Cursus:
Artificiële Intelligentie

versie 1.6.180.340

Willem M. A. Van Onsem BSc

Katholieke Universiteit Leuven
Academiejaar 2009-2010

Figuur 1: Support CopyLeft: All Wrongs Reserved!

Inhoudsopgave

Notities vooraf

De index op het einde is bedoelt als een woordenlijst, indien de lezer het grote deel van deze termen kan thuisbrengen en duiden, is het waarschijnlijk dat hij zal slagen voor het examen.

Deze cursus is hoofdzakelijk gebaseerd op de presentaties van Prof. D. De Schreye. Verder is deze cursus ook gebaseerd op de presentatie rond "Metaheuristieken" van Prof. P. De Causmaecker.

Indien de LaTeX-broncode hierbij gewenst is, kunt u mailen naar `vanonsem.willem@gmail.com`.

De auteur garandeert de juistheid van deze cursus niet. Hoewel deze cursus met de nodige zorg is samengesteld, is het niet ondenkbaar dat er fouten in staan. Errata/opmerkingen/suggesties kunnen altijd doorgestuurd worden naar `vanonsem.willem@gmail.com`*, deze worden dan in de volgende versie verbeterd.*

Over de auteur *valt eigenlijk niet veel te zeggen, behalve dat hij geen exemplaren signeert.*

Speciale dank gaat naar (in alfabetische volgorde) "Blind Guardian", "The Electric Light Orchestra", "Joy", "Piknik", "Secret Service", "Sektor Gaza", "The Scorpions" en "Zombie Nation" voor de muziek tijdens de 7 nachten waarin deze cursus tot stand kwam.

Kudos gaan verder uit naar de volgende groepen/personen (in alfabetische volgorde):

Derde Bachelor Wiskunde Minor Informatica K.U.Leuven 2010-2011 *Die waarschijnlijk het gedeelte over metaheuristieken wetenschappelijk onverantwoord zullen vinden.*

`DhfCYscOCwsXB` *Voor een glimlach van tijd tot tijd.*

Donald Knuth *Voor een norm hoe cursussen er horen uit te zien, en de ontwikkeling van TeX.*

Ingmar Dasseville *Debugger van deze cursus en pianist die het algemene karma verstoort.*

Jonas Vanthornhout *Invoeren van de term "Überalgemene Modus Ponens" (zie 6.3.8).*

`Narod.Ru` *Voor culturele ontdekkingen.*

Personen die errata indienden *Harm De Weirdt, Ingmar Dasseville, Maarten Dhondt, Pieter-Jan Vuylsteke, Robin De Croon.*

Tweede Bachelor Informatica KULAK 2010-2011 *Die waarschijnlijk zullen proberen om deze cursus te lezen, en daar hopelijk ook in slagen.*

Voorbeschouwing

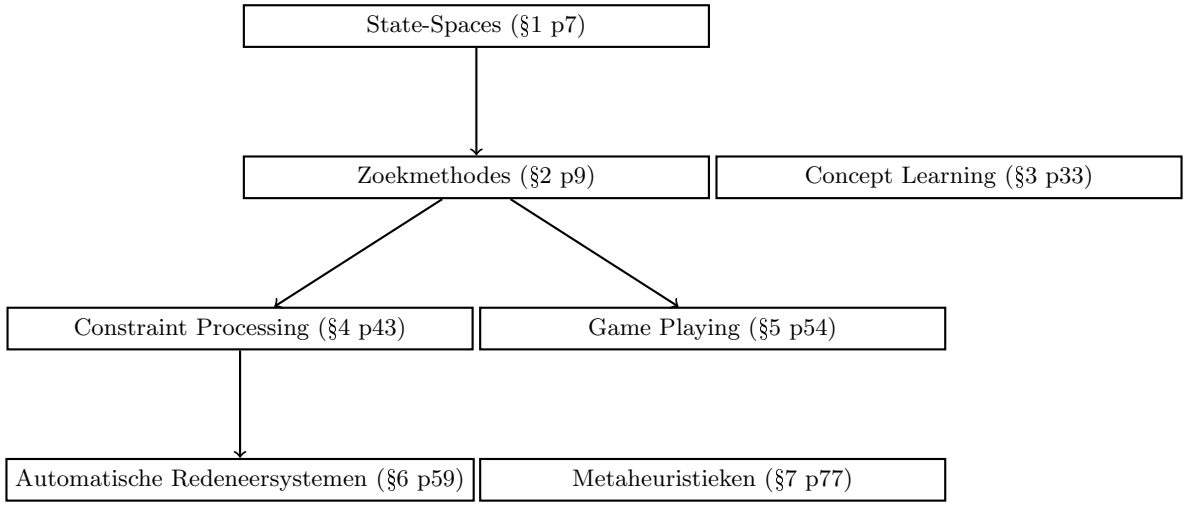
> *De vraag of computers kunnen denken is net als de vraag of duikboten kunnen zwemmen.*
>
> - Edsger Dijkstra, Nederlands informaticus (1930-2002)

Artificiële Intelligentie is een gebied binnen de Informatica die vele banden heeft met talloze wetenschappelijke domeinen (bijvoorbeeld Psychologie). De bedoeling van dit domein is het generen van effecten, die door mensen als intelligent worden beschouwd. Dit wil daarom nog niet zeggen dat de structuren die gebouwd worden dit ook effectief zijn (Wat is überhaupt intelligent? De hersenen zijn uiteindelijk ook slechts een machine). Meestal worden deze effecten gegenereerd door een combinatie van enerzijds brute rekenkracht die het niveau van het brein ver overstijgt. Anderzijds door het toepassen van algoritmen die meestal snel naar een oplossing kunnen convergeren.

Deze cursus behandelt enkele algemene gebieden in de Artificiële Intelligentie. Daarnaast biedt het ook een korte inleiding op verschillende gebieden die later in andere cursussen verder uitgebreid worden (bijvoorbeeld Machine Learning en Genetische Algoritmen). Deze gebieden zijn:

- State-Spaces
- Zoekmethodes
- Concept Learning (Machine Learning)
- Constraint Processing
- Game Playing
- Automatische Redeneersystemen
- Metaheuristieken

Uiteraard staan deze gebieden niet los van elkaar. Afhankelijkheden van deze verschillende gebieden worden weergegeven op figuur 2. Het is aangewezen om eerst de afhankelijkheden van een bepaald deel te studeren alvorens het deel zelf. De cursus is dan ook zo opgebouwd, dat de afhankelijkheden ondersteund worden. Per hoofdstuk zijn er één of meer leidende voorbeelden die gedurende dit hoofdstuk uitgewerkt worden. Alle concepten in het hoofdstuk worden dan op deze voorbeelden toegepast.



Figuur 2: Indeling en afhankelijkheden van deze cursus

Layout en Stijl

> *De eerste eigenschap van stijl is helderheid.*
>
> - Aristoteles, Grieks filosoof (384 v.C. - 322 v.C.)

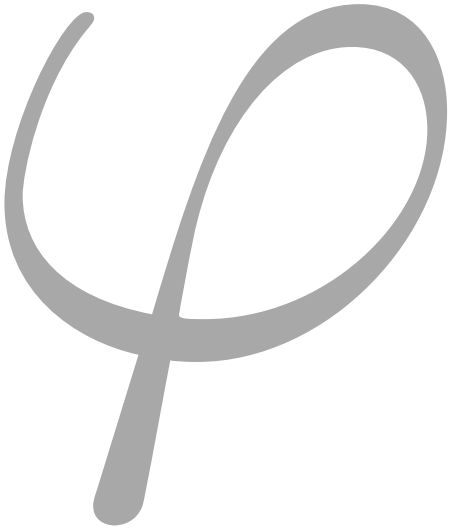
Naast een cursus met een goeie inhoud, is ook de opmaak een belangrijk aspect. Om het leerproces zoveel mogelijk te bevorderen heeft de auteur volgende overwegingen gemaakt wat betreft de layout.

Alle pagina's zijn afdrukbaar met een zwart-wit printer. Dit drukt de eventuele kosten bij een afdruk van deze cursus. Bovendien verhoogt dit de leesbaarheid bij kleurenblindheid. Op het einde van de cursus staan enkele samenvattende schema's waar ook versies in kleur van beschikbaar zijn.

Deze cursus wordt geïllustreerd met talloze afbeeldingen. Deze zijn allemaal tot stand gekomen met het grafisch pakket Ti*k*Z[1]. Alle afbeeldingen zijn bijgevolg vectorieel.

In elk hoofdstuk worden de technieken besproken aan de hand van een leidend voorbeeld. Tekst gerelateerd aan het leidend voorbeeld wordt zoals hier weergegeven met een zwarte rand. Indien de techniek in kwestie duidelijk is, mogen deze stukken overgeslagen worden.

Belangrijke **terminologie** wordt in het vetjes zonder schreven[2] gezet. Deze terminologie komt ook in de index achteraan de cursus. Deze index verwijst dus niet noodzakelijk naar de definities van de termen. Maar naar de pagina waar de term in kwestie voor het eerst voorkomt.



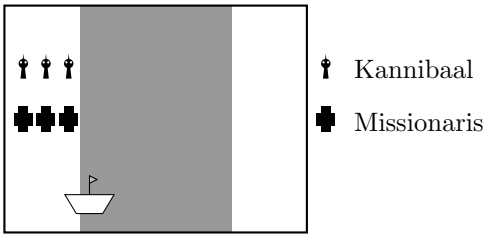
[1] TikZ ist kein Zeichenprogramm
[2] De zogenoemde "pootjes" die letters krijgen. (Engels: serifs)

1 State-Spaces

> *Voor elke toestand, hoe moeilijk ook is er een uitweg. Het komt er alleen op aan een besluit te nemen.*
>
> - Leo Tolstoy, Russisch romanschrijver (1828-1910)

1.1 Leidend Voorbeeld: Missionaries & Cannibals

Als leidend voorbeeld voor dit eerste hoofdstuk opteren we voor het Missionaries & Cannibals probleem. Dit probleem kunnen we als volgt beschrijven: we beschouwen twee oevers, aan één kant staan drie missionarissen en drie kannibalen. Aan de oever ligt ook een boot, deze kan maximum twee personen (kannibalen of missionarissen) tegelijk overzetten. De boot kan alleen de andere oever bereiken indien minstens één persoon zich in de boot bevindt. Het is de bedoeling dat zowel de drie missionarissen als de kannibalen de andere oever bereiken. Indien echter aan een oever meer kannibalen dan missionarissen staan, verliezen deze de macht over de kannibalen, en worden ze opgegeten, en is het spel ten einde. Dit is schematisch weergegeven op figuur 3.



Figuur 3: Begintoestand van het Missionaries & Cannibals probleem

1.2 Wat is een State-space?

een **state-space** is de verzameling van de verschillende staten (toestanden) waarin een probleem zich kan bevinden, waarbij de staat deterministisch en ondubbelzinnig uitgedrukt wordt. Daarnaast bevat een state-space ook een set **productieregels**: condities om van de ene staat van het probleem naar een andere te gaan. Verder bezit een state-space ook een set **begin sta(a)t(en)**, en een set voorwaarden waarbij het **doel** is bereikt (we het probleem dus opgelost hebben).

Hoe zouden we nu het Missionaries & Cannibals probleem kunnen omzetten naar een statespace? Het enige wat we eigenlijk moeten weten is hoeveel missionarissen en cannibalen er aan één van de twee oevers staan, en aan welke oever de boot ligt. Daarom introduceren we een notatie die er als volgt uit ziet: $\langle M, K, B\rangle$ hierbij stellen M het aantal missionarissen, en K het aantal kannibalen voor op de linkeroever. En schrijven we voor B, l indien de boot zich op de linkeroever, en r indien de boot zich op de rechteroever bevindt. De begintoestand noteren we dus als $\langle 3, 3, l\rangle$. Het doel is logischerwijs: $\langle 0, 0, r\rangle$.

Zoals reeds eerder vermeld, is dit probleem gebonden aan enkele voorwaarden. Zo mogen er nooit meer kannibalen dan missionarissen op één van de oevers aanwezig zijn, tenzij er zich geen missionarissen op deze oever bevinden. Dit vertaald zich naar twee regels die we hieronder beschrijven:

$$\langle M, K, B\rangle \text{ is geldig} \Leftrightarrow \begin{cases} M = 0 \vee M \geq K & \{\text{Linker oever}\} \\ \qquad \wedge & \\ M = 3 \vee M \leq K & \{\text{Rechter oever}\} \end{cases} \tag{1}$$

Welke productieregels definiëren we? Indien de boot links staat dienen we minstens één en maximum twee personen over te zetten. Omgekeerd indien de boot aan de rechter oever ligt, dienen we één of twee personen aan de linkerover toe te voegen. Indien we deze regels formaliseren, bekomen we volgende regels:

$$\begin{cases} \langle M, K, l\rangle & \Rightarrow & \langle M-\delta_m, K-\delta_k, r\rangle \mid \delta_m, \delta_k \in \{0,1,2\} \wedge 1 \leq \delta_m + \delta_k \leq 2 \\ \langle M, K, r\rangle & \Rightarrow & \langle M+\delta_m, K+\delta_k, l\rangle \mid \delta_m, \delta_k \in \{0,1,2\} \wedge 1 \leq \delta_m + \delta_k \leq 2 \end{cases} \tag{2}$$

1.3 Wat zijn de keuzes bij een State-space?

Er zijn verschillende keuzes te maken:

- Welke regel passen we eerst toe (proberen we eerst uit)
- Zoeken we in een boom of een graaf
- Zijn we op zoek naar de optimale oplossing, of is een oplossing voldoende
- Kunnen we het probleem opdelen in verschillende deelproblemen
- Zoeken we vooruit of achteruit

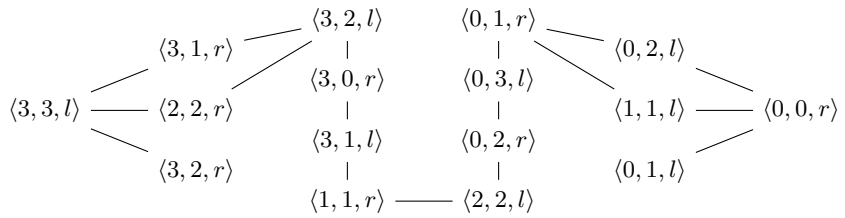
Indien we voor een optimale oplossing kiezen moeten we bijhouden wat de geaccumuleerde kost (zie 2.4) precies inhoud.

Indien we een keuze maken tussen **voorwaarts redeneren** of **achterwaarts redeneren**, moeten we vooral kijken naar de volgende criteria:

- Zijn de regels achterwaarts te construeren: (bij schaken kan je niet achteruit denken)
- **Branching factor**: het gemiddelde aantal staten die direct afgeleid kan worden uit een staat.
- Aantal begintoestanden ten opzichte van het aantal eindtoestanden
- Een achterwaartse redenering kan meestal meer zijn keuzes verklaren

Een andere mogelijkheid is **Middle-out reasoning** waarbij vanuit een bepaalde staat naar het doel en een begin gegraven wordt.

Het is duidelijk dat de het Missionaries & Cannibals probleem een graaf is, immers kunnen we indien we een bepaalde transactie gedaan hebben, telkens deze ongedaan maken door de overgezette personen terug over te zetten. Op zich zijn we niet op zoek naar de kortste oplossing. We kunnen echter wel oplossingen genereren die heel veel redundant werk doen (door bijvoorbeeld een aantal keer op stappen terug te keren). Omdat dit probleem relatief simpel is, is het niet verder op te delen. We kunnen het probleem ook trachtten op te lossen door achterwaarts vanuit de oplossing te redeneren. Maar deze oplossing is eigenlijk volledig identiek. Op figuur 4 wordt de graaf weergegeven die alle toestanden van het probleem weergeeft.



Figuur 4: Graaf van de verschillende toestanden

2 Zoekmethodes

> *Toen ik moe was van zoeken, leerde ik vinden.*
>
> - Friedrich Nietzsche, Duits dichter en filosoof (1844-1900)

Indien we een state-space gebouwd hebben, kunnen we vervolgens een algoritme ontwikkelen die door deze state-space zoekt naar een oplossing. Hiervoor bestaan er verschillende methodes die kunnen gebruikt worden: De **basic search methods**.

Evaluatie-criteria We kunnen een zoekmethode evalueren op volgende criteria:

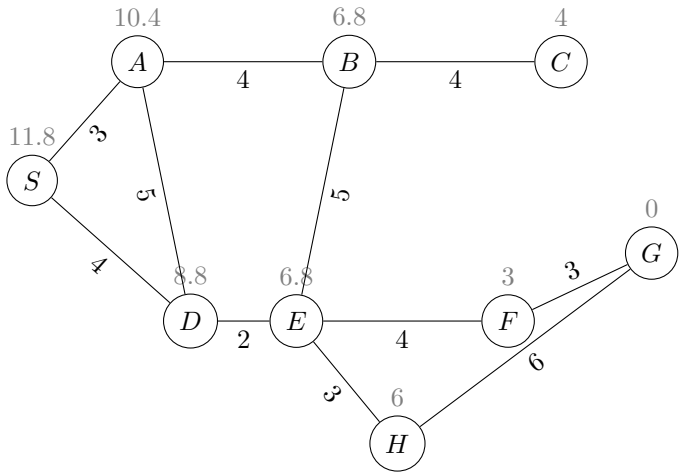
- **Completeness**: vindt het algoritme altijd een oplossing (indien er een oplossing bestaat)
- **Speed**: de worst-case tijdscomplexiteit
- **Memory**: de worst-case geheugencomplexiteit
- **Optimaliteit**: zoeken we de optimale oplossing, of is één oplossing voldoende

Voor complexiteitsanalyses zullen we gebruikmaken van onderstaande parameters:

- d: de diepte van de boom
- b: de vertakkingsfactor
- m: de diepte van de minst diepe oplossing
- δ: de kleinste kost tussen twee staten (zie 2.4)

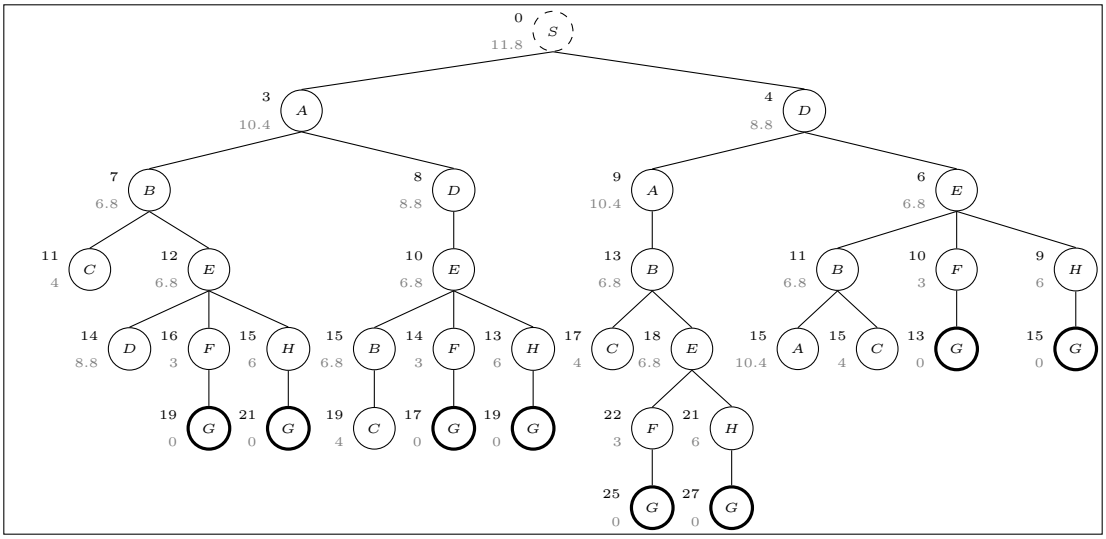
2.1 Leidend Voorbeeld: Wegenplan

Doorheen deze sectie zullen we vaak concepten verduidelijken met behulp van een voorbeeld. Dit voorbeeld wordt hieronder kort toegelicht. Het betreft een sterk vereenvoudigd wegenplan. Hierbij willen we vanuit S (de begintoestand), G (het doel) bereiken. We gaan er vanuit dat er slechts 9 steden bestaan: A (Antwerpen), B (Brussel), C (Charleroi), D (Durbuy), E (Eeklo), F (Fosses-la-Ville), G (Geel), H (Hoei) en S (Sint-Niklaas). Verder bestaan er alleen wegen die tussen twee steden lopen met een bepaalde afstand in kilometer. Dit wordt weergegeven op figuur 5.



Figuur 5: Voorstelling van een door ons beschouwd wegenplan

Als overgangsregels beschouwen we iedere weg naar iedere stad die we tot dan toe nog niet bezocht hebben. Vanuit S kunnen we dus naar twee andere toestanden: A en D. Indien we vervolgens verder de productieregels op deze toestanden toepassen, bekomen we een boom zoals op figuur 6.



Figuur 6: Zoekboom van het wegenplan

Als heuristiek (zie 2.3.1) nemen we tot slot de afstand in vogelvlucht tussen de stad in kwestie, en G zelf. Deze wordt in het grijs op het wegenplan aangegeven. We zullen in deze sectie heel wat zoekalgoritmen uitvoeren op deze boom, om zo de concrete werking te verduidelijken. Deze afbeeldingen staan niet in de tekst zelf, maar staan in 2.7 op pagina 26.

2.2 Blind Search Methods

Indien we geen specifieke kennis over het probleem hebben, of deze te complex/lastig is om te formuleren kunnen we een **Blind Search Method** gebruiken. Het is evident dat zoeken zonder probleemgerelateerde informatie inefficiënt is. Bij deze methodes zijn we ook niet geïnteresseerd in de optimale oplossing.

2.2.1 Depth-First Search

Bij **Depth-First Search** of **Diepte-eerst zoeken** volgen we een bepaalde productieregel vanaf de begintoestand. Vervolgens blijven we deze regel recursief toepassen tot we vast zitten. Indien we een doel gevonden hebben is het algoritme ten einde. Indien het niet het doel is, zal het algoritme de laatste productieregel ongedaan maken en een andere in de plaats stellen. Indien er geen ongebruikte regels meer voor handen zijn maken we de regels daarvoor ongedaan tot we terug een regel kunnen kiezen. Een andere naam hiervoor is **Chronological Backtracking**. Deze zoekmethode wordt geïllustreerd in **Algorithm 1**. We voorkomen dat we in een lus terechtkomen (indien de productieregel dit toelaten) door telkens we een toestand genereren te controleren of één van zijn voorouders identiek aan die toestand is. Indien dit zo is, zullen we die toestand niet meer verder evalueren. Dit systeem noemen we **Loopdetectie**.

Complexiteiten We zullen nu de tijd- en geheugencomplexiteit voor Depth-First analyseren. Hierbij zullen we systemen zoals het loopdetectie systeem buiten beschouwing laten. We gaan er dus vanuit dat we een pad analyseren in $\mathcal{O}(1)$. Dit is uiteraard niet het geval, en zal in andere algoritmes zelfs oplopen tot lange procedures. We zijn dus meer geïnteresseerd in het aantal maal we de while-lus uitvoeren, en beschouwen dit als tijdscomplexiteit.

Algorithm 1 Depth-First zoekalgoritme

```
1: Queue ← Path containing the root
2: while notEmpty (Queue) ∧ ¬goalReached (Queue) do
3:     r ← dequeue (Queue)
4:     R ← createNewPaths (r)
5:     removeLoops (R)
6:     enqueueFront (Queue, R)
7: end while
8: if goalReached (Queue) then
9:     return success
10: else
11:    return failure
12: end if
```

Indien we dit toepassen op Depth-First stellen we vast dat we een tijdscomplexiteit van $\mathcal{O}\left(b^d\right)$ uitkomen. Vrij vertaald betekent dit dus dat we de volledige boom doorlopen. Dit is te beargumenteren: stel dat de oplossing zich in de meest rechtse tak bevindt. In dat geval, zal Depth-First eerst alle andere paden van links naar rechts doorlopen. Het maakt niet zoveel uit op welke diepte de oplossing zich bevindt (indien niet onmiddellijk onder de wortel). Deze eigenschap kan ook de compleetheid van het algoritme ook in gevaar brengen. Indien we met een oneindig diepe boom te maken hebben. Waarbij we telkens verder op een bepaalde knoop kunnen produceren, zal het algoritme nooit stoppen. We moeten dus over productieregels beschikken die vroeg of laat ieder pad doen stoppen, om dit algoritme volledig te doen werken.

Het geheugengebruik van Depth-First is beter. Op ieder niveau is op ieder moment hooguit één knoop geëxpandeert, bijgevolg zitten er in de Queue hooguit $\mathcal{O}\left(b \cdot d\right)$ knopen. Namelijk per diepte b verschillende knopen.

Indien we dit samenvatten bekomen we volgende criteria:

Compleet:	Ja, indien geen oneindig diepe boom
Snelheid:	$\mathcal{O}\left(b^d\right)$
Geheugen:	$\mathcal{O}\left(b \cdot d\right)$
Optimaal:	Nee

We illustreren de werking van dit algoritme op het wegenplan in figuur 9 op pagina 26.

2.2.2 Breadth-First Search

Bij **Breadth-First Search** of de **Breedte-eerst zoeken** zoekmethode wordt een laag aan oplossingen afgewerkt alvorens er een nieuwe laag wordt bekeken. Concreet betekent dit dus dat de kinderen van een geëvalueerde knoop achteraan in een wachtrij geplaatst worden, en dus pas behandeld worden wanneer alle andere knopen van dezelfde boom toegevoegd zijn. Dit wordt geïllustreerd in **Algorithm 2**.

Complexiteiten In tijdscomplexiteit is een breedte eerst benadering in de meeste gevallen beter. Het algoritme zal immers alleen de boom expanderen tot de diepte waar een doel zich bevindt. De boom die we bekomen tot een diepte m telt $\mathcal{O}\left(b^m\right)$ knopen. Bijgevolg hebben we dus ook $\mathcal{O}\left(b^{m-1}\right)$ evaluaties gedaan. Of eenvoudiger $\mathcal{O}\left(b^m\right)$. Een ander voordeel van een Breedte-Eerst benadering is dat ook oneindig diepe bomen geëvalueerd kunnen worden. Bovendien zal het algoritme als oplossing een pad teruggeven met het minimale aantal stappen om tot een oplossing te komen (niet te verwarren met goedkoopste bij optimaal zoeken, het is niet omdat we het minste aantal steden passeren, dat we ook het minste kilometers afleggen).

Algorithm 2 Breadth-First zoekalgoritme

```
1: Queue ← Path containing the root
2: while notEmpty (Queue) ∧ ¬goalReached (Queue) do
3:     r ← dequeue (Queue)
4:     R ← createNewPaths (r)
5:     removeLoops (R)
6:     enqueueBack (Queue, R)
7: end while
8: if goalReached (Queue) then
9:     return success
10: else
11:     return failure
12: end if
```

De ommezijde van deze medaille is dat het geheugen erg onefficiënt gebruikt wordt. Dit moet immers telkens alle paden bijhouden op een diepte m. Dit is uiteraard eveneens gelijk aan $\mathcal{O}(b^m)$. Omdat bij een hoge m of b het geheugengebruik makkelijk grote proporties aanneemt waarbij zelf moderne geheugens soms moeten afhaken is Breedte-Eerst alleen geschikt voor kleine problemen.

Samenvattend bekomen we dus volgende criteria:

Compleet:	Ja, zelfs de kortste
Snelheid:	$\mathcal{O}(b^m)$
Geheugen:	$\mathcal{O}(b^m)$
Optimaal:	Nee, wel het kortst

We illustreren de werking van dit algoritme op het wegenplan in figuur 10 op pagina 27.

2.2.3 Non-Deterministic Search

Bij **Non-Deterministic Search** of de **Niet-deterministisch zoeken** zoekmethode worden de gegenereerde kinderen op toevallige plaatsen in de wachtrij gezet. Het gevolg is dus dat de we toevallig de boom expanderen. Soms kunnen deze methodes tot snelle resultaten leiden, anderzijds is de tijdsduur niet te berekenen. Dit wordt geïllustreerd in **Algorithm 3**.

Algorithm 3 Non-Deterministic zoekalgoritme

```
1: Queue ← Path containing the root
2: while notEmpty (Queue) ∧ ¬goalReached (Queue) do
3:     r ← dequeue (Queue)
4:     R ← createNewPaths (r)
5:     removeLoops (R)
6:     enqueueRandom (Queue, R)
7: end while
8: if goalReached (Queue) then
9:     return success
10: else
11:     return failure
12: end if
```

Complexiteiten Het berekenen van de complexiteiten is behoorlijk eenvoudig: we weten eenvoudigweg niet hoe de knopen geëvalueerd zullen worden. In het slechtste geval wordt telkens een foute knoop gekozen, en moeten we dus eerst de volledige boom evalueren. De tijdscomplexiteit is dus $\mathcal{O}\left(b^d\right)$. Dit algoritme zal echter niet vastlopen op oneindig diepe bomen. Vroeg of laat zal het de juiste combinatie van knopen evalueren om tot een oplossing te komen. Hoe lang dit duurt is echter onmogelijk te voorspellen.

Verder kunnen we ook eerst alle knopen evalueren zodat we de onderste laag van onze boom in het geheugen opslaan. De geheugencomplexiteit is dus ook gelijk aan $\mathcal{O}\left(b^d\right)$.

Indien we dus het loopdetectie-systeem even buiten beschouwing laten in onze tijds- en geheugencomplexiteitsanalyse bekomen we volgende criteria:

Compleet:	Ja
Snelheid:	$\mathcal{O}\left(b^d\right)$
Geheugen:	$\mathcal{O}\left(b^d\right)$
Optimaal:	Nee

Een mogelijk scenario van dit algoritme is weergegeven in figuur 11 op pagina 28.

2.2.4 Iterative Deepening Search

Een in het algemeen interessante zoekmethode is **Iterative Deepening Search** of **Iteratief verdiepend zoeken**. Deze methode maakt gebruik van het **Depth-limited search** algoritme, dit is een algoritme die een implementatie is van Depth First, maar waarbij vanaf een bepaalde diepte gestopt wordt. Iterative Deepening Search maakt van deze methode gebruik door de diepte-limiet parameter op een bepaalde beginwaarde te zetten, en deze telkens te verhogen zolang er niets gevonden wordt. Dit wordt geïllustreerd in **Algorithm 4**.

Algorithm 4 Iterative Deepening zoekalgoritme

```
1:  Queue ← ∅
2:  depth ← 1{Or any other offset depth}
3:  while ¬goalReached (Queue) do
4:      {Start Depth-Limited Search}
5:      Queue ← Path containing the root
6:      while notEmpty (Queue) ∧ ¬goalReached (Queue) do
7:          r ← dequeue (Queue)
8:          if depth (r) < depth then
9:              R ← createNewPaths (r)
10:             removeLoops (R)
11:             enqueueFront (Queue, R)
12:         end if
13:     end while
14:     {End Depth-Limited Search}
15:     depth ← depth + 1
16: end while
```

Complexiteiten De complexiteit van Iterative Deepening berekenen we als volgt. We genereren een iteratie op Depth-First, hierbij zullen we de diepte blijven ophogen tot m (op deze hoogte vinden we dus ons doel). Daarom berekenen we de tijdscomplexiteit als de som van alle Diepte-Eerst pogingen tot m:

$$\sum_{i=1}^{m} b^i = \frac{b^{m+1} - b}{b - 1} = \mathcal{O}\left(b^m\right) \tag{3}$$

Bij het berekenen van de geheugencomplexiteit zullen we eenvoudig het maximum van de geheugencomplexiteit van de verschillende Diepte-Eerst pogingen nemen. Omdat $b \geq 1$. Is dit dus gelijk aan de laatste: de geheugencomplexiteit is bijgevolg $\mathcal{O}\left(b \cdot m\right)$.

Indien we dus het loopdetectie-systeem even buiten beschouwing laten in onze tijds- en geheugencomplexiteitsanalyse bekomen we volgende criteria:

Compleet:	Ja, zelfs de kortste
Snelheid:	$\mathcal{O}(b^m)$
Geheugen:	$\mathcal{O}(b \cdot m)$
Optimaal:	Nee

In veel opzichten is dit het beste blind search algoritme. Het is compleet en vindt zelfs de kortste route, daarnaast is het ook nog het snelste en het meest geheugenefficiënte algoritme.

2.2.5 Bi-Directional Search

Bi-Directional Search of de **bidirectioneel zoeken** zoekmethode is een methode waarbij tegelijk vanuit de bron als uit het doel gerekend wordt, en dit tot een gemeenschappelijke toestand gevonden wordt. Meestal wordt hiervoor breedte-eerst gebruikt, andere implementatie zijn echter ook mogelijk. Dit wordt geïllustreerd in **Algorithm 5**.

Algorithm 5 Bi-Directional zoekalgoritme (met Breedte-Eerst)

```
1:  Queue_1 ← Path containing the root
2:  Queue_2 ← Path containing the goal
3:  while notEmpty(Queue_1) ∧ notEmpty(Queue_2) ∧ ¬shareState(Queue_1, Queue_2) do
4:      r ← dequeue(Queue_1)
5:      R ← createNewPaths(r)
6:      removeLoops(R)
7:      enqueueBack(Queue_1, R)
8:      r ← dequeue(Queue_2)
9:      R ← createNewReversedPaths(r)
10:     removeLoops(R)
11:     enqueueBack(Queue_2, R)
12: end while
13: if shareState(Queue_1, Queue_2) then
14:     return success
15: else
16:     return failure
17: end if
```

Complexiteiten Opnieuw is het berekenen van de complexiteiten gebaseerd op eerder berekende resultaten. We verwachten dat bij een Breedte-eerst benadering we een gemeenschappelijke toestand vinden rond $m/2$. Indien een Breedte-Eerst implementatie eindigt op deze diepte, bekomen we een tijdscomplexiteit $\mathcal{O}\left(b^{m/2}\right)$ en een geheugencomplexiteit $\mathcal{O}\left(b^{m/2}\right)$. Dat we dit algoritme tweemaal toepassen, maakt geen verschil.

Indien we het loopdetectie-systeem even buiten beschouwing laten in onze tijds- en geheugencomplexiteitsanalyse bekomen we volgende criteria:

Compleet:	Ja
Snelheid:	$\mathcal{O}\left(b^{m/2}\right)$
Geheugen:	$\mathcal{O}\left(b^{m/2}\right)$
Optimaal:	Nee

2.3 Heuristic Search Methods

2.3.1 De heuristische functie

Heuristische zoekmethodes of **Heuristic Search Methods** maken gebruik van een **heuristische functie** $h : \mathbb{S} \to \mathbb{R} : s \mapsto r = h(s)$. Deze functie vertaalt een bepaalde toestand s in een getal r, die een schatting is van hoe goed de toestand is. Dit stelt ons in staat om generische zoekalgoritmen te ontwikkelen die toch **probleem-specifieke kennis** in zich dragen. In het algemeen houdt een heuristiek in dat je een

schatting geeft hoeveel productieregels nog op een toestand moeten worden toegepast, alvorens tot het doel te transformeren. Meestal is dit aantal echter niet gekend en blijft het bij een schatting. Maar zelfs vage schattingen kunnen zoekalgoritmen en enorme snelheidswinst geven.

2.3.2 Hill climbing

Vervolgens kunnen we onze zoekmethodes die we eerder gedefinieerd hebben omzetten naar algoritmen die het heuristische aspect bevatten. **Hill Climbing** is de variant van Depth First met heuristische functie. In plaats van echter de productieregels van links naar rechts te volgen zoals bij depth first het geval is, zullen we altijd eerst de gegenereerde kinderen sorteren op heuristiek en zo toevoegen. Zoals formeel gesteld in **Algorithm 6**. Over het algemeen werkt een algoritme gebaseerd op een heuristiek beter.

Algorithm 6 Hill Climbing zoekalgoritme

```
1: Queue ← Path containing the root
2: while notEmpty (Queue) ∧ ¬goalReached (Queue) do
3:     r ← dequeue (Queue)
4:     R ← createNewPaths (r)
5:     removeLoops (R)
6:     quickSort (R, h (R_i))
7:     enqueueFront (Queue, R)
8: end while
9: if goalReached (Queue) then
10:    return success
11: else
12:    return failure
13: end if
```

Indien de heuristiek echter slecht gekozen is kan dit resulteren in dezelfde complexiteiten als voor diepte eerst:

Compleet:	Ja
Snelheid:	$\mathcal{O}\left(b^d\right)$
Geheugen:	$\mathcal{O}\left(b \cdot d\right)$
Optimaal:	Nee

We illustreren de werking van dit algoritme op het wegenplan in figuur 12 op pagina 29.

2.3.3 Beam Search

Beam Search kan gezien worden als de tegenhanger van breedte-eerst. Omdat breedte-eerst slechts weinig ruimte geeft om er een heuristiek aan toe te voegen is het concept licht gewijzigd. In plaats van alle knopen op een bepaald niveau te onderzoeken worden alleen de w beste (laagste h-waarde) onderzocht. Dit resulteert in een zoekalgoritme dat niet altijd een oplossing garandeert. Anderzijds indien de heuristiek goed genoeg gekozen is, zodat een pad naar de oplossing telkens gekozen wordt, kan dit algoritme snel tot een oplossing komen. Dit algoritme is weergegeven in **Algorithm 7** Indien we het loopdetectie-systeem even buiten beschouwing laten in onze tijds- en geheugencomplexiteitsanalyse, en we gaan ervan uit dat het algoritme succesvol is, bekomen we volgende criteria:

Compleet:	Nee
Snelheid:	$\mathcal{O}\left(w \cdot m\right)$
Geheugen:	$\mathcal{O}\left(w\right)$
Optimaal:	Nee

We illustreren de werking van dit algoritme op het wegenplan in figuur 13 op pagina 29.

Algorithm 7 Beam Search zoekalgoritme

```
1: nextdepth ← Path containing the root
2: Queue ← ∅
3: while notEmpty (nextdepth) ∧ ¬goalReached (Queue) do
4:     Queue ← nextdepth
5:     nextdepth ← ∅
6:     while notEmpty (Queue) ∧ ¬goalReached (Queue) do
7:         r ← dequeue (Queue)
8:         R ← createNewPaths (r)
9:         removeLoops (R)
10:        nextdepth ← nextdepth ∪ R
11:        while #nextdepth > w do
12:            {Verwijder pad met de hoogste heuristiek uit nextdepth}
13:            nextdepth ← nextdepth \ {max_h (R)}
14:        end while
15:    end while
16: end while
17: if goalReached (Queue) then
18:     return success
19: else
20:     return failure
21: end if
```

2.3.4 Hill Climbing 2

Hill Climbing 2 is een specifiek geval van Beam Search met $w = 1$. De naam doet nochtans vermoeden dat het een variant van Hill Climbing is. We kunnen Hill Climbing 2 echter ook zien als een vorm van Hill Climbing zonder backtrackingsmechanisme. Dit algoritme is weergegeven in **Algorithm 8** Indien we

Algorithm 8 Hill Climbing 2 zoekalgoritme

```
1: r ← root
2: while ¬goalReached (r) do
3:     R ← createNewPaths (r)
4:     removeLoops (R)
5:     if notEmpty (R) then
6:         r ← min_h (R)
7:     else
8:         return failure
9:     end if
10: end while
11: return success
```

het loopdetectie-systeem even buiten beschouwing laten in onze tijds- en geheugencomplexiteitsanalyse, en we gaan ervan uit dat het algoritme succesvol is, bekomen we volgende criteria:

Compleet:	Nee
Snelheid:	$\mathcal{O}(m)$
Geheugen:	$\mathcal{O}(b)$
Optimaal:	Nee

Hill Climbing 2 introduceert echter ook enkele bekende problemen:

- **Foothills**: Dit zijn lokale minima die echter niet naar het doel zelf leiden. Dergelijke minima werken als een val die Hill Climbing 2 van het doel doet afwijken
- **Plateaus**: Soms kan het gebeuren dat alle kinderen dezelfde heuristische waarde als hun ouder hebben, in dat geval weet Hill Climbing 2 niet wat kiezen en is de kans op een foute keuze groot

- **Ridges**: De kinderen hebben een lagere heuristische waarde maar zijn equivalent aan elkaar. Opnieuw is de keuze waarschijnlijk fout.

