Kunstmatige Intelligentie (AI)

Walter Kosters

voorjaar 2024

www.liacs.leidenuniv.nl/~kosterswa/AI/totaal.pdf

DeepNash

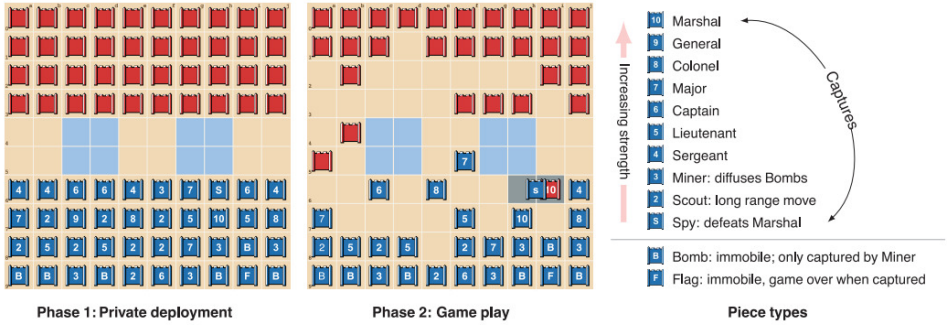
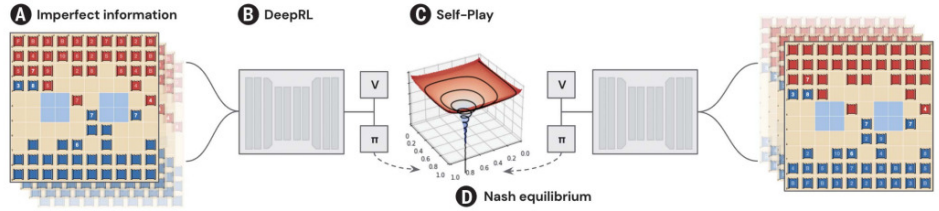


Fig. 1. Stratego is a two-player board game in which players try to capture the opponent's flag. Initially, the players secretly deploy 40 pieces of diverse strengths on the board. Then, they take turns moving pieces, possibly encountering an opponent piece that reveals both piece identities, and then the weaker piece is removed. Two lakes (indicated in blue) cannot be crossed by any piece. The complete rules are defined by the International Stratego Federation.



Replicator dynamics: $\frac{d}{d\tau}\pi^i_\tau(a^i) = \pi^i_\tau(a^i)\left[Q^i_{\pi_\tau}(a^i) - \sum_{b^i}\pi^i_\tau(b^i)Q^i_{\pi_\tau}(b^i)\right]$

Reward transformation: $r^i(\pi^i, \pi^{-i}, a^i, a^{-i}) = r^i(a^i, a^{-i}) - \eta\log\left(\frac{\pi^i(a^i)}{\pi^i_{\text{reg}}(a^i)}\right) + \eta\log\left(\frac{\pi^{-i}(a^{-i})}{\pi^{-i}_{\text{reg}}(a^{-i})}\right)$

breaking news

↓ link

RESEARCH

MACHINE LEARNING

Mastering the game of Stratego with model-free multiagent reinforcement learning

Julien Perolat*†, Bart De Vylder*†, Daniel Hennes, Eugene Tarassov, Florian Strub, Vincent de Boer‡, Paul Muller, Jerome T. Connor, Neil Burch, Thomas Anthony, Stephen McAleer, Romuald Elie, Sarah H. Cen, Zhe Wang, Audrunas Gruslys, Aleksandra Malysheva, Mina Khan, Sherjil Ozair, Finbarr Timbers, Toby Pohlen, Tom Eccles, Mark Rowland, Marc Lanctot, Jean-Baptiste Lespiau, Bilal Piot, Shayegan Omidshafiei, Edward Lockhart, Laurent Sifre, Nathalie Beauguerlange, Remi Munos, David Silver, Satinder Singh, Demis Hassabis, Karl Tuyls*†

We introduce DeepNash, an autonomous agent that plays the imperfect information game Stratego at a human expert level. Stratego is one of the few iconic board games that artificial intelligence (AI) has not yet mastered. It is a game characterized by a twin challenge: It requires long-term strategic thinking as in chess, but it also requires dealing with imperfect information as in poker. The technique underpinning DeepNash uses a game-theoretic, model-free deep reinforcement learning method, without search, that learns to master Stratego through self-play from scratch. DeepNash beat existing state-of-the-art AI methods in Stratego and achieved a year-to-date (2022) and all-time top-three ranking on the Gravon games platform, competing with human expert players.

