik Jan Hoogeboom and Walter A. Kosters. How to construct tetris configurations. *Int. J. Intell. Games & Simulation*, 3(2):97–105, 2004. (document), 4.3.2, 4.7

[Kay00a] Richard Kaye. Minesweeper is NP-complete. *Mathematical Intelligencer*, 22(2):19–15, 2000. (document), 1, 4, 5.3, 5.2, 5.2, 5.5, 5.6, 5.7, 5.10, 5.11, 5.3.1, 5.12, 5.13, 5.14, 5.15, 6, 6.1, 10

[Kay00b] Richard Kaye. Some minesweeper configurations, 2000. Technical report, The University of Birmingham. (document), 5.4, 5.2, 5.2, 5.16, 5.17, 6.1

[Kui01] Bart Kuijpers. *Logica en Moddeleren*. 2001. 3.1

[McP03] B.P. McPhail. The complexity of puzzles: NP-completeness results for nurikabe and minesweeper, 2003. Reed College, Undergraduate Thesis. 2.1, 3.2.1, 5.2, 5.3.1, 5.3.2, 5.4, 5.4.2, 6.2.1

[NIM] Game of NIM. `http://www.cut-the-knot.com/nim_theory.shtml`. 4.2

[Rob71] R.M. Robinson. Undecidability and non periodicity for tilings in the plane. *Inv. Math.*, 12:177–209, 1971. 4.4.1

[Rob84] J. M. Robson. N by n checkers is exptime complete. *SIAM J. Comput.*, 13(2):252–267, 1984. 4.7

[SC79] Larry J. Stockmeyer and Ashok K. Chandra. Provably difficult combinatorial games. *SIAM Journal on Computing*, 8(2):151–174, 1979. 4.7

[Sip96] Michael Sipser. *Introduction to the Theory of Computation: Preliminary Edition*. 1996. (document), 2.1, 2.1.1, 2.1.2, 2.2.1, 2.2.3, 2.2.4, 2.4, 2.5, 3.1

[TUN] A mind stimulating game: In Memory of Late Dr. C. K. Tung. `http://www.codecphone.com/micard/NIM/index.html`. 4.2

[vEB96] P. van Emde Boas. The convenience of tilings. *Complexity, Logic and Recursion Theory, Lecture Notes in Pure and Applied Mathematics*, 187:331–363, 1996. (document), 4.1, 4.4.1, 4.9, 4.8

[Wan60] Hao Wang. Proving theorems by pattern recognition I. *Commun. ACM*, 3(4):220–234, 1960. 4.4.1

[Wes] Frank Wester. The minesweeper page. `http://www.frankwester.net/winmine.html`. (document), 9, 9.1, 9.2, 9.2, 9.2, 9.3, 9.4