Hill Climbing 2 is een vorm van **Local Search**, dit is een zoekstrategie waarbij de queue telkens maar 1 element bevat. Een ander voorbeeld van een Local Search Algoritme is **Minimal Cost Search**. Hierbij nemen we telkens het kind met de kleinste kost.

We illustreren de werking van dit algoritme op het wegenplan in figuur 14 op pagina 30.

2.3.5 Greedy Search

Greedy Search is samen met Hill Climbing de enige complete heuristische oplossing. Greedy Search expandeert telkens de knoop die de kleinste heuristiek heeft, daar deze waarschijnlijk het snelst tot de oplossing komt. Het houdt echter de andere knopen nog steeds in zijn queue. Dit algoritme is weergegeven in **Algorithm 9** Over het algemeen levert greedy search zeer competitieve resultaten. In het slechtste

Algorithm 9 Greedy zoekalgoritme

```
1: Queue ← Path containing the root
2: while notEmpty (Queue) ∧ ¬goalReached (Queue) do
3:     r ← dequeue (Queue)
4:     R ← createNewPaths (r)
5:     removeLoops (R)
6:     enqueue (Queue, R){Doesn't matter where}
7:     quickSort (Queue, h)
8: end while
9: if goalReached (Queue) then
10:    return success
11: else
12:    return failure
13: end if
```

geval klopt de schatting echter niet, en dwingt deze het algoritme de volledige boom te doorzoeken. We bekomen dus volgende criteria:

Compleet:	Ja
Snelheid:	$\mathcal{O}\left(b^d\right)$
Geheugen:	$\mathcal{O}\left(b^d\right)$
Optimaal:	Nee

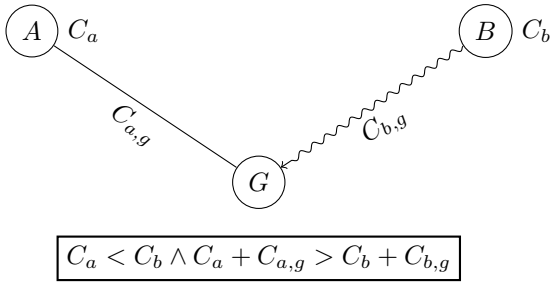
2.4 Optimal Search Methods

Onder **Optimal Search** of **Optimaal zoeken** verstaan we het zoeken naar een oplossing die daarenboven de kleinste **accumulatieve kost** bevat. Dit leidt tot meer complexe algoritmen die meestal ook gebruikmaken van heuristische functies. De kostfunctie definiëren we als $c : \mathbb{P} \to \mathbb{R} : p \mapsto r = c(p)$. Hierbij is de kost van een pad p te berekenen door de som tussen iedere 2 staten te bereken:

$$c(p) = \sum_i c(p_i, p_{i+1}) \qquad \Big| \; c(p_i, p_{i+1}) \geq 0 \tag{4}$$

2.4.1 Uniform Cost Algorithm

Het **Uniform Cost Algorithm**, ook wel **Uniform best-first** algoritme genoemd is een variant van het greedy algoritme waarbij de queue niet gesorteerd wordt op de heuristiek, maar op de geaccumuleerde kost. Dit levert echter niet altijd de optimale oplossing op. Het kan immers zijn dat een toestand die tot dan toe de goedkoopste was een doel bereikt, maar hiervoor een zware prijs betaald. Terwijl een andere toestand in de queue aanvankelijk duurder was, maar naar het doel convergeert zodat de totale kost toch lager is. Dit wordt geïllustreerd in figuur 7, die een algemeen geval beschouwd. We zijn immers meer



Figuur 7: Algemeen geval van falen bij Uniform Cost Search

geneigd om hierbij A te evalueren met een lagere kost. Een pad dat echter vanuit B vertrekt, lijkt op het eerste zicht duurder, maar kost uiteindelijk minder.

We passen bovendien dit algoritme toe op ons wegenplan, en bekomen een niet optimaal pad of figuur 15 op pagina 30, waar de kost 15 is, in plaats van 13.

Dit algoritme geeft wel in de meeste gevallen al een behoorlijk optimale oplossing. Dit algoritme wordt beschreven in **Algorithm 10**. Indien we het loopdetectie-systeem even buiten beschouwing laten

Algorithm 10 Uniform Cost zoekalgoritme

```
1:  Queue ← Path containing the root
2:  while notEmpty (Queue) ∧ ¬goalReached (Queue) do
3:      r ← dequeue (Queue)
4:      R ← createNewPaths (r)
5:      removeLoops (R)
6:      enqueue (Queue, R){De plaats doet er niet toe, de Queue wordt nog gesorteerd}
7:      quickSort (Queue, c)
8:  end while
9:  if goalReached (Queue) then
10:     return success
11: else
12:     return failure
13: end if
```

in onze tijds- en geheugencomplexiteitsanalyse, bekomen we volgende criteria:

Compleet: Ja
Snelheid: $\mathcal{O}\left(b^d\right)$
Geheugen: $\mathcal{O}\left(b^d\right)$
Optimaal: Nee

2.4.2 Optimal Uniform Cost Algorithm

Het **Optimal Uniform Cost Algorithm**, garandeert wel een optimale oplossing, dit wordt gegarandeerd door het **Branch-and-Bound Principe**. Dat principe zegt dat eenmaal we een pad dat naar een doel wijst gevonden hebben, we alle paden met een accumulatieve kost groter dan dat pad niet meer dienen te beschouwen (de kost neemt immers alleen maar toe). Indien onze Queue een pad bevat dat naar een doel wijst, en dit komt als eerste voor, weten we zeker dat dit het optimale pad is. Gebasseerd op deze theorie herschrijven we dus onze terminatieconditie van het Uniform Cost Algorithm. Dit algoritme wordt beschreven in **Algorithm 11**. Indien we het loopdetectie-systeem even buiten beschouwing laten in onze tijds- en geheugencomplexiteitsanalyse, bekomen we volgende criteria:

Compleet: Ja
Snelheid: $\mathcal{O}\left(b^d\right)$
Geheugen: $\mathcal{O}\left(b^d\right)$
Optimaal: Ja

Algorithm 11 Optimal Uniform Cost zoekalgoritme

```
1: Queue ← Path containing the root
2: while notEmpty (Queue) ∧ ¬goalReached (Queue [0]) do
3:     r ← dequeue (Queue)
4:     R ← createNewPaths (r)
5:     removeLoops (R)
6:     enqueue (Queue, R){De plaats doet er niet toe, de Queue wordt nog gesorteerd}
7:     quickSort (Queue, c)
8: end while
9: if goalReached (Queue [0]) then
10:    return success
11: else
12:    return failure
13: end if
```

Een concrete uitwerking van dit algoritme wordt weergegeven op figuur 16 op pagina 31. We merken op dat deze gegarandeerde optimaliteit echter een grote hoeveelheid extra werk kan betekenen (66% extra werk voor een oplossing die slechts 13% beter is).

2.4.3 Extended Uniform Cost

Het Optimal Uniform Cost algoritme werkt goed zolang de constraint $c(p_i, p_{i+1}) > 0$ stand houdt, met andere woorden dat de kost tussen elke boog groter is dan 0. Het nadeel van dit zoekalgoritme is echter dat er opnieuw weinig probleem gerelateerde informatie aan te pas komt. Een oplossing bestaat erin een nieuwe functie te definiëren: de **Extended Uniform Cost** $f : \mathbb{P} \to \mathbb{R} : p \mapsto r = f(p)$:

$$f(p) = c(p) + h(p_{\text{end state}}) \tag{5}$$

Dit is dus de som van de reeds gemaakte kosten c, en een schatting h van de nog te maken kosten tot de oplossing. Samen vormt dit dus een schatting van de kosten die we zullen betalen voor een pad naar het doel, met als subpad het huidige pad. Met deze functie kunnen we algoritmen ontwikkelen die gerichter op zoek gaan naar een oplossing, maar waarbij de optimaliteit behouden blijft.

2.4.4 Estimate-extended Uniform Cost Algorithm

Een algoritme die gebruik maakt van de Extended Uniform Cost is het **Estimate-extended Uniform Cost Algorithm**. Dit is een variant op het Optimal Uniform Cost Algorithm, maar waarbij de knopen op de Extended Uniform Cost f gesorteerd worden in plaats van de Uniform Cost c. Dit algoritme wordt beschreven in **Algorithm 12**. Dit algoritme komt tot een optimale oplossing wanneer we echter nog

Algorithm 12 Estimate-extended Uniform Cost zoekalgoritme

```
1: Queue ← Path containing the root
2: while notEmpty (Queue) ∧ ¬goalReached (Queue [0]) do
3:     r ← dequeue (Queue)
4:     R ← createNewPaths (r)
5:     removeLoops (R)
6:     enqueue (Queue, R){De plaats doet er niet toe, de Queue wordt nog gesorteerd}
7:     quickSort (Queue, f)
8: end while
9: if goalReached (Queue [0]) then
10:    return success
11: else
12:    return failure
13: end if
```

een extra constraint plaatsen op de heuristiek. Als we het terminatiecriterium bekijken moet de meest

optimale oplossing op het einde van het algoritme vooraan staan. Eventueel kunnen paden, die op dat moment echter een ongunstige heuristiek hebben, toch nog optimalere oplossingen opleveren. Het zou kunnen gebeuren dat optimale paden uit de boot vallen, indien op dat moment de heuristiek de nog te maken kosten overschat. De oplossing is het stellen dat de heuristiek altijd een **Onderschatting** moet zijn van de nog te nemen kost tot een doel. Met andere woorden:

$$h(T) \leq c\left(T_{\text{end point}} \cdots \text{goal}\right) \tag{6}$$

Over het algemeen presteren deze algoritmen goed indien een goede heuristische functie gekozen wordt. Het berekenen van de functie impliceert echter ook overhead, bijgevolg is de beste heuristiek niet altijd geschikt om te zoeken. Indien we het loopdetectie-systeem even buiten beschouwing laten in onze tijds- en geheugencomplexiteitsanalyse, bekomen we volgende criteria:

Compleet: Ja
Snelheid: $\mathcal{O}\left(b^d\right)$
Geheugen: $\mathcal{O}\left(b^d\right)$
Optimaal: Ja (mits onderschatting)

Een uitgewerkt voorbeeld van Estimate-Extended Uniform Cost Search op het wegenplan wordt beschreven op figuur 17 op pagina 32.

2.4.5 Path Deletion

Een andere uitbreiding die we kunnen toepassen en die algemeen toepasbaar is, is **Path Deletion**. Bij het genereren van de boom genereren we vaak dezelfde toestand in verschillende takken van de boom (volgens het principe "vele wegen leiden naar Rome"). We kunnen echter door de evaluatie van één van deze staten tot een oplossing komen. Evaluatie van de andere versies leidt alleen maar tot overhead, en enkel de versie van de staat met de tot dan toe laagste cummulatieve kost komt in aanmerking voor een optimale oplossing. Path Deletion verwijdert de paden uit de queue die overhead veroorzaken. Indien er al een pad bestaat met dezelfde eindstaat, en de kost van dat pad is lager, is het niet nodig om de paden met eenzelfde eindpunt maar een hogere kost in de wachtrij te houden. Deze zullen immers altijd een hogere kost behouden. Of meer formeel in **Algorithm 13**.

Algorithm 13 Path Deletion Principe

1: **if** $P \in \text{Queue} \wedge P = (\text{root}, \ldots, I) \wedge Q \in \text{Queue} \wedge Q = (\text{root}, \ldots, I, \ldots) \wedge c(P) \geq c(Q)$ **then**
2: **delete** (Queue, P)
3: **end if**

2.4.6 A* Search

Het populairste zoekalgoritme bij uitstek is **A* Search**, die eigenlijk een combinatie is van het beste uit verschillende zoekalgoritmen. A* is een uitbreiding op het Extended Uniform Cost Search Algoritme waarbij men het Branch-and-Bound principe, Heuristische Underestimate en de Redudant Path Deletion uitbreidingen toevoegt. Dit algoritme wordt beschreven in **Algorithm 14**. Een uitgewerkt voorbeeld staat op figuur 18 op pagina 32. Vanwege de populariteit van A* zullen we wat dieper ingaan op enkele eigenschappen van A*.

Bewijs van Optimaliteit Alvorens we A* kunnen gebruiken, moeten we eerst een bewijs kunnen formuleren die borg staat dat A* altijd tot een optimale oplossing zal komen. Hiervoor maken we gebruik van enkele eigenschappen die we ook formeel aantonen. Hierbij stelt B het meest optimale pad van de bron naar het doel voor.

theorema 1. *Voor ieder subpad p van B is de uitgebreide kost van p kleiner of gelijk aan de kost van B. $\forall p$ subpath of $B : f(p) \leq c(B)$*

Algorithm 14 A* zoekalgoritme

```
1: Queue ← Path containing the root
2: while notEmpty (Queue) ∧ ¬goalReached (Queue [0]) do
3:     r ← dequeue (Queue)
4:     R ← createNewPaths (r)
5:     removeLoops (R)
6:     enqueue (Queue, R){De plaats doet er niet toe, de Queue wordt nog gesorteerd}
7:     quickSort (Queue, f)
8:     {Path Deletion}
9:     for all P, Q ∈ Queue do
10:        if P = (root, ..., I) ∧ Q = (root, ..., I, ...) ∧ c (P) ≥ c (Q) then
11:            delete (Queue, P)
12:        end if
13:    end for
14: end while
15: if goalReached (Queue [0]) then
16:    return success
17: else
18:    return failure
19: end if
```

Bewijs.

$$f(p) = c(p) + h\left(p_{\text{end point}}\right) \wedge h\left(p_{\text{end point}}\right) \leq c\left(p_{\text{end point}} \dots \text{goal}\right)$$
$$\Downarrow$$
$$f(p) \leq c(p) + c\left(p_{\text{end point}} \dots \text{goal}\right) = c(p, \dots, \text{goal}) = c(B)$$

□

theorema 2. *Tijdens iedere while test van A* bevat de Queue een subpad van B*

Bewijs.

bewijs door middel van inductie :

Initieel: S = pad dat de bron bevat $\in$ Queue $\wedge$ S is een subpad van B
$\Downarrow$
subpad van $B \in$ Queue

Iteratie: indien een subpad van B verwijdert wordt

- het subpad heeft ten minste een kind (anders bestaat er geen B)
- bovendien bevat dat kind geen lus (B bevat geen lus)

$\Rightarrow$ Dit kind wordt weer toegevoegd aan de Queue
$\Rightarrow$ een subpad van $B \in$ Queue

□

corollary 3. *Voor ieder pad p zodat $f(p) > c(B)$, kan p nooit geselecteerd worden (Het zal nooit als eerste in de Queue zitten).*

Bewijs.

altijd een subpad q van B met $f(q) \leq c(B)$ $\wedge$ $f(q) \leq c(B) < f(p)$
$\Downarrow$
p wordt nooit geselecteerd

□

Bewijs van terminatie Stel d is een natuurlijk getal $> c(B)/\delta$ (hierbij is δ de kleinste kost van alle bogen), en stel P een pad van lengte d. In dat geval geldt:

$$f(P) = c(P) + h(P) \geq c(P) \geq d \cdot \delta > c(B) = f(B) \tag{7}$$

We weten dus dat het pad P nooit geselecteerd zal worden (B is immers een pad dat het doel bereikt, en heeft een lagere f-waarde). Omdat we uitgaan van een eindige vertakkingsfactor b, zijn er dus maar een eindig aantal knopen die in de Queue geplaatst kunnen worden. Omdat telkens wanneer we de lus uitvoeren er één element uit de Queue gehaald wordt, zal de Queue op een gegeven moment leeg zijn, waardoor de lus doorbroken wordt en het algoritme stopt.

Bewijs van compleetheid Door theorema 2 kunnen we bewijzen dat het pad dat uiteindelijk zal evolueren naar een doel altijd in de Queue zit, bijgevolg zal A* altijd het doel vinden.

Selectiegedrag van A* Welke knopen zullen geëxpandeert worden? We kunnen de impliciete boom onderverdelen in drie verschillende groepen:

- **Nooit geselecteerde knopen**: Hierbij is $f(p) > c(B)$ of een redundant pad.
- **Misschien geselecteerd knopen**: Hierbij is $f(p) = c(B)$
- **Geselecteerde knopen**: Hierbij is $p \in \text{Queue} \wedge p$ is niet redundant $\wedge f(p) < c(B)$

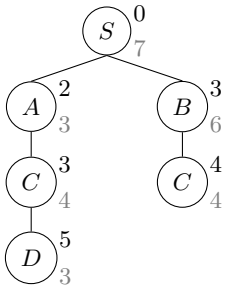
Heuristische kracht **Heuristical Power** gaat over in welke mate de heuristische functie belangrijk is voor het A* algoritme, en of we twee heuristische functies op de één of andere manier kunnen vergelijken met elkaar. Door de constraint dat iedere heuristische functie alleen strikt positieve getallen moet teruggeven, en daarenboven een onderschatting moet zijn van de nog te nemen kost, kunnen we aantonen dat de heuristische functie het A* algoritme in ieder geval niet slechter kan doen presteren dan bijvoorbeeld het Optimal Uniform Cost algoritme. Indien we echter naar een vergelijking tussen twee heuristische functies streven, is het zo dat een functie die bij iedere toestand een grotere waarde retourneert dan zijn tegenhanger per definitie beter is. Met andere woorden:

$$h_1 \text{ is beter dan } h_2 \leftrightarrow \forall s \in \mathbb{S} : h_1(s) \geq h_2(s) \tag{8}$$

Dit kunnen we beargumenteren omdat een heuristische functie per definitie een onderschatting moet zijn voor de werkelijke nog te betalen kost, en dus de heuristiek waarschijnlijk realistischer zal zijn. Verder kunnen we ook nog formeel aantonen dat indien h_1 beter is dan h_2 en er bestaat een pad van de bron naar het doel, dan zal A* met h_2 alle knopen evalueren die A* onder h_1 geëvalueerd zou hebben, met andere woorden A* zal zo goed als altijd langer werken met een slechtere heuristiek dan met een betere, volgens onze definitie van beter.

Minimale kost paden en monotoniciteit Als we **Algorithm 13** dichter bekijken zien we een asymmetrie in het algoritme, dit resulteert in het feit dat niet elke redundante knoop ontdekt wordt. Indien we bijvoorbeeld een pad beschouwen (S, A, C, D) dat bij C een geaccumuleerde kost heeft van 5, en we expanderen een ander pad zodat er een kind-pad ontstaat (S, B, C) met een geaccumuleerde kost van 4 zal het eerste pad niet verwijdert worden (zie figuur 8, hierbij gebeurt evaluatie als volgt: S, A, C, D en B). Het is immers al het verst uitgebreid, en de kost is niet lager. Deze situatie kan voorkomen indien de heuristiek van B bijvoorbeeld heel hoog was, terwijl deze van A en C veel kleiner waren. We kunnen het Path Deletion algoritme in principe niet al te veel aanpassen zodat het hier ook rekening mee houdt. Dit komt omdat deze knoop al geëxpandeert is en we dus dubbel werk willen vermijden. Anderzijds kunnen we een constraint opleggen op onze heuristiek zodat deze nooit tot dergelijk situaties kan leiden. Deze constraint heet **Monotoniciteit** of **Monotonicity**. Deze stelt dat indien we een staat A hebben, en B is hiervan een kind, dan moet de heuristische waarde van A kleiner of gelijk zijn aan de heuristische waarde van B en de kost van A naar B, of meer formeel:

$$B \text{ is een kind van } A \rightarrow h(A) \leq h(B) + c(A, B) \tag{9}$$



Figuur 8: Concreet geval van een niet monotone heuristiek

Opnieuw is deze maatregel redelijk: indien immers niet aan de monotoniciteitsregel wordt voldaan betekent dit dat de heuristiek van B een nog grotere onderschatting van het pad is dan de heuristiek van A. We weten na de heuristiek van A te hebben geëvalueerd reeds dat we meer kosten zullen hebben dan die heuristiek. Indien B een grotere onderschatting is, weten we eigenlijk helemaal niets nieuws. De heuristiek van B zou in dat geval informatieloos zijn. A* zal indien we de monotoniciteitsregel volgen altijd een knoop selecteren met het meest efficiëntste pad tot de knoop. Verder kunnen we ook het Redudant Path algoritme optimaliseren onder deze voorwaarde, zoals in **Algorithm 15**. Hierdoor wordt

Algorithm 15 Geoptimaliseerd Path Deletion Principe

```
1: ℌ ← ∅
2: if A* selecteerd een pad p then
3:     s ← p_end state
4:     if hashtableContains (ℌ, s) then
5:         do not evaluate p
6:     else
7:         insertHashtable (ℌ, s)
8:     end if
9: end if
```

path deletion vrijwel altijd gedaan in $\mathcal{O}(1)$.

Pathmax **Pathmax** is een variant van A* die deze monotoniciteitsregel afdwingt. Dit kan op verschillende manieren gedaan worden:

- Corrigeren van de waarde van f: Indien B een kind is van A en $f(S, \ldots, A, B) < f(S, \ldots, A,)$ dan nemen we voor $f(S, \ldots, A, B)$ dezelfde waarde als $f(S, \ldots, A)$.
- f herdefiniëren: we genereren zelf een functie f' die we definiëren als

$$f'(S, \ldots, A, B) = \max(c(S, \ldots, A, B) + h(B), f'(S, \ldots, A)) \tag{10}$$

2.4.7 Iterative Deepening A* (IDA*)

Een variant van A* is **Iterative Deepening A*** of afgekort **IDA***. Deze heeft tot doel het geheugengebruik van A* meer onder controle te houden (we kunnen A* immers in de verte vergelijken met breedte-eerst, en dit algoritme is niet zuinig met geheugen). Net als Iterative Deepening een beperking op de diepte stelt voor Depth-First, stelt IDA* een beperking op de f waarde. Indien we verwachten dat de oplossing bijvoorbeeld een f-waarde zal hebben van 100 zal het algoritme de kinderen van geëvalueerde knopen niet meer opslaan indien deze een f-waarde groter hebben dan 100. Uiteraard kan het gebeuren dat onze schatting niet toereikend was om tot een oplossing te komen, in dat geval wordt de f-waarde verhoogt en voeren f-**limited Search** opnieuw uit. Meestal wordt als begin f-waarde de f-waarde van de bron $f(S)$ genomen, al dan niet vermenigvuldigt met een factor. Verder wordt de f-limiet meestal verhoogt naar de minimale f die groter was dan de f-limiet. Dit wordt beschreven in **Algorithm 16**. Indien we het loopdetectie- en path deletion-systeem even buiten beschouwing laten in onze tijds- en geheugencomplexiteitsanalyse, bekomen we volgende criteria:

Algorithm 16 Iterative Deepening A* zoekalgoritme

```
1: f-bound ← f(S){Or another f-bound-offset value}
2: Queue ← ∅
3: while ¬goalReached(Queue) do
4:     {Begin f-limited Search}
5:     Queue ← Path containing the root
6:     f-new ← ∞
7:     while notEmpty(Queue) do
8:         r ← dequeue(Queue)
9:         R ← createNewPaths(r)
10:        removeLoops(R)
11:        f-new ← min(f-new, f(min_f(select(R, (R_i) > f-bound))))
12:        removeIf(R, f(R_i) > f-bound)
13:        {Because we will evaluate them all the nodes lower than}
14:        {a certain f-value we don't have to sort them in any order}
15:        enqueueFront(Queue, R)
16:        {Path Deletion}
17:        if P, Q ∈ Queue ∧ P = (root, ..., I) ∧ Q = (root, ..., I, ...) ∧ c(P) ≥ c(Q) then
18:            delete(Queue, P)
19:        end if
20:    end while
21:    {Einde f-limited Search}
22:    f-bound ← f-new
23: end while
```

Compleet: Ja
Snelheid: $\mathcal{O}\left(N^2\right)$
Geheugen: $\mathcal{O}\left(b \cdot c\left(B\right)/\delta\right)$
Optimaal: Ja

2.4.8 Simplified Memory-bounded A* (SMA*)

Een gelijkaardig effect probeert **Simplified Memory-bounded A*** (of kortweg **SMA***) te bereiken. Hierbij is het geheugen echter begrensd, en moeten we proberen om in deze beperkende omstandigheden toch een optimale oplossing te vinden. Dit behoorlijke complexe algoritme heeft als basisidee dat indien het geheugen vol is, en we moeten een knoop evalueren, de knopen met de hoogste f-waarde tijdelijk uit het geheugen worden gelaten. Om echter geen overbodig werkt te doen in het herberekenen van alle knopen veranderen we de f-waarde van de ouder. De ouder van een te verwijderen knoop houdt telkens de beste (laagste) f-waarde van van zijn kinderen bij. Deze f-waardes spelen vervolgens nog een rol om de knopen opnieuw te evalueren, en te zoeken naar een eventueel doel.

Een belangrijk aspect hierbij is dat kinderen één voor één gegenereerd worden om geen **Memory Overflow** te bekomen. Indien het geheugen vol is, moet een knoop wijken voor de te genereren knoop. Verder kan dit algoritme uiteraard geen doelen vinden waarbij het pad naar het doel meer geheugen vereist dan er beschikbaar is. Het algoritme is echter wel compleet wanneer er voldoende geheugen voorradig is.

2.5 Besluitende commentaren

2.5.1 Niet Optimale Varianten

Tot dusver zochten we altijd naar optimale oplossingen, soms is het echter beter om niet de meest optimale oplossing te zoeken indien dit tijdswinst kan opleveren. Indien we gebruik maken van **Non-optimal variants**, is dit meestal door de heuristiek aan te passen. Deze zullen meestal het onderschattings-aspect achterwege laten, en worden meestal **Non-admissible heuristics** genoemd. Een andere mogelijkheid is het

toepassen van een ander zoekalgoritme. Een variant van A*, het **Non-admissible A* Search Algorithm** is hier een voorbeeld van. Hierbij wordt f als volgt gedefinieerd:

$$f(p) = \omega \cdot c(p) + h(p_{\text{end state}}) \quad | \, \omega \in [0,1] \tag{11}$$

Met ω als een parameter. Indien $\omega = 1$ verkrijgen we opnieuw het A* algoritme, indien we $\omega = 0$ nemen bekomen we opnieuw Greedy Search.

2.5.2 Complexiteit in het algemeen

Bij nature is het vinden van een optimaal pad dat naar een oplossing voor een dergelijke probleem NP-compleet. Er bestaan echter wel parallelle[3] algoritmen die in polynomiale tijd kunnen werken.

Meestal leidt dit tot een trade-off waarbij we de optimaliteit reduceren tot een behoorlijke optimale oplossing en hierbij een grote hoeveelheid tijd besparen naar het zoeken van de oplossing. Een voorbeeld hierbij is het instellen van ω bij Non-admissible A*: hoe meer ω naar 1 gaat, hoe meer we in de buurt van een optimale oplossing komen.

2.6 Samenvatting

Name	Compleet	Optimaal	Snelheid	Geheugen	Alg.(p.)
Blind Search					
- Depth First	Ja	Nee	$\mathcal{O}(b^d)$	$\mathcal{O}(b \cdot d)$	1(11)
- Breadth First	Ja	Nee	$\mathcal{O}(b^m)$	$\mathcal{O}(b^m)$	2(12)
- Neen-Deterministic	Ja	Nee	$\mathcal{O}(b^d)$	$\mathcal{O}(b^d)$	3(12)
- Iterative Deepening	Ja	Nee	$\mathcal{O}(b^m)$	$\mathcal{O}(b \cdot m)$	4(13)
- Bi-Direction Search	Ja	Nee	$\mathcal{O}(b^{m/2})$	$\mathcal{O}(b^{m/2})$	5(14)
Heuristic Search					
- Hill Climbing	Ja	Nee	$\mathcal{O}(b^d)$	$\mathcal{O}(b \cdot d)$	6(15)
- Beam Search	Nee	Nee	$\mathcal{O}(w \cdot m)$	$\mathcal{O}(w)$	7(16)
- Hill Climbing 2	Nee	Nee	$\mathcal{O}(m)$	$\mathcal{O}(b)$	8(16)
- Greedy Search	Ja	Nee	$\mathcal{O}(b^d)$	$\mathcal{O}(b^d)$	9(17)
Optimal Search					
- Uniform Cost	Ja	Nee	$\mathcal{O}(b^d)$	$\mathcal{O}(b^d)$	10(18)
- Optimal Uniform Cost	Ja	Ja	$\mathcal{O}(b^d)$	$\mathcal{O}(b^d)$	11(19)
- Estimate-extended Uniform Cost	Ja	Ja	$\mathcal{O}(b^d)$	$\mathcal{O}(b^d)$	12(19)
- A*	Ja	Ja	$\mathcal{O}(b^d)$	$\mathcal{O}(b^d)$	14(21)
- Iterative Deepening A*	Ja	Ja	$\mathcal{O}(N^2)$	$\mathcal{O}(b \cdot c(B)/\delta)$	16(24)
- Simplified Memory-bounded A*	Ja	Ja	$\mathcal{O}(b^d)$	$\mathcal{O}(1)$	

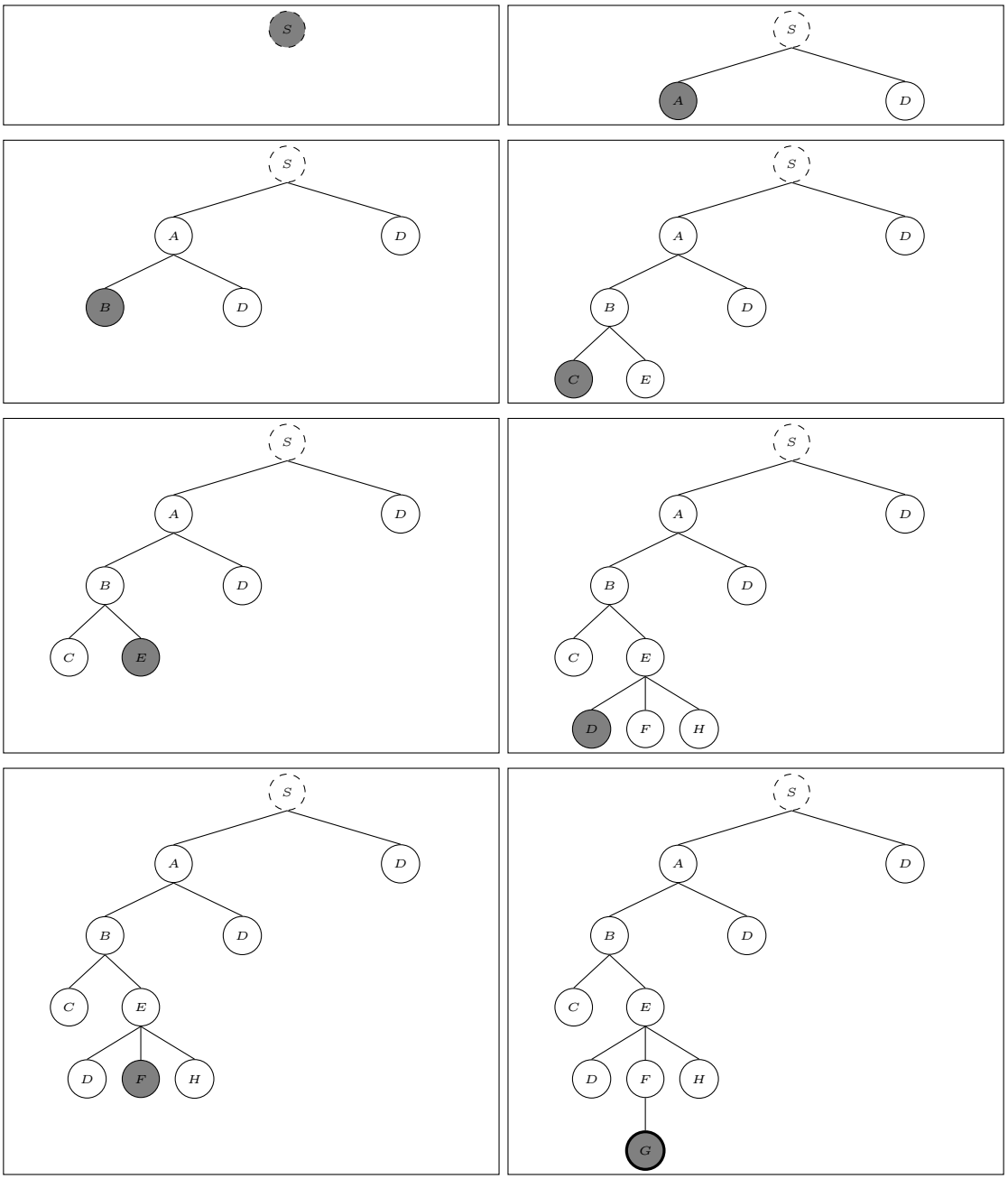
Tabel 1: Samenvatting van de zoekmethodes

Een samenvattend schema is te vinden in appendix A.1 op pagina 86.

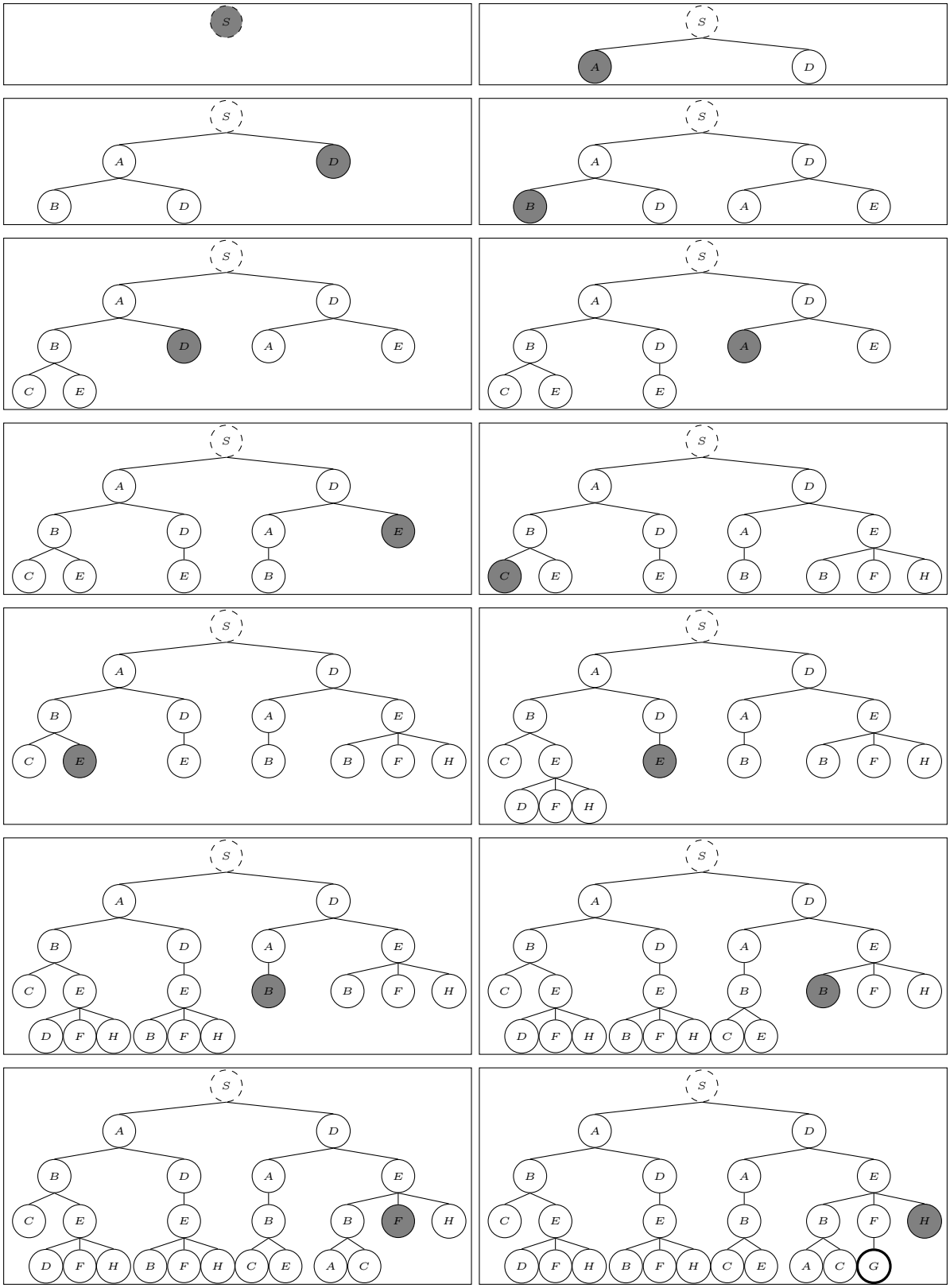
[3] Algoritmen die we kunnen uitvoeren op bijvoorbeeld vector-, DNA- en kwantumcomputers met massaparallellisme.