Perolat et al., Science 378 (2022) 990–996: Mastering the game of Stratego . . .

Deep neural nets, reinforcement learning, selfplay.

Kunnen computers denken? Diplomacy! En ChatGPT?

Huishoudelijke mededelingen

Lectures are in DUTCH!

De colleges zijn op woensdagen, 7 februari tot en met 22 mei 2024 (niet op 27 maart), 13:15–15:00 uur, Gorlaeus zaal C1. De colleges zijn ook achteraf op video te bekijken, zie de website — ook voor oudere video's.
En de werkcolleges: donderdagen, 8 februari tot en met 23 mei 2024 (niet op 28 maart en 9 mei), 13:15–15:00 uur, in de computerzalen 302/306/307 van het Snellius.

Schriftelijk tentamen (tijdig aanmelden in MyStudyMap):

- donderdag 13 juni 2024, 9:00–12:00 uur;
  Universitair Sportcentrum

- hertentamen: maandag 8 juli 2024, 9:00–12:00 uur;
  Universitair Sportcentrum

We maken gebruik van het volgende boek:

Stuart J. Russell en Peter Norvig
Artificial Intelligence, A Modern Approach
fourth edition
Pearson, 2020

Afkorting: [RN]. Er zijn vele andere boeken, zie de website.

Het practicum kent vier opgaven met strikte deadlines:

- 28 februari 2024, 13:15 uur: Monte Carlo en Tetris
- 20 maart, 13:15 uur: Agenten / Robots
- 17 april, 13:15 uur: A*
- 13 mei, 13:15 uur: Neuraal netwerk

Assistentie: Raaf van Beusekom, Xander Lenstra. Justin de Rooij en Michael de Rooij.

We gebruiken de programmeertaal C++, via Linux of Windows-WSL2 of Windows-Code::Blocks of de Mac, zie:

www.liacs.leidenuniv.nl/~kosterswa/pm/

Het eindcijfer E is het cijfer T van het schriftelijk tentamen, *mits* de vier practicumopgaven alle als voldoende beoordeeld zijn; 6 EC. Als $T \leq 5$ is dit het eindcijfer.

Als het gemiddelde practicumcijfer G groter is dan T en $T \geq 5.5$, wordt het eindcijfer E hun gemiddelde, $(G+T)/2$, waarbij dit maximaal 1 punt meer mag worden dan T.

En E wordt afgerond naar het dichtstbijzijnde getal uit de verzameling $\{1, 2, 3, 4, 5,\ 6, 6.5, 7, 7.5, 8, 8.5, 9, 9.5, 10\}$.

Bij elke practicumopgave, te maken in $\leq$tweetallen, moet een circa zes pagina's tellend verslag in LaTeX gemaakt worden (zie `voorbeeldfile` op de website). Globale opzet:

1. Inleiding
2. Uitleg probleem
3. Relevant werk (= Related work)
4. Aanpak
5. Implementatie (kort!)
6. Experimenten (tabel, grafiek)
7. Conclusie (inclusief verder onderzoek)

Referenties

Appendix: programma — eigen code

Het tekstverwerkingssysteem LaTeX is gebaseerd op Donald Knuth's TeX, en draait op "iedere" computer.

Je maakt met een editor naar keuze een ASCII-file `iets.tex` met je eigen tekst, "latex-t" deze met `pdflatex iets.tex` naar een printbare PDF-file `iets.pdf`.
Internet-versie: www.overleaf.com

Handleiding:

www.liacs.leidenuniv.nl/~kosterswa/stuva1/lshort.pdf

met name Hoofdstuk 1, 2 en 3 (plaatjes: 4)
of: lees `sample2e.tex`

```
% commentaar: voorbeeldfile
\documentclass[12pt]{article}
\author{P.~Puk}
\title{Kabouters in de Tweede Kamer}
\frenchspacing

\begin{document}  %\maketitle
\section*{Verkiezingen}
In 2024--25 w\'e\'er \emph{ver-kie\-zingen}
--- geldt $a^n-b_n=c\times{}n$?
```

Verkiezingen

In 2024–25 wéér *ver-kiezingen* — geldt $a^n - b_n = c \times n$?