2.7 Toepassingen van de zoekalgoritmen (Afbeeldingen)

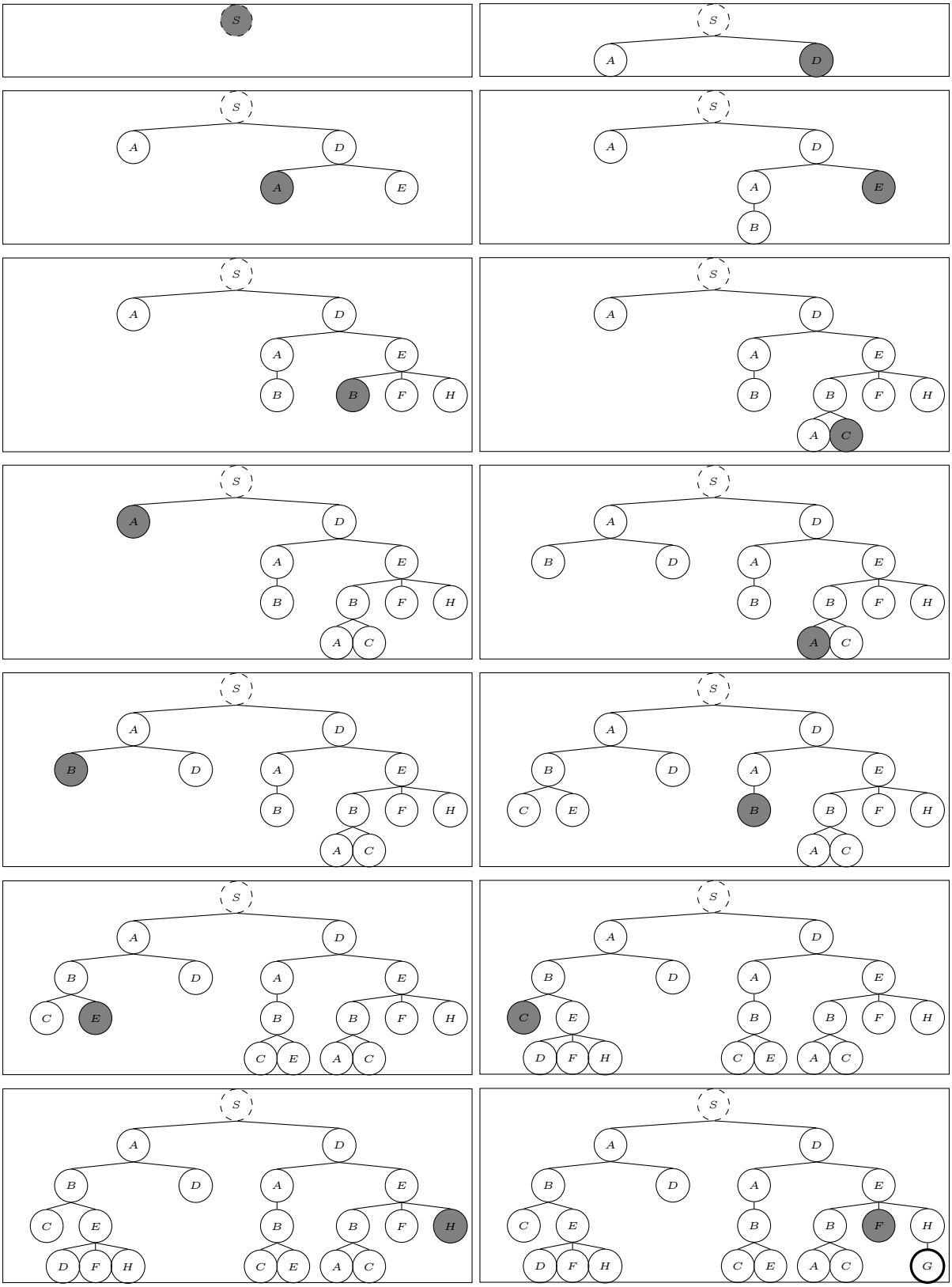
2.7.1 Blind Search Methods



Figuur 9: Depth-First toegepast op het wegenplan

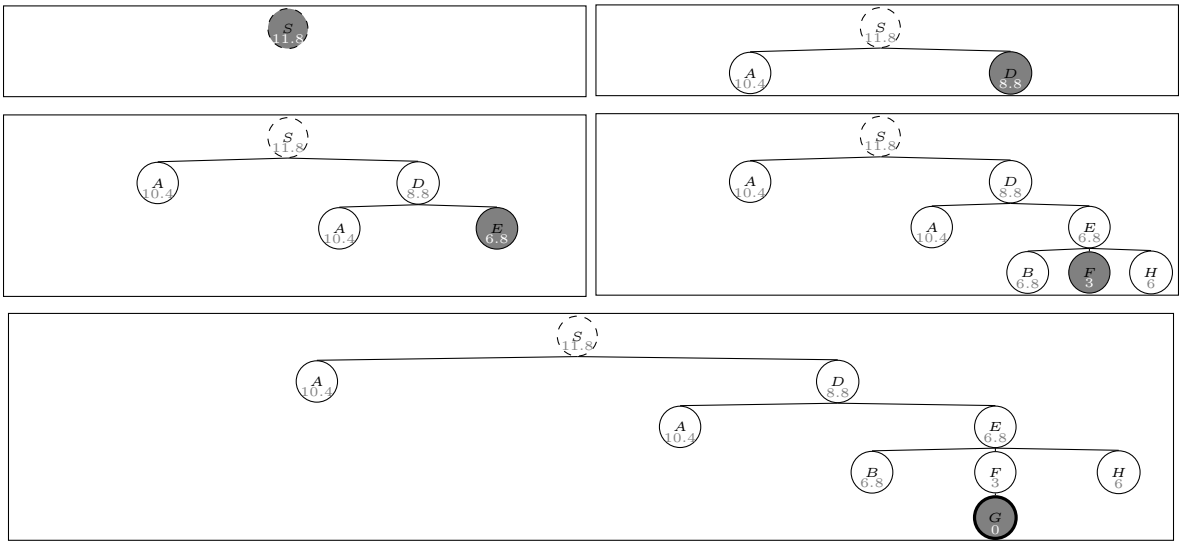


Figuur 10: Breadth-First toegepast op het wegenplan

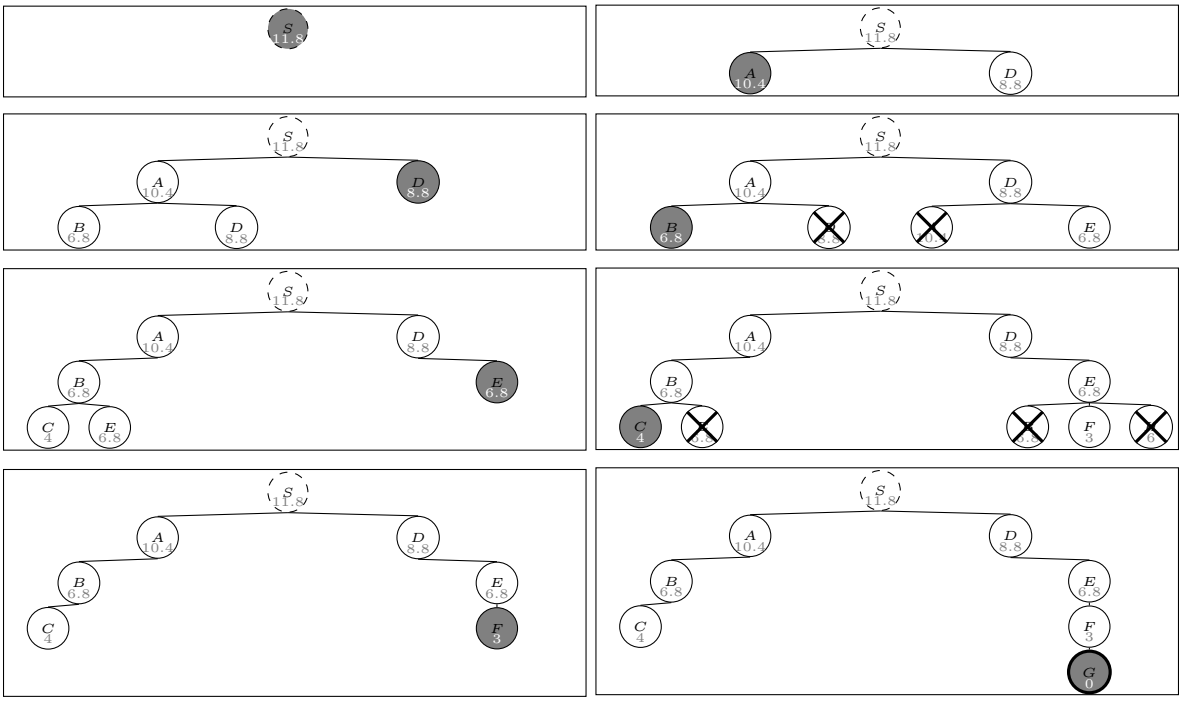


Figuur 11: Een mogelijk scenario van Non-Deterministic Search toegepast op het wegenplan

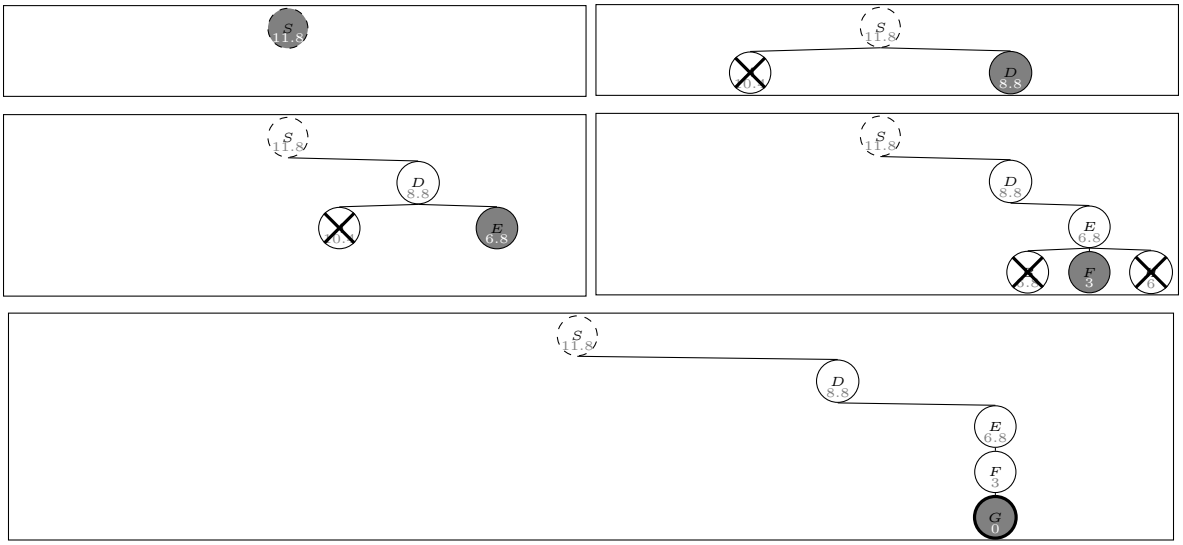
2.7.2 Heuristic Search Methods



Figuur 12: Hill Climbing toegepast op het wegenplan

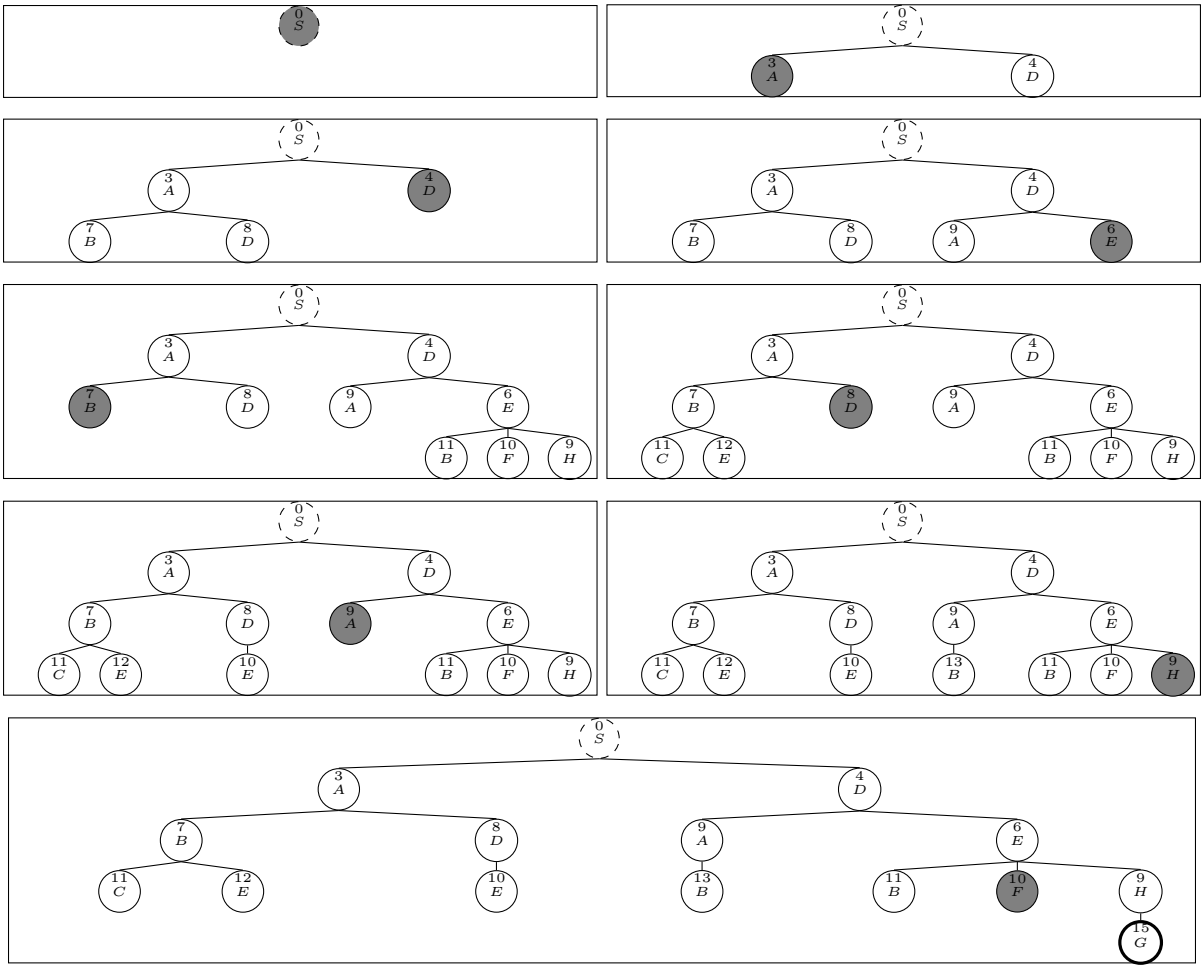


Figuur 13: Beam Search toegepast op het wegenplan (met $w = 2$)

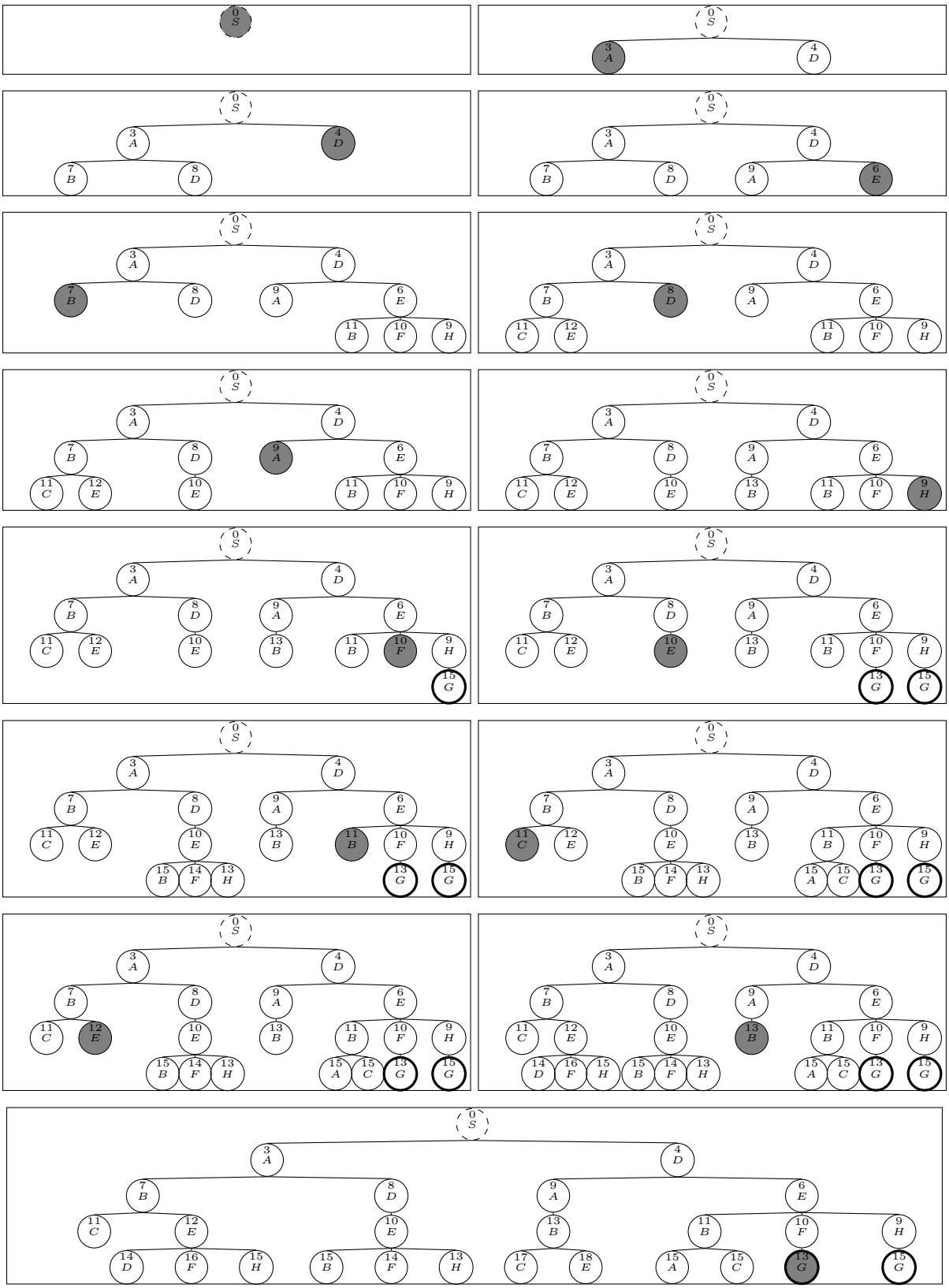


Figuur 14: Hill Climbing 2 toegepast op het wegenplan

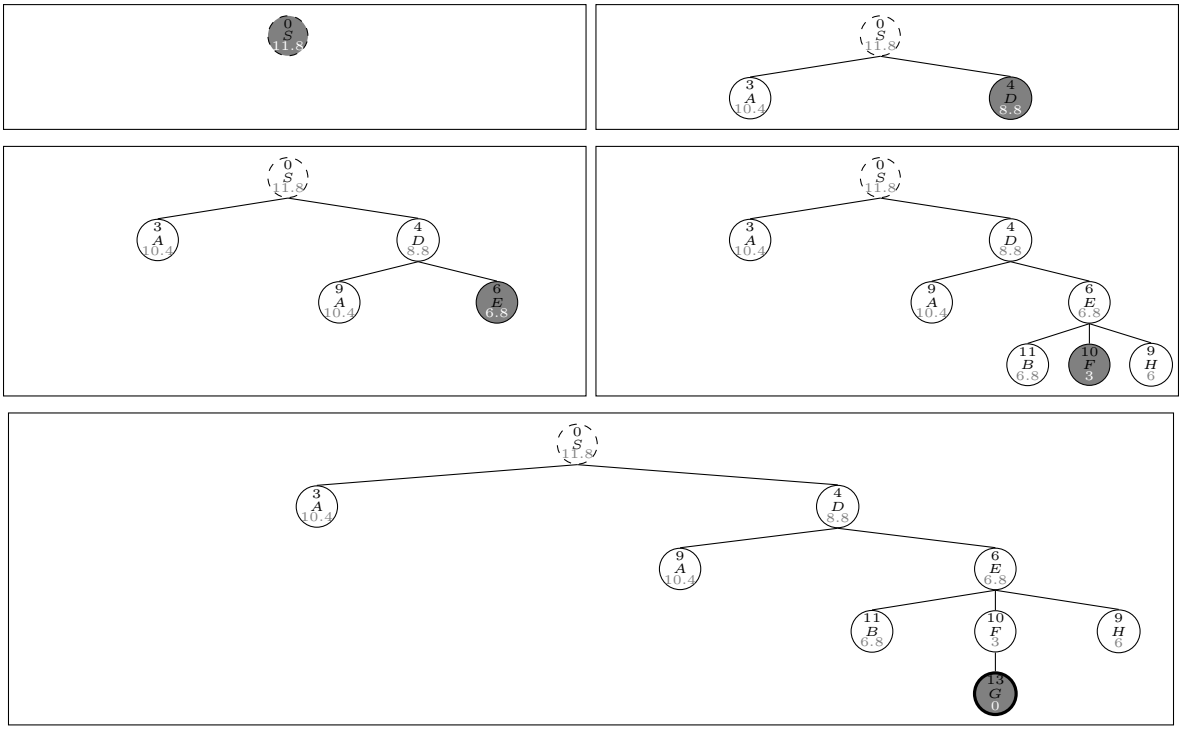
2.7.3 Optimal Search Methods



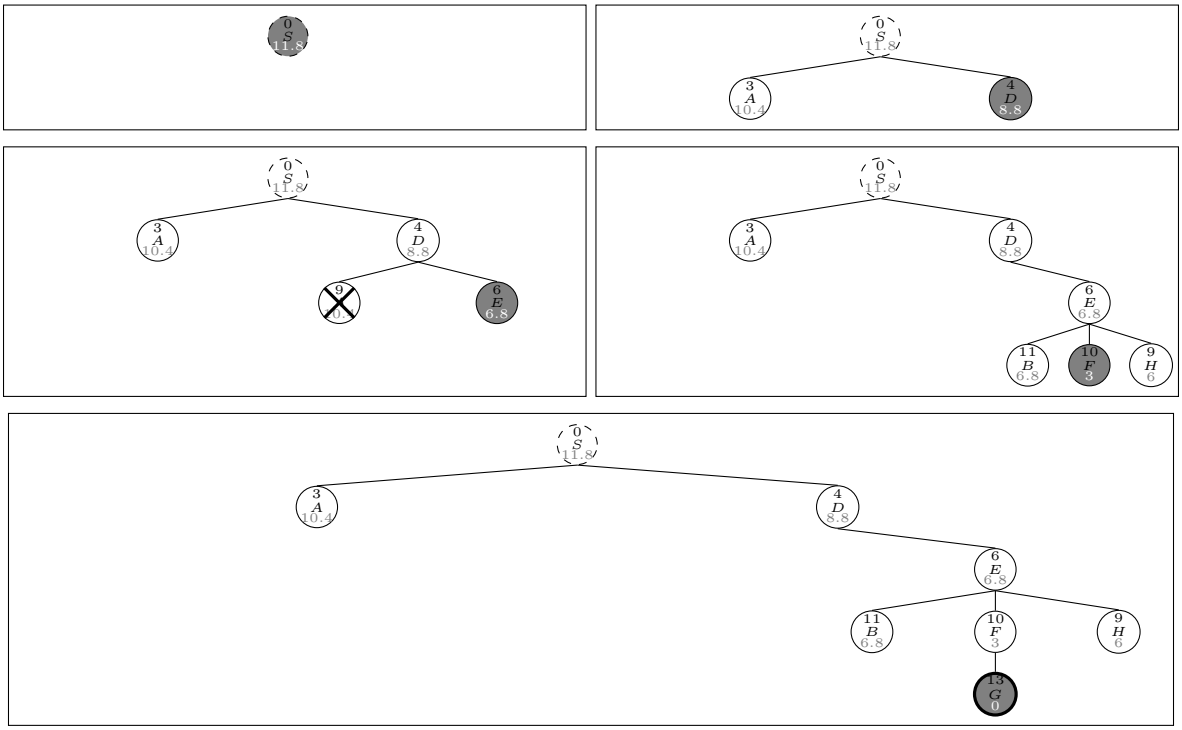
Figuur 15: Uniform Cost Search toegepast op het wegenplan



Figuur 16: Optimal Uniform Cost Search toegepast op het wegenplan



Figuur 17: Estimate-Extended Uniform Cost Search toegepast op het wegenplan



Figuur 18: A* Search toegepast op het wegenplan

3 Concept Learning (Machine Learning)

> *Een hypothese vormt een stimulans om naar bepaalde eigenschappen te zoeken of ze juist te verwaarlozen, of zelfs te ontkennen.*
>
> - Umberto Eco, Italiaans schrijver en criticus (1932-)

Machine learning is een probleem waarbij we proberen een machine in staat te stellen zelf te leren vanuit ervaring. In deze cursus gaan we alleen dieper in op **Concept Learning**. Hierbij proberen we de machine een concept te laten leren. Dit moet vervolgens resulteren in een algoritme die het beslissingsprobleem kan oplossen: "behoort een bepaalde situatie tot het gekende concept?". Met andere woorden we geven een bepaalde situatie s en we verwachten dat de machine kan antwoorden met juist of fout of de situatie die we tonen wel degelijk tot het concept behoort. Om de machine te trainen in het begrijpen van het concept geven we een hoeveelheid testdata: een reeks situaties waarbij we zelf vermelden of de situatie deel uitmaakt van het beoogde concept. Algemeen kunnen we dus stellen dat het algoritme in staat moet zijn om de verzameling van situaties op te splitsen in juiste of foute. Dit proces is uiteraard niet deterministisch, we hebben immers alleen maar de volledige notie van het concept indien we de volledige verzameling situaties als testdata zouden ingeven, in dat geval is het algoritme uiteraard zinloos.

Formeel kunnen we ieder probleem waarbij we de machine het concept willen laten leren als volgt voorstellen:

- Een set van "**events**" $\mathbb{E}$ dit zijn de parameters waarover we informatie hebben in ons probleem. Bijvoorbeeld "First Name", "Country" en "Job".
- Een ongekende doelfunctie c, deze functie beslist iedere situatie in het probleem. $c : \mathbb{E} \to \mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$
- Een set testdata D: situaties waarvoor c wel gekend is
- Een hypothesetaal die kan uitdrukken welke situaties wel en welke niet tot het concept behoren.

We zijn op zoek naar een hypothese h zodat geldt voor alle testdata D:

$$\forall d \in D : d \text{ is waar volgens } h \leftrightarrow c\,(d) \tag{12}$$

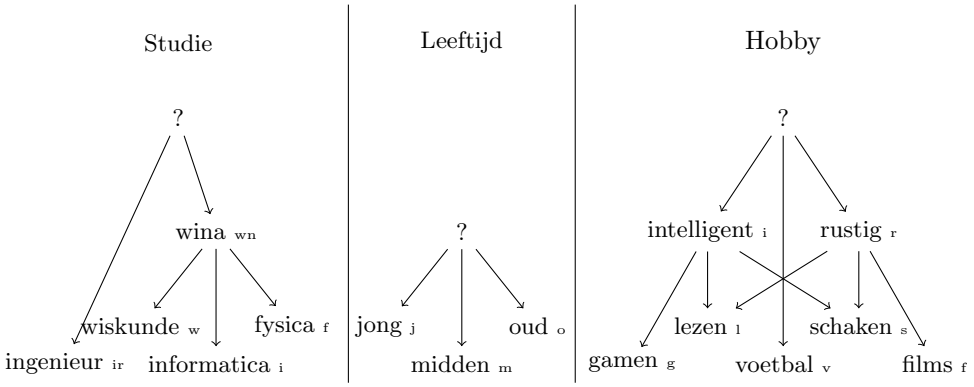
Voor dit probleem zijn in de loop der tijd verschillende oplossingen bedacht zoals neurale netwerken. De oplossing die we hier behandelen heet **Version Spaces**.

3.1 Leidend Voorbeeld: Solliciteren

Een bedrijf wil enkele nieuwe werknemers aanwerven. Er reageren echter een groot aantal mensen op de vacature. Manueel selecteren van de sollicitanten die een gesprek krijgen lijkt dan ook onbegonnen werk. We zullen in dit hoofdstuk een manier uitwerken zodat een programma op basis van enkele beslissingen die door de personeelsdirecteur zijn gemaakt, de rest van de sollicitanten zal indelen. Er worden slechts enkele parameters uit het curriculum vitae gebruikt: de studie, de leeftijd (dit is overigens verboden) en de hobby van de persoon in kwestie. Bovendien zullen we in dit voorbeeld slechts enkele waarden invullen. We ordenen en groeperen de verschillende mogelijke waarden volgens figuur 19. De testdata wordt weergegeven in tabel 2.

3.2 Version Spaces

Version spaces is een symbolische oplossingsmethode, waarbij aan de hand van een **hypothesetaal** een concept geleerd wordt, die vervolgens de overige data kan beoordelen.



Figuur 19: Conceptboom van de sollicitatie

#	Studie	Leeftijd	Hobby	+/−
1	fysica	midden	voetbal	−
2	informatica	jong	schaken	+
3	ingenieur	oud	gamen	−
4	wiskunde	jong	lezen	+
5	fysica	jong	gamen	+
6	ingenieur	jong	gamen	−

Tabel 2: Testdata van het sollicitatie voorbeeld

3.2.1 Naïeve benadering

Een eerste en zeer naïeve oplossingsmethode is ervan uitgaan dat alleen de situaties die we gezien hebben in de testdata die positief zijn, ook de situaties zijn die tot het concept behoren. In dat geval bouwen we niets anders dan een zoekalgoritme die kijkt of de ingegeven situatie bij de testdata positief vermeld stond en dan juist teruggeeft, indien niet geeft het vals terug. Een even slechte keuze is natuurlijk net het omgekeerde: alle situatie voldoen aan het concept tenzij de testdata dit tegenspreekt. Aan deze oplossingen hebben we natuurlijk helemaal niets: het is de bedoeling dat we iets kunnen zeggen over situaties die we nog niet kennen. Bij het eerste geval moet we immers het concept **veralgemenen** (verbreden), in het tweede voorbeeld moet we het concept kunnen **specificeren** (vernauwen).

> Concreet betekent dit dus dat we bij het voorbeeld alleen juist teruggeven op (informatica, jong, schaken), (wiskunde, jong, lezen) en (fysica, jong, gamen). In het tweede geval is het resultaat altijd juist, behalve indien er (fysica, midden, voetbal), (ingenieur, oud, gamen) en (ingenieur, jong, gamen) ingevoerd wordt.

3.2.2 Oplossing: bouwen van een hypothesetaal

De oplossing van dit probleem bestaat eruit een taal te ontwikkelen, die een concept kan uitleggen: de hypothesetaal. Deze taal moet het mogelijk maken een verzameling van hypotheses of beschrijvingen te genereren: de **hypotheseruimte**. Vervolgens kan een concept alleen uitgelegd worden aan de hand van één van de elementen uit de hypotheseruimte. Een goede hypothesetaal moet het probleem van zinloze beschrijvingen tegengaan, zoals gesteld in 3.2.1. Daarnaast moet het de machine tot veralgemening of specifiëring dwingen, die in de meeste gevallen nuttig is (zo is bijvoorbeeld een veralgemening naar een concept “alles is toegelaten”, meestal geen goed idee).

In deze cursussen ontwikkelen we een hypothesetaal die gezien kan worden als een conjunctie van gelijkheden. Hierbij zien we een hypothese als een rij waarbij ieder element een bepaalde eigenschap is van de situatie die we kunnen meten.

Concreet kunnen volgende voorbeelden, hypotheses zijn voor het leidend voorbeeld met als parameters [Studie, Leeftijd, Hobby] zijn:

- [wiskunde, jong, lezen]: Alle jonge wiskundigen die als hobby lezen
- [informatica, ?, ?]: Alle informatici
- [wina, ?, ?]: Alle mensen die wiskunde informatica of fysica gestudeerd hebben (de twee vorige hypotheses behoren hier dus ook toe)
- [ingenieur, oud, ?]: Alle oude ingenieurs
- [?, jong, rustig]: Alle jonge mensen die een rustige hobby uitoefenen
- [?, ?, ?]: Iedereen
- $\perp$: Niemand

Hierbij beperken we de concepten dus niet noodzakelijk tot een equivalentie van alle waarden. Het vraagteken (?) betekent dat deze parameter niet relevant is voor het concept, in dat geval betekent dit dat wat er ook staat op deze plaats in de situatie, dit niet zal bepalen of we de situatie verwerpen. Anderzijds kunnen we ook een hiërarchie opbouwen in waarden die bij de parameters horen. Zo staat "rustig" duidelijk boven "lezen" en "schaken", en staat "wina" boven "fysica" en "wiskunde" (zie ook de hiërarchie van het leidend voorbeeld op figuur 19). Tot slot bestaat er ook nog een lege hypothese: $\perp$, in dat geval zal iedere situatie niet tot het concept behoren. Het spreekt voor zich dat [?, ?, ?] en $\perp$ niet echt de beoogde oplossingen zijn. Daar we hierbij eenvoudigweg stellen dat respectievelijk alles en niets tot het concept behoren.

3.2.3 De Inductieve Leer-hypothese

Zal deze taal wel volstaan om een bepaald concept te leren? Hierbij beroepen we ons op de Inductieve Leer-hypothese, dat is een theorema dat stelt:

theorema 4. *Indien we een hypothese h vinden die de doelfunctie c op de testdata D voldoende benadert, dan zal h c ook voldoende benaderen op situaties die niet behoren tot D.*

Met dit theorema zijn we in staat een minder naïef algoritme, **Find-S**, te bouwen die een concept leert door middel van generalisatie. Dit algoritme vertrekt van een $\perp$ hypothese, met andere woorden het verwerpt iedere situatie die het tegenkomt. Wanneer we de testdata evalueren zal het bij een positieve test-situatie de hypothese minimaal generaliseren. Dit resulteert in een hypothese die altijd waar is bij positieve situaties in de testdata. Dit is echter onvoldoende omdat negatieve test-situaties niet altijd als fout beoordeeld zullen worden (we kunnen dus stellen dat de hypothese de testdata niet voldoende benadert). Toch zal dit algoritme één van de elementen vormen in de oplossing. Dit algoritme wordt beschreven in **Algorithm 17**. Uiteraard is het kiezen van de minst veralgemenende hypothese niet

Algorithm 17 Find-S

```
1: h ←⊥
2: for all d ∈ D ∧ c(d) do
3:     if ¬hypothesisCovers(h, d) then
4:         h ← minimalGeneralisationThatCovers(h, d)
5:     end if
6: end for
```

deterministisch bepaald. Algemeen zijn er echter een aantal richtlijnen:

- Events die voldoende algemeen waren om de situatie toch te doen slagen worden niet aangepast
- Events die in de hypothese niet voldoende algemeen waren worden veralgemeend door een ouderwaarde te kiezen die zowel de oorspronkelijke waarde bevat en de waarde van de falende situatie

Concreet betekent dit dus dat bij de eerste veralgemening eenvoudigweg de situatie als nieuwe hypothese gezien wordt. Verder zijn er soms verschillende ouder-waarden die de twee waarden ondersteunen. In dat geval dient er eenvoudigweg één van de mogelijkheden gekozen te worden.

Find-S is hierdoor niet geschikt om tot een goede hypothese te komen, zowel het niet-deterministisch karakter als het afleveren van foute hypotheses maken het ongeschikt. Verder is het algoritme niet in staat inconsistenties in de data te detecteren, wanneer eenzelfde situatie eerst positief en daarna negatief voorkomt. Voordelen van Find-S zijn dan weer dat het zowel een goede tijds- $\mathcal{O}(n)$ als geheugencomplexiteit $\mathcal{O}(1)$ bevat.

Indien we Find-S toepassen op het leidend voorbeeld, kan dit tot een scenario leiden zoals bij tabel 3. Merk op dat de keuze van generalisatie bij sample 4 niet deterministisch bepaald is. We kunnen dus afhankelijk van de details van de implementatie van Find-S tot een andere hypothese komen.

#	Hypothese	Alternatieven
I	$\bot$	
2	[informatica, jong, schaken]	
4	[wina, jong, rustig]	[wina, jong, intelligent]
5	[wina, jong, ?]	

Tabel 3: Find-S toegepast op het leidend voorbeeld.

De tegenhanger van Find-S is **Dual Find-S** hierbij wordt uitgegaan van het meest algemene geval ($[?, ?, \ldots, ?]$), vervolgens wordt bij iedere negatieve test-situatie die niet aan de hypothese voldoet de hypothese minimaal gespecificeerd. Zoals beschreven in **Algorithm 18**. Opnieuw gelden hier dezelfde

Algorithm 18 Dual Find-S

```
1: h ← [?, ?, ..., ?]
2: for all d ∈ D ∧ ¬c(d) do
3:     if hypothesisCovers(h, d) then
4:         h ← minimalSpecialisationthatNotCovers(h, d)
5:     end if
6: end for
```

opmerkingen als bij Find-S.

We passen opnieuw het leidend voorbeeld toe, op Dual Find-S. Het resultaat hiervan wordt weergegeven in tabel 4. Opnieuw stellen we vast dat we tussen verschillende opties kunnen kiezen. Bovendien kan de uiteindelijke hypothese onmogelijk voldoen. Indien we de positieve voorbeelden hierop testen falen deze allemaal!

#	Hypothese	Alternatieven
I	[?, ?, ?]	
1	[?, ?, intelligent]	[ir, ?, ?], [w, ?, ?], [i, ?, ?], [?, j, ?], [?, o, ?], [?, ?, r]
3	[?, midden, intelligent]	[wina, ?, ?], [?, ?, lezen], [?, ?, schaken]
6	[?, midden, intelligent]	(vorige hypothese voldoet)

Tabel 4: Dual Find-S toegepast op het leidend voorbeeld.

3.2.4 Version Spaces: het idee

Het idee van Version Spaces is nochtans gebaseerd op Find-S en Dual Find-S. Maar de twee algoritmen worden tegelijk uitgevoerd, en bovendien worden alle nieuwe minimale generalisaties en specificaties

bijgehouden in een boomstructuur in tegenstelling tot het kiezen van eentje bij Find-S en Dual Find-S. Nadien worden alle mogelijke generalisaties afgewogen tegen alle mogelijke specificaties. We zullen het algoritme hiervoor stap voor stap opbouwen en verschillende optimalisaties implementeren. Hiervoor beginnen we met een grofstructuur van hoe het algoritme er dient uit te zien in **Algorithm 19**. Hierbij

Algorithm 19 Grofstructuur van het Version-Spaces algoritme

```
{Verzameling van hypotheses die vanuit de meest algemene hypothese vertrekken}
G ← {[?, ?, ..., ?]}
{Verzameling van hypotheses die vanuit de meest specifieke hypothese vertrekken}
S ← {⊥}
for all d ∈ D do
    if ¬notEmpty (G) ∨ ¬notEmpty (S) then
        {Dit treed op indien de testdata tegenstrijdige informatie bevat}
        return failure
    end if
    if c (d) then
        {positieve test-situatie}
        for all h ∈ S do
            if ¬hypothesisCovers (h, d) then
                {Een hypothese heeft een probleem met de test-situatie}
                {Oplossing}
                ...
            end if
        end for
        {Alle problemen zijn opgelost}
    else
        {negatieve test-situatie}
        for all h ∈ G do
            if hypothesisCovers (h, d) then
                {Een hypothese heeft een probleem met de test-situatie}
                {Oplossing}
                ...
            end if
        end for
        {Alle problemen zijn opgelost}
    end if
end for
```

wordt iedere test-situatie behandelt. Indien de test-situatie een positief voorbeeld is, controleren we of alle hypothesen in S wel de test-situatie bevatten. Indien dit niet zo is, zullen we hiervoor een oplossing moeten bedenken. Omgekeerd indien het voorbeeld een negatief voorbeeld is, mag geen enkele van de hypotheses in G deze situatie bevatten. Indien één van de hypotheses dit toch doet, dient deze gespecificeerd te worden.

Indien we een hypothese tegenkomen in G die een bepaalde situatie positief beoordeelt terwijl deze negatief is, betekent dit dat de hypothese te algemeen is, we moeten met andere woorden de hypothese vervangen door een hypothese die minimaal gespecificeerd wordt. Indien er meerdere minimale specificaties mogelijk zijn, dienen deze allemaal toegevoegd te worden. De falende hypothese moet vervolgens uit G verwijdert worden. Indien we echter al deze hypothesen zomaar toevoegen riskeren we een grote aantal hypotheses te bekomen die indien we later G en S tegen elkaar uitspelen nutteloos blijken te zijn. We hoeven enkel hypothesen toe te voegen die minstens de generalisatie zijn van één hypothese in S (generalisatie betekent dat iedere mogelijke situatie die door de eerste hypothese aanvaardt wordt, ook door de tweede aanvaardt zal worden). Anders komen we immers tegenstrijdige hypothesen aan de twee kanten uit.

Omgekeerd kunnen we uiteraard stellen dat een hypothese in S die een gegeven situatie negatief beoordeelt terwijl deze positief is, veralgemeent dient te worden. Opnieuw indien er meerdere minimale veralgemeningen zijn dienen deze allemaal toegevoegd te worden, maar alleen indien deze hypothese minstens een veralgemening in G vindt. Dergelijke regels lijken misschien niet veel uit te maken. Stel echter dat we over veel test-data beschikken kunnen uit deze extra (en foute) hypothesen heel wat nieuwe hypothesen gegenereerd worden, die ook fout zijn. Dit kan op termijn een grote overhead veroorzaken.

Een andere optimalisatie bestaat erin om indien we bijvoorbeeld een negatieve test-situatie tegenkomen te controleren of alle hypothesen in S deze niet bevatten. Indien er een hypothese is die deze test-situatie toch positief beoordeelt, is deze duidelijk te algemeen. Deze kan dus bijgevolg weggegooid worden. Dit principe heet **Pruning**. Uiteraard werkt dit ook met positieve voorbeelden in de omgekeerde richting.

Indien een hypothese in G specifieker is dan een andere hypothese in G hebben we overduidelijk te maken met redundantie. De meest specifieke hypothese zou immers nooit gegenereerd mogen worden, omdat we alleen specifieker werken indien de algemenere hypotheses falen. Dit probleem kan echter voorkomen indien twee verschillende hypotheses eenzelfde minimale specificatie hebben, en op een gegeven moment één van de hypotheses niet meer geldig is. Het verwijderen van **redundante hypotheses** kan dan ook een behoorlijke snelheidswinst inhouden. We verwijderen hier dus de meest specifieke hypothese uit G. Ook dit principe werkt in de andere richting. Eventueel kan een hypothese na verloop van tijd zowel in S als G voorkomen, in dat geval spreken we van **convergentie**.

Indien we alle voorgaande principe en systemen nu implementeren in de grofstructuur van **Algorithm 19**. Dan bekomen we een algoritme zoals beschreven in **Algorithm 20**.

Nu we dit algoritme gedefinieerd hebben, kunnen we de karakteristieken wat dichterbij bekijken. Een eerste aspect dat duidelijk opvalt is de symmetrie. In het algoritme kan de behandeling van positieve situaties volledig vergeleken worden met de behandeling van negatieve situaties. Een tweede aspect is dat dit algoritme in principe geen geheugen heeft voor de eerder geëvalueerde test-situaties. Met andere woorden, de situaties kunnen één-voor-één ingevoerd en geëvalueerd worden, zonder dat ze ergens bewaard dienen te worden. Men kan hier dan ook dynamisch mee omspringen door bijvoorbeeld eerst een kleine hoeveelheid test-data te evalueren, op basis van het geleerde concept andere situaties te beoordelen, en vervolgens indien de resultaten niet bevredigend zijn meer test-data in te voeren. De hypothese-sets fungeren immers als een soort van geheugen hiervoor. Version Spaces kan echter net als Find-S en Dual Find-S niet overweg met fouten (**noise**) in de test-data. Indien een bepaalde situatie tweemaal voorkomt bijvoorbeeld in zowel een positieve als een negatieve gedaante, zal één van de sets leeggehaald worden, en het falen van het algoritme tot gevolg hebben. Indien de test-data geen inconsistenties bevat, maar gewoon fouten zullen deze fouten mee in rekening gebracht worden en dus tot fouten bij het beslissen leiden. Andere technieken zoals Neurale Netwerken zijn wel in staat om fouten en inconsistenties te behandelen. Door een forgive-and-forget systeem, zal de invloed van een bepaalde test-situatie verminderen. Indien er dus een fout in de test-data zit, kan door een grote stroom aan correcte data, deze fout rechtgezet worden.

Het algoritme kan eindigen omwille van twee redenen: ofwel omdat alle test-data geëvalueerd is, ofwel omdat minstens één van de hypothese-sets leeg is. Indien er geen test-data meer is, kunnen we bewijzen dat ons algoritme per definitie een juist resultaat zal geven op de ingevoerde test-data. Indien het algoritme op een andere manier beëindigd wordt, kan één van de hierboven opgesomde problemen aan de oorzaak liggen, ofwel is de hypothesetaal eenvoudigweg niet toereikend, om het concept te vatten, dit probleem wordt behandeld in 3.2.5.

Het algoritme is verder ook in staat om zelf hints te geven welke test-data het best zou krijgen. Indien een bepaalde situatie op een moment door bijvoorbeeld de helft van de hypotheses positief beoordeeld wordt, en door de andere helft negatief, is het logisch dat het antwoord op deze situatie de grootste aanpassing van de hypothese-sets zal teweegbrengen. In dat geval kan het programma eventueel een hint aan de gebruiker geven dat deze situatie best als test-data ingevoerd wordt.

Algorithm 20 Complete implementatie van het Version-Spaces algoritme

```
1: G ← {[?, ?, ..., ?]}
2: S ← {⊥}
3: for all d ∈ D do
4:     if ¬notEmpty(G) ∨ ¬notEmpty(S) then
5:         return failure
6:     end if
7:     if c(d) then
8:         for all h ∈ S do
9:             if ¬hypothesisCovers(h, d) then
10:                S+ ← minimalGeneralisationsThatCovers(h, d)
11:                removeIf(S+, ∀g ∈ G : ¬isSpecialisationOf(S+[i], g))
12:                removeIf(S+, ∃s ∈ S ∪ S+ : isGeneralisationOf(S+[i], s))
13:                S ← S ∪ S+
14:            end if
15:        end for
16:        for all h ∈ G do
17:            if ¬hypothesisCovers(h, d) then
18:                G ← G \ {h}{Pruning}
19:            end if
20:        end for
21:    else
22:        for all h ∈ G do
23:            if hypothesisCovers(h, d) then
24:                G+ ← minimalSpecialisationsThatNotCovers(h, d)
25:                removeIf(G+, ∀s ∈ S : ¬isGeneralisationOf(G+[i], s))
26:                removeIf(G+, ∃g ∈ G ∪ G+ : isSpecialisationOf(G+[i], g))
27:                G ← G ∪ G+
28:            end if
29:        end for
30:        for all h ∈ S do
31:            if hypothesisCovers(h, d) then
32:                S ← S \ {h}{Pruning}
33:            end if
34:        end for
35:    end if
36: end for
```

Op figuur 20 passen we nu het volledige Version Space algoritme toe op het leidend voorbeeld. Initieel starten we dus met twee bomen die langs de G kant de $[?, ?, ?]$ hypothese bevat. Langs de S kant rekenen we vanaf de $\perp$ hypothese. Het eerste sample is een negatief voorbeeld. We zien dat G niet meer toereikend is, en splitsen de hypothese op in alle mogelijke nieuwe hypotheses. Bij het tweede sample dienen we S te veralgemenen. Het eerste positieve voorbeeld zal telkens het volledige sample kopiëren. Verder dienen we sommige hypotheses uit G te verwijderen. Deze hypotheses worden door pruning verwijdert, ze kunnen immers onmogelijk de hypothese vormen. Het derde sample is opnieuw negatief. Hierbij voldoet $[?, ?, \text{intelligent}]$ niet langer. We genereren hier dus opnieuw deelhypotheses uit. Sommige van deze hypothese kunnen echter niet bestaan. Er is immers voor deze hypothesen geen specifieker geval aan de zijde van S. Bijvoorbeeld $[?, ?, \text{lezen}]$. Deze hypothese worden dan ook onmiddellijk verwijdert. Andere hypothese zoals $[?, \text{jong}, \text{intelligent}]$ dienen we ook te verwijderen. Deze hypothese is redundant aan de hoger gelegen $[?, \text{jong}, ?]$ hypothese. We houden in totaal één nieuwe hypothese over. Het vierde sample zal vervolgens S verder veralgemenen. Er zijn echter twee mogelijk veralgemeningen. Verder zullen we door pruning de $[\text{informatica}, ?, ?]$ hypothese in G moeten verwijderen. Het volgende sample is ook positief. $[\text{wina}, \text{jong}, \text{rustig}]$ voldoet niet langer en veralgemenen we naar $[\text{wina}, \text{jong}, ?]$. Deze hypothese is echter een veralgemening van een hypothese

die wel kan blijven bestaan. Bijgevolg verwijderen we ook de nieuwe hypothese. Het laatste sample ten slotte dwingt [?, jong, ?] zich verder te specifiëren. Slechts één van de nieuwe hypothesen zal geen enkel positief sample negatief beoordelen. We bekomen dus als resultaat situatie F zoals op figuur 20.

Het beslissingsalgoritme Naast het leren van een concept moeten Version Spaces uiteindelijk ook kunnen beslissen of een gegeven situatie wel degelijk deel uitmaakt van het concept. Indien de gebruiker vraagt of een bepaalde situatie tot het concept behoort, zal het beslissingalgoritme testen of iedere hypothese (zowel in G als in S) deze situatie juist beslist. Nochtans hoeven we enkel de hypotheses van S te testen om te besluiten dat een situatie wel degelijk positief is, G bevat immers alleen maar generalisaties van S en zal dus bijgevolg ook altijd juist testen. Omgekeerd hoeven we alleen de hypotheses van G te testen om te besluiten dat iedere hypothese de situatie verwerpt. Het probleem is uiteraard dat een situatie wel eens door verschillende hypotheses anders beoordeeld kan worden. In dat geval wordt meestal de ratio tussen positieve en negatieve beoordelingen berekend en wordt afhankelijk hiervan een positief of negatief resultaat teruggegeven, meestal met vermelding van de probabiliteit. Indien er echter evenveel positieve als negatieve beoordelingen zijn wordt er meestal geen conclusie getrokken. Een compacte implementatie hiervoor wordt beschreven in **Algorithm 21**.

Algorithm 21 Conclusie trekken uit een Version-space

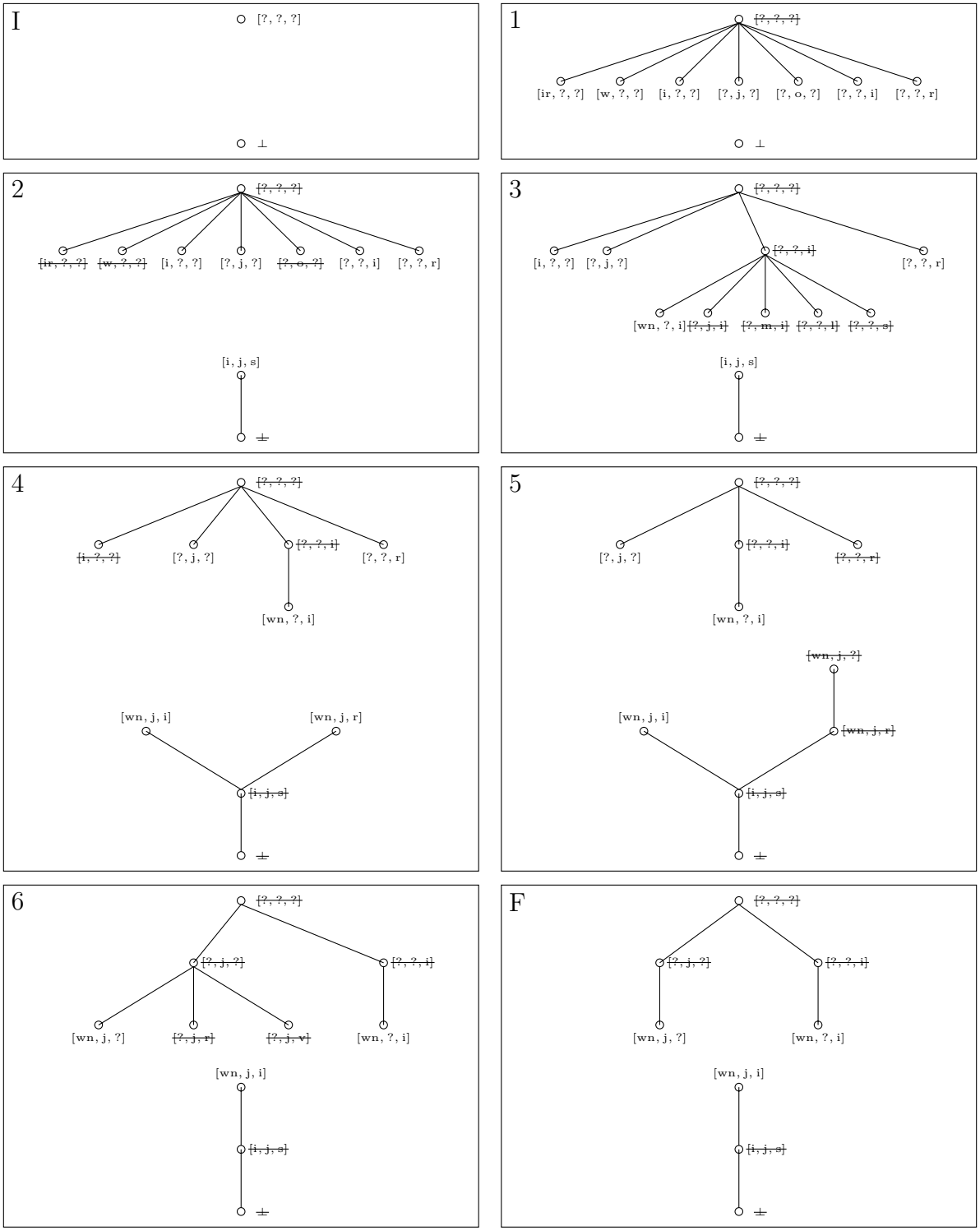
```
1:  n+ ← 0
2:  n− ← 0
3:  for all h ∈ S ∪ G do
4:      if hypothesisCovers (h, d) then
5:          n+ ← n+ + 1
6:      else
7:          n− ← n− + 1
8:      end if
9:  end for
10: if n− = 0 then
11:     return true
12: else if n+ = 0 then
13:     return false
14: else if n+ > n− then
15:     return "true with a probability of n+/(n+ + n−)"
16: else if n+ < n− then
17:     return "false with a probability of n−/(n+ + n−)"
18: else {n+ = n−}
19:     return "no conclusion"
20: end if
```

3.2.5 De relevantie van de hypothesetaal

De hierboven gestelde hypothesetaal slaagt er niet in om alle mogelijke concepten te leren (omdat er bijvoorbeeld enkel generalisaties zijn die later tegengesproken worden door een ander voorbeeld). Kunnen we met een expressievere taal eventueel betere concepten aanleren? Stel dat we een nieuwe taal definiëren die opnieuw werkt door middel van symbolen. Hierbij definiëren we elementen van de vorm < parameter >=< waarde >. Verder laten we conjuncties (die ook al toegelaten waren), disjuncties en negaties toe. Deze taal kan met andere woorden iedere deelverzameling van mogelijke staten bevatten (elke staat valt te beschrijven door een conjunctie van al haar parameters, met een disjunctie kunnen we deze combineren). Het probleem is dat we weer tot dezelfde resultaten komen als onze naïeve implementatie in het begin (zie 3.2.1). En we dus met andere woorden niets kunnen leren. We geven enkel waar indien we de situatie al positieve gedaante zagen in de testdata, en negatief indien de situatie als negatief voorbeeld in de test-data stond. We kunnen dus stellen dat we nood hebben aan een **Inductieve**

BIAS, dit is een aanname dat het concept kan geleerd worden door een bepaalde set regels voor een hypothesetaal (Zo is de hypothesetaal hierboven te beschrijven als een conjunctie van eigenschappen).

Hoe weten we echter of onze hypothesetaal zal volstaan? Dit is uiteraard niet op voorhand te bepalen. In de loop der tijd zijn er echter verschillende hypothesetalen ontstaan. We kunnen een orde specifiëren in de talen naar representatievermogen. Veel praktische algoritmen hebben dan ook een set van dergelijke talen. Indien de gebruiker niet tevreden is met de resultaten van een beperkte hypothesetaal, verandert het algoritme de taal naar een hypothesetaal met een breder representatievermogen. Dit vermijdt de keuze van een taal, bovendien zal de minst representatieve taal het meest algemene concept genereren, waardoor het concept waarschijnlijk dan ook het best gedefinieerd zal zijn.



Figuur 20: Verloop van Version Spaces op de testdata.

4 Constraint Processing

> *Alle beperking maakt gelukkig.*
>
> - Arthur Schopenhauer, Duits filosoof (1788-1860)

Een alternatief voor zoekalgoritmen bij het oplossen van zoekproblemen, is **Constraint Processing**. Zoals de naam al doet vermoeden kan constraint processing alleen maar een bepaald type zoekproblemen oplossen: **Constraint Problems**.

4.1 Wat zijn Constraint Problems?

Constraint problemen zijn problemen die bestaan uit:

- een set variabelen: $z_1, z_2, \ldots, z_n$
- een eindig domein per variabele: $\mathrm{dom}\,(z_i) = d_i = \{a_{i\,1}, a_{i\,2}, \ldots, a_{i\,n_i}\}$
- een set constraints ($c : d_i \times d_j \times \ldots \times d_l \to \mathbb{B}$): een functie die waar teruggeeft indien aan een bepaalde eis gerelateerd aan de variabelen van de constraint voldaan wordt, anders vals. Constraints worden opgesplitst worden in drie types:
  - **Unaire constraints**: iedere variabele z_i heeft een constraint $c\,(z_i)$
  - **Binaire constraints**: voor iedere twee variabelen z_i en z_j met $i < j$: $c\,(z_i, z_j)$
  - Optioneel **Meervoudige constraints**: $c\,(z_i, z_j, \ldots, z_l)$, deze constraints worden echter buiten beschouwing gelaten en zijn eerder uitzonderlijk

Formeel stelt het probleem concrete waarden voor de variabelen te vinden die een element zijn van hun respectievelijk domein, en daarenboven zo dat aan alle constraint voldaan wordt. Dit kan uiteraard gedaan worden met de reeds behandelde zoekalgoritmen. Voor dergelijke problemen bestaan er echter efficiëntere oplossingsmethodes.

4.2 Leidend Voorbeeld: 4-Teachers

We introduceren een fictief probleem (omdat dit probleem relatief eenvoudig is, en vele concepten illustreert). We stellen 4 professoren (A,B,C,D). En 5 lokalen. Elke professor dient een bepaalde les te geven. Uiteraard kunnen er geen twee lessen in eenzelfde lokaal plaatsvinden. Daarnaast blijken de professoren nog enkele voorkeuren te hebben:

1. professor A wil geen les geven in lokaal 3 en wil hoogstens 2 lokalen verwijdert zijn van prof. B. Het lokaalnummer dient minstens twee groter te zijn dan dat van professor D.
2. professor B wil geen les geven in lokaal 5. En in een lokaal met een nummer dat minstens 2 groter is dan dat van professor C.
3. professor C wil alleen les geven in lokalen kleiner of gelijk aan 3. Met minstens één lokaal tussen professor A.
4. professor D wil les geven in een lokaal dat exact 2 kleiner is dan het lokaal van professor B.

De enige oplossing voor dit probleem is: $(A, B, C, D) = (5, 4, 1, 2)$. Formeel drukken we deze situatie nu als volgt uit: We beschikken over vier variabelen A, B, C en D. Elke variabele heeft hetzelfde domein, $d_A = d_B = d_C = d_D = \{1, 2, 3, 4, 5\}$. Als unaire constraints hebben we:

$$\begin{cases} c(A) \leftrightarrow A \neq 3 \\ c(B) \leftrightarrow B \neq 5 \\ c(C) \leftrightarrow C \leq 3 \\ c(D) \leftrightarrow \textbf{true} \end{cases} \tag{13}$$

Als binaire constraint beschikken we over:

$$\begin{cases} c(A,B) \leftrightarrow |A-B| \in \{1,2\} \\ c(A,C) \leftrightarrow |A-C| > 1 \\ c(A,D) \leftrightarrow D < A-1 \\ c(B,C) \leftrightarrow C < B-1 \\ c(B,D) \leftrightarrow D = B-2 \\ c(C,D) \leftrightarrow C \neq D \end{cases} \tag{14}$$

4.3 Node-consistency

Unaire constraints zijn slechts gerelateerd aan 1 variabele, hierdoor kunnen we deze al doen kloppen nog voor we een van de oplossingsmethodes gebruiken. Hiervoor gebruiken we **Node-consistency**, ook wel **1-consistency** genoemd. Deze techniek reduceert eenvoudigweg het domein van de variabele tot enkel die waarden waarvoor de constraint klopt. Voor iedere variabele z_i genereren we dus een nieuw domein d'_i met:

$$d'_i = \{a | a \in d_i \wedge c\,(z_i = a)\} \tag{15}$$

Toegepast op ons voorbeeld genereren we dus een nieuw domein voor iedere variable met:

$$\begin{array}{llll} d_A = \{1,2,3,4,5\} & \wedge & c\,(A) \leftrightarrow A \neq 3 & \Rightarrow \quad d'_A = \{1,2,4,5\} \\ d_B = \{1,2,3,4,5\} & \wedge & c\,(B) \leftrightarrow B \neq 5 & \Rightarrow \quad d'_B = \{1,2,3,4\} \\ d_C = \{1,2,3,4,5\} & \wedge & c\,(C) \leftrightarrow C \leq 3 & \Rightarrow \quad d'_C = \{1,2,3\} \\ d_D = \{1,2,3,4,5\} & \wedge & c\,(D) \leftrightarrow \textbf{true} & \Rightarrow \quad d'_D = \{1,2,3,4,5\} \end{array} \tag{16}$$

4.4 Visuele voorstelling van het probleem

We kunnen een dergelijk probleem visueel voorstellen: door middel van een **OR-tree** of met een **Constraint Network**. Deze twee voorstellen vertegenwoordigen bovendien twee totaal verschillende strategieën om een dergelijk probleem op te lossen. Voorbeelden van de twee voorstellingen zijn respectievelijk te zien op figuren 21 en 22.

4.4.1 De OR-tree voorstelling

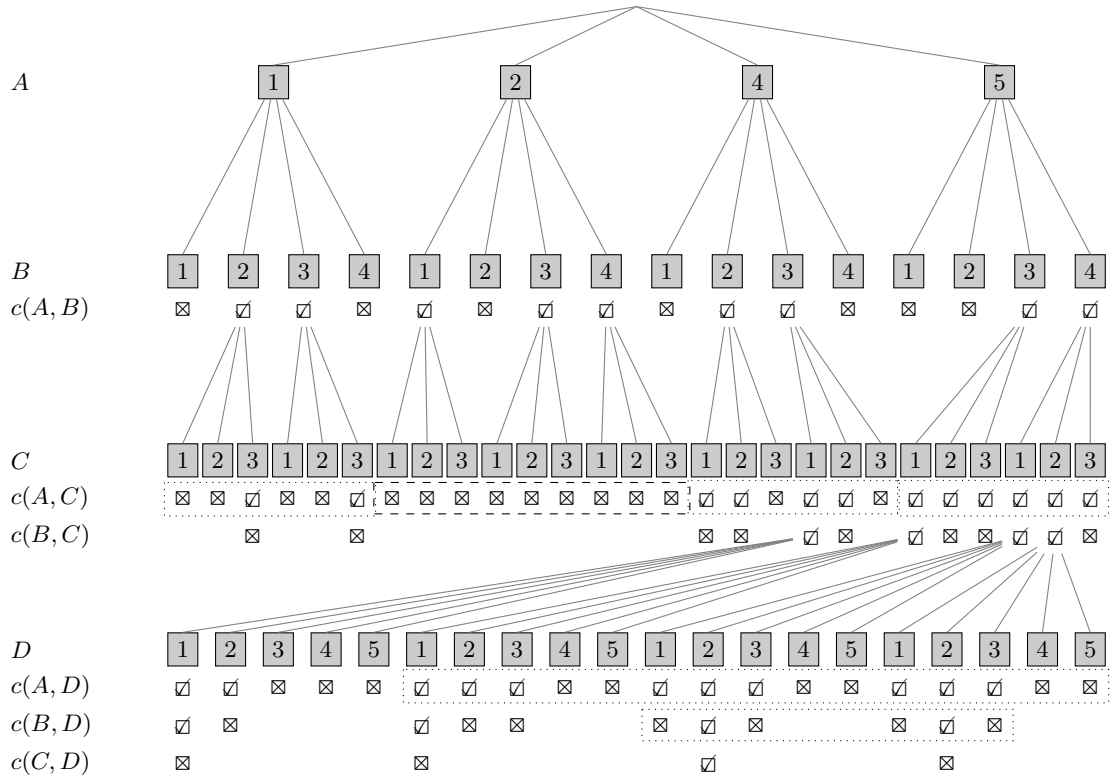
Bij een OR-tree selecteren we een orde in de variabelen, en vervolgens wijzen we aan de hand van die orde een bepaalde laag i van de boom toe aan variabele z_i. Onder iedere knoop die in de laag erboven $i-1$ aan alle constraints $\forall j < i-1 : c\,(z_j, z_{i-1})$ voldoet, genereren we vervolgens knopen, waarbij elke knoop één van de domeinwaarden van z_i draagt. Daaronder testen we vervolgens de constraints op de dan reeds ingevulde waarden: $\forall j < i : c\,(z_j, z_i)$. We passen dit proces iteratief toe in de diepte van de boom.

Het zoeken naar een oplossing met behulp van een dergelijke boom, werkt met ongeveer dezelfde principes als de zoekalgoritmen beschreven in sectie 2. We kunnen echter enkele optimalisaties toepassen door het kiezen van goede **backtrack**-algoritmen. Dit zijn algoritmen die geactiveerd worden op het moment dat we in een knoop zitten waar we geen verdere onderliggende knopen meer kunnen evalueren en dus een niveau terug moeten keren, deze technieken worden besproken in subsectie 4.5.

Een uitgewerkte OR-tree voor het 4-Teachers probleem staat op figuur 21.

4.4.2 Het Constraint Network

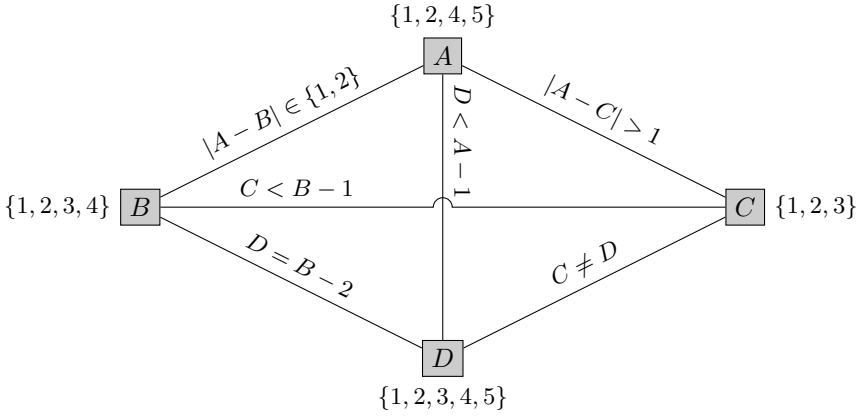
Bij een constraint network stellen we het probleem voor als een grafe. Hierbij vertegenwoordigt iedere knoop een bepaalde variabele. Bogen tussen verschillende punten, vertegenwoordigen dan weer de constraint tussen de twee variabelen van die punten. Verder kennen we ook een verzameling mogelijk waarden



Figuur 21: OR-tree van het 4-teachers probleem.

toe aan een punt, initieel is deze gelijk aan het domein van deze variabele. Vervolgens overlopen we iedere constraint. Indien bepaalde elementen in het domein van een variabele niet mogelijk zijn, worden deze uit het domein verwijdert. Dit proces heet **relaxation** of **relaxatie** en wordt verder beschreven in subsectie 4.6. Het verwijderen van enkele variabelen kan een kettingreactie tot gevolg hebben. Dit resulteert in het beste geval tot één element per variabele. Meestal loopt het echter niet zo'n vaart, en wordt het probleem opgelost met een backtrackingsalgoritme die met de gereduceerde domeinen het probleem sneller oplost.

Indien we dit toepassen op de het 4-Teachers probleem bekomen we een Constraint Network zoals beschreven op figuur 22.



Figuur 22: Constraint Network van het 4-teachers probleem.

4.5 Backtrack algoritmen

Indien we het probleem oplossen met de OR-tree benadering, komen we vaak situaties tegen waarbij we geen kinderen van een bepaalde knoop meer kunnen evalueren. Dit komt bijvoorbeeld omdat alle kinderen geëvalueerd zijn. In zo'n geval is het de bedoeling dat we één of meer niveaus terugspringen. Hiervoor bestaan verschillende implementaties die vooral tot doel hebben om nutteloos op constraints te controleren te vermijden. Deze worden beschreven in subsubsecties 4.5.1 tot 4.5.5.

4.5.1 Chronological Backtracking

Chronological Backtracking is veruit het eenvoudigste algoritme. Indien we dit algoritme toepassen bekomen we niets minder dan diepte-eerst zoeken toegepast op de OR-boom. Met andere woorden hierbij wordt eenvoudigweg naar de ouder teruggekeerd, en de knoop rechts van de huidige knoop wordt vervolgens geëvalueerd. Indien ook in de ouder geen knoop meer te evalueren is, treed er een recursief effect op, waarbij we telkens een niveau hoger gaan tot er uiteindelijk weer een volgende knoop beschikbaar is. Hierbij voeren we een iteratie uit op een bepaalde diepte i over alle waarden in het domein d_i. Indien

Algorithm 22 chronologicalBacktracking (depth)

```
1:  for k = 1 to n_depth do
2:      {Iteratie over alle knopen op niveau depth}
3:      z_depth ← a_depth k {Geef z_depth een concrete waarde}
4:      b ← true
5:      {Controleer alle constraints met z_i}
6:      for j = 1 to depth − 1 do
7:          b ← b ∧ c(z_j, z_i)
8:      end for
9:      if b then
10:         {Alle constraints kloppen}
11:         if depth = n then
12:             {Laatste niveau bereikt, oplossing gevonden!}
13:             return (z_1, z_2, ..., z_n)
14:         else
15:             {Naar het volgende niveau}
16:             chronologicalBacktracking (depth + 1)
17:         end if
18:     end if
19: end for
```

vervolgens alle constraints $\forall j < i : c(z_j, z_i)$ kloppen, dan evalueren we recursief de variabele op het volgende niveau. Indien we de laatste variabele succesvol kunnen toewijzen, beschouwen we dit als een oplossing. Indien we echter vruchteloos alle waarden geëvalueerd hebben, keren we terug naar het vorige niveau (waar de recursieve oproep vandaan kwam).

4.5.2 Backjumping

Een probleem met Chronological Backtracking is dat heel wat constraints nutteloos gecontroleerd worden. Stel dat we een keuze maken voor een waarde van z_3, en alle waarden die we toekennen uit het domein falen op de eerste constraint $c(z_1, z_3)$. In dat geval heeft het geen zin om de volgende waarde voor z_2 te kiezen. Het probleem ligt immers bij z_1 die op dat moment een waarde heeft waarbij geen enkele waarde voor z_3 de constraint kan doen kloppen. Dit fenomeen heet **trashing** en introduceert een grote hoeveelheid nutteloos werk (vooral wanneer z_2 hier een groot domein heeft). In dat geval dienen we eenvoudig weg een nieuwe waarde voor z_1 te kiezen, en dus terug te springen naar niveau 1. Dit concept heet **backjumping**.

Indien we het bovenstaande formeler uitdrukken kunnen we zeggen dat indien op een niveau i alle constraints gefaald hebben voor een zeker niveau k, dan springen we terug naar k. Deze conditie kunnen

we wiskundig als volgt uitdrukken:

$$\forall x \in d_i, \exists l \leq k < i : \neg c\,(z_l, z_i = x) \tag{17}$$

Het volledige concept wordt vervolgens uitgedrukt in **Algorithm 23**.

Algorithm 23 backjumping (depth, **out:** l)

```
1:  l ← 0
2:  for k = 1 to n_depth do
3:      {Iteratie over alle knopen op niveau depth}
4:      z_depth ← a_depth k {Geef z_depth een concrete waarde}
5:      b ← true
6:      {Controleer alle constraints met z_i}
7:      for j = 1 to depth − 1 do
8:          if b ∧ ¬c(z_j, z_i) then
9:              {Eerst falende conditie}
10:             b ← false
11:             l ← max(l, j)
12:         end if
13:     end for
14:     if b then
15:         {Alle constraints kloppen}
16:         if depth = n then
17:             {Laatste niveau bereikt, oplossing gevonden!}
18:             return (z_1, z_2, ..., z_n)
19:         else
20:             {Naar het volgende niveau}
21:             backjumping (depth + 1, out: L)
22:             if L < depth then
23:                 return
24:             end if
25:         end if
26:     end if
27: end for
```

Een concreet voorbeeld van trashing vinden we op figuur 21. Op het moment dat $A = 2$ testen we tot drie maal toe het volledige domein van C tegenover A. Deze checks staan in een gestreepte rechthoek. We hadden echter na één keer reeds kunnen besluiten dat andere waardes voor B dit probleem niet zouden oplossen. Hoewel trashing in dit didactisch voorbeeld slechts éénmaal voorkomt, is het een probleem die indien n groot wordt frequent voorkomt, en dus een grote overhead veroorzaakt. Backjumping zal hier dus op reageren door onmiddelijk na éénmaal volledig C tegenover A te hebben gecontroleerd, een nieuwe waarde voor A te nemen. Dit bespaart ons in totaal 9 controles.

4.5.3 Backmarking

Met backjumping wordt een groot aantal nutteloze controles vermeden. Toch doet dit algoritme nog steeds vaak dubbel werk. Als we immers een waarde voor z_3 kiezen controleren we deze allereerst tegenover z_1, dit doen we echter telkens opnieuw met de keuze van een andere z_2. We controleren met andere woorden vaak tweemaal eenzelfde constraint met dezelfde waarden. Indien deze constraint moeilijk te berekenen is, zorgt dit voor de nodige overhead. Bovendien kan het aantal dubbele controles hoog oplopen wanneer we ons diep in de boom bevinden en bijvoorbeeld z_{15} eindeloos moeten controleren tegenover z_1 tot z_{13}, terwijl we het antwoord al zouden kunnen kennen. Deze **redundante controles** of **redundant checks** kunnen bijgevolg op termijn het algoritme een behoorlijke vertraging opleveren. **Backmarking** is een techniek die probeert het dubbele werk te verminderen, en tegelijk niet al teveel tijd te verliezen bij het opzoeken van de vorige resultaten.

Hiervoor maakt backmarking gebruik van twee tabellen:

- **Backup**(k), deze ééndimensionale rij houdt bij welke variabelen verandert zijn tussen twee controles op een diepte k
- **Checkdepth**(k, l) deze tweedimensionale tabel houdt bij vanaf welke diepte het resultaat van de constraints onzeker is. Alle voorgaande constraints zijn per definitie juist. Op een diepte k met een variabele-waarde $a_{k\,l}$

Het algoritme houdt dus bij hoe diep de constraints klopten, de vorige keer dat ze gecontroleerd werden. Als er in tussentijd niets verandert is aan de waarden van de variabelen, hoeven we de voorgaande controles niet meer te doen. Indien dit niet het geval is doen we de controles wel, en houden we de resultaten bij voor verder gebruik.

Initieel zullen we deze tabellen opvullen met énen[4]. We weten immers niets over de constraints, en hebben nog geen variabelen een waarde toegekend. Na verloop van tijd zal deze tabel wel informatie bevatten. Vervolgens zullen we op een diepte k iedere mogelijke waarde $a_{k\,l}$ aan z_k toekennen, indien hierbij Checkdepth $(k, l) \geq$ Backup (k), betekent dit dat we een aantal constraints opnieuw zullen moeten evalueren vanaf Backup (k) (de minst diepe variabele die gewijzigd werd). Anderzijds indien Checkdepth $(k, l) <$ Backup (k) slaagden de vorige keer niet alle constraints, en zijn hierbij de schuldige variabelen nog niet aangepast, in dat geval hoeven we helemaal niets te evalueren. Dit wordt formeel toegelicht met **Algorithm 24**.

Algorithm 24 backmarking (depth)

```
1:  for k = 1 to n_depth do
2:      z_depth ← a_depth k
3:      if Checkdepth (depth, k) ≥ Backup (depth) then
4:          {Variabelen van eerder falende constraints zijn aangepast, constraints herevalueren}
5:          ok ← true
6:          j ← Backup (depth)
7:          while ok ∧ j < depth do
8:              Checkdepth (depth, k) ← j{Gecontroleerde diepte aanpassen}
9:              ok ← ok ∧ c (z_j, z_depth)
10:             j ← j + 1
11:         end while
12:         if ok then
13:             {alle constraints zijn geaccepteerd}
14:             if depth = N then
15:                 {diepte bereikt, rapporteren}
16:                 return z_1, z_2, ..., z_n
17:             else {Verder uitdiepen}
18:                 backmarking (depth + 1)
19:             end if
20:         end if
21:     end if
22:     for j = depth to N do
23:         Backup (j) ← depth − 1{Variabele aangepast (diepte naar omhoog)}
24:     end for
25: end for
```

Voorbeelden van redundante controles die met backmarking vermeden worden zijn op figuur 21 weergegeven in gestippelde rechthoeken. Deze controles kunnen dus, na éénmaal berekend, overgeslagen worden. Omdat zoals we merken de resultaten zich toch herhalen.

[4] variabelen beginnen vanaf 1: $z_1, z_2, \ldots$

Tabulation Een andere manier om redundante checks te vermijden is uiteraard een tabel op te bouwen, die iedere controle tussen iedere waarde van 2 verschillende variabele bijhoudt. dit principe wordt **Tabulation** genoemd. Dit kan in theorie een behoorlijke snelheidswinst opleveren. Anderzijds is de kans groot dat dit algoritme de tijd die het daar eventueel mee zou kunnen uitsparen nodig zal hebben om in de tabel te zoeken. Bovendien impliceert deze methode veel geheugengebruik.

4.5.4 Intelligent Backtracking

Een andere strategie die gebruikt kan worden is **Intelligent Backtracking**. Deze strategie vertaalt zich echter niet onmiddellijk in een algoritme, het is eerder een collectie algoritmen die meestal probleemafhankelijk zijn. Hierbij gaat men uit van het idee dat men tijdens de constructie van de boom op situaties stuit die sowieso niet tot de oplossing kunnen leiden. Deze situaties worden vervolgens opgeslagen als "**no-goods**". We kunnen een no-good dus beschouwen als een set waarden voor bepaalde variabelen (over het algemeen niet alle variabelen), waarvan we zeker zijn dat ze nooit tot een oplossing kunnen komen. Op het moment dat we tijdens het verdere verloop door de boom op een dergelijke situatie botsen, zullen we deze niet verder behandelen, maar eenvoudigweg een backjumping uitvoeren naar de laatste fout gekozen variabele. Het probleem is echter dat we geen generisch algoritme kunnen ontwikkelen die de rede tot falen bij een situatie kan achterhalen. De generatie van "no-goods" moet dus gedaan worden op basis van probleem-gebonden kennis.