Programma 2024

#	Datum	Onderwerp	Deadline
1	7.2	Algemene introductie [RN1]	
2	14.2	Intelligente agenten [RN2]	
3	21.2	Logische agenten [RN7;8]	
4	28.2	Probleemoplossen en zoeken [RN3]	Opgave 1
5	6.3	Geinformeerd zoeken [RN4]	
6	13.3	Spel(l)en [RN5]	
7	20.3	idem, vervolg	Opgave 2
8	3.4	CSP's [RN6]	
9	10.4	Leren [RN19]	
10	17.4	Deep learning [RN21]	Opgave 3
11	24.4	idem, vervolg: Reinforcement Learning [RN22]	
12	1.5	Locaal zoeken en Optimalisatie [RN4.1]	
13	8.5	Bayesiaanse netwerken [RN12;13] (geen werkcollege)	
14	15.5	Oude tentamens (geen (werk)college 22/23.5)	Opgave 4

Optioneel: Natural Language Processing [RN23,24], Robotica [RN26].

werkcollege "sommen op papier"

Idee:

- De werkcolleges zijn bedoeld om aan de programmeeropgaven en de bijbehorende verslagen te werken.
- Vier werkcolleges bestaan deels uit het maken van sommen: 7 maart, 4 april, 2 mei en 16 mei.
- Tijdens de werkcolleges: vragen stellen, samenwerken, peer review, milestones, . . .
- Er is aanwezigheidscontrole.

Informele introductie

Kunstmatige intelligentie is een verzamelnaam voor een heel breed vakgebied, met onder andere:

- *robotica*: Hoe programmeer je een robot?
- *data mining*: Wat is verborgen in/op WikiLeaks?
- *rechtspraak*: Kan een machine rechter zijn? NLP ↙
- *linguistiek*: "the spirit is willing but the flesh is weak" $\rightarrow \ldots \rightarrow$ "the vodka is good but the meat is rotten"?
- *computer games*: Kan een computer Fortnite spelen?
- *neurale netwerken*: Kun je beurskoersen voorspellen?
- *cognitie*: Kunnen we het menselijk brein nadoen?

Je kunt op minstens twee manieren naar Kunstmatige intelligentie kijken:

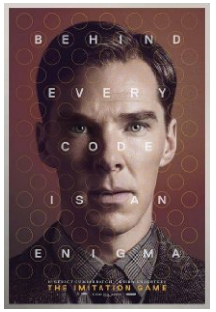
1. vanuit een meer *psychologische* of *filosofische* richting:
   Wat is het verschil tussen een mens en een computer?
   Kan een computer denken?

2. vanuit een meer *technische* richting:
   Hoe werkt een schaakprogramma?
   Hoe werkt een Marsrobot?

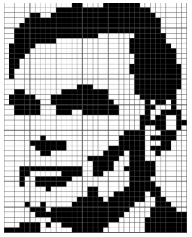
"Do androids dream of electric sheep?" $\longrightarrow$

Kunstmatige intelligentie laat computers zich zo gedragen dat het intelligent zou heten als mensen het op die manier zouden doen.

De beroemde Turing-test uit 1950 vraagt (“the imitation game”):