4.5.5 Dynamic Search Rearrangement

Een andere strategie om heel wat nutteloos werk te vermijden is **Dynamic Search Rearrangement**. Hierbij maken we gebruik van een backtrack algoritme, maar staat de orde van de variabelen nog niet vast. Doel hiervan is om de vertakkingsfactor b te verkleinen met behulp van het **First-Fail Principle**. Dit principe zegt dat we beter constraints laten falen op het hogere dan op lagere niveaus. In het laatste geval dreigen er immers vele varianten te ontstaan die omwille van dezelfde reden toch allemaal op dezelfde constraint zullen falen. Al deze varianten controleren en er vervolgens backtracking op uitvoeren leidt alleen maar tot overhead. Meestal kijkt men dus naar de variabele met de meest **Non-triviale constraints** (constraints die niet altijd **true** teruggeven). Een andere parameter die helpt bij het kiezen, is de grote van het domein: variabelen met kleinere domeinen worden meestal beter eerst geëvalueerd. Nog een andere manier is een keuze maken aan de hand van een heuristiek. De meeste algoritmen zullen een combinatie van de drie vorige technieken hanteren om uiteindelijk een volgorde te bepalen.

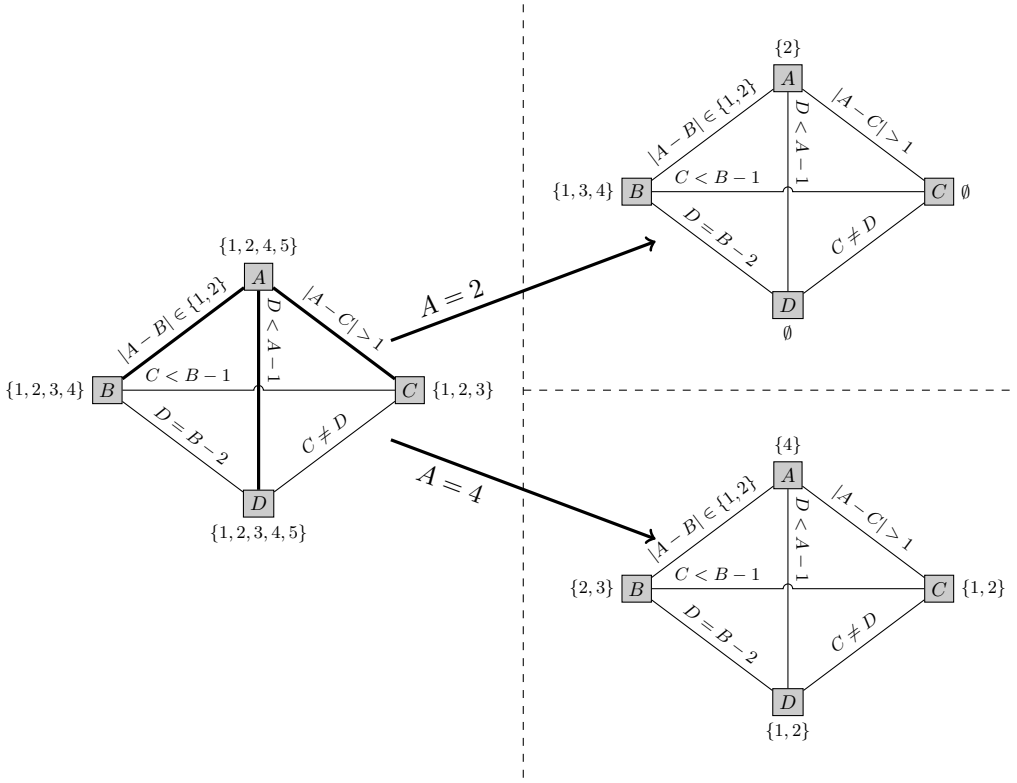
4.6 Relaxatie en Hybrid Constraint Processing

Indien we gebruik maken van Constraint-Networks hebben we een verzameling andere algoritmen nodig om een probleem op te lossen. Deze technieken zullen in de volgende subsubsecties verder toegelicht worden.

4.6.1 Weak Relaxation

Weak Relaxation is een concept waarbij we in staat zijn het domein van één of meerdere variabelen te verkleinen, maar waarbij in het algemeen het constraint netwerk na het toepassen van de algoritmen niet consistent is. Door deze technieken echter voldoende te herhalen door **Arc Consistency Technieken** kunnen we erin slagen een netwerk toch consistent te maken.

Forward Check **Forward Check** is een techniek waarbij we een variabele z_i een bepaalde waarde toekennen $z_i = a_{i\,j}$. Vervolgens controleren we alle constraints die gerelateerd zijn aan die variabele namelijk $c\,(z_i, z_k)$ en $c\,(z_k, z_i)$. We testen met andere woorden alle element in het domein van z_k en elimineren de waarden waarvoor de constraint niet meer geldt. Deze techniek veronderstelt echter wel dat we dus de waarde van z_i kennen. Een oplossing hiervoor is door middel van backtracking, alle mogelijk waarden voor z_i bekijken. Een ander nadeel aan deze techniek is dat niet alle relaxatie is gedaan (Kleinere domeinen kunnen een kettingreactie van relaxatie in beweging brengen, Forward Check gaat hier echter niet op in). Indien we een ongeldige waarde voor z_i kiezen, kan dit bovendien resulteren in het volledig leeghalen van een domein, en dus tot een inconsistente situatie leidt. In de meeste gevallen zullen niet



Figuur 23: Forward Check toegepast op het 4-Teachers probleem met $A = 2$ en $A = 4$.

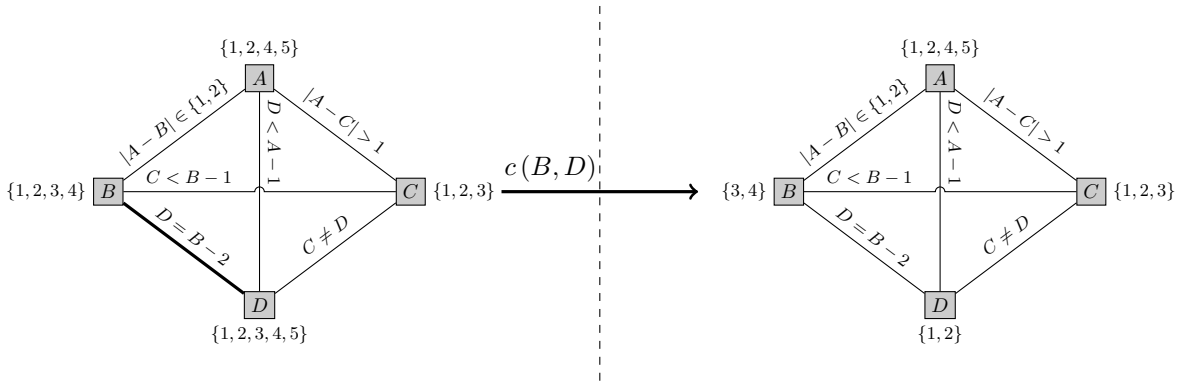
alle nieuwe domeinen van de variabelen slechts één element bevatten, en dus biedt deze techniek geen echte oplossing.

Op het leidend voorbeeld stellen we enerzijds $A = 2$ en anderzijds $A = 4$ zoals op figuur 23. Indien we $A = 2$ nemen bekomen we voor C en D lege verzamelingen, dit betekent dus dat we aan deze variabelen geen enkele waarde kunnen toekennen om aan de constraints te voldoen. We besluiten dus dat $A = 2$ onmogelijk waar kan zijn. Anderzijds indien $A = 4$ reduceren we de domeinen van de andere variabelen. Uit de OR-tree konden we al afleiden dat er geen oplossing bestaat voor $A = 4$. We zullen dus een andere variabele ook een waarde moeten toekennen om dit te besluiten.

Look Ahead Check Een andere techniek die de domeinen tracht te verkleinen is **Look Ahead Check**. Hierbij ligt de focus niet op een variabele, zoals bij Forward Check, maar bij de constraints zelf. We itereren over iedere constraint $c\,(z_i, z_j)$ en kennen z_i een bepaalde waarde toe. Indien voor om het even welke waarde van $z_j \in d_j$ de constraint blijft falen, is het duidelijk dat we de waarde van z_i uit zijn domein d_i mogen schrappen. Logischerwijs kunnen we dezelfde. Uiteraard werken we ook in de omgekeerde richting en zullen we zo ook het domein d_j van z_j zo kunnen verkleinen. Ook dit algoritme zal in het algemeen niet tot een consistente staat komen, de kettingreactie wordt immers opnieuw niet verwerkt. Een belangrijke eigenschap van dit algoritme is dat de volgorde waarin de constraints geëvalueerd worden, het resultaat kan beïnvloeden. Indien we bijvoorbeeld eerst het domein van een variabele verkleinen en daarna nog een andere constraint gerelateerd aan deze constraint onderzoeken, zullen we over het algemeen meer inconsistenties vinden. Formeel zeggen we dus dat voor een gegeven variabele z_i we $a_{i\,k}$ kunnen verwijderen uit het domein indien geldt:

$$\exists c(z_i, z_j) : \forall x \in d_j : \neg c\,(z_i = a_{i\,k}, z_j = x) \tag{18}$$

Of anders gezegd, er bestaat een constraint waarbij voor iedere waarde die de andere variabele aanneemt de constraint faalt.



Figuur 24: Look Ahead Check toegepast op het 4-Teachers probleem met $c(B, D)$.

We zullen dit concept illustreren door een Look Ahead Check uit te voeren op het 4-Teachers probleem. Als constraint nemen we $c(B, D)$. Zoals we op figuur 24 zien, worden zowel de domeinen van B als D gereduceerd.

4.6.2 Arc Consistency Technieken

Om de domeinen verder te verkleinen, en vervolgens tot een consistente staat te komen waarbij aan iedere constraint voldaan wordt, zijn een andere collectie algoritmen nodig, deze worden Arc Consistency Techniques of **2-consistency** genoemd. Het principe is gebaseerd op herhalen van Weak Relaxation technieken tot een consistente staat bekomen wordt. We bespreken hieronder twee technieken.

AC1 Een eenvoudig algoritme dat ontwikkeld werd door Mackworth[5] is **AC1** of **Arc Consistency 1**. Hierbij passen we het Look Ahead Algoritme toe tot geen enkel domein meer verkleind wordt. Het resultaat na het toepassen van dit algoritme, is een consistent netwerk van variabelen met een (verkleind) domein. Meestal bereiken we echter geen concrete oplossing. Bovendien is AC1 niet efficënt: Alle constraints worden gecontroleerd, ook degene die die niet gerelateerd zijn aan een variabele waar het domein verkleind is. In dat geval zal een extra evaluatie van de constraint, geen nieuwe inconsistenties opleveren, en dus nutteloos blijken.

AC3 Een oplossing voor dit efficiëntieprobleem werd geïmplementeerd met **Arc Consistency 3** of kortweg **AC3**. Hierbij worden alle constraints in een wachtrij geplaatst. Vervolgens worden constraints in de wachtrij één voor één geëvalueerd. Indien deze constraint inconsistente waarden uit één van de domeinen verwijdert, worden alle constraints die gerelateerd zijn aan de variabele met het gereduceerde domein in de wachtrij geplaatst. Het algoritme gaat door tot de wachtrij uiteindelijk leeg is. Dit is formeel beschreven in **Algorithm 25**. Dit algoritme geeft exact hetzelfde resultaat als AC1 maar in de meeste gevallen slechts de helft aan constraint-evaluaties. Opnieuw vinden we echter geen resultaat. Om dit probleem op te lossen, is een combinatie van backtracking algoritmen en relaxatie-algoritmen vereist. Deze worden nu behandeld in subsubsectie 4.6.3.

4.6.3 Hybrid Constraint Processing

Zoals reeds eerder vermeld zullen Arc Consistency technieken en Weak Relaxation algoritmen in het algemeen niet tot een concrete oplossing komen. Hiervoor is een combinatie van backtracking zoekmethoden en relaxatie nodig. Er zijn verschillende combinaties mogelijk, de meest relevante worden hieronder kort weergegeven.

[5] Alan K. Mackworth

Algorithm 25 Arc Consistency 3

```
1: Queue ← {c(z_i, z_j) | ∀i, j : i < j}
2: while notEmpty(Queue) do
3:     c(z_i, z_j) ← dequeue(Queue)
4:     d'_i ← domain(z_i)
5:     d'_j ← domain(z_j)
6:     removeInconsistentValuesOf(z_i, z_j, c(z_i, z_j))
7:     if domain(z_i) ≠ d'_i then
8:         enqueue(Queue, {c(z_k, z_i) | ∀k : k < i} ∪ {c(z_i, z_k) | ∀k : i < k})
9:     end if
10:    if domain(z_j) ≠ d'_j then
11:        enqueue(Queue, {c(z_j, z_k) | ∀k : j < k} ∪ {c(z_k, z_j) | ∀k : k < j})
12:    end if
13: end while
```

Forward Checking **Forward Checking** (Niet te verwarren met Forward Check) is, zoals de naam al doet vermoeden, een combinatie van backtracking en Forward Check. Hierbij geeft het Chronologisch Backtrackingsalgoritme aan een bepaalde variabele een waarde, waarna Forward Check uitgevoerd wordt. Dit reduceert vervolgens de domeinen van de andere variabelen. Vervolgens zullen we een niveau lager in de boom zoeken, en een andere variabele een waarde toewijzen krijgt, en de lus opnieuw begint. Indien we tot situaties komen waarbij we een domein volledig leeghalen, kunnen we hieruit geen oplossing meer genereren, bijgevolg voeren we een backtracking uit. Belangrijk hierbij is op te merken dat de domeinen bij backtracking terug hun oorspronkelijke waarde moeten bekomen. En men dus niet eenvoudigweg de domeinen telkens mag verkleinen, deze hangen immers af van de gekozen waarden. Eventueel kan van het Dynamic Search Rearrangement Concept gebruik gemaakt worden.

Lookahead Checking **Lookahead Checking** (Niet te verwarren met Look Ahead Check) is, zoals de naam al doet vermoeden, een combinatie van backtracking en Lookahead Check. De werking van dit algoritme is behoorlijk analoog aan dat van Forward Checking, met als enig verschil dat Lookahead Check uitgevoerd wordt en niet Forward Checking. Ook hier kan een Dynamic Search Rearrangement nog extra tijdswinst opleveren.

Welk algoritme is het beste? Zowel Forward Checking als Lookahead Checking slagen er in de meeste gevallen in, tot een resultaat te komen binnen een redelijke tijd. Toch ligt de nadruk bij de twee algoritmen op verschillende plaatsen. Forward Checking zal over het algemeen minder consistency checks doen, met het gevolg dat de boom meer vertakkingen hebben. Anderzijds zal Lookahead Checking veel tijd investeren in het zoeken van inconsistenties, maar wint deze terug door minder grote bomen te genereren. Algemeen echter is Forward Checking beter, Lookahead Checking controleert immers alle constraints, ook wanneer er geen variabele aangepast is, en er dus geen verkleining van het domein mogelijk is. Indien een probleem echter veel (niet-triviale) constraints bevat kan Lookahead Checking beter presteren.

4.7 k-consistency

Naast 1-consistency, en 2-consistency, kan men dit principe uiteraard veralgemenen naar k-**consistency**. In dat geval gelden er ook constraints, die uitspraken doen over de relatie van k variabelen ten opzicht van elkaar. In OR-bomen wordt een dergelijke constraint geëvalueerd vanaf het moment dat alle variabelen die gerelateerd zijn aan de constraint een waarde toegewezen krijgen. Meestal komt dit er dus op neer dat de constraint zich op het niveau bevindt van de variabele met het hoogste index-cijfer. Ook Weak-Relaxation technieken kunnen veralgemeend worden om met deze situaties om te gaan, in het algemeen zal dit echter resulteren in een groot verlies van performantie. Dit omwille van twee redenen: de problemen zijn eenvoudigweg complexer en er is tot nu toe minder onderzoek in geïnvesteerd.

4.8 Toepassingen en Alternatieven

Constraint processing heeft toepassingen in allerhande industriële takken van de samenleving gaande van het opstellen van uurroosters over het laden van trucks tot het programmeren van robotarmen. Een bekende taal om constraint systemen te beschrijven is Prolog[6].

We drukken het 4-Teachers probleem hieronder formeel uit in SWI-Prolog:

```
:- use_module(library(clpfd)).        %laad de constraint processing module in

fourTeachers([A,B,C,D]) :-
    Vars = [A,B,C,D],                 %variabelen initialiseren
    Vars ins 1..5,                    %domein specifieren
    A #\= 3,                          %c(A)
    B #\= 5,                          %c(B)
    C #=< 3,                          %c(C)
    abs(A-B) #>= 1, abs(A-B) #=< 2,   %c(A,B)
    abs(A-C) #> 1,                    %c(A,C)
    D #< A-1,                         %c(A,D)
    C #< B-1,                         %c(B,C)
    D #= B-2,                         %c(B,D)
    C #\= D,                          %c(C,D)
    label(Vars).                      %los het constraint probleem op.
```

We kunnen dan vervolgens een query aanroepen in SWI-Prolog die ons het resultaat van de puzzel teruggeeft:

```
?- fourTeachers(S).
S = [5, 4, 1, 2].
```

Indien we de laatste opdracht `label/1` weglaten dan wordt het constraint probleem niet opgelost. Dit vertelt ons echter iets meer over de werking van SWI-Prolog: Er wordt een Look Ahead Check uitgevoerd op het probleem, in de volgorde dat de constraints worden gedefinieerd. In dat geval wordt een reeks constraints als uitvoer geschreven die het probleem beschrijft inclusief de optimalisaties gegenereerd door Look Ahead Check:

```
?- fourTeachersNoLabel([A,B,C,D]).
A in 4..5,
_G3451+B#=A,
abs(A-C)#>=2,
abs(A-B)#>=1,
D in 1..2,
C#\=D,
D+2#=B,
C in 1..2,
C#=<B+ -2,
B in 3..4,
_G3451 in 0..2.
```

Alternatieven komen vooral uit de hoek van **Linear Programming**. Dit zijn een reeks numerieke en algebraïsche technieken die over het algemeen zeer efficiënt werken voor linear constraints (Constraints waar alleen vermenigvuldigingen met constanten en optellingen in voorkomen), een voorbeeld hiervan is het "Simplex Algoritme", indien we echter niet-lineare constraints beschouwen is Constraint Processing de aangewezen methode, verder kan Constraint Processing ook overweg met niet numerieke data (zoals bijvoorbeeld tekst, lijsten,...).

[6] Programmation en Logique

5 Game Playing

> *Het is geen schande gespeeld te hebben, maar wel om geen einde te maken aan het spel.*
>
> - Horatius, Romeins dichter (65 v.C. - 8 v.C.)

Een andere tak van de Artificiële intelligentie is **Game Playing**. Dit probleem is net als Constraint-Processing een speciaal geval van zoekproblemen die speciale eisen stelt. Game Playing stelt ons verder ook in staat Artificiële systemen te beoordelen. Dit komt omdat spelletjes zich meestal heel goed lenen tot State-Spaces: De regels zijn zeer duidelijk (ze worden zelf opgesteld, soms in tegenstelling tot de echte wereld) en er is makkelijk een representatie voor te bedenken.

Indeling van games We kunnen games indelen met behulp van twee parameters:

- Determinisme: is er een kans-element in het spel of niet
- Informatie Symmetrie: indien alle spelers even veel weten spreken we van "Perfecte Informatie" (voorbeeld: Schaken), Bij "Inperfecte Informatie" kennen de spelers slechts een deel van de informatie in het spel. (voorbeeld: Poker, Stratego).

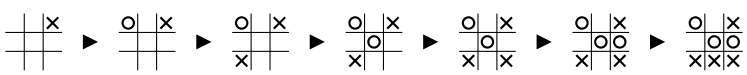
Waarom zijn nieuwe technieken vereist? Er zijn twee grote problemen waarom zoekalgoritmen meestal niet goed werken:

- Het **"Contingency" probleem**[7]: Verschillende spelers hebben tegenstrijdige belangen, de ene speler stelt een andere doel dan de andere speler, bovendien zal de tegenstander niet noodzakelijk de voor hem beste zet doen.
- De grote van het zoekprobleem: Spelletjes zijn er meestal op gericht er voor te zorgen dat zoeken onmogelijk is. Een voorbeeld is schaken waar de vertakkingsfactor gemiddeld $b = 15$ is en de diepte zo'n $d = 80$ zetten. Dit leidt dus tot $b^d = 15^{80} \approx 1.223 \times 10^{94}$ knopen, geen enkele hedendaagse machine is in staat een dergelijke boom binnen een redelijke tijd af te zoeken.

De oplossing bestaan er dan ook uit om bomen slechts tot een bepaalde diepte te onderzoeken, vanaf de huidige toestand. Door middel van een **evaluatie-functie** eval (S) wordt vervolgens de situatie beoordeeld. Dit oordeel wordt vervolgens naar boven gepropageerd in de boom, waardoor we bij iedere knoop op het eerste niveau een waarde bekomen. Afhankelijk van deze waarden kunnen we vervolgens een keuze maken.

5.1 Leidend Voorbeeld: Tic-Tac-Toe

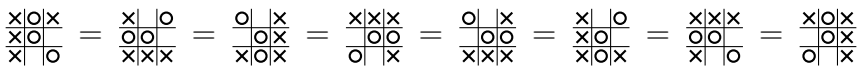
Als voorbeeld zullen we het Tic-Tac-Toe spel bespreken. Dit spel, ook wel "Boter, kaas en Eieren" genoemd wordt gespeeld op een 3×3 bord. De spelers dienen beurtelings respectievelijk een kruis of cirkel op het bord plaatsen. Bedoeling is drie van de eigen tekens op één lijn te plaatsen. Dit spel is eenvoudig voor pedagogische voorbeelden omdat de regels simpel zijn, en het spel beperkt is tot 765 mogelijke bordposities. Een mogelijk spelverloop beschrijven we of figuur 25.



Figuur 25: Een mogelijk spelscenario bij Tic-Tac-Toe

[7] Voor meer hierover: zie publicaties van John Nash over het "Nash-evenwicht"

We zullen in onze experimenten met dit voorbeeld geroteerde situaties als equivalent beschouwen. Borden die dus door enkele rotaties van 90 graden gelijk aan elkaar zijn beschouwen we als equivalent. Daarnaast worden ook spiegelingen over het middelste vak (horizontaal en verticaal op de tweede rij, kolom of diagonaal) op dezelfde manier behandeld. Dit is redelijk omdat de rotatie en diagonale spiegeling bij dit spel geen enkele rol speelt. Dit illustreren we in figuur 26.



Figuur 26: Equivalente situaties bij Tic-Tac-Toe

We hebben nood aan een evaluatie-functie om een bepaalde toestand te evalueren. De drijvende kracht achter Tic-Tac-Toe zijn de verschillende lijnen. We berekenen dus voor elke lijn een bepaalde score. De evaluatie-functie is vervolgens de som van de score van de 8 lijnen. Een lijn krijgt een score 0 indien deze zowel een kruis als cirkel bevat. Indien enkel één soort tekens aanwezig is, geven we de lijn een score afhankelijk van het aantal: $l(n) = 10^{n-1}$. We nemen X als het positieve teken, en O als het negatieve teken.

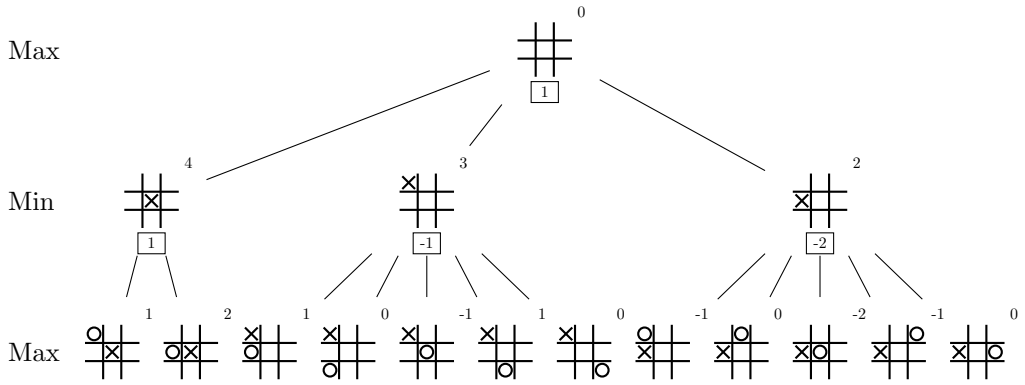
5.2 Mini-Max

De propagatie naar boven in de boom verloopt volgens het **Mini-Max**-principe. Hierbij gaan we ervan uit dat de computer de situatie beoordeelt, volgens hoe goed hij er zelf voor staat. De computer streeft met andere woorden naar een zo maximaal mogelijke evaluatie. De tegenstander anderzijds streeft naar een zo laag mogelijke evaluatie. Dit resulteert in het feit dat bij een knoop waar de computer een keuze dient te maken, hij de maximale waarde verder naar boven zal propageren. De tegenstander daarentegen zal de minimale evaluatie van zijn kinderen overnemen. Dit idee wordt formeel weergegeven in **Algorithm 26**, waarbij we de methode initieel aanroepen met depth = 0 and board met de huidige staat van het bord.

Algorithm 26 minimax (board, depth)

```
1:  if depth < depthbound then
2:      {Einde nog niet bereikt}
3:      if isMaxLevel (depth) then
4:          M ← −∞
5:          for all c ∈ children (board) do
6:              M=max (M, minimax (c, depth + 1))
7:          end for
8:          return M
9:      else
10:         m ← ∞
11:         for all c ∈ children (board) do
12:             m=min (m, minimax (c, depth + 1))
13:         end for
14:         return m
15:     end if
16: else
17:     {Eind-diepte bereikt}
18:     return eval (board)
19: end if
```

We kunnen nu een boom opstellen die het Mini-Max principe illustreert bij Tic-Tac-Toe voor een diepte $d = 2$. Waarbij we ook de propagatie naar boven toe tonen. Rechtsboven ieder bord tonen we de evaluatiefunctie toegepast op dit bord. Onder ieder bord, staat de gepropageerde waarde. Volgens deze boom is het plaatsen van een X in het midden de beste zet.



Figuur 27: Mini-Max boom van Tic-Tac-Toe.

5.3 Alpha-Beta cut-offs

Minimax algoritmen hebben het nadeel de ze eerst de hele boom opbouwen alvorens ze propagatie toepassen. Hierdoor worden vaak heel wat knopen geëvalueerd waarvan we al kunnen weten dat ze niet meer instressant zijn. **Alpha-Beta Cut-offs** proberen een oplossing te bieden voor deze overhead. Stel dat we reeds weten wat de waarde is van één van de kinderen van een maximum-knoop, dan weten we per definitie dat deze knoop een waarde zal teruggeven die groter of gelijk is aan die waarde. Stel nu dat een ander kind, een **minimum-knoop**, op dat moment geëvalueerd wordt, en we weten de waarde van ten minste één kind. We weten zeker dat de waarde die de minimum-knoop zal doorgeven kleiner of gelijk is aan die waarde. Als de bovengrens van de minimumwaarde echter kleiner is dan de ondergrens van de **maximum-knoop** heeft verdere evaluatie geen zin. Wat de waarde van de minimum-knoop ook mag zijn. De maximumknoop zal een getal retourneren die groter is dan zijn ondergrens.

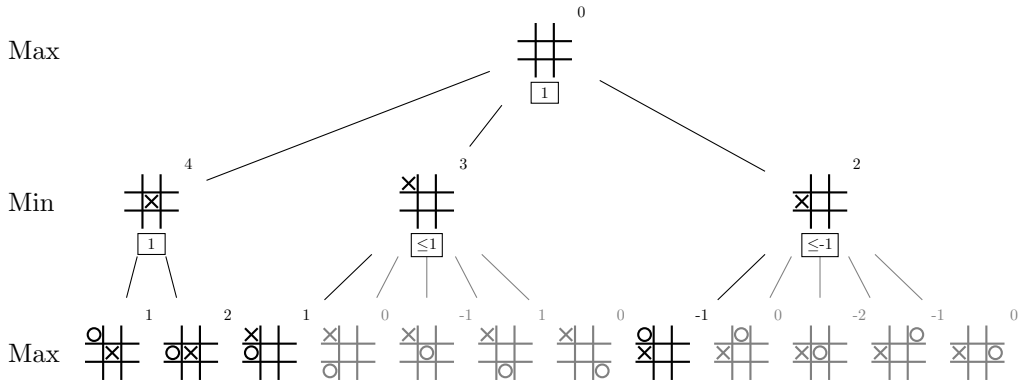
Alpha-Beta Cut-offs lossen het probleem als volgt op, er wordt een diepte-eerst benadering gebruikt. Hierbij wordt telkens wanneer een kind geëvalueerd is, de boven- (**Beta-waarde** genoemd) of ondergrens (**Alpha-waarde** genoemd) van de respectievelijk minimum- of maximum-knoop van de ouder aangepast. Indien een kind een Alpha-waarde heeft die groter of gelijk is aan de Beta-waarde van een ondergeschikte knoop, heeft verdere evaluatie geen zin. In dat geval spreken we van een **Alpha-cut** of **Alpha-snede**. Analoog indien de Beta-waarde kleiner of gelijk is aan de Alpha-waarde van een ondergeschikte knoop, zullen we deze niet verder evalueren. Dit is dan een **Beta-cut** of **Beta-snede**.

Zoals de vorige paragraaf reeds doet blijken, gaat dit principe verder dan alleen de rechtstreekse ouder. Ook Alpha-waardes die verschillende niveaus hoger opgeslagen zitten, hebben een invloed op Beta-waardes dieper in de boom, en kunnen zo nog steeds Alpha-snedes veroorzaken (dit principe geldt natuurlijk ook omgekeerd). Dit fenomeen wordt **Deep cut-offs** genoemd.

In het beste geval, wanneer de beste keuze voor zowel computer als tegenstander zicht telkens in het eerste kind bevindt, kunnen we telkens met een behoorlijke snelheid alle andere kinderen elimineren. In dat geval zakt het aantal evaluaties van $\mathcal{O}\left(b^d\right)$ naar $\mathcal{O}\left(b^{d/2}\right)$. Wat neerkomt op een worteltrekking van de eerste evaluaties! Meestal loopt het echter niet zo'n vaart maar is de winst toch zeker opmerkelijk.

Zelfs in de spelboom op figuur 27 is er sprake van beta snedes. Zoals weergegeven op figuur 28 dienen we slechts 4 van de 12 borden op het laagste niveau te evalueren. Indien we immers het eerste bord geëvalueerd hebben met de X geplaatst in de hoek, weten we dat de gepropageerde minimax waarde hoogstens 1 blijft. Dit is bijgevolg kleiner of gelijk aan de reeds berekende minimax waarde bij een X in het midden. Omdat we weten dat de waarde alleen maar meer in ons nadeel kan evalueren, hoeven we deze tak niet verder te evalueren. Dit principe werkt ook analoog op de tak waarbij we X aan de rand plaatsen. Uiteraard kan geargumenteerd worden dat dit voorbeeld geen spectaculaire versnellingen teweeg brengt. Als we echter in het achterhoofd houden dat de gemiddelde spelboom

veel groter is, en de evaluatiefunctie soms erg complex en moeilijk te berekenen is, levert dit principe makkelijk een significante bijdrage aan de performantie.



Figuur 28: Een voorbeeld van beta-cuts bij Tic-Tac-Toe.

5.4 Het Horizon-effect

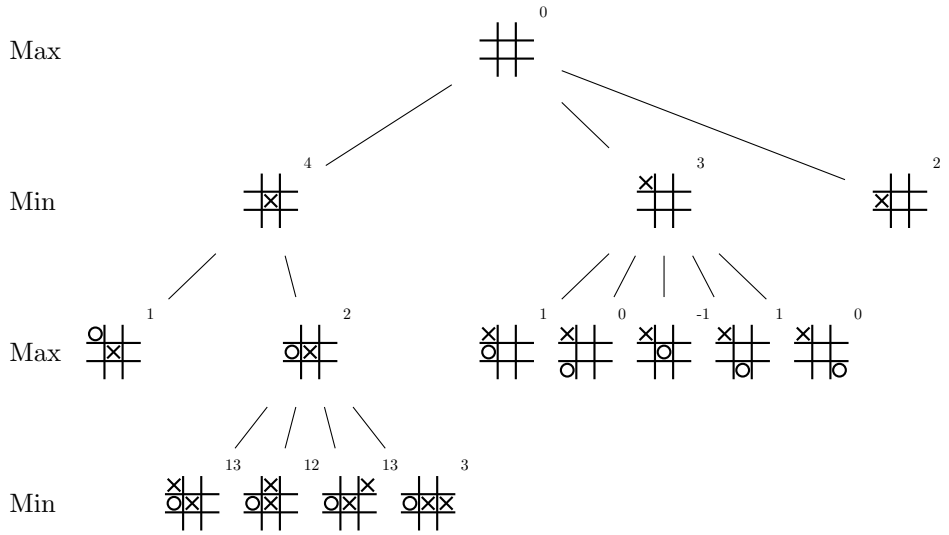
Een groot nadeel aan het opbouwen van een boom die op een bepaalde diepte stopt is het **Horizon-effect**. Dit effect stelt dat we volledig blind zijn voor wat onder de diepte-grens gebeurt. Vooral bij strategische spellen kunnen dergelijke effect nefast zijn: posities die er de eerste 4 zetten goed uitzien kunnen uiteindelijk tot een catastrofe leiden. Een oplossing bied het **Heuristic Continuation**-principe. Dit principe stelt dat we bepaalde paden die er tot aan de horizon goed uitzien, beter verder onderzoeken. Dit privellege wordt alleen verleent aan paden die ofwel aan de horizon tot één van de beste behoren, of waarbij een heuristiek alarmeert voor een strategisch belangrijke fase (bijvoorbeeld onmiddellijk stukkenverlies, koning in gevaar,...)

Hoe wordt een dergelijke Heuristic Continuation dan georganiseerd? Een oplossing hiervoor is **Tapering Search**. Hierbij evalueren we een knoop, met een zekere interne diepte-limiet. Vervolgens generen we de kinderen van deze knoop, en sorteren ze op hun evaluatie. Het beste kind erft vervolgens de diepte limiet over van zijn ouder vermindert met 1. Het tweede beste kind met een vermindering met twee, enzovoort tot de dieptelimiet 0 bedraagt. Deze methode roepen we dan ook aan startend vanuit de huidige toestand met over het algemeen een grote diepte limiet. Dit wordt formeel beschreven in **Algorithm 27**.

Algorithm 27 taperingSearch (state, depth, depthbound)

```
1: if depth < depthbound then
2:     {Einde nog niet bereikt}
3:     C ← children (state)
4:     quickSort (C, eval (C_i))
5:     for i = 0 to min (#C, depthbound) − 1 do
6:         taperingSearch (R [i] , depth + 1, depthbound − 1 − i)
7:     end for
8: end if
```

Bij wijze van voorbeeld passen we Tapering Search toe bij Tic-Tac-Toe met een begindiepte van $d = 3$. Zoals te zien op figuur 29. We expanderen eerst de eerste knoop, hierbij zien we dat bij X in het midden we de beste evaluatiefunctie bekomen. Deze knoop zullen we dus verder expanderen met een diepte $d' = 2$. Bij X in een hoek bekomen we ook een goede heuristiek. Deze knoop expanderen we met $d' = 1$. Tenslotte expanderen we het bord met X aan een rand niet meer verder. Op analoge manier expanderen we vervolgens de onderliggende knopen.



Figuur 29: Tapering Search met $d = 3$ bij Tic-Tac-Toe.

5.5 Tijdslimieten

Zoeken met een diepte-limiet, is niet altijd de beste methode voor tijdsgebonden spelletjes. Dergelijke zoekmethodes kunnen immers erg variëren in tijdsgebruik, men loopt dus de kans dat het algoritme nog geen oplossing heeft wanneer de tijd op is. Een oplossing biedt het Iterative Deepening concept. Hierbij wordt de dieptegrens telkens met een factor verhoogt. Indien we initieel een lage diepte-grens hanteren, hebben we meestal een eerste oplossing in een kwestie van milliseconden. Vervolgens verdiepen we, indien we hierbij tot een resultaat komen, zullen we dit accepteren als nieuwe oplossing. Indien de tijd verstreken is, retourneren we de oplossing van onze laatste diepte. Hierdoor hebben we altijd een oplossing klaar indien de tijd opraakt.

5.6 Kansspelen

Bij kansspelen is het verloop van het spel nog moeilijker te bepalen. In dat geval kunnen we de dobbelsteen of een andere toevalsgenerator, beschouwen als een derde speler. Deze speler krijgt vervolgens een aparte soort knopen toegewezen: **expectimin-knopen** en **expectimax-knopen**, afhankelijk van de plaats waar de dobbelsteen zich tussen de spelers bevindt (expectimax onder een maximum-knoop en expectimin-onder een minimum-knoop). Vervolgens zal deze knoop het gewogen gemiddelde van respectievelijk het minimum of het maximum van de knopen genereren, of meer formeel met d_i de waarde van de toevalsgenerator:

$$\begin{aligned} \text{expectimin}\,(C) &= \sum_i P\,(d_i) \cdot \min\,(\text{eval}\,(s) |\, \forall s \in \text{children}\,(d_i)) \\ \text{expectimax}\,(C) &= \sum_i P\,(d_i) \cdot \max\,(\text{eval}\,(s) |\, \forall s \in \text{children}\,(d_i)) \end{aligned} \tag{19}$$

6 Automatische Redeneersystemen

> *Logica is de architectuur van de menselijke rede.*
>
> - Evelyn Waugh, Brits schrijfster (1903-1966)

Naast zoekproblemen en het oplossen van constraint netwerken, blijkt de Artificiële intelligentie ook een uiterst geschikt domein te zijn voor logica. Al in de jaren '60 had men het droombeeld van de computer als stellingbewijzer. Met logica kan immers alles formeel uitgedrukt worden. Bijgevolg, zo redeneerde men, kan men een machine bouwen die door het toepassen van simpele regels, in staat moet zijn om met een stelsel uitspraken, een afgeleide uitspraak te kunnen bewijzen. Deze hypothese is werd gestaafd met volgende argumenten:

- Logica is de assembleertaal van de kennis en ligt bovendien heel dicht bij de natuurlijke taal: bijgevolg kan men bijna elke vorm van kennis omzetten in formele ondubbelzinnige logica.
- Als computers kennis moeten verwerken, dan moet die kennis formeel en ondubbelzinnig zijn.
- Door middel van **Logische Deductie** moeten we op een systematische manier nieuwe kennis uit bestaande kennis kunnen synthetiseren.

De vraag is dus of we deze deductie kunnen automatiseren.

6.1 Leidend voorbeeld: Marcus Brutus

Stel we beschikken over volgende bestaande kennis:

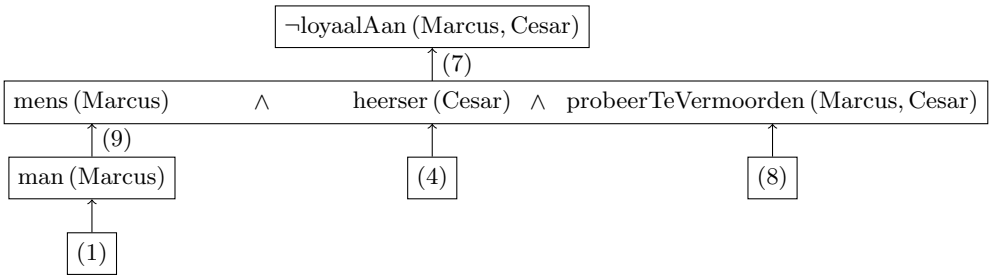
1. Marcus was een man
2. Marcus was een Pompeiër
3. Alle Pompeiërs waren Romeinen
4. Cesar was een Heerser
5. Alle Romeinen waren ofwel loyaal aan Cesar of probeerden hem te vermoorden
6. Iedereen is loyaal aan iemand
7. Mensen proberen alleen heersers te vermoorden waar ze niet loyaal aan zijn
8. Marcus probeerde Cesar te vermoorden
9. Iedere man is een mens.