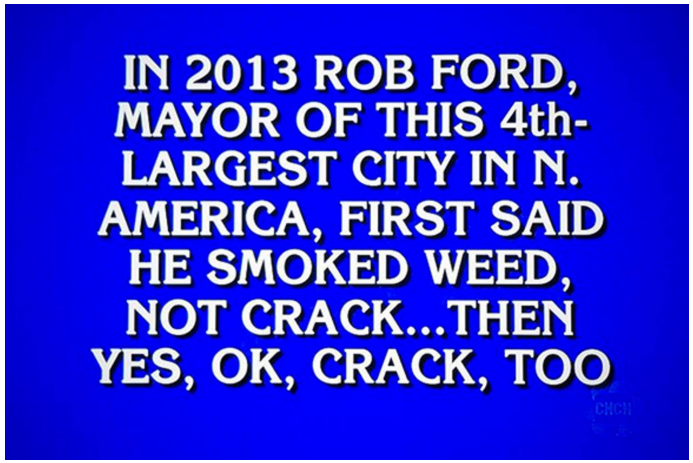


> In een afgesloten kamer bevindt zich een mens *of* een computer, waarmee we alleen via toetsenbord en beeldscherm contact hebben.
> Is het een mens of juist een computer?



Het originele probleem was overigens met man $\leftrightarrow$ vrouw.

IBM heeft in 2011 een computer "Jeopardy!" laten spelen:

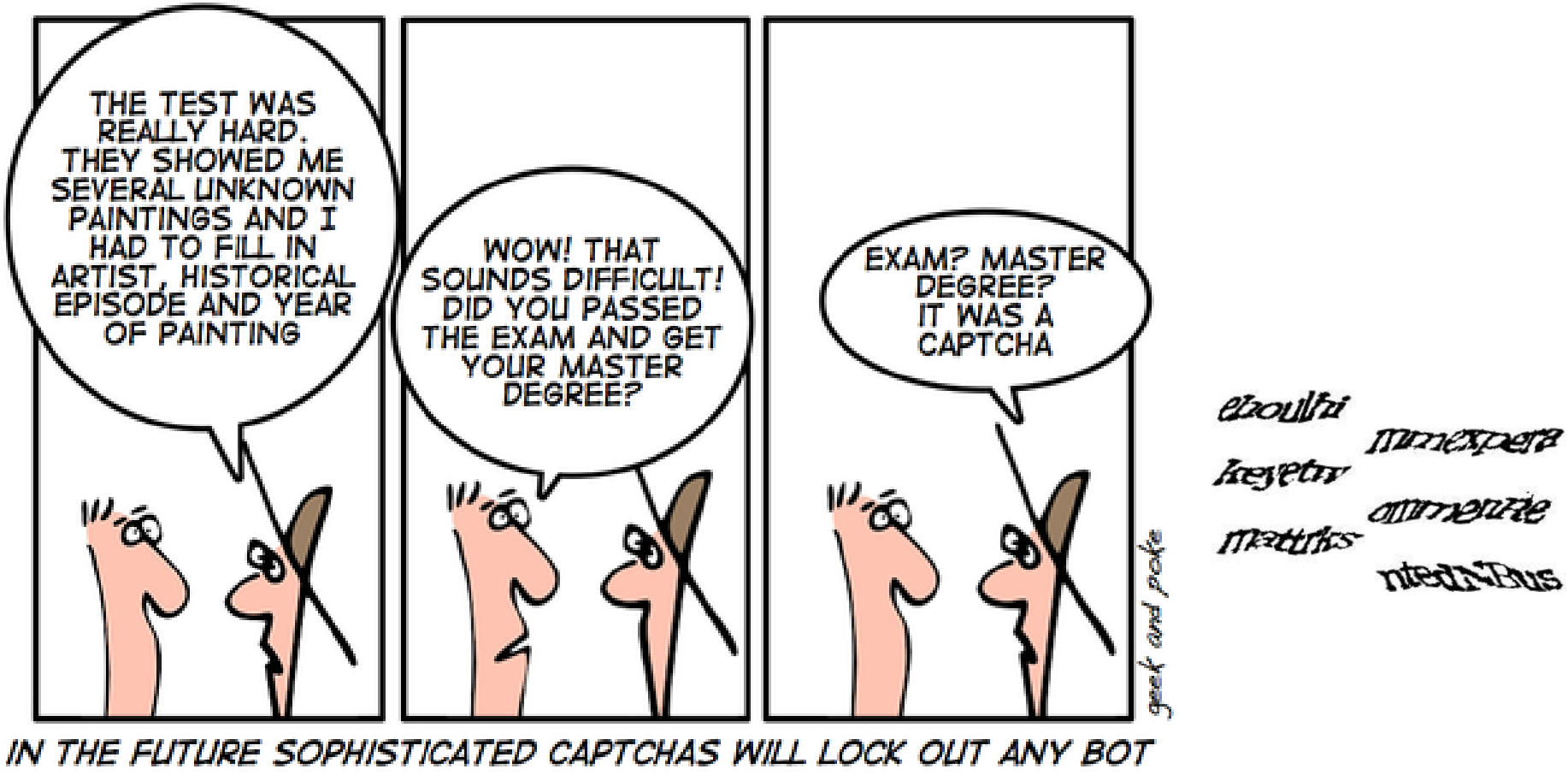


In 1964–66 schreef Joseph Weizenbaum van het MIT in Boston het beroemde programma ELIZA/DOCTOR, dat een psychotherapeut "speelt" — met veel succes.
Een voorbeeldgesprek (Jason Hutchens' HAL/HeX, 1997):

my name is bob
`Pleased to meet you.`
how are you doing, jas?
`I am just fine, thanks! And you?`
i am feeling great!
`If you are feeling great, it's your business, mate!`

Dit soort computerprogramma's doet mee in de jaarlijkse Loebner Prize.

Captcha


THE TEST WAS REALLY HARD. THEY SHOWED ME SEVERAL UNKNOWN PAINTINGS AND I HAD TO FILL IN ARTIST, HISTORICAL EPISODE AND YEAR OF PAINTING
WOW! THAT SOUNDS DIFFICULT! DID YOU PASSED THE EXAM AND GET YOUR MASTER DEGREE?
EXAM? MASTER DEGREE? IT WAS A CAPTCHA
geek and poke
IN THE FUTURE SOPHISTICATED CAPTCHAS WILL LOCK OUT ANY BOT

Een robot is een "actieve, kunstmatige agent wiens omgeving de fysieke wereld" is. Het woord stamt uit 1921 (of eerder), en is gemaakt door de Tsjechische broers Čapek. Nu kennen we ook softbots: internet programma's.

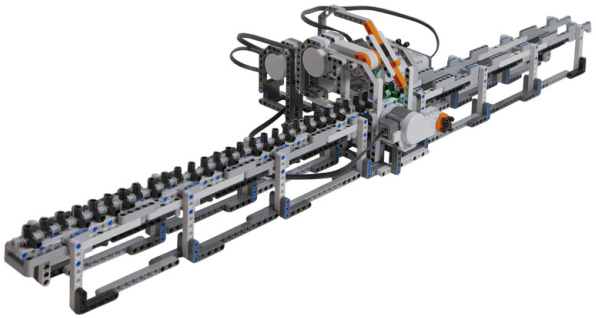
Van de science fiction schrijver Isaac Asimov (auteur van "I, Robot") zijn de drie wetten van de robotica:

1. Een robot mag een mens geen kwaad doen.
2. Een robot moet menselijke orders gehoorzamen (tenzij dat tegen 1. ingaat).
3. Een robot moet zichzelf beschermen (tenzij dat tegen 1. of 2. ingaat).

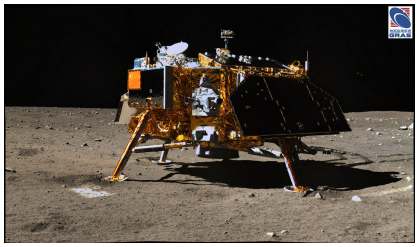
Lego-robots

Een simpel programma voor een Lego-robot, dat deze random laat lopen, is:

```
task main ( ) {
  while ( true ) {
    OnFwd (OUT_A + OUT_C);
    Wait (Random (100) + 40);
    OnRev (OUT_A);
    Wait (Random (85) + 30);
  }
}
```



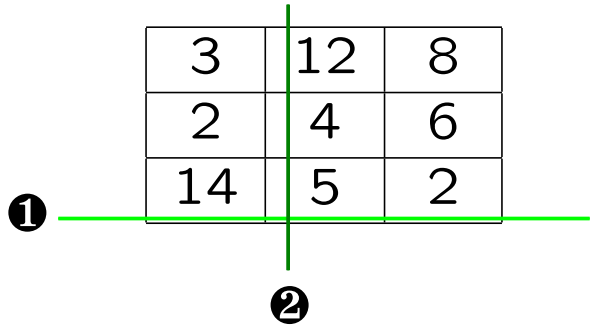
Lego Turing machine



Chang'e-3 Moon lander

Pepper robot van Softbank (2016)

Maxi en Mini spelen het volgende eenvoudige spel: Maxi wijst eerst een (horizontale) rij aan, en daarna kiest Mini een (verticale) kolom:



Bijvoorbeeld: Maxi ❶ kiest rij 3, daarna kiest Mini ❷ kolom 2; dat levert einduitslag 5.

Maxi wil graag een zo groot mogelijk getal, Mini juist een zo klein mogelijk getal.

Hoe analyseren we dit?

Als Maxi rij 1 kiest, kiest Mini kolom 1 (levert 3); als Maxi rij 2 kiest, kiest Mini kolom 1 (levert 2); als Maxi rij 3 kiest, kiest Mini kolom 3 (levert 2). Dus kiest Maxi rij 1!

3	12	8
2	**?**	**?**
14	5	2

Nu merken we op dat de analyse hetzelfde verloopt als we niet eens weten wat onder de twee vraagtekens zit. Het α-β-algoritme onthoudt als het ware de beste en slechtste mogelijkheden, en kijkt niet verder als dat toch nergens meer toe kan leiden.

Ieder (?) schaakprogramma gebruikt deze methode.

Ook aan een spel als Tetris kleven allerlei vragen:

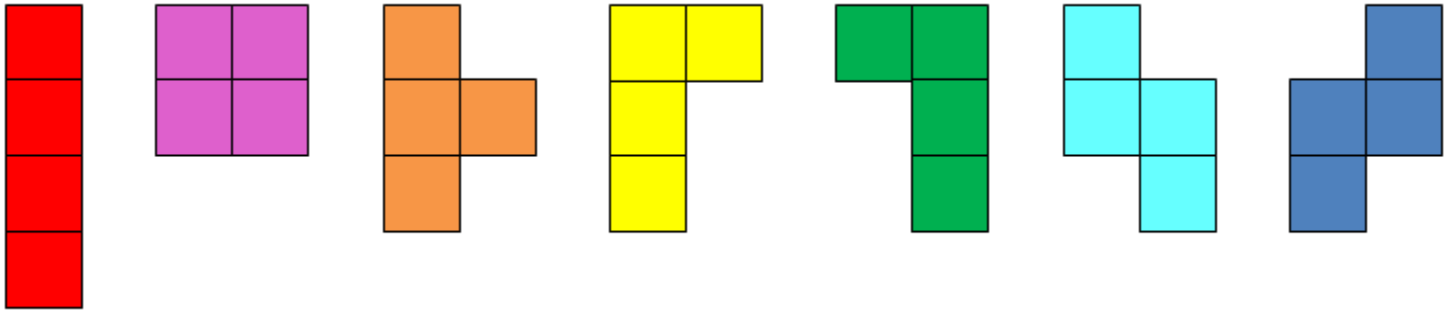
- Hoe speel je het zo goed mogelijk? (AI)
- Hoe moeilijk is het? (complexiteit)
- Wat kan er allemaal gebeuren?

Zo is bijvoorbeeld bewezen dat sommige Tetris-problemen NP-volledig zijn, dat je bijna alle configuraties kunt bereiken, maar dat niet alle problemen "beslisbaar" zijn, zie:

www.liacs.leidenuniv.nl/~kosterswa/tetris/

Tetris: NP-volledig?

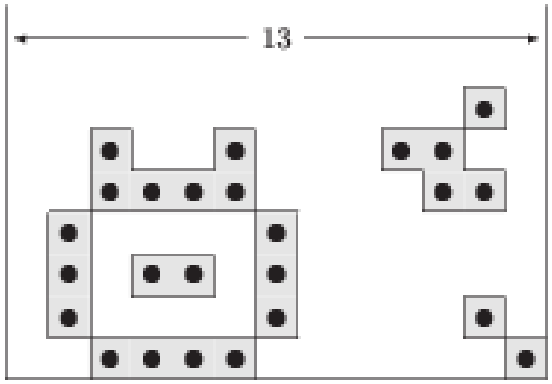
De 7 Tetris-stukken:



Stukken vallen random; volle regels worden verwijderd.
De vraag "Kun je met een gegeven serie (inclusief volgorde) van deze stukken een bord helemaal leeg spelen?" is NP-volledig.
Als iemand het bord leeg speelt kun je dat eenvoudig controleren. Als het *niet* kan, kan men (tot nu toe) niks beters verzinnen dan alle mogelijkheden één voor één na te gaan!