We willen een antwoord vinden op de vraag "Was Marcus Loyaal aan Cesar?". Indien we dit zelf uitwerken zullen we meestal eerst alle uitspraken omzetten naar **Eerste Orde logica**:

1. $\text{man}(\text{Marcus})$
2. $\text{pompeiër}(\text{Marcus})$
3. $\forall x : \text{pompeiër}(x) \rightarrow \text{romein}(x)$
4. $\text{heerser}(\text{Cesar})$
5. $\forall x : \text{romein}(x) \rightarrow \left(\begin{array}{c} \text{loyaalAan}(x, \text{Cesar}) \wedge \neg\text{probeerTeVermoorden}(x, \text{Cesar}) \\ \vee \\ \neg\text{loyaalAan}(x, \text{Cesar}) \wedge \text{probeerTeVermoorden}(x, \text{Cesar}) \end{array} \right)$
6. $\forall x, \exists y : \text{loyaalAan}(x, y)$

7. $\forall x, y : \text{mens}(x) \wedge \text{heerser}(y) \wedge \text{probeerTeVermoorden}(x, y) \rightarrow \neg\text{loyaalAan}(x, y)$
8. $\text{probeerTeVermoorden}(\text{Marcus}, \text{Cesar})$
9. $\forall x : \text{man}(x) \rightarrow \text{mens}(x)$

We proberen dan te bewijzen dat Marcus loyaal was aan Cesar, of formeler $\text{loyaalAan}(\text{Marcus}, \text{Cesar})$. Anderzijds kunnen we ook door middel van **Achterwaarts redeneren** proberen te bewijzen dat Marcus niet loyaal is aan Cesar en tot een inconsistentie trachtten te komen. Indien we $\neg\text{loyaalAan}(\text{Marcus}, \text{Cesar})$ als beginexpressie kiezen, kunnen we een substitutie uitvoeren met behulp van 7. Hierbij vervangen we x door Marcus en y door Cesar. Vervolgens bekomen we $\text{mens}(\text{Marcus}) \wedge \text{heerser}(\text{Cesar}) \wedge \text{probeerTeVermoorden}(\text{Marcus}, \text{Cesar}) \rightarrow \neg\text{loyaalAan}(\text{Marcus}, \text{Cesar})$. Indien we deze drie componenten kunnen aantonen kunnen we bewijzen, hebben we meteen bewezen dat Marcus niet loyaal is aan Cesar. Aangezien 9 geldt en we weten door 1 dat Marcus een man is, kunnen we het eerste statement vervangen door $\text{man}(\text{Marcus})$. Aangezien dit laatste een feit is (door 1), en de twee andere aspecten in de vergelijking ook feiten zijn (door 4 en 8). Kunnen we vervolgens bewijzen dat Marcus niet loyaal was aan Cesar. Schematisch wordt dit bewijs ook getoond op figuur 30.



Figuur 30: Bewijs voor disloyaliteit van Marcus jegens Cesar

Bovenstaand voorbeeld legt echter onmiddellijk een paar problemen bloot:

1. Kennisrepresentatie:
   (a) Imprecies taalgebruik: niet alle taal is machinaal onmiddellijk te vervangen door logica (bijvoorbeeld 7)
   (b) Evidente informatie: vooral 9 wordt bijvoorbeeld makkelijk vergeten omdat deze vanzelfsprekend is
   (c) Niet alle informatie is makkelijk voor te stellen met logica (zoals "bijna","meestal",...)
   (d) Logica is niet handig vanuit "Sofware-engineering" perspectief (lijkt te veel op assembler)
2. Problem solving:
   (a) Een groot aantal tradeoffs die ook bij zoekmethodes voorkwamen (zie 1.3)
   (b) Welke deductieregels hebben we nodig (Modus Ponens, Modus Tollens,...)
   (c) Hoe behandelen we $\forall x$ en $\exists y$
   (d) Hoe berekenen we substituties (in het voorbeeld eenvoudig, in de praktijk soms oneindig veel mogelijkheden)
   (e) Wat bewijzen we? "Het te bewijzen" of het omgekeerde?
   (f) Hoe behandelen we gelijkheid van statements (twee verschillende expressies kunnen soms wel equivalent zijn, door symmetrie, reflexiviteit, transitiviteit)
   (g) Hoe garanderen we de volledigheid en de correctheid?

6.2 De formele model semantiek van Logica

Alvorens al die problemen op te lossen is het zinvol eerst opnieuw stil te staan bij het wiskundige concept van logica. Deze subsectie mag dan ook overgeslagen worden indien de propositie- en predicatenlogica reeds in andere cursussen behandeld werd.

6.2.1 Propositielogica

In propositielogica zijn we in staat simpele expressies voor te stellen door middel van een beperkt **alfabet** (reeks toegelaten karakters). Dit alfabet kunnen we onderverdelen in drie subsets:

- **Atomaire Proposities**: Dit zijn expressies waar we een bepaalde eigenschap aan geven. Hun namen worden door de gebruiker zelf gekozen (bijvoorbeeld $\{\text{weer\_is\_regenachtig}, \text{Jos\_draagt\_paraplu}\}$)
- **Connectieven**: Om atomaire proposities met elkaar te combineren gebruiken we connectieven. Deze set staat vast en bevat volgende elementen: $\{\wedge, \vee, \neg, \rightarrow, \leftarrow, \leftrightarrow\}$
- **Punctuaties**: Bij connectieven gelden bepaalde voorrangsregels, om deze aan te passen beschikken we over punctuaties of haakjes, deze set staat vast en bevat volgende elementenen: $\{(,)\}$

Een **Goed Gevormde Formule** in de propositielogica is een expressie die enkel gebruik maakt van het alfabet van de propositielogica, en daarbij de regels respecteert van de connectieven en de punctuaties.

Hieronder worden enkele concrete schoolvoorbeelden van propositielogica weergegeven:

$$\begin{cases} \text{afstandsbediening\_is\_kapot} \vee \neg\text{tv\_werkt} \\ \text{schilderij\_is\_gestolen} \rightarrow \neg\text{schilderij\_hangt\_hier} \\ \text{Gabriela\_tennist} \wedge \text{Judith\_schaakt} \\ \text{er\_loopt\_stroom} \rightarrow \text{draad\_wordt\_warm} \\ \text{Jos\_draagt\_paraplu} \leftarrow \text{weer\_is\_regenachtig} \end{cases} \tag{20}$$

Semantiek We kunnen de betekenis of **semantiek** van een expressie vastleggen op twee manieren: Door middel van natuurlijke taal, of door middel van **transformationele semantiek**. In dat laatste geval beschrijven we de statements door een daar uit afgeleid mathematisch object. Zo kunnen we bijvoorbeeld uit een reeks statements alle atomaire proposities die waar zijn doormiddel van **logisch gevolg** weergeven. Indien we vervolgens een expressie controleren kunnen we deze opbouwen vanuit de waarheidstabel.

Logisch Gevolg Maar hoe definiëren we dan logisch gevolg? intuïtief denken we aan alles wat we logisch gezien willen vastleggen maar dit is niet het geval. Op dit ogenblik hebben we immers nog geen verzameling afleidingsregels. Dat is immers wat we willen vastleggen! Een logisch gevolg wordt bereikt door **Interpretatie** en een **Model**:

Interpretatie: Is een functie die aan iedere atomaire formule een bepaalde waarheidswaarde toekent, en vervolgens alle statements coontroleert. We genereren dus een **waarheidstabel**.

Model: Een model toetst vervolgens deze statements op hun waarheid. Configuraties waarbij geen statements falen worden hierbij als een logisch gevolg gezien. Een model M is dus een subset van een interpretatie I indien alle expressies waarop de expressie gebaseerd zijn, waar zijn.

Of meer formeel: Gegeven een stel formules (statements) S en een formule F. F is een logisch gevolg van S ($S \vDash F$) indien elk model van S ook F waarmaakt.

6.2.2 Predicatenlogica

Hoewel propositielogica reeds heel wat problemen kan oplossen, kunnen we het leidende voorbeeld er niet in uitdrukken. Hiervoor hebben we een algemenere taal nodig, de **predicatenlogica**. Het alfabet van de predicatenlogica bestaat uit de volgende elementen:

- **Variabelen** (zoals x en y): deze vervullen een tijdelijke rol met behulp van kwantoren

- **Constanten** (zoals Cesar en Marcus): hun rol is globaal en dus gedefinieerd over alle statements
- **Functie-symbolen** (zoals vader, hart,...): om bepaalde relaties ten opzichte van variabelen en constanten te definiëren
- **Predicaatsymbolen** (zoals mens, loyaalAan,...): gebruikers gedefinieerde symbolen om een algoritme te kunnen laten werken, deze kennen bovendien een waarheidswaarde toe.
- Connectieven ($\wedge$, $\neg$,...): om meer geavanceerde uitspraken te doen (net als bij propositielogica)
- Punctuaties ((en)): het geven van prioriteit aan een bepaald deel van de expressie
- **Quantoren** ($\forall$ en $\exists$): het geven van een invulling aan de variabelen

Verder introduceren we nog een nieuw concept in de predicatenlogica: Een **term**. Een term is een variabele, een constante of een functiesymbool voorzien van met termen (wat cascadering van functies mogelijk maakt) zoveel als het verwachtte aantal voor de functie. Concrete voorbeelden zijn moordenaar (Cesar), dochter (x, Cesar) en hart (vader (Marcus)). Een goed-gevormde formule in predicatenlogica wordt geconstrueerd uit predicaat-symbolen voorzien met termen als argumenten en uit connectoren, quantoren en punctuaties, en dit volgens de regels die connectoren opleggen.

Een Goed Gevormde Formule in predicatenlogica is een expressie die zich tot het alfabet van de predicatenlogica beperkt, en daarnaast de regels voor connectieven en punctiaties respecteren, verder bevatten ze geen variabelen die niet door een quantor gedefinieerd zijn (in de expressie zelf). Hoe moeten we nu deze formules interpreteren? We kunnen een expressie in predicatenlogica zien als een combinatie van enkele factoren:

- Een verzameling D die we het domein noemen waarin alle constanten zitten.
- De Functie-symbolen als wiskundige functies die van $D \to D$ gedefinieerd zijn
- De predicaatsymbolen als wiskundige functies die van $D \to \mathbb{B}$ gedefinieerd zijn ($\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$ is de verzameling booleans)
- De quantoren op variabelen lezen we als volgt:
  - $\forall x\ f(x)$ is waar indien $\forall d \in D : I(f(d)) = \mathbf{true}$ (hierbij is I de identiteitsfunctie die **true** op **true** en **false** op **false** afbeeldt)
  - $\exists x\ f(x)$ is waar indien $\exists d \in D : I(f(d)) = \mathbf{true}$

6.3 Automatisch Redeneer-systemen voor eerste-orde predicatenlogica

In deze subsectie zullen we een automatisch rendeneer-systeem voor eerste-orde predicaten logica ontwerpen. Hierbij bezitten we een kennisbank geschreven in predicatenlogica die we T noemen. Dit is een verzameling van eerste-orde formules en wordt formeel ook wel eens de "**Theorie**" genoemd. Verder beschikken we over een bijkomende eerste-orde predicatenlogica-formule F. De vraag die beantwoordt dient te worden is of T, F impliceert, of formeler: $T \vDash F$. We zoeken dus een afleidingstechniek die dit kan uitmaken voor iedere F en T. Daarnaast zijn ook correctheid, volledigheid en efficientie vereiste criteria.

6.3.1 Beslisbaarheid

Is iedere formule te beslissen? Volgens de **stelling van Church**, bestaat er geen enkel algoritme die in staat is voor iedere T en F uit te maken of $T \vDash F$. Dit ondermijnt dus onmiddellijk ons eerder gestelde doel: Niet elk probleem is beslisbaar. Anderzijds zegt de **Volledigheidsstelling van Gödel** dat er een redeneertechniek bestaat waarbij voor iedere T en F waarbij $T \vDash F$, deze techniek dit kan bewijzen. Dit wordt **semi-beslisbaarheid** genoemd. Dus indien F uit T volgt, is dit bewijsbaar. Indien we noch $T \vDash F$ noch $T \vDash \neg F$ kunnen bewijzen, kunnen we bijgevolg niets zeggen over F.

6.3.2 Achterwaartse Resolutie

In het algemeen wordt voor automatisch redeneren in de eerste orde slechts één aanpak gehanteerd: **Achterwaartse resolutie**. Deze techniek bevat verschillende technische componenten, die we stap voor stap zullen uitwerken, en waarbij we de predicatentaal zullen uitbreiden van **Gegronde Horn Clause Logica** over **Horn Clause logica** en **Clausale logica** tot uiteindelijk de volledige predicatenlogica. Voor elk van deze talen zullen we een semi-beslisbare procedure bouwen, die we verder zullen moeten verfijnen indien we de taal uitbreiden. De achterwaartse resolutie is gebaseerd op de volgende principes:

- **Clausale vorm** (in 6.3.5 op pagina 64)
- **Bewijs door inconsistentie** (in 6.3.6 op pagina 65)
- **Unificatie** (in 6.3.8 op pagina 67)
- De **Resolutie stap** (in 6.3.9 op pagina 70)
- De **Resolutie bewijzen** (in 6.3.10 op pagina 72)
- **Normalisatie** (in 6.3.11 op pagina 72)

Nadat we eerst de verschillende beperkte predicaattalen uitleggen in subsubsecties 6.3.3, 6.3.4 en 6.3.5, worden deze principes in de verdere subsubsecties verder uitgelegd. Na een korte introductie zal vervolgens dieper ingegaan op de materie. Eventueel kan de lezer bij ieder item eerst de eerste paragraaf lezen, en vervolgens zijn kennis uitbreiden met de overige paragrafen.

6.3.3 Gegronde Horn Clause Logica

Een formaat die ons toelaat een eerste de basis van redeneren te definiëren is de Gegronde Horn Clausale vorm. Hierbij worden geen variabelen toegelaten. Alleen predicaatsymbolen met constanten en functies worden toegelaten. Een typische Gegronde Horn Clausale expressie ziet er dus als volgt uit:

$$A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n \tag{21}$$

Hierbij zijn $A, B_1, B_2, \ldots, B_n$ **atomen**. Een atoom is een formule van de vorm $p\left(t_1, t_2 \ldots, t_m\right)$ waarbij p een predicaatsymbool is, en $t_1, t_2 \ldots, t_m$ termen zijn (termen laten uiteraard ook variabelen toe, in de gegronde Horn Clause Logica staan er echter geen quantoren, bijgevolg is het gebruik van variabelen niet mogelijk). Het atoom in het linkerdeel A noemen we ook het **hoofd** van de gegronde Horn Clause, de atomen aan de rechterkant $B_1, B_2, \ldots, B_n$ worden ook wel de **body-atomen** genoemd. Tussen body-atomen zijn alleen conjuncties ($\wedge$) toegelaten, negaties ($\neg$) en disjunties ($\vee$) zijn verboden. Indien $n = 0$ (er dus niets aan de rechterkant staat), spreken we over een **feit**, en is A per definitie waar (mits geen inconsistentie). Uiteraard beperkt het verbod op variabelen de expressieve kracht dit formaat, bijgevolg is niet iedere expressie uit te drukken in gegronde Horn Clause Logica.

Volgende expressies staan in Gegronde Horn Clause Logica:

$$\begin{cases} \text{man}\left(\text{Marcus}\right) \leftarrow \\ \text{probeerTeVermoorden}\left(\text{Marcus}, \text{Cesar}\right) \leftarrow \\ \text{loyaalAan}\left(\text{vader}\left(\text{Marcus}\right), \text{Cesar}\right) \leftarrow \text{loyaalAan}\left(\text{Marcus}, \text{Cesar}\right) \\ \text{romeinsHeerser}\left(\text{Cesar}\right) \leftarrow \text{romein}\left(\text{Cesar}\right) \wedge \text{heerser}\left(\text{Cesar}\right) \end{cases} \tag{22}$$

6.3.4 Horn Clause Logica

Indien we de gegrondheid laten vallen, bekomen we de Horn Clause Logica. Voortaan zijn variabelen wel toegestaan, zolang deze echter universieel gekwantificeerd (alleen $\forall x$ is toegelaten, $\exists y$ is verboden) zijn. Bovendien worden alle variabelen vooraan in de expressie gedefinieerd (twee variabelen met dezelfde naam zijn bijgevolg niet mogelijk). Hierdoor kunnen we in principe de $\forall x$-definities weglaten (Daar

slechts één quantor toegelaten, kunnen we deze impliciet schrijven voor iedere variabele in de expressie). We beschrijven dan ook het formaat als volgt:

$$\forall x_1, \forall x_2 \ldots, \forall x_k \ A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n \tag{23}$$

Naast de hierboven beschreven **Implicatieve vorm** van de Horn Clause logica, Bestaat er ook nog een **Conjunctieve vorm** die er als volgt uitziet:

$$\forall x_1, \forall x_2 \ldots, \forall x_k \ A \vee \neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_n \tag{24}$$

Representatiekracht De representatiekracht van de Horn Clauses is echter nog steeds beperkt. Hoewel veel expressies ongevormd kunnen worden naar Horn Clauses, zijn sommige expressies niet weer te geven in Horn Clausale Logica.

Volgende expressies zijn voorbeelden van Horn Clauses:

$$\begin{cases} \forall x : \text{romein}(x) \leftarrow \text{pompeiër}(x) \\ \forall x, \forall y : \text{moeder}(x, y) \leftarrow \text{ouder}(x, y) \wedge \text{vrouw}(y) \\ \forall x : \text{probeerTeVermoorden}(x, \text{Cesar}) \vee \text{probeerTeVermoorden}(\text{Cesar}, x) \end{cases} \tag{25}$$

De taal is zoals eerder vermeld niet krachtig genoeg om de hele predicatenlogica te omvatten. Zoals bij volgende expressies:

$$\begin{cases} \forall x \ \text{mens}(x) \rightarrow \text{man}(x) \vee \text{vrouw}(x) \\ \forall x \ \text{hond}(x) \wedge \neg\text{abnormaal}(x) \rightarrow \text{heeftVierPoten}(x) \end{cases} \tag{26}$$

Voor deze expressies bestaat geen equivalent in de Horn Clausale Logica.

6.3.5 Clausale Logica

Een nog breder formaat dan de Horn Clause Logica is de Clausale Logica. Dit formaat is in staat ieder predicaat weer te geven, en heeft bijgevolg dezelfde representatiekracht (mits een conversiealgoritme) als de predicatenlogica. Hierbij zijn we in staat het hoofd van de expressie uit te breiden tot een reeks disjuncte ($\vee$) atomen. Zoals onderstaande vorm weergeeft:

$$\forall x_1, \forall x_2 \ldots, \forall x_k \ A_1 \vee A_2 \vee \ldots \vee A_m \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n \tag{27}$$

Hierbij zijn alleen disjuncties ($\vee$) in het hoofd, en conjuncties ($\wedge$) in het lichaam toegelaten. Bovendien blijven alle variabelen aan het begin van de expressie universieel gequantificeerd. We kunnen dus Horn Clausale Logica zien als een speciaal geval met $m = 1$.

Hieronder staan enkele voorbeelden van expressies en hun Clausaal equivalent:

$$\begin{cases} \neg\exists u \ r(f(u)) \\ \forall z \ \neg q(z) \\ \forall y \ p(f(y)) \\ \forall x \ \text{mens}(x) \rightarrow \text{man}(x) \vee \text{vrouw}(x) \end{cases} \Leftrightarrow \begin{cases} \forall u \ \textbf{false} \leftarrow r(f(u)) \\ \forall z \ \textbf{false} \leftarrow q(z) \\ \forall y \ p(f(y)) \leftarrow \textbf{true} \\ \forall x \ \text{man}(x) \vee \text{vrouw}(x) \leftarrow \text{mens}(x) \end{cases} \tag{28}$$

Uiteraard is dit meestal een arbeidsintensief werk, een algemeen algoritme die expressies kan omvormen wordt weergegeven in 6.3.11.

Het formaat van F Het te bewijzen deel heeft daarintegen een aangepast formaat in tegenstelling tot de theorie T zijn alle variabelen existentieel gekwantiseerd, bijgevolg geldt:

$$\exists x_1, \exists x_2 \ldots, \exists x_k \ A_1 \wedge A_2 \wedge \ldots \wedge A_m \tag{29}$$

Verder kunnen we hierbij opmerken dat er geen pijl in voorkomt, en dat enkel conjuncties ($\wedge$) tussen de termen zijn toegelaten. Uiteraard geldt ditzelfde formaat ook voor de gegronde Horn Clausale Logica (zonder variabelen) en de Horn Clausale Logica.

6.3.6 Bewijs door inconsistentie

In het algemeen werken automatische redeneersystemen meestal volgens het **"Bewijs door inconsistentie"-principe**. Hierbij bewijzen we F niet vanuit de axioma's. We voegen eenvoudigweg $\neg F$ toe aan T, als er in dat geval een inconsistentie in T ontstaat kunnen we besluiten dat F waar is. Omgekeerd geldt echter niet dat indien T niet inconsistent is, dat $\neg F$ waar is. Indien we dit formeel stellen bekomen we volgend theorema:

theorema 5. *Stel T een theorie en F een formule. T impliceert F als en slechts als $T \cup \{\neg F\}$ inconsistent is.*

Het bewijs hiervoor wordt hieronder weergegeven:

$$\begin{array}{lcll} T \vDash F & \Leftrightarrow & \forall m \in \text{model}\,(T) : T \rightarrow_m F & \{\text{Elk model } m \text{ van } T \text{ maakt } F \text{ waar}\} \\ & \Leftrightarrow & \forall m \in \text{model}\,(T) : \neg\,(T \rightarrow_m \neg F) & \{\text{Elk model } m \text{ van } T \text{ maakt } \neg F \text{ onwaar}\} \\ & \Leftrightarrow & \neg \exists m \in \text{model}\,(T \cup \{\neg F\}) & \{\text{Er bestaat geen model voor } T \cup \{\neg F\}\} \\ & \Leftrightarrow & \text{inconsistent}\,(T \cup \{\neg F\}) & \{T \cup \{\neg F\} \text{ is inconsistent}\} \end{array} \tag{30}$$

6.3.7 Modus Ponens

Nu we alle formaten gedefinieerd hebben, kunnen we een strategie opbouwen waarbij we doormiddel van de combinatie van twee expressies tot nieuwe expressies komen. Een van de basis deductieregels die we eenvoudig kunnen toepassen op Gegronde Horn Clausale Logica is de **Modus Ponens**. De Modus Ponens stelt dat indien we weten dat $A \leftarrow B$, en we weten dat B waar is, we hieruit kunnen besluiten dat A ook waar is. Of formeler gesteld:

$$\frac{\begin{array}{r} B \\ \therefore \quad A \leftarrow B \end{array}}{A} \tag{31}$$

Deze regels heeft echter weinig praktisch nut bij Gegronde Horn Clause logica, omdat het formaat toelaat dat we conjuncties in het lichaam plaatsen. We hebben dus nood aan een algemenere deductieregel die hiermee overweg kan. Hiervoor definiëren we de **Veralgemeende Modus Ponens**, een regel die stelt dat indien we een expressie $A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n$ kennen, en we weten dat $B_1, B_2, \ldots, B_n$ allemaal waar zijn, we ook weten dat A waar is. Deze regel kunnen we eenvoudig bewijzen door middel van inductie. Formeel definiëren we dus de Veralgemeende Modus Ponens als:

$$\frac{\begin{array}{r} B_1 \\ B_2 \\ \vdots \\ B_n \\ \therefore \quad A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n \end{array}}{A} \tag{32}$$

Voorwaartse redeneermethode Met deze deductieregel zijn we in staat op een eenvoudige manier een Gegronde Horn Clausale expressie te bewijzen. Indien we de Modus Ponens herhaaldelijk toepassen op T, waarbij we nieuwe feiten genereren, en deze vervolgens weer toelaten de Modus Ponens regel toe te passen op andere expressies. Een algemeen algoritme hiervoor wordt weergegeven in **Algorithm 28**. Dit algoritme is correct omdat de Veralgemeende Modus Ponens correct is, en we eigenlijk niets anders doen dan deze herhaaldelijk uitvoeren. Het algoritme is ook volledig, omdat er slechts een eindig aantal Gegronde Horn Clauses in T zitten, bijgevolg zijn er maar een eindig aantal atomaire gevolgen. Efficiënt is dit algoritme echter niet, het gaat niet gericht op zoek naar een oplossing maar probeert gewoon tot het uiteindelijk bij toeval de oplossing vindt. Er kunnen dus heel wat atomaire gevolgen afgeleid worden waar we eigenlijk niets aan hebben. Om dit probleem op te lossen zijn achterwaartse implementaties beter, waarbij men kijkt welke atomen men nodig heeft, en deze vervolgens probeert te zoeken.

Algorithm 28 Voorwaartse oplossingsstrategie voor gegronde horn clause

Require: $F = C_1 \wedge C_2 \wedge \ldots \wedge C_m$

```
Derived ← ∅
repeat
    selectOne (T, A ← B_1 ∧ B_2 ∧ ... ∧ B_n, A ∉ Derived ∧ ∀i : B_i ∈ Derived)
    Derived ← Derived ∪ {A}
until ∀i : C_i ∈ Derived ∨ ¬selectionPossible (T, A ← B_1 ∧ B_2 ∧ ... ∧ B_n, A ∉ Derived ∧ ∀i : B_i ∈ Derived)
if ∀i : C_i ∈ Derived then
    return T ⊨ F
else
    return failure
end if
```

achterwaartse redeneermethode Gebaseerd op het principe van bewijs door inconsistentie (zie 6.3.6) kunnen we dit algoritme versnellen, en dus een bewijs proberen op te stellen die met een zekere zin gedreven wordt, en niet door toeval. Hiervoor voegen we $\neg F$ toe aan T. We botsen echter op het probleem dat $\neg F$ geen geldige Clausale logica is, bijgevolg is het zeker geen Gegronde Horn Clausale Logica. Dit kunnen we echter omzeilen door niet $\neg F$ in te brengen in T, maar **false** $\leftarrow F$. Hierbij zien we **false** als een predicaatsymbool die onder elke interpretatie onwaar is. Dit kunnen we formeel bewijzen, we stellen dat $F = \exists x_1 \exists x_2 \ldots \exists x_m \; B_1 \wedge B_2 \wedge \ldots \wedge B_n$. Met simpele manipulatie van de negatie van deze expressie bekomen we:

$$\neg \left(\exists x_1 \exists x_2 \ldots \exists x_m \; B_1 \wedge B_2 \wedge \ldots \wedge B_n\right) \Leftrightarrow \forall x_1 \forall x_2 \ldots \forall x_m \; \textbf{false} \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n \tag{33}$$

We kunnen dus stellen dat $\neg F$ ook een Horn Clause is, en dus een element van T kan zijn. Om achterwaarts te kunnen redeneren hebben we verder nog nood aan een **Verder Veralgemeende Modus Ponens**. Hierbij gaan we uit van twee expressies $A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_i \wedge \ldots \wedge B_n$ en $B_i \leftarrow C_1 \wedge C_2 \wedge \ldots \wedge C_m$. Deze twee expressies kunnen we vervolgens omzetten naar een algemene expressie: $A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_{i-1} \wedge C_1 \wedge C_2 \wedge \ldots \wedge C_m \wedge B_{i+1} \wedge \ldots \wedge B_n$. Immers indien aan alle C condities voldaan wordt weten we dat B_i waar is, bijgevolg kunnen we B_i vervangen door alle C-condities:

$$\begin{array}{rr} & A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_i \wedge \ldots \wedge B_n \\ \therefore & B_i \leftarrow C_1 \wedge C_2 \wedge \ldots \wedge C_m \\ \hline & A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_{i-1} \wedge C_1 \wedge C_2 \wedge \ldots \wedge C_m \wedge B_{i+1} \wedge \ldots \wedge B_n \end{array} \tag{34}$$

Indien we nu F in zijn negatieve Horn Clause vorm zetten (**false** $\leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n$), kunnen we vervolgens vanuit F op zoek gaan naar substituties. Op het eerste zicht kan dit het probleem alleen maar erger lijken te maken. Vergelijking 34 lijkt telkens maar meer argumenten aan F te zullen toevoegen waardoor we het probleem juist telkens moeilijker zullen kunnen oplossen. Anderzijds kan $m = 0$ (een feit dus). Hiermee elimineren we dus een argument. Op basis van dit gegeven kunnen we nu een algoritme ontwikkelen die vertrekt vanuit $\neg F$ en stelselmatig substituties uitvoert totdat het rechterlid van $\neg F$ leeg is. In dat geval staat er **false** $\leftarrow$. Dit is per definitie fout, bijgevolg kunnen we stellen dat er een inconsistentie in $T \cup \{\neg F\}$ zit en dus dat $T \vDash F$. Formeel wordt hiervoor een algoritme gedefinieerd in **Algorithm 29**. Het probleem is dat dit algoritme niet altijd tot de oplossing komt,

Algorithm 29 Achterwaardse oplossingsstrategie voor gegronde horn clause

Require: $F = B_1 \wedge B_2 \wedge \ldots \wedge B_n$

```
goal := false ← B_1 ∧ B_2 ∧ ... ∧ B_n
repeat
    selectOne (T, B_i ← C_1 ∧ C_2 ∧ ... ∧ C_m, B_i ∈ body (goal))
    bodyReplace (goal, B_i, C_1 ∧ C_2 ∧ ... ∧ C_m)
until ¬selectionPossible (T, B_i ← C_1 ∧ C_2 ∧ ... ∧ C_m, B_i ∈ body (goal)) ∨ goal = false ←
```

stel dat er verschillende expressies zijn waarmee we een zekere B_i kunnen vervangen. In het geval dat we een foute kiezen, waardoor we niet ieder element in het lichaam van goal kunnen vervangen, zal

het algoritme vastlopen. En op het moment dat er uiteindelijk geen vervanging meer mogelijk is, zal het stoppen, zonder resultaat. Dit impliceert echter niet dat er geen oplossingsmethode is. We zullen dus een Backtrackingsalgoritme rond dit algoritme moeten schrijven, dit kiest welke expressie we zullen gebruiken om de query te vervangen (regel 3 bij **Algorithm 29** dus). Indien het algoritme dan na alle alternatieven te hebben uitgeprobeerd nog steeds nergens goal = **false** ← tegenkwam, kunnen we stellen dat er geen inconsistentie in $T \cup \{\neg F\}$ zit, en kunnen we dus niet kunnen besluiten dat F waar is.

Hoewel we door backtracking het omgekeerde verwachten is achterwaarts redeneren efficiënter. We gaan immers geen regels meer toepassen die niets met F te zien hebben. Bovendien bestaan er enkele regels die snel kunnen detecteren of een tak al dan niet gaat falen (zo moet voor ieder nieuw lid in het lichaam van goal, een expressie bestaan die deze term opnieuw kan substitueren, in sommige gevallen is dit dan een feit). Dergelijke detectiesysteem wordt **Atoom-selectie** genoemd, en kan meestal in de keuze betrokken worden.

Is dit algoritme volledig? Het antwoord is dat het afhangt van de backtrackingtechniek die gehanteerd wordt. Indien we bijvoorbeeld een stelsel opstellen:

$$\begin{cases} \textbf{false} \leftarrow p \\ p \leftarrow p \\ p \leftarrow \end{cases} \tag{35}$$

Kan een slecht zoekalgoritme in principe telkens regel 2 kiezen en deze vervangen. Het gevolg is dat het algoritme nooit ten einde loopt, en goal onverandert blijft. Een volledig zoekalgoritme zal echter af en toe een andere mogelijkheid proberen, en zo uiteindelijk de inconsistentie bewijzen.

6.3.8 Unificatie

Unificatie is een procedure waarbij we substitutie van variabelen mogelijk maken. Gegeven bijvoorbeeld twee expressies $p(f(A), y)$ en $p(x, g(x))$. We willen de **Meest Algemene Gemeenschappelijke Instantiatie** ook wel **Meest Algemene Unifier**, **Most General Unifier** of **mgu** genoemd vinden. In dit voorbeeld zullen we x vervangen door $f(A)$, hierdoor moeten we $g(x)$ vervangen door $g(f(A))$. Tot slot kunnen we vervolgens y vervangen door $g(f(A))$. En bekomen we als algemene expressie: $p(f(A), g(f(A)))$. In dat geval is de mgu dus: $\theta = \{x/f(A), y/g(f(A))\}$.

Substituties In de vorige subsubsectie behandelden we de Verder Veralgemeende Modus Ponens. Hiermee waren we in staat op een redeneersystemen te ontwikkelen voor de Gegronde Horn Clausale Logica. Nu voegen we echter variabelen toe. Het systeem kan immers niet detecteren dat verschillende expressies door het veralgemenen van de variabelen, eigenlijk op hetzelfde neerkomen. Hiervoor gebruiken we dus Unificatie. Een unificatie zal twee expressies veralgemenen zodat ze tot dezelfde expressie omgevormd kunnen worden. Het resultaat is een **substitutie**. Een substitutie is een eindige verzameling koppels van het type Variabele/Term. Hierbij verschilt iedere variabele van alle termen in de substitutie. Verder mag een variabele die aan de linkerkant gedefinieerd wordt, niet in een term aan de rechterkant voorkomen (ook niet bij een ander koppel). We noteren een substitutie meestal met een kleine griekse letter.

Hieronder staan enkele voorbeelden van substituties:

$$\begin{cases} \theta = \{x/g(z), y/B\} \\ \varphi = \{x/h(g(A)), y/g(A), z/w\} \\ \kappa = \{x/\text{Cesar}, y/\text{Marcus}\} \\ \sigma = \{x/\text{vader}(\text{Marcus}), y/\text{dochter}(\text{Cesar}, z)\} \end{cases} \tag{36}$$

Substituties toepassen Het toepassen van substituties op enkelvoudige expressies (atomen of termen) is zeer eenvoudig. Alle variabelen in de expressie die ook ergens aan de linkerkant van een koppel in de substitutie staan worden eenvoudigweg vervangen door wat er aan de rechterkant van deze term staat.

Dit wordt geïllustreerd met onderstaande voorbeelden:

$$\begin{cases} \theta = \{x/g\,(z)\,, y/B\} & p\,(x, f\,(y, z)) \cdot \theta = & p\,(g\,(z)\,, f\,(B, z)) \\ \varphi = \{x/h\,(g\,(A))\,, y/g\,(A)\,, z/w\} & p\,(x, f\,(y, z)) \cdot \varphi = & p\,(h\,(g\,(A))\,, f\,(g\,(A)\,, w)) \\ \kappa = \{x/\text{Cesar}, y/\text{Marcus}\} & \text{pompeiër}\,(x) \cdot \kappa = & \text{pompeiër}\,(\text{Cesar}) \end{cases} \tag{37}$$

Gebruik van Unifiers We willen substituties gebruiken om twee atomen in twee verschillende clauses gelijk te maken. Hiervoor moeten we dus op zoek gaan naar de meest algemene gemeenschappelijke unifier. Een **unifier** is een substitutie met een extra eigenschap:

theorema 6. *Een substitutie θ is een unifier van een verzameling enkelvoudige expressies S indien geldt: $S \cdot \theta$ is een* **singleton**.

We zijn echter enkel geïntresseerd in de Meest Algemene Unfier, dit is een unifier, die uiteraard nog een extra eigenschap heeft:

theorema 7. *Een unifier θ is een meest algemene unifier voor een verzameling enkelvoudige expressies S indien voor iedere andere unifier φ van S er een substitutie σ bestaat waarvoor geldt: $S \cdot \varphi = (S \cdot \theta) \cdot \sigma$*

Het sleutelidee achter de Meest Algemene Unifier is dan ook om minimale instantiaties te criëren, en dus geen overbodige substituties uit te voeren, of substituties naar zeer complexe termen terwijl dit eigenlijk niet nodig is. De Meest Algemene Unifier van een verzameling enkelvoudige expressies S wordt genoteerd als $\text{mgu}\,(S)$, indien we de Meest Algemene Unfier van twee enkelvoudige expressies A en B nemen wordt dit ook genoteerd als $\theta = \text{mgu}\,(A, B)$. Er bestaat echter enkel een Meest Algemene Unifier tussen A en B indien deze hetzelfde predicaat hebben. Anders heeft de substitutie van variabelen immers geen zin, de twee expressies zullen telkens in predicaat verschillen, en de verzameling zal nooit herleid kunnen worden tot een singleton.

Überalgemene Modus Ponens Nu we een unifier hebben gedefinieerd kunnen we de Verder Veralgemeende Modus Ponens verder veralgemenen. Hiervoor definiëren we de **Überalgemene Modus Ponens** als volgt:

$$\frac{\begin{array}{r} A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_i \wedge \ldots \wedge B_n \\ \therefore \qquad B_i' \leftarrow C_1 \wedge C_2 \wedge \ldots \wedge C_m \end{array}}{\begin{array}{c} (A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_{i-1} \wedge C_1 \wedge C_2 \wedge \ldots \wedge C_m \wedge B_{i+1} \wedge \ldots \wedge B_n) \cdot \theta \\ \theta = \text{mgu}\,(B_i, B_i') \end{array}} \tag{38}$$

Er is echter één voorwaarde: B_i en B_i' moeten hetzelfde predicaat hebben, anders bestaat er eenvoudigweg geen Meest Algemene Unifier. Verder volgt de correctheid van deze formule uit de correctheid van de Verder Veralgemeende Modus Ponens. We kunnen immers eerst θ toepassen op de twee expressies. In dat geval bekomen we een geval volkomen identiek aan de Verder Veralgemeende Modus Ponens. Verder zullen we ook eerst de namen van de variabelen van de twee expressies aanpassen, alvorens we de Überalgemene Modus Ponens toepassen, om **clashes** met gelijknamige variabelen te vermijden. We kunnen nu **Algorithm 29** uitbreiden zodat we ook met unifiers kunnen werken, hierdoor bekomen we **Algorithm 30**. Uiteraard dienen we opnieuw een backtrackingsalgoritme rond deze implementatie

Algorithm 30 Achterwaardse oplossingsstrategie voor Horn Clause (met substitutie)

Require: $F = B_1 \wedge B_2 \wedge \ldots \wedge B_n$
1: goal := **false** $\leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n$
2: **repeat**
3: **selectOne** $(T, B_i' \leftarrow C_1 \wedge C_2 \wedge \ldots \wedge C_m, \exists B_i \in \textbf{body}\,(\text{goal}) \wedge \exists\theta = \text{mgu}\,(B_i, B_i'))$
4: goal := **false** $\leftarrow (B_1 \wedge B_2 \wedge \ldots \wedge B_{i-1} \wedge C_1 \wedge C_2 \wedge \ldots \wedge C_m \wedge B_{i-1} \wedge \ldots \wedge B_n) \cdot \theta$
5: **until** $\neg$**selectionPossible** $(T, B_i' \leftarrow C_1 \wedge C_2 \wedge \ldots \wedge C_m, \exists B_i \in \textbf{body}\,(\text{goal}) \wedge \exists\theta = \text{mgu}\,(B_i, B_i')) \vee$ goal = **false** $\leftarrow$

te schrijven omdat we een foute keuze kunnen maken bij regel 3. Verder hangt de volledigheid van dit algoritme opnieuw af van de volledigheid van het zoekalgoritme (volledigheid met oneindig diepe takken). Een bijkomende vereiste is hierbij echter dat we bij de expressie eerst de variabelen hernoemen.

Generatie van een Meest Algemene Unifier Tot nu toe hebben we altijd de Meest Algemene Unifier beschouwd zonder ons af te vragen, hoe we deze kunnen vinden. Hiervoor is een speciaal algoritme nodig die met de invoer van twee enkelvoudige expressies A en B er in slaagt om een Meest Algemene Unfier te genereren. Hierbij vertrekken we van een substitutie mgu $= \{A/B\}$. Uiteraard zal een dergelijke substitutie niet werken (strikt gezien is het geen substitutie, daar aan de linkerkant een variabele hoort te staan, en geen enkelvoudige expressie), immers horen de variabelen in andere element van de expressie ook vervangen te worden door de termen in de substitutie. Bijgevolg zal het algoritme stap voor stap dit koppel opdelen in meer koppels om te eindigen met een echte substitutie: de Meest Algemene Unifier. Dit algoritme wordt formeel beschreven in **Algorithm 31**. Dit algoritme wordt het **Martelli-Montanari**

Algorithm 31 Generatie van de Meest Algemene Unifier **mgu** (A, B)

```
1: mgu ← {A/B}
2: stop ← false
3: error ← false
4: while ¬stop ∧ ¬error do
5:     if selectionPossible (mgu, s/t, s ∉ variables ∧ t ∈ variables) then
6:         select (mgu, s/t, s ∉ variables ∧ t ∈ variables)
7:         mgu ← (mgu \ {s/t}) ∪ {t/s}
8:     else if selectionPossible (mgu, s/t, s ∈ variables ∧ s = t) then
9:         select (mgu, s/t, s ∈ variables ∧ s = t)
10:        mgu ← mgu \ {s/t}
11:    else if selectionPossible (mgu, s/t, s ∈ variables ∧ t ∉ variables ∧ s ∈ t) then
12:        error ← true{s is een parameter in t}
13:    else if selectionPossible (mgu, s/t, s ∈ variables ∧ ∃u/v ∈ mgu ∧ s/t ≠ u/v ∧ (s ∈ u ∨ s ∈ v))
       then
14:        select (mgu, s/t, s ∈ variables ∧ ∃u/v ∈ mgu ∧ s/t ≠ u/v ∧ (s = u ∨ s = v))
15:        θ ← {s/t}
16:        for all u/v ∈ mgu do
17:            if s/t ≠ u/v then
18:                mgu ← (mgu \ {u/v}) ∪ ({u/v} · θ){pas substitutie toe op alle elementen}
19:            end if
20:        end for
21:    else if selectionPossible (mgu, s/t, s = f (s1, s2, ... sn) ∧ t = g (t1, t2, ... tm)) then
22:        select (mgu, s/t, s = f (s1, s2, ... sn) ∧ t = g (t1, t2, ... tm))
23:        if f ≠ g ∨ m ≠ n then
24:            error ← true{Verschillende functies of ander aantal parameters}
25:        else
26:            mgu ← (mgu \ {s/t}) ∪ {s1/t1, s2/t2, ..., sn/tn}
27:        end if
28:    else
29:        stop ← true{Niets meer aan te passen, algoritme is ten einde}
30:    end if
31: end while
32: if ¬error then
33:    return mgu
34: else
35:    return failure
36: end if
```

algoritme[8] genoemd in is verder uitbreidbaar voor meer dan twee expressies. Het algoritme kan stoppen omwille van twee redenen: ofwel zijn er geen regels meer toe te passen, in dat geval is stop = **true**, anders is error = **true**, in dat geval kunnen we geen Meest Algemene Unifier genereren.

[8] Naar Alberto Martelli en Ugo Montanari

Bij wijze van voorbeeld zullen we twee expressies met elkaar unificeren: $\text{kent}\,(\text{Cesar}, x)$ en $\text{kent}\,(y, \text{moeder}\,(y))$. Dit doen we stap per stap. We beginnen met deze expressies in de mgu te plaatsen:

$$\text{mgu}_1 = \{\text{kent}\,(\text{Cesar}, x)\,/\text{kent}\,(y, \text{moeder}\,(y))\} \tag{39}$$

Vervolgens halen we de twee predicaten uiteen en instantiëren we de argumenten met elkaar:

$$\text{mgu}_2 = \{\text{Cesar}/y, x/\text{moeder}\,(y)\} \tag{40}$$

We evalueren hier de elementen van links naar rechts. Het eerste element dient omgewisseld te worden:

$$\text{mgu}_3 = \{y/\text{Cesar}, x/\text{moeder}\,(y)\} \tag{41}$$

Nu we het eerste element op de juiste manier geïnstantieerd hebben, zullen we deze substitutie toepassen op de andere elementen:

$$\text{mgu}_4 = \{y/\text{Cesar}, x/\text{moeder}\,(\text{Cesar})\} \tag{42}$$

Eenmaal we deze toestand bereikt hebben, valt er niets meer aan te passen. We hebben dus de Meest Algemene Unifier gevonden: $\varphi = \{y/\text{Cesar}, x/\text{moeder}\,(\text{Cesar})\}$. Indien we deze substitutie op de termen toepassen bekomen we $\text{kent}\,(\text{Cesar}, \text{moeder}\,(\text{Cesar}))$.

6.3.9 De Resolutie stap en Factoring

De Resolutie stap Nadat we de Modus Ponens-regel en het vervangen van variabelen doormiddel van Unficiatie behandeld hebben, verbreden we ons taalbegrip verder naar de Clausale vorm. Hierbij staan we dus toe dat het hoofd van de expressies een disjunctie ($\vee$) is van verschillende atomen. We weten echter niet welke uitspraak in het hoofd van de expressie nu eigenlijk waar is, indien alle atomen waar zijn in het lichaam. Een oplossing kan er uit bestaan, om deze expressie met behulp van de disjunctieve vorm om te zetten. Een expressie $\forall x_1, \forall x_2 \ldots, \forall x_k\ A_1 \vee A_2 \vee \ldots \vee A_m \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n$ wordt dan gelijk aan:

$$\begin{gathered} \forall x_1, \forall x_2 \ldots, \forall x_k\ A_1 \vee A_2 \vee \ldots \vee A_i \vee \ldots \vee A_m \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n \\ \Updownarrow \\ \forall x_1, \forall x_2 \ldots, \forall x_k\ A_i \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n \wedge \neg A_1 \wedge \neg A_2 \wedge \ldots \neg A_{i-1} \wedge \neg A_{i+1} \wedge \ldots \wedge A_m \end{gathered} \tag{43}$$

Omdat negaties echter niet toegestaan zijn in de Clausale vorm, en er verder geen algemeen algoritme bestaat die de negatie van een bepaalde uitdrukking kan berekenen (en inwendige negatie soms onmogelijk kan zijn), zullen we dit probleem op een andere manier moeten oplossen. Daarom veralgemenen we verder de oorzaak van het probleem: De Überalgemene Modus Ponens. Omdat het resultaat eigenlijk de geest van de Modus Ponens-regel niet meer bevat spreken we vanaf nu van een Resolutie. De resolutie wordt gedefinieerd als:

$$\frac{\begin{array}{rr} & A_1 \vee A_2 \vee \ldots \vee A_m \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_i \wedge \ldots \wedge B_n \\ \therefore & C_1 \vee C_2 \vee \ldots \vee C_j \vee B_i' \vee C_{j+1} \vee \ldots \vee C_k \leftarrow D_1 \wedge D_2 \wedge \ldots \wedge D_l \end{array}}{\begin{array}{r} (A_1 \vee A_2 \vee \ldots \vee A_m \vee C_1 \vee C_2 \vee \ldots \vee C_j \vee C_{j+1} \vee \ldots \vee C_k \\ \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_{i-1} \wedge D_1 \wedge D_2 \wedge \ldots \wedge D_m \wedge B_{i+1} \wedge \ldots \wedge B_n) \cdot \theta \\ \theta = \text{mgu}\,(B_i, B_i') \end{array}} \tag{44}$$

De resolutie is te bewijzen door alle C-elementen door middel van negatie naar het lichaam van de expressie te brengen. Vervolgens passen we de Überalgemene Modus Ponens toe, op de twee expressies, en kunnen we de negaties weer terug naar het hoofd van de bekomen expressie schuiven. Een andere mogelijkheid is om de expressies om te zetten in een conjunctieve normaalvorm. Vervolgens de conjunctie van de twee expressies te nemen, om daarna terug naar de implicatieve vorm te gaan.

Factoring De resolutie introduceert echter een nieuw probleem. We dreigen in onze expressie kopieën te generen van atomen, of afwijkende atomen die onder substitutie met de Meest Algemene Unifier

equivalent worden. We dienen dus situaties als de volgende te vermijden waarbij er een Meest Algemene Unifier $\exists\theta = \text{mgu}\,(B_i, B_i')$ of $\exists\varphi = \text{mgu}\,(A_i, A_i')$ bestaat:

$$\begin{cases} A_1 \vee A_2 \vee \ldots A_m \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_i \wedge \ldots \wedge B_i' \wedge \ldots \wedge B_n \\ A_1 \vee A_2 \vee \ldots \vee A_i \vee \ldots \vee A_i' \vee \ldots A_m \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n \end{cases} \tag{45}$$

Een nieuwe deductieregel genaamd **Factoring** rekent met deze problemen af. Factoring blijkt bovendien noodzakelijk te zijn in zowat elk probleem, dit kunnen we aantonen door de resolutie te analyseren op het aantal atomen:

$$\begin{array}{rr|l} & A_1 \vee A_2 \vee \ldots \vee A_m \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_i \wedge \ldots \wedge B_n & N = m + n \\ \therefore & C_1 \vee C_2 \vee \ldots \vee C_j \vee B_i' \vee C_{j+1} \vee \ldots \vee C_k \leftarrow D_1 \wedge D_2 \wedge \ldots \wedge D_l & M = k + l + 1 \\ \hline & (A_1 \vee A_2 \vee \ldots \vee A_m \vee C_1 \vee C_2 \vee \ldots \vee C_j \vee C_{j+1} \vee \ldots \vee C_k & \\ & \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_{i-1} \wedge D_1 \wedge D_2 \wedge \ldots \wedge D_m \wedge B_{i+1} \wedge \ldots \wedge B_n) \cdot \theta & N + M - 2 \\ & \theta = \text{mgu}\,(B_i, B_i') & \end{array} \tag{46}$$

Hierbij rekenen we **false** niet als een atoom. Indien we dus een inconsistentie willen aantonen moeten we vroeg of laat een expressie bekomen van lengte 0 (**false** $\leftarrow$). We kunnen er echter vanuit gaan dat zowel $M \geq 1$ als $N \geq 1$. Alleen wanneer we dus twee expressies combineren door resulutie waarbij het eerste van de vorm **false** $\leftarrow p$ en het tweede van de vorm $p \leftarrow$ is, kunnen we doormiddel van resolutie de inconsistentie bewijzen. In alle andere gevallen (waar we waarschijnlijk meer in geïntresseerd zijn) zal het algoritme nooit tot een oplossing komen.

Daarom definiëren we twee nieuwe deductieregels, die de problemen in vergelijking 45 oplossen:

$$\begin{array}{rr} \therefore & A_1 \vee A_2 \vee \ldots \vee A_m \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_i \wedge \ldots \wedge B_i' \wedge \ldots \wedge B_n \\ \hline & (A_1 \vee A_2 \vee \ldots \vee A_m \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_i \wedge \ldots \wedge B_n) \cdot \theta \\ & \theta = \text{mgu}\,(B_i, B_i') \end{array}$$

$$\begin{array}{rr} \therefore & A_1 \vee A_2 \vee \ldots \vee A_i \vee \ldots \vee A_i' \vee \ldots \vee A_m \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n \\ \hline & (A_1 \vee A_2 \vee \ldots \vee A_i \vee \ldots \vee A_m \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n) \cdot \varphi \\ & \varphi = \text{mgu}\,(A_i, A_i') \end{array} \tag{47}$$

Algemene Implementatie Nu we in staat zijn om theoriën op te lossen, kunnen we een algemeen algoritme bouwen, die in staat is de inconsistentie van ons gewijzigde theorie ($S = T \cup \{\textbf{false} \leftarrow F\}$) aan te tonen. Dit algoritme beschrijven we in **Algorithm 32**. Merk op dat hierbij het lineare karakter verloren

Algorithm 32 Generatie van de Meest Algemene Unifier **mgu** (A, B)

Require: S\{Theorie op inconsistentie te bewijzen\}
consistent $\leftarrow$ **false**
inconsistent $\leftarrow$ **false**
while $\neg$consistent $\wedge$ $\neg$inconsistent **do**
  if (**false** $\leftarrow$) $\in S$ **then**
    inconsistent $\leftarrow$ **true**
  else if $\exists G, H \in S$: **resolvable** $(G, H) \wedge \neg$**resolved** (G, H) **then**
    select $(S, (G, H), G, H \in S$: **resolvable** $(G, H) \wedge \neg$**resolved** $(G, H))$
    $I \leftarrow$ **factor** (**resolvent** (G, H))
    $S \leftarrow S \cup \{I\}$
  else
    consistent $\leftarrow$ **true**
  end if
end while

gaat. Onder Horn Clause (**Algorithm 30**) vertrokken we vanuit goal := **false** $\leftarrow F$, en combineerden we deze expressie voortdurend met andere expressies, waardoor we een nieuw doel bekwamen. Bij het algoritme met de resolutie, wordt het doel niet altijd betrokken in de functie. De Clausale resolutie is dan ook niet linear en kan soms heel wat expressies met elkaar combineren die uiteindelijk helemaal niets

opleveren. Dit maakt dit algoritme erg onefficiënt. Bovendien krijgt het het een erg non-deterministisch karakter (vooral door het **select** () statement). Een bewijs is immers niet meer een lineare tak, maar een deel van de impliciet gegenereerde graaf. Omdat we in principe op situaties kunnen botsen waarbij we zo oneindig veel expressies genereren, kunnen we ons de vraag stellen of dit algoritme volledig is of niet. Er bestaat in ieder geval een implementatie, de **Herbrand Stellingbewijzer**[9], die een volledige zoekstrategie voor dit problemen teruggeeft. Verder kunnen we concluderen dat deze methode correct is. Enkel indien inconsistent = **true** heeft het algoritme resultaat, in dat geval weten we dat **false** $\leftarrow$ ergens toegevoegd is aan S. Aangezien onze resolutiestap correct is, weten we bijgevolg ook dat deze expressie correct is, en dat alle modellen van S deze expressie vroeg of laat zullen toevoegen. Bijgevolg kunnen we stellen dat S geen modellen heeft. Indien het algoritme stopt omwille van consistent = **true**, dan weten we dat het algoritme alle mogelijkheden heeft geprobeerd. Immers alle combinaties die nog niet geprobeerd werden, worden uitgevoerd, maar **false** $\leftarrow$ wordt eenvoudigweg niet bereikt. Indien de theorie toch inconsistent zou zijn, zou een volledige strategie deze vroeg of laat moeten ontdekken wat dus niet gebeurt. Gebaseerd op de kleinste-model semantiek kunnen we besluiten dat F niet klopt.

6.3.10 Resolutie bewijzen

Hoe bewijzen we nu de inconsistentie in onze gemodificeerde theorie T' (waarbij we **false** $\leftarrow F$ hebben toegevoegd)? We kunnen de inconsistentie aantonen door twee expressies uit T' te nemen, waar resolutie op mogelijk is. In dat geval voeren we de resolutie uit tussen die twee expressies, en voegen het resultaat toe aan T' samen met de twee oorspronkelijke expressies. Indien de resolutie resulteert in de expressie **false** $\leftarrow$ hebben we een inconsistentie gevonden. Bijgevolg kunnen we zeggen dat de laatste set T' inconsistent is, omdat deze set equivalent is met de initiële gemodificeerde set T' kunnen we zeggen dat T' inconsistent is in het algemeen. En dit impliceert bijgevolg $T \vDash F$.

6.3.11 Normalisatie

Zoals reeds vermeld kunnen we iedere expressie in eerste-orde predicatenlogica omzetten naar Clausale logica. Bijgevolg hebben we dus geen extra componenten meer nodig om tot een oplossingsstrategie te komen, deze kan nu immers alle problemen oplossen. Door die equivalentie kunnen we dus stellen dat:

theorema 8. *Elke theorie in Eerste-Orde Predicatenlogica kan automatisch omgezet worden in een Clausale theorie T' zodat: isInconsistent*$(T) \Leftrightarrow$ *isInconsistent*(T')

We kunnen dus bijgevolg een algoritme ontwikkelen die een gegeven theorie T in predicatenlogica omzet naar een theorie T' in Clausale vorm, in deze subsubsectie zullen we trachtten zo'n algoritme te ontwikkelen, en in grote lijnen formeel te beschrijven.

We kunnen iedere formule in predicatenlogica omzetten naar een formule die een conjunctie van disjuncties zijn (waarbij de atomen eventueel van een negatie worden voorzien), we bekomen dus een situatie zoals beschreven in volgende vergelijking:

$$(A_1 \vee A_2 \vee \ldots \vee A_n) \wedge (B_1 \vee B_2 \vee \ldots \vee B_m) \wedge \ldots \wedge (C_1 \vee C_2 \vee \ldots \vee C_k) \tag{48}$$

Dit wordt meestal aangeduid als de **Conjunctieve Normaal Vorm**. Dit kunnen we bekomen in 4 stappen vanuit iedere expressie in predicaatlogica:

1. Zet $\leftrightarrow$ om in $\rightarrow$: $p \leftrightarrow q \Rightarrow p \rightarrow q \wedge q \rightarrow p$
2. Zet $\rightarrow$ om naar zijn disjunctie vorm: $p \rightarrow q \Rightarrow \neg p \vee q$
3. Breng de negaties zover mogelijk naar binnen
4. Gebruik de distributiviteit van $\wedge$ en $\vee$ om de uiteindelijk vorm te bekomen

[9] Genoemd naar Jacques Herbrand (1908-1931).

Nu we een conjunctie van disjunties bereikt hebben, moeten we de quantoren zien om te vormen. Hierbij zetten we onze expressie eerst om naar **Prenex Normaal Vorm**. Hierbij staan alle quantoren vooraan in de expressie. Indien er verschillende lokale quantoren in de expressie staan die dezelfde naam dragen, dienen we deze te hernoemen. Iedere variabele dient een andere naam te hebben. We bekomen dus volgende vorm:

$$(Q_1\ x_1)\,(Q_2\ x_2)\ldots(Q_l\ x_l)\ \ (A_1 \vee A_2 \vee \ldots \vee A_n) \wedge (B_1 \vee B_2 \vee \ldots \vee B_m) \wedge \ldots \wedge (C_1 \vee C_2 \vee \ldots \vee C_k) \quad (49)$$

Hierbij zijn de Q_i's ofwel een $\forall$ ofwel $\exists$ karakter.

De Clausale logica laat echter maar één quantor toe: $\forall$. We moeten dus de de $\exists$ kunnen elimineren. Indien deze onafhankelijk zijn van de rest van de expressie, is er in principe niet echt een probleem. Onder het **"Geef het kind een naam"-principe** vervangen we gewoon de variabele door een constante (een **skolem constante**). Indien we echter met geneste quantoren te maken hebben wordt de situatie complexer. Indien we bijvoorbeeld de volgende vergelijking hebben:

$$\forall x \exists y\ \text{mens}\,(x) \rightarrow \text{hart}\,(y) \wedge \text{heeft}\,(x, y) \quad (50)$$

Hierbij kunnen we y niet zomaar door een hart vervangen, anders zou dit betekenen dat iedere persoon met hetzelfde hart rondloopt! In dat geval dienen we een **skolem functie** te definiëren, deze functie bezit evenveel parameters als $\forall$-quantoren waarin het omsloten is. We lossen dus het bovenstaande probleem op met een skolem-functie H:

$$\forall x\ \text{mens}\,(x) \rightarrow \text{hart}\,(H\,(x)) \wedge \text{heeft}\,(x, H\,(x)) \quad (51)$$

Nu we van de $\exists$-quantoren verlost zijn, moeten we alleen nog onze expressie omzetten naar een Clausale Implicatieve Vorm. Hierbij zullen we onze expressie opsplitsen in verschillende expressies, ter hoogte van de $\wedge$-connectoren. Optioneel kunnen we bovendien in elk van de nieuwe expressies de $\forall$-quantoren weglaten (we weten immers dat iedere quantor die in de expressie gebruikt wordt, universieel gedefinieerd is). Concreet betekent dit dus dat we onderstaande vorm toepassen:

$$(A_1 \vee A_2 \vee \ldots \vee A_n) \wedge (B_1 \vee B_2 \vee \ldots \vee B_m) \wedge \ldots \wedge (C_1 \vee C_2 \vee \ldots \vee C_k) \Rightarrow \begin{cases} A_1 \vee A_2 \vee \ldots \vee A_n \\ B_1 \vee B_2 \vee \ldots \vee B_m \\ \vdots \\ C_1 \vee C_2 \vee \ldots \vee C_k \end{cases} \quad (52)$$

Om vervolgens tot een implicatief formaat te komen brengen we vervolgens alle negatie-atomen naar de rechterkant van de pijl, de andere atomen blijven aan de linkerkant van de pijl staan. Vervolgens plaatsen we de juiste connectoren ($\vee$ aan de linkerkant, $\wedge$ aan de rechterkant): een concreet voorbeeld wordt hieronder weergegeven:

$$A_1 \vee A_2 \vee \ldots \vee \neg A_i \vee \ldots \vee A_n \Rightarrow A_1 \vee A_2 \vee A_{i-1} \vee A_{i+1} \vee \ldots \vee A_n \leftarrow A_i \quad (53)$$

Volledige Procedure Indien we nu alle stappen samenvatten in één procedure bekomen we het volgende stappenplan (dit kan geïmplementeerd worden in een algoritme, en dus machinaal omgevormd worden):

1. Zet $\leftrightarrow$ om in $\rightarrow$: $p \leftrightarrow q \Rightarrow p \rightarrow q \wedge q \rightarrow p$
2. Zet $\rightarrow$ om naar zijn disjunctie vorm: $p \rightarrow q \Rightarrow \neg p \vee q$
3. Breng de negaties zover mogelijk naar binnen
   - $\neg\,(\neg p) \Rightarrow p$
   - $\neg\,(p \vee q) \Rightarrow \neg p \wedge \neg q$
   - $\neg\,(p \wedge q) \Rightarrow \neg p \vee \neg q$
   - $\neg \exists x\ p\,(x) \Rightarrow \forall x\ \neg p\,(x)$

- $\neg\forall x\ p(x) \Rightarrow \exists x\ \neg p(x)$

4. Standaardiseer variabele namen (geen twee lokale variabelen met dezelfde naam)
5. Quantoren naar voren (naar Prenex Normaal Vorm)
6. Elimineer $\exists$ (invoering van Skolems)
7. Disjuncties naar binnen (naar Conjunctieve Normaal Vorm)
8. Expressie opsplitsen ter hoogte van de $\wedge$-connectoren
9. $\forall$-quantoren weglaten
10. Negatie atomen naar de rechterkant van de pijl, en de connectoren vervangen door de juiste (naar Clausale Vorm)

Bij wijze van voorbeeld zullen we de meest complexe expressie (5) uit het leidend voorbeeld normaliseren naar zijn Clausale Vorm:

$$\forall x : \text{romein}(x) \rightarrow \left(\begin{array}{c} \text{loyaalAan}(x, \text{Cesar}) \wedge \neg\text{probeerTeVermoorden}(x, \text{Cesar}) \\ \vee \\ \neg\text{loyaalAan}(x, \text{Cesar}) \wedge \text{probeerTeVermoorden}(x, \text{Cesar}) \end{array} \right) \tag{54}$$

Eerst werken we de implicatie om naar zijn disjuncte vorm:

$$\forall x : \neg\text{romein}(x) \vee \left(\begin{array}{c} \text{loyaalAan}(x, \text{Cesar}) \wedge \neg\text{probeerTeVermoorden}(x, \text{Cesar}) \\ \vee \\ \neg\text{loyaalAan}(x, \text{Cesar}) \wedge \text{probeerTeVermoorden}(x, \text{Cesar}) \end{array} \right) \tag{55}$$

We kunnen een groot aantal stappen overslaan, zo bevat de expressie geen lokale variabelen, en staan alle quantoren reeds vooraan. Vervolgens dienen we de expressie om te zetten naar een een conjunctieve normaalvorm. Dit levert ons de volgende expressie op:

$$\forall x : \left(\begin{array}{c} (\neg\text{romein}(x) \vee \text{loyaalAan}(x, \text{Cesar}) \vee \text{probeerTeVermoorden}(x, \text{Cesar})) \\ \wedge \\ (\neg\text{romein}(x) \vee \neg\text{loyaalAan}(x, \text{Cesar}) \vee \neg\text{probeerTeVermoorden}(x, \text{Cesar})) \end{array} \right) \tag{56}$$

We splitsen vervolgens deze expressie op ter hoogte van de conjuncties ($\wedge$):

$$\left\{ \begin{array}{l} \forall x : \neg\text{romein}(x) \vee \text{loyaalAan}(x, \text{Cesar}) \vee \text{probeerTeVermoorden}(x, \text{Cesar}) \\ \forall x : \neg\text{romein}(x) \vee \neg\text{loyaalAan}(x, \text{Cesar}) \vee \neg\text{probeerTeVermoorden}(x, \text{Cesar}) \end{array} \right. \tag{57}$$

We laten nu de quantoren weg en schrijven ten slotte de expressies in hun implicatieve vorm:

$$\left\{ \begin{array}{l} \text{loyaalAan}(x, \text{Cesar}) \vee \text{probeerTeVermoorden}(x, \text{Cesar}) \leftarrow \text{romein}(x) \\ \textbf{false} \leftarrow \text{romein}(x) \wedge \text{loyaalAan}(x, \text{Cesar}) \wedge \text{probeerTeVermoorden}(x, \text{Cesar}) \end{array} \right. \tag{58}$$

Indien we het volledige systeem normaliseren bekomen we volgende theorie T:

$$T = \left\{ \begin{array}{l} \text{man}(\text{Marcus}) \\ \text{pompeiër}(\text{Marcus}) \\ \text{romein}(x) \leftarrow \text{pompeiër}(x) \\ \text{heerser}(\text{Cesar}) \\ \text{loyaalAan}(x, \text{Cesar}) \vee \text{probeerTeVermoorden}(x, \text{Cesar}) \leftarrow \text{romein}(x) \\ \textbf{false} \leftarrow \text{romein}(x) \wedge \text{loyaalAan}(x, \text{Cesar}) \wedge \text{probeerTeVermoorden}(x, \text{Cesar}) \\ \text{loyaalAan}(x, \text{loyaalPersoon}(x)) \\ \textbf{false} \leftarrow \text{mens}(x) \wedge \text{heerser}(y) \wedge \text{probeerTeVermoorden}(x, y) \wedge \text{loyaalAan}(x, y) \\ \text{probeerTeVermoorden}(\text{Marcus}, \text{Cesar}) \\ \text{mens}(x) \leftarrow \text{man}(x) \end{array} \right. \tag{59}$$

6.4 Logisch Programmeren

Naast het bewijzen van stellingen worden Automatische Redeneersystemen ook gebruikt bij **Logisch Programmeren**. Hierbij beperken we ons redeneersysteem opnieuw tot Horn Clauses, maar voegen drie belangrijke componenten toe die Logisch programmeren mogelijk moeten maken:

- Het teruggeven van **Antwoord Substituties**
- De **Kleinste Model Semantiek**, in plaats van de standaard Eerste Orde Logica semantiek
- Uitbreiding van de Horn Clause Logica met **Negatie als Eindige Faling**

Deze aspecten worden in de volgende subsubsecties behandeld.

6.4.1 Antwoord Substituties

Meestal zijn we in een programma niet geïntresseerd of onze stelling F klopt, maar willen we een concrete situatie terugkrijgen waarin F klopt. In de meeste gevallen willen we zelfs alle mogelijke antwoorden terugkrijgen. Hierdoor vormen Logische Programmeertalen de bais van enkele General Purpose Programmeertalen zoals Prolog, XSB en Haskell. In theorie kunnen deze talen zelfs resultaten teruggeven met een efficiëntie gelijkaardig of zelfs sneller dan C. Een praktijkvoorbeeld bij een project leert echter dat dergelijke talen er makkelijk een zevenvoud aan tijd voor nodig hebben in verhouding tot objectgerichte talen!

Hieronder volgt een concreet voorbeeld die de werking van Antwoord Substitutie illustreert. Hierbij is onze theorie

$$\left\{ \text{ dubbelPlus1}\,(x, y) \leftarrow y = 2 \cdot x + 1 \right. \tag{60}$$

Indien we volgende expressie invoeren in een Automatisch Redeneersysteem, zal dit enkel antwoorden met "**true**". Meestal zijn we hier echter niet in geïntresseerd, en willen we de waarde van z kennen. Een logische programmeertaal zal dan ook antwoorden met "$\mathbf{true}, z = 11$":

$$\mathbf{false} \leftarrow \text{dubbelPlus1}\,(5, z) \tag{61}$$

6.4.2 Kleinste Model Semantiek

Indien uit een theorie T niet volgt dat F, dan zal volgens de klassieke Eerste Orde Logica het resultaat zijn dat we het eenvoudig niet kunnen weten. Het idee van de Kleinste Model Logica is dat indien we iets niet weten, we er eenvoudigweg van uitgaan dat het niet waar is. Concreet betekent dit dus dat het model in Eerste Orde Logica zegt dat we weten wat we weten en daarbuiten we niets weten. Logisch Programmeren gaat er vanuit dat alles wat we niet weten per definitie niet waar is. Hierdoor kunnen uiteraard vreemde situaties ontstaan waarbij zowel $T \models F$ als $T \models \neg F$ allebei als **false** worden beoordeeld.

Dergelijke filosofie wordt ook wel de **Gesloten Wereld Assumptie** genoemd. Het is een compacte manier om **Volledige Kennis** over iets uit te drukken (we gaan er immers vanuit dat er naast de theorie T, in de wereld geen kennis bestaat). En dus indien niet gezegd werd dat iets waar is, het per definitie niet waar is. Logisch programmeren ondersteund dan ook het formuleren van definities van je concepten. Alles wat niet aan de definities beantwoord kan dan niet aan het concept voldoen. Dit gaat in tegen het **Monotoniciteitsprincipe** dat zegt dat we alleen maar iets mogen zeggen waarover we iets weten.

Voordeel Deze assumptie mag vreemd lijken, maar laat ons toe om concepten volledige te **axiomatiseren**. In Logisch Programmeren kunnen immers de natuurlijke getallen volledig geaxiomatiseerd worden. Iets wat niet in Eerste Orde Logica kan (door de Onvolledigheidsstelling van Kurt Gödel).

6.4.3 Negatie als Eindige Faling

In grote lijnen komt de representatiekracht van Logisch Programmeren overeen met Horn Clausale Logica. Een poging om hieraan te ontsnappen is de Negatie als Eindige Faling. Hierbij breiden we de representatiekracht uit met volgende items:

- Disjuncties ($\vee$) worden toegelaten in de hoofden
- Negaties ($\neg$) in de atomen van het lichaam

Indien we ons de beperkingen van de Horn Clausale Logica herinneren lijkt het alsof we de taal uitgebreid hebben naar de volledige eerste orde logica. Bovendien hoeft men maar één van de twee regels toe te laten (deze zijn immers equivalent aan elkaar). Omdat we echter werken met de Kleinste Model Semantiek zullen we toch een situatie bekomen die er enigsinds anders uitziet.

De negatie van een bepaald atoom betekent hier immers niet de standaard negatie (zoals deze voorgesteld wordt in een waarheidstabel). De negatie van een bepaald atoom is waar in de Kleinste Model Semantiek indien alle pogingen om dat atoom te bewijzen falen. Of formeler gesteld:

theorema 9. *Als alle pogingen om B te bewijzen, gebruik makend van de* **Linear Programming Resolutie** *(gelijkaardig aan de Überalgemene Modus Ponens) na eindige tijd allemaal falen, besluiten we $\neg B$.*

Concreet betekent dit dus dat indien we een theorie $T = \{A \leftarrow \neg B\}$ hebben, en we willen **false** $\leftarrow A$ bewijzen. Het algoritme "**true**" zal teruggeven. We weten immers niets over B, dus redeneert Linear Programming weten we dat $\neg B$ klopt, en dus klopt A.

6.4.4 Combinatie van Eerste-Orde Logica en Logisch Programmeren

Logisch Programmeren is dus heel nuttig indien we volledig kennis hebben over onze concepten. Eerste Orde Logica is nuttig indien we beperkte kennis hebben. Het komt er dus op aan om de twee te combineren. Dit resulteert in **Open Logisch Programmeren**, hierbij is het de bedoeling dat we Logisch Programmeren indien we volledige kennis hebben over het concept. Indien we een concept behandelen waar we onvolledige kennis van hebben, zoeken we onze toevlucht tot Eerste Orde Logica.

6.4.5 Constraint Logic Programming

Meestal worden heel wat technieken uit Constraint Processing gebruikt bij logisch programmeren (zoals bijvoorbeeld Backtracking). In dat geval spreken we over **Constraint Logic Programming**. Dit concept bied het beste uit de twee werelden. Zo is het vaak eenvoudiger om kennis in logica uit te drukken, dan met constraints. Anderzijds erven we de efficiëntie van Constraint Processing over. Er zijn dan ook al heel wat talen hierop gebaseerd zoals: CHIP, Prolog III, Eclipse,...

7 Metaheuristieken

> *Ik heb geen oplossing, maar ik bewonder het probleem.*
>
> - Anthelme Brillat Savarin, Frans politicus en advocaat (1755-1826)

Een tak van de Arificiële Intelligentie die zich vooral bezighoudt met optimalisatieproblemen is de **Metaheuristiek** (niet te verwarren met Heuristieken bij zoeken). Optimalisatieproblemen zoals bijvoorbeeld het **Traveling Salesman Problem (TSP)**, het **Knapsack Problem**, enzovoort. Hierbij is het niet de bedoeling dat we een oplossing vinden (zoals in de voorgaande secties eerder het geval was). Maar waarbij we dynamisch tussen oplossingen kunnen wisselen, en waar we op zoek zijn naar een oplossing die één of meerdere parameters optimaliseert. Problemen die moeilijk te implementeren constraints op de oplossing opleggen zijn dus beter te vermijden. Typisch zal een Metaheuristiek dit probleem proberen op te lossen in een polynomiale tijd. Terwijl de problemen meestal een exponentieel gedrag kennen, indien we de optimale oplossing willen kennen. Bijgevolg zal een Metaheuristiek niet altijd de meest optimale oplossing vinden, maar meestal een acceptabele oplossing in ruil voor een realistische rekentijd.

Algemeen kunnen we dus stellen dat Metaheuristieken hoofdzakelijk bedoeld zijn om **NP-Complete Problemen** op een redelijke manier op te lossen in min of meer constante tijd (de meeste Metaheuristieken zullen immers na een bepaalde tijd meestal niet meer met betere resultaten komen). Verder hebben de meeste Metaheuristieken een toevalscomponent, dit **Stochastisch gedrag** resulteert over het algemeen in een beter en sneller resultaat. In tegenstelling tot deterministische heuristieken die af en toe in vallen kunnen terechtkomen waar ze niet uitgeraken. Metaheuristieken zijn ten slotte ook algemeen en niet probleem-specifiek. De meeste Metaheuristieken vragen een functie die de **Utility-waarde** van een oplossing weergeeft een eventueel een set aan constraints waaraan een oplossing hoort te voldoen.

Hoewel het concept nog behoorlijk recent is (De term "Metaheuristiek" duikt voor het eerst op in 1986), bestaan er tientallen verschillende technieken. Metaheuristieken aan zich zijn dan ook eerder een "Framework" of een set van ontwerp richtlijnen voor het oplossen van een probleem. De meeste van deze Metaheuristieken zijn bovendien gebaseerd op fenomenen uit de Natuurkunde, Biologie, Wiskunde of Scheikunde. Hieronder een beperkte lijst met metaheuristieken:

- **Genetische Algoritmen** (zie 7.2 op pagina 78)
- **Evolutionary Algoritms**
- **Tabu Search** (zie 7.3 op pagina 80)
- **Simulated Annealing** (zie 7.4 op pagina 81)
- **Great Deluge**
- **Hyperheuristics** (zie 7.6 op pagina 85)
- **Variable Neighbourhood Search** (zie 7.5 op pagina 82)
- **Late Acceptance**
- **Extreme Optimisation**
- **Neural Networks**
- **Electrostatic Potential**

7.1 Leidend Voorbeeld: Het binair knapzakprobleem

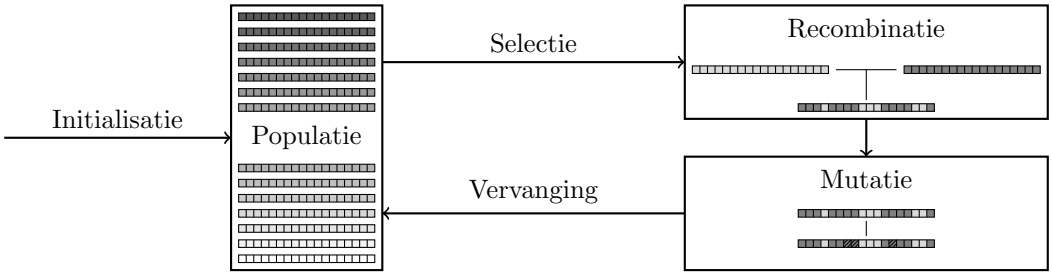
Als illustrerend voorbeeld kiezen we voor het binair knapzakprobleem. Hierbij beschikken we over een knapzak en een reeks objecten. Een knapzak heeft een bepaalde capaciteit c. Elk object i wordt gekenmerkt door een utility-waarde u_i en door een gewicht w_i. Het is de bedoeling een zo groot mogelijke utility in de knapzak te plaatsen, waarbij het totale gewicht de capaciteit niet overschrijdt. Of formeler:

$$\begin{cases} \max \sum_i a_i \cdot b_i \\ \sum_i w_i \cdot b_i \leq c \end{cases} \tag{62}$$

Hierbij is b een lijst van booleans indien element i zich in de knapzak bevindt is $b_i = 1$ anders is $b_i = 0$.

7.2 Genetische Algoritmen

Genetische Algoritmen zijn gebaseerd op de genetica die ontwikkeld werd door Darwin[10] en Mendel[11]. Hierbij zien we een oplossing als een individu. Dit individu moet om te zetten zijn in een lijst van bits met vaste lengte. Dit kunnen we als het DNA van het individu beschouwen. Het algoritme werkt in 6 fases: Initialisatie, Evaluatie, Selectie, Recombinatie, Mutatie en Vervanging. Deze worden hieronder kort uitgelegd. En geschematiseerd op figuur 31. Daarna volgen nog enkele bemerkingen.



Figuur 31: Concept van genetische algoritmen.

7.2.1 Initialisatie

Bij de **Initialisatie**-stap crieëren we een set van verschillende oplossingen, die we de **populatie** of **population** noemen. Deze populatie wordt meestal per toeval gegenereerd. Anderzijds kan deze uiteraard gegenereerd worden op basis van probleemkennis.