Tetris: bereikbaar?

Een "willekeurige" configuratie:

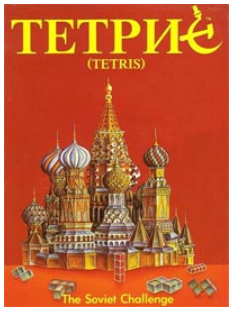


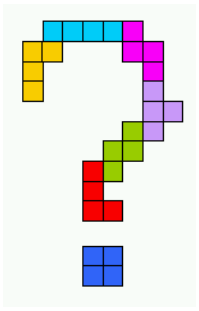
Deze kan gemaakt worden door 276 geschikte Tetris-stukken op de juiste plaats te laten vallen, zie de website.
Let op: alleen geheel gevulde regels verdwijnen, alles daarboven zakt *één rij*.

Claim: op een bord van oneven breedte kan elke configuratie bereikt worden!

Programmeeropgave 1: Tetris

De eerste programmeeropgave (deadline: 28 februari 2024, 13:15 uur; denk ook aan het verslag) gaat over Tetris.



YouTube

Schrijf een C++-programma dat redelijk Tetris speelt. Gebruik verschillende eenvoudige ‘strategieën” (random, Monte Carlo, slim), en beschrijf experimenten.

Zie verder, ook voor voorbeeldcode:

www.liacs.leidenuniv.nl/~kosterswa/AI/teet2024.html

Bij de Monte Carlo (MC) techniek laat je voor elk van de mogelijke zetten een aantal (1000 (?), de "playouts") potjes *random* (?) tot het eind doorspelen. Je kiest dan de zet met de hoogste gemiddelde (?) uitkomst.
Dit heet *Pure Monte Carlo search*.

Thema werkcollege 8.2: MC; 15.2: "slim"; 22.2: verslag.

1. Wat is het verschil tussen een mens en een computer?

2. Kan een computer denken?

3. Hoe werkt een Marsrobot?

4. Hoe werkt een vertaalprogramma?

5. Hoe bedenkt een computer een zet bij vier-op-een-rij?

En schaken? En Diplomacy? En ChatGPT? En Tetris?

Serieuze introductie

Kunstmatige Intelligentie (AI)

Hoofdstuk 1 van Russell/Norvig = [RN]

Introductie

voorjaar 2024

College 1, 7 februari 2024

www.liacs.leidenuniv.nl/~kosterswa/AI/

Definitie AI

Er zijn vele verschillende definities van AI. Ze vallen in vier categorieën uiteen:

Systemen die denken als mensen	Systemen die redelijk denken
Systemen die handelen als mensen	Systemen die redelijk handelen

Boven: gedachten-processen, redeneren; onder: gedrag.
Links: meet succes af aan mens; rechts: aan ideaal beeld.

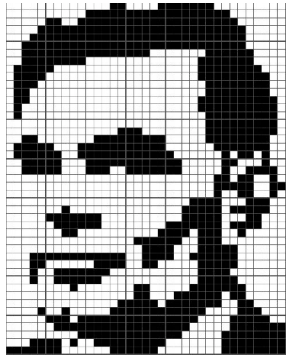
Voorbeeld: een autonome auto.

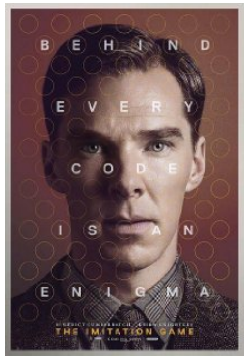
Turing-test

Als je de definitie "Systemen die handelen (act) als mensen" aanhangt, kom je bij de Turing-test — zie ook eerder.