7.2.2 Evaluatie

Vervolgens evalueren we in de **Evaluatie**-stap alle individuen in de populatie. Dit doen we hoofdzakelijk op basis van hun utility-waarde. Deze wordt ook wel **fitness value** genoemd.

7.2.3 Selectie

Uit de populatie kiezen we in de **Selectie** enkele goede oplossingen. Hierbij hoeven we echter niet per definitie de beste individuen te selecteren. Soms is het beter, om nog een extra toevalscomponent te laten spelen, waardoor we soms gewoon een goede kandidaat uitkiezen.

7.2.4 Recombinatie

De kandidaten die we uit de populatie gekozen hebben, zullen we vervolgens **recombineren**. Hierbij crieëren we nieuwe oplossingen uit twee bestaande oplossingen, door karakteristieken uit de twee DNA's te combineren, en daaruit nieuwe DNA's te bouwen.

Hoe worden twee bit-lijsten nu concreet gecombineerd? Meestal wordt hiervoor een techniek gebruikt die we **Crossover** noemen. Hierbij hebben we vooraf bepaalde punten bepaald, k-**points** genoemd, waarbij we de bron veranderen. Afwisselend tussen deze punten levert ofwel de ene ofwel de andere ouder de code voor het kind. Meestal wordt hierbij een ander kind gegenereerd, die bij ieder stuk de code van de ouder andere krijgt. Deze techniek is bovendien uitbreidbaar zodat een kind meer dan twee ouders kan hebben. Een andere methode die ook wel eens gehanteerd wordt is dat bij iedere bit elke ouder 50% kans maakt om deze te leveren. Bij een nog andere methode overlopen we de bitlijst, en bij een zekere kans beschouwen we een Crossover.

[10] Charles Robert Darwin (1809-1882)
[11] Gregor Johann Mendel (1822-1884)

7.2.5 Mutatie

Meestal wordt er enige **mutatie** toegepast op de nieuwe kinderen. Hierbij wordt het DNA door middel van toeval licht gewijzigd. Dit heeft meestal tot doel om bij populaties die erg monotoon zijn toch de nodige "inspiratie" te genereren.

Indien alle ouders op eenzelfde bit dezelfde waarde hebben, zal deze bit bij het kind ook deze waarde krijgen. Het probleem is echter dat indien het grootste (en vooral beste) deel van de populatie deze waarde deelt, we op den duur deze waarde niet meer kunnen veranderen. Dit kan resulteren in het feit dat we een collectie aan slechte oplossingen onderzoeken. Een oplossing voor dit probleem is om sommige bits van de gegenereerde kinderen te doen veranderen. Een zogenoemde **flipover**. Dit doen we door iedere bit te overlopen en indien een toevallig gegenereerd getal aan een bepaalde conditie voldoet (onder een zeer kleine kans), veranderen we de waarde van deze bit naar het omgekeerde.

7.2.6 Vervanging

In de **vervanging**sfase worden de kinderen in de populatie opgenomen, meestal worden er vervolgens individuen uit de populatie gezet, zodat de populatie niet al te groot wordt. Het verwijderen van individuen gebeurt opnieuw meestal op basis van de fitness-waarde. Maar ook hier weer hoeft dit geen deterministisch proces te zijn. Nadat een nieuwe generatie oplossingen in de populatie is opgenomen, beginnen we opnieuw met de evaluatie, en begint de lus opnieuw. Tijdens de uitvoer van het algoritme wordt meestal de tot dan toe beste oplossing telkens bijgehouden. Na een aantal maal het hele proces herhaald te hebben. Wordt dit resultaat vervolgens teruggegeven als eindoplossing.

In de praktijk worden er zo'n drietal vervangingsstrategieën gebruikt:

- **Delete All** We verwijderen de volledige populatie, en voegen de niewe kinderen toe als nieuwe populatie.
- **Steady-State** Hierbij verwijderen we telkens n oude leden uit de populatie, de criteria hier voor zijn opnieuw zelf in te stellen (ouders, slechtste, beste,...) en voegen de nieuwe kinderen toe.
- **Steady-State No Duplicates** Soms kunnen gegenereerde kinderen kopieën zijn van reeds bestaande individuen, in dat geval stijgt de kans op selectie, wat eventueel de evolutie van de populatie kan ondermijnen (indien een bepaald individu succesvol is, riskeren we dat na enige tijd de hele populatie uit dat individu bestaat). Hierbij worden dus de duplicaten verwijdert.

7.2.7 Bemerkingen

Genetische Algoritmes zijn vooral geschikt voor **sheduling** (het opstellen van planningen). Maar ze convergeren gegarandeert naar de meest optimale oplossing voor ieder probleem. Meestal wordt een genetisch algoritme hierbij niet alleen toegepast maar wordt het gecombineerd met een ander techniek zoals bijvoorbeeld Local Search. Dit produceert in het algemeen betere resultaten. In dat geval spreken we van **Memetic Algorithms**.

Parameters Parameters die we meestal zelf kunnen kiezen bij genetische algoritmes zijn:

- De grootte van de populatie
- Kans op mutatie
- Crossover: k-punten of de kansen op een crossover.

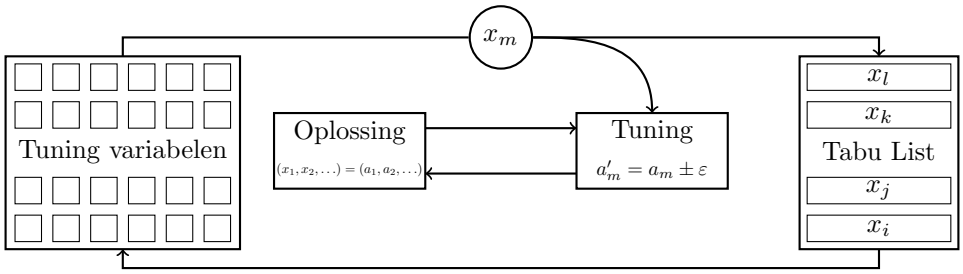
Deze parameters worden meestal bepaald door experimenten op test-data (waarbij deze parameters vaak toevallig gezet worden, en de resultaten met andere vergeleken worden), in dat geval zijn ze statisch. Soms zijn ze echter dynamisch, en zal het algoritme ze zelf aanpassen, indien de gebruiker bijvoorbeeld niet tevreden is.

Door de binaire lijst b_i als het DNA voor een oplossing te nemen bekomen we een eenvoudige representatie. We vullen initieel de populatie door een collectie aan toevallige binaire lijsten te genereren. Oplossingen die niet aan de gewichtsvoorwaarde voldoen worden eenvoudigweg uit de oplossing verwijdert. Als selectie nemen we eenvoudigweg de utility-waarde van de knapzak. Dit kan eventueel aangevuld worden met een stochastisch element, waardoor ook niet optimale oplossingen kunnen geselecteerd worden. Als recombinatie wordt per element in de nieuwe binaire lijst voor 50% van de gevallen de ene ouder en in het ander geval de andere ouder als bron gebruikt. Eventueel kunnen we dus ook meerdere kinderen uit dezelfde ouders genereren. Als mutatie zullen we een bit omdraaien in de lijst met een kans van 0.1%. Oplossingen die niet aan de voorwaarden voldoen worden niet toegevoegd aan de populatie. Als vervangstrategie gebruiken we Steady-State.

7.3 Tabu Search

Tabu Search is een techniek waarbij we ervan uitgaan dat oplossingen die dicht bij elkaar liggen, meestal ongeveer dezelfde Utility-factor bezitten. We zouden dus de ruimte van oplossingen als een landschap kunnen zien, waarbij goede oplossingen een heuvel voorstellen. Tabu Search probeert hierbij op zoek te gaan naar de top van de hoogste berg. Indien we dit met behulp van een heuristiek zouden oplossen (we nemen de richting waarin we het snelst stijgen) dreigen we echter al snel op een lokaal maximum te stuiten. Een plaats waarbij alle dichte oplossingen slechter zijn, maar waarbij de eigen oplossing niet de beste is. Ook Simulated Annealing (zie 7.4) werkt op basis van hetzelfde principe.

De oplossing die Tabu Search biedt, is gebaseerd op het principe van **tuning**. We bekijken nu een oplossing als een lijst van parameters (uiteraard kunnen we voor iedere parameter een bit nemen, en bekomen we opnieuw een lijst van bits). we zorgen er voor dat onze initiële oplossing aan alle constraints voldoet, vervolgens passen we een bepaalde parameter aan zodat we een nieuwe geldige toestand bekomen. Deze parameter wordt vervolgens in een lijst opgenomen, de **Tabu List**. Dit is een lijst van parameters (en dus bewegingen in de oplossing) die tijdelijk niet mogen veranderen. Bij de volgende keuze van de parameter, mogen we alle parameters kiezen behalve deze in de Tabu List. Uiteraard worden de Tabu parameters na enige tijd weer verwijdert uit de lijst. Meestal heeft de lijst een vaste grootte die meestal vastgelegd wordt op een bepaalde factor van het aantal parameters. Bij de keuze welke parameters de lijst mogen verlaten indien de lijst overloopt zijn er enkele technieken: De parameter die er het langst in zit (de lijst is in dat geval een Queue), of een toevallige parameter zijn hierbij de populairste keuzes. Figuur 32 vat het concept achter Tabu Search schematisch samen.



Figuur 32: Concept van Tabu Search.

7.3.1 Bemerkingen

Tabu Search is een techniek die vooral gebruikt wordt bij ruimteproblemen zoals het berekenen van de kortste route, het vullen van magazijnen,...

De implementatie van Tabu Search voor het knapzakprobleem is relatief eenvoudig. Als variabelen beschouwen we alle elementen uit de binaire rij. We zullen dus eenvoudigweg een willekeurig element omdraaien, en vervolgens de index hiervan aan de Tabu List toevoegen. Als grootte van de Tabu List nemen we bijvoorbeeld 10% van de lengte van de binaire rij.

7.4 Simulated Annealing

Ook Simulated Annealing, ontwikkeld in 1983 door Kirkpatrick[12], maakt gebruik van het dichte oplossings principe. Hierbij werd de inspiratie gehaald bij het stollen van kristallen. Kristallen verkrijgen bij een traag stollingsproces, de meest optimale geometrische structuur op atomair niveau. Dit effect probeert men met Simulated Annealing na te bootsen. Een kristal streeft immers naar een zo minimaal mogelijke potentiële energie. Hoewe we bij optimalisatie meestal op zoek zijn naar de maximale waarde, is het gebruik van een eenvoudig minteken voor de utility-functie, voldoende om van een optimaal minimum naar een optimaal maximum te gaan.

De **Vaste Stoffen Fysica** ofwel **Solid State Physics** stelt dat de kans op een overgang tussen twee verschillende geometrieën g_1 en g_2 gekenmerkt wordt door volgende expressie:

$$P(g_1 \to g_2) = \begin{cases} 1 & \text{if } U(g_2) \leq U(g_1) \\ e^{U(g_1)-U(g_2)/k_B \cdot T} & \text{otherwise} \end{cases} \tag{63}$$

De geometrieën zelf liggen over het algemeen niet ver uit elkaar (in de fysische wereld verschillen ze hoogstens enkele kwantum-niveaus). Verder is $U(g)$ de potentieële energie van een bepaalde geometrie g, k_B is de Boltzmann constante en T is de temperatuur in Kelvin op dat moment. We merken hierbij op dat indien we naar een energetisch lagere geometrie gaan de kans $P = 1$, en dus bijgevolg deze transactie altijd ondersteund wordt. Indien we naar een hogere energetische waarde gaan is de kans eerder beperkt. Deze kans hangt dan ook grotendeels af van de heersende temperatuur: onder extreem hoge temperatuur worden alle overgangen nagenoeg toegestaan, bij zeer lage temperaturen zijn dergelijke overgangen quasi onmogelijk. Indien we de temperatuur maar traag genoeg doen zakken zal de geometrie bij iedere temperatuur een staat bereiken die we **Thermal Equilibrium** noemen. Hierbij is de kansverdeling voor iedere mogelijke transitie gegeven door:

$$P(X = i) = \frac{e^{-U_i/k_B \cdot T}}{\sum_j e^{-U_j/k_B \cdot T}} \tag{64}$$

Het **Metropolis Algoritme** is een gekende generator van nieuwe staten in een problemen vertrekkende vanaf een bepaalde staat. Door dit concept toe te passen op een oplossingsverzameling bereiken we meestal zeer goede resultaten, indien we eerst onze staat naar een Thermisch Evenwicht laten convergeren kunnen we zelfs de meest optimale oplossing bereiken. Formeel kunnen we dit concept uitdrukken in **Algorithm 33**. Een conceptueel schema staat op figuur 33. Hierbij is c een controleparameter die we

Algorithm 33 Simulated Annealing

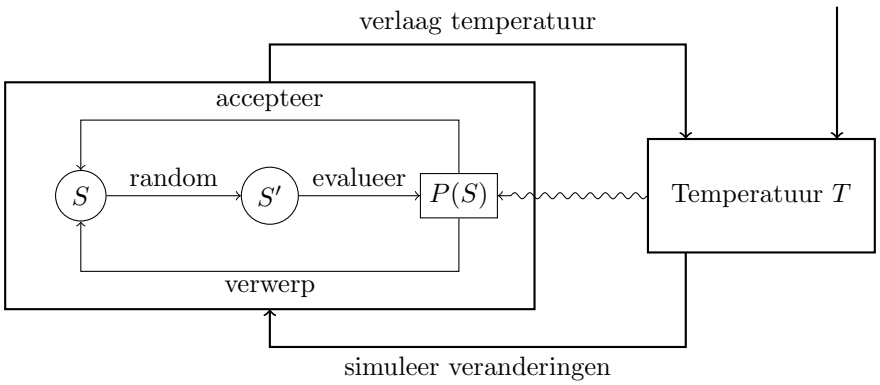
```
1: S ← S_start {S is de huidige oplossings-staat}
2: c ← c_i {zet de controle parameter op een initiele waarde}
3: L ← L_i {initiële herhalingslengte}
4: while c > c_f do
5:     for l = 1 to L do
6:         S_j ← generateNextRandomStep(S)
7:         if random(0,1) < e^{f(S_j)−f(S)/c} then
8:             S ← S_j
9:         end if
10:    end for
11:    c ← changeControl(c)
12:    L ← changeLength(L)
13: end while
```

fysisch kunnen zien als $k_B \cdot T$.

7.4.1 Bemerkingen

Simulated Annealing is een techniek die vooral intressant is bij problemen met grafen (kleuren van grafen) en sheduling. Daarintegen blijkt Simulated Annealing niet echt geschikt te zijn voor problemen

[12]Scott Kirkpatrick



Figuur 33: Concept van Simulated Annealing.

zoals het Handelsreizigersprobleem. In de praktijk wordt het dan ook bijvoorbeeld gebruikt voor Image Processing. Hierbij werkt het meestal beter dan tijdsequivalente methodes, en ondergaat het minder problemen wanneer de grootte van het probleem opgedreven wordt. Een probleem is echter dat de kwaliteit van belangrijke factoren zoals het regelen van de temperatuur, en het bepalen van wat dichte oplossingen zijn, hoofdzakelijk bepaald worden door de vaardigheden van de programmeur. Een foute keuze kan tot zeer slechte resultaten leiden.

We gaan van een toestand in een andere door een object toe te voegen aan onze knapzak, en indien hierbij de gewichtvoorwaarde geschonden wordt toevallige objecten uit de knapzak te halen tot het gewicht weer klopt. Als begintemperatuur stellen we bijvoorbeeld 300 K in. Per temperatuur zullen we 25 keer van configuratie veranderen. Vervolgens vermenigvuldigen we de temperatuur met 0.9. We laten het algoritme stoppen indien de temperatuur onder de 1 K. In totaal zullen we dus 1375 van toestand proberen te veranderen.

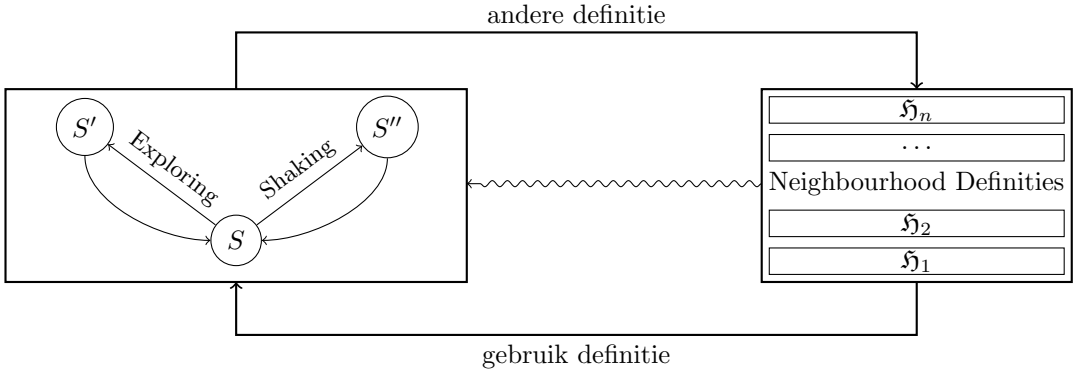
7.5 Variable Neighbourhood Search

Net als Tabu Search en en Simulated Annealing beroept ook Variable Neighbourhood Search zich op het lokaliteitsprincipe. Het stelt dus dat goede oplossingen meestal in de buurt liggen van elkaar. Om echter het probleem met lokale maxima op te lossen, probeert men niet het algoritme te dwingen naar alternatieven te kijken, het stelt gewoon de op dat moment geldende definitie van buurt (lokaliteit) in vraag. Indien we deze definitie veranderen kunnen twee oplossingen die eerst dicht bij elkaar lagen, ineens geen echte band meer hebben. Indien we dus op een lokaal maximum botsen, kunnen we eenvoudigweg door de definitie om te vormen een nieuw landschap bouwen waar het uiteindelijk de bedoeling is om de hoogste berg te beklimmen. Bij iedere definitie zoeken we telkens naar de beste buur, dit proces wordt **Exploring** genoemd. Het algoritme is formeel te beschreven in **Algorithm 34**. Meestal ordent men hier

Algorithm 34 Variable Neighbourhood Search

```
1: ℌ ←{Set van neighbourhood structuren}
2: x ← x_0{Initiele oplossing}
3: l ← 0
4: while l < #ℌ do
5:     x' ← bestNeighbourByStructure (x, ℌ [l])
6:     if utility (x') > utility (x) then
7:         l ← 0
8:         x ← x'
9:     else
10:        l ← l + 1
11:    end if
12: end while
```

de neighbourhood definties zodaning dat de eerst een kleine regio rond de huidige oplossing onderzoekt, en dus snel tot een resultaat komt. De laatste definitie zoekt daarintegen meestal een zeer groot gebied af. Verder wordt dit algoritme meestal herhaalt in een extra lus die pas stopt indien aan een bepaald terminatiecriterium voldaan is. Naast het zonet beschreven algoritme bestaan er heel wat varianten van Variable Neighbourhood Search, de belangrijkste sommen we hieronder op. Het algemene concept van Variable Neighbourhood Search wordt geschematiseerd in figuur 34.



Figuur 34: Concept van Variable Neighbourhood Search.

7.5.1 Variable Neighbourhood Search Reduced

Variable Neighbourhood Search Reduced is een snellere variant. Hierbij wordt niet gezocht naar de beste buur, maar kiezen we gewoon een toevallige buur. Het generen van een toevallige buur wordt **Shaking** genoemd. Alsof een beving in het landschap ons naar dichtbij gelegen punt brengt. Uiteraard leidt dit er toe dat we vaker van Neighbourhood Structure moeten wisselen. Anderzijds kan het generen van een random buur meestal in $\mathcal{O}(1)$ gedaan worden, terwijl het zoeken naar de beste buur $\mathcal{O}(n)$ vereist (waarbij n zeer groot kan zijn). Dit algoritme komt meestal tot een minder goede oplossing, maar werkt meestal opvallend sneller. Bovendien kan het resultaat die we hier mee bereikt hebben opnieuw ingegeven worden in een andere implementatie van Variable Neighbourhood Search die de oplossing verder optimaliseert. Dit algoritme wordt beschreven in **Algorithm 35**.

Algorithm 35 Variable Neighbourhood Search Reduced

```
1: 𝔥 ←{Set van neighbourhood structuren}
2: x ←{Initiele oplossing}
3: l ← 0
4: while l < #𝔥 do
5:     x' ← randomNeighbourByStructure(x, 𝔥[l])
6:     if utility(x') > utility(x) then
7:         l ← 0
8:         x ← x'
9:     else
10:        l ← l + 1
11:    end if
12: end while
```

7.5.2 Variable Neighbourhood Search Basic

Variable Neighbourhood Search Basic is een combinatie van Variable Neighbourhood Search en Variable Neighbourhood Search Reduced. Hierbij genereren we een toevallige buur, waarna we vanuit deze buur een beste buur zoeken. Deze tweede buur wordt dan vervolgens geëvalueerd en verworpen of aangenomen. Dit staat beschreven in **Algorithm 36**.

Algorithm 36 Variable Neighbourhood Search Basic

```
1: ℌ ←{Set van neighbourhood structuren}
2: x ←{Initiele oplossing}
3: l ← 0
4: while l < #ℌ do
5:     x′ ← randomNeighbourByStructure (x, ℌ [l])
6:     x″ ← bestNeighbourByStructure (x′, ℌ [l])
7:     if utility (x″) > utility (x) then
8:         l ← 0
9:         x ← x″
10:    else
11:        l ← l + 1
12:    end if
13: end while
```

7.5.3 Variable Neighbourhood Search General

Een implementatie van Variable Neighbourhood Search waarij we blijven zoeken tot we $k_{\max}$ keer geen verbetering zien optreden heet **Variable Neighbourhood Search General**. Deze implementatie kan in sommige gevallen zelfs ertoe leiden dat men de omhullende lus die het uiteindelijke stopcriterium bepaald, weglaat. Indien $k_{\max}$ goed gekozen is (te klein levert een slechte oplossing, te groot zorgt voor de nodige overhead), kan dit algoritme in de meeste gevallen het meeste efficiënt met zijn tijd omspringen. Omdat naast de Neighbourhood Structuren we alleen maar een parameter $k_{\max}$ nodig hebben, kan deze parameter vaak eenvoudig door testdata bepaald worden. **Algorithm 37** beschrijft dit concept formeel.

Algorithm 37 Variable Neighbourhood Search General

```
1: ℌ ←{Set van neighbourhood structuren}
2: x ←{Initiele oplossing}
3: l ← 0
4: k ← 0
5: while k < k_max do
6:     x′ ← x
7:     while l < #ℌ do
8:         x″ ← bestNeighbourByStructure (x′, ℌ [l])
9:         if utility (x″) > utility (x′) then
10:            l ← 0
11:            x′ ← x″
12:        else
13:            l ← l + 1
14:        end if
15:    end while
16:    if utility (x′) > utility (x) then
17:        k ← 0
18:        x ← x′
19:    else
20:        k ← k + 1
21:    end if
22: end while
```

7.5.4 Bemerkingen

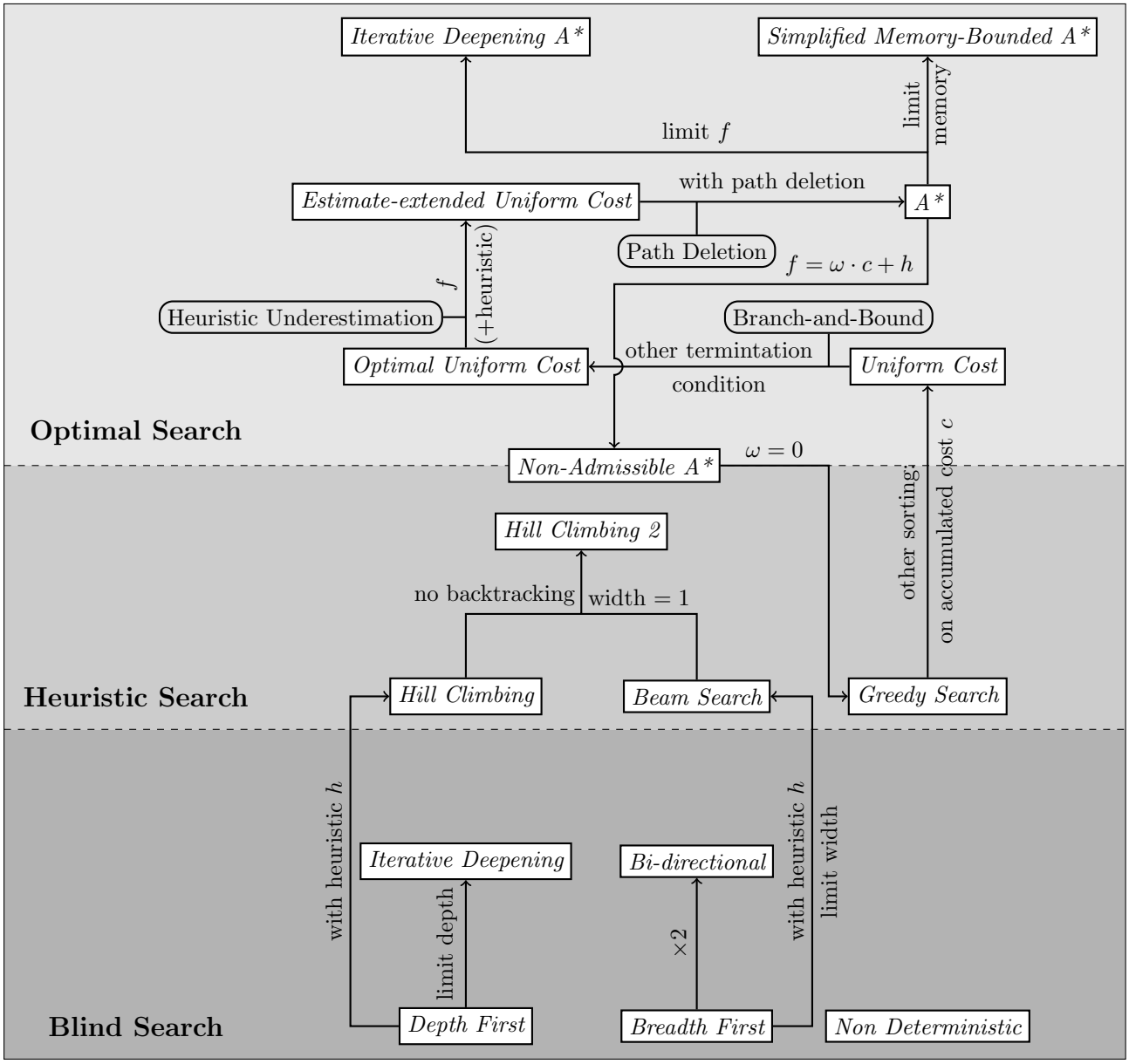
Variable Neighbourhood Search is een techniek die vooral gespecialiseerd is in het oplossen van problemen in verband met grafen, sheduling en locatieproblemen.

7.6 Hyperheuristics

De meeste heuristieken hebben slechts een specialisatie op een bepaald gebied. Dit leid er toe dat we eigenlijk nog steeds geen algemeen algoritme hebben. Hyperheuristics is een poging om tot een algoritme te komen die in principe alle problemen met een zekere optimaliteit kan oplossen. Hyperheuristics zelf is echter geen algoritme, het is een techniek die allerhande Metaheuristieken (zoals deze die in de vorige subsecties besproken werden) combineert, om zo telkens tot de optimale mix te komen. Hierbij definiëren we een set van Metaheuristieken die een bepaald probleem kunnen oplossen. Door deze Metaheuristieken op enkele testproblemen te laten lopen, komen we te weten hoe goed iedere Metaheuristiek presteert op het soort probleem dat we willen oplossen. Het komt er dan ook op neer dat we de heuristieken op de één of andere manier gaan combineren met elkaar. Volgens een bepaalde keuzefunctie kunnen we dan vervolgens selecteren hoe vaak we een bepaalde Metaheuristiek op het probleem loslaten, alvorens we een andere aan de beurt laten. Doormiddel van Machine Learning kunnen we deze keuzefunctie dan verbeteren, tot we een algoritme bekomen die over het algemeen een zeer optimale oplossing teruggeeft.

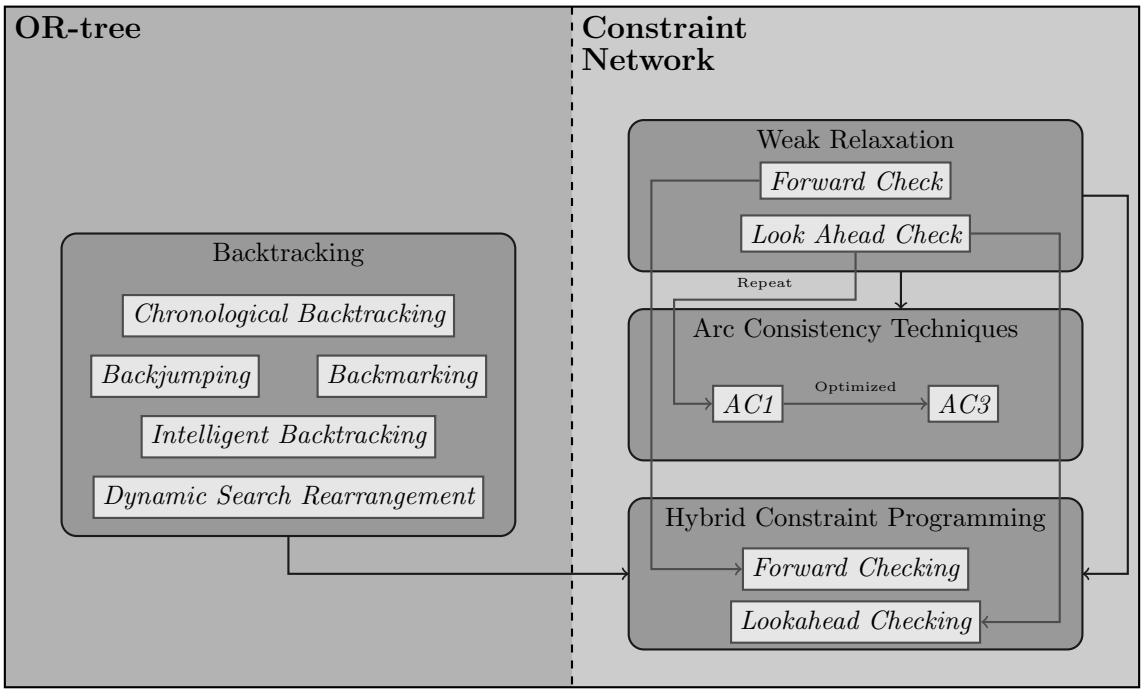
A Samenvattende Schema's

A.1 Zoekmethodes



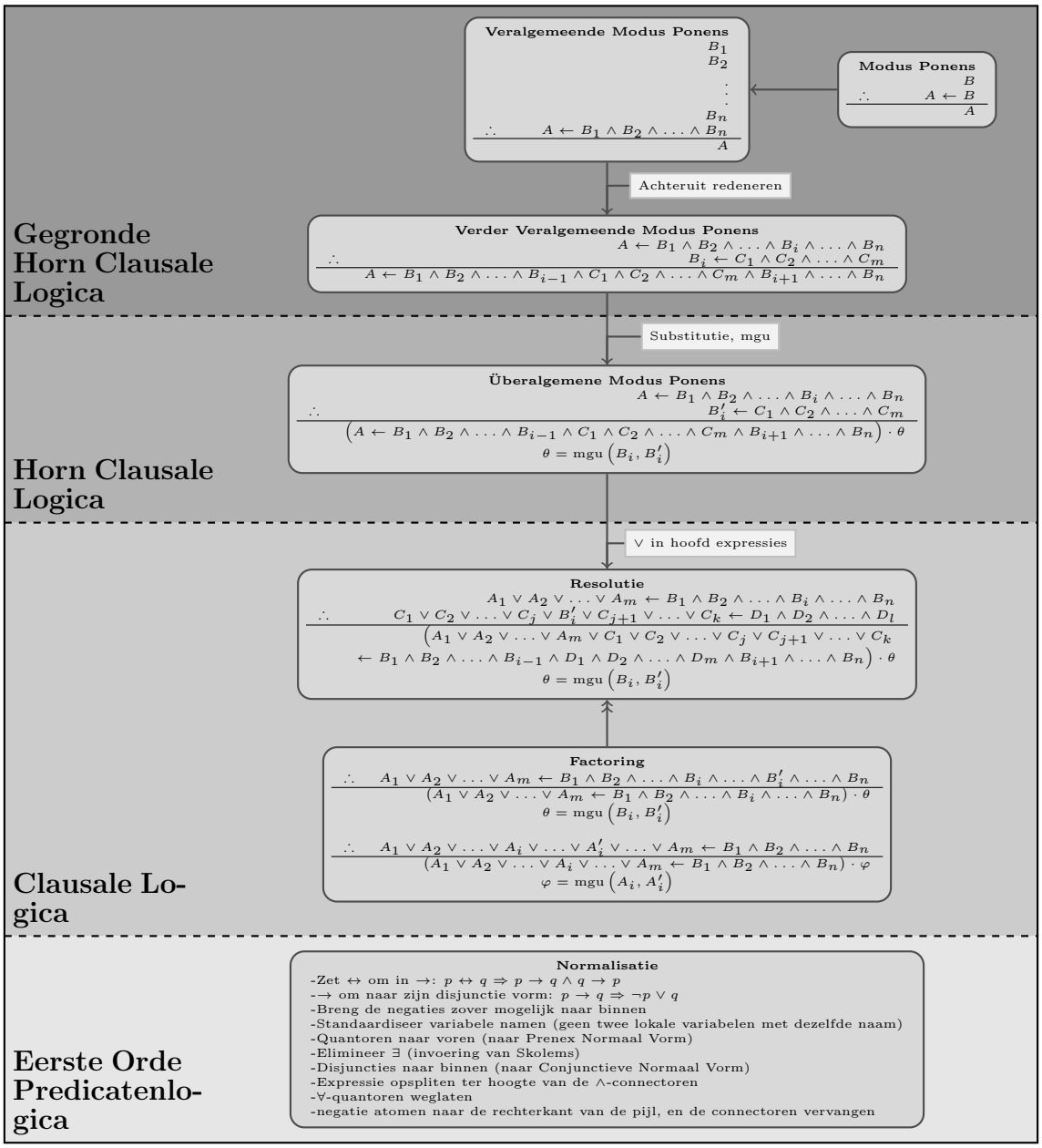
Figuur 35: Samenvattend schema van Zoekmethodes

A.2 Constraint Processing



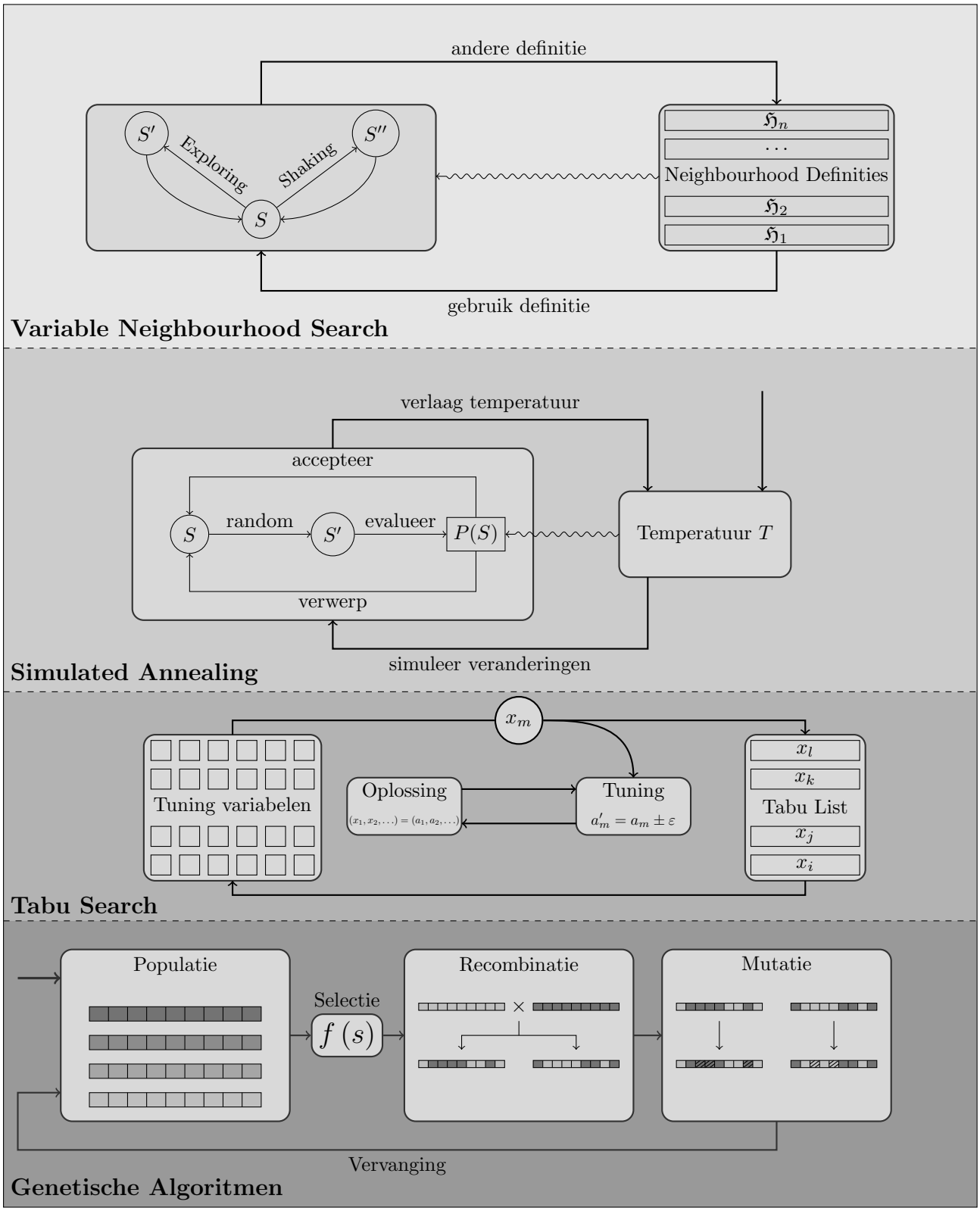
Figuur 36: Samenvattend schema van Constraint Processing

A.3 Automatisch Redeneren



Figuur 37: Samenvattend schema van Automatisch Redeneren

A.4 Metaheuristieken



Figuur 38: Samenvattend schema van Metaheuristieken

Lijst van figuren

Index

“And now something completly different”

Catharsis

$$\Delta x \cdot \Delta p_x \geq \frac{\hbar}{2}$$

Vrijheid van **Heisenbergh**

$$-\frac{\hbar^2}{2m}\frac{\partial^2 \Psi}{\partial x^2} + U \cdot \Psi = E \cdot \Psi$$

Alles gaat in golven van **Schrödinger**

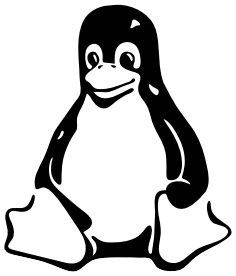
$$x' = \frac{x - v \cdot t}{\sqrt{1 - \frac{v^2}{c^2}}}$$

Alles is relatief van **Einstein**

$$E = m \cdot c^2$$

Alles is energie van **Einstein**

W.V.D.
C.C.C.P.



Figuur 39: Created with GNU/Linux!