Een computer slaagt voor deze test als hij/zij/het voor een menselijke ondervrager (met "typemachine-contact") niet van een mens te onderscheiden is. Noodzakelijk:

- verwerking van natuurlijke taal
- kennis-representatie
- geautomatiseerd redeneren
- machine learning





Alan Turing, 1912–1954

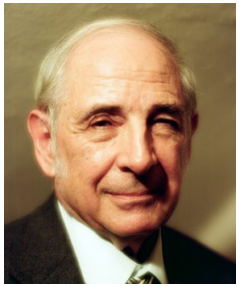
Voor de totale Turing-test (met video, en doorgeven van fysieke voorwerpen) komen daar nog bij: computer-zien en robotica.

De Chinese kamer

De Turing-test lijkt op het beroemde Chinese kamer gedachten-experiment van John Searle uit 1980.

Hierbij zit iemand in een kamer, krijgt geschreven Chinese opdrachten, zoekt in een (groot) boek het juiste "antwoord" — inclusief bijvoorbeeld manipulaties met stapeltjes en pennen —, en geeft dit door naar de buitenwereld.

Volgens Searle betekent het correct verwerken van het juiste programma niet noodzakelijk begrip, of bewustzijn.



En ChatGPT?

Een agent is iets wat handelt. Een rationele agent is iets wat handelt om de beste (verwachte) uitkomst te bereiken. Er is dus een doel.

AI "is" dan de studie van dit soort agenten.

We streven daarbij naar *perfecte* rationele agenten. In de praktijk nemen we — door bijvoorbeeld tijdsbeperking — genoegen met wat minder.

In de 21ste eeuw werd AI de studie van agenten die "het goede/juiste doen" (oftwel "do the right thing").

Grondslagen van AI

De grondslagen van AI zijn te vinden in:

- Filosofie
- Wiskunde (algoritme, Kurt Gödel's stelling, NP-volledigheid (Karp, Cook, Garey, Johnson))

- Economie
- Neuro-wetenschap
- Psychologie (behaviorisme van Watson, cognitieve psychologie)

- Informatica (John von Neumann, . . .)
- Control theorie en Cybernetica (Wiener)
- Linguistiek (Noam Chomsky)

De beginperiode: 1943–1956

1943 — McCulloch en Pitts — artificiële neuronen

1949 — leerregel van Hebb

schaakprogramma's van Shannon en Turing (en de test!)

1951 — Minsky en Edmonds — neurale netwerk computer

1956 — geboorte van de AI: Dartmouth

1956 — Newell en Simon — Logic Theorist (LT)

Groot enthousiasme: 1952–1969
John McCarthy noemde dit het "Look ma, no hands" tijdperk.

1952 — Samuel — checkers

1957 — Newell en Simon — General Problem Solver (GPS)

1958 — McCarthy (MIT, Stanford; 1927–2011) — de taal Lisp, time sharing, Advice Taker

1958 — Minsky (MIT) — microwerelden (Slagle's SAINT (1963), Evans' ANALOGY (1968), . . .): blokkenwereld (Huffman, Waltz, Winston, Winograd's SHRDLU, . . .)

1960–62 — Widrow en Hoff — adalines; Rosenblatt — perceptrons

Een dosis realiteit: 1966–1973

Spoedig bleek dat alle optimistische claims niet werden waargemaakt. Een beroemd voorbeeld is de volgende vertaling van Engels via Russisch terug naar Engels: "the spirit is willing but the flesh is weak" naar "the vodka is good but the meat is rotten". Alle subsidiëring in de VS werd in die tijd stop gezet.

Problemen: er werd geen "kennis" opgeslagen, maar louter syntactisch gemanipuleerd; en de echte problemen waren te groot. Zo experimenteerde men al in 1958 (weinig succesvol) met Genetische Algoritmen. Het boek van Minsky en Papert uit 1969 toonde ook zekere grenzen aan.

Daarna vonden/vinden allerlei ontwikkelingen plaats:

1969–1986 — kennis-gebaseerde systemen (zoals expert-systemen: MYCIN (Feigenbaum, . . .))

1986–. . . — neurale netwerken zijn weer terug en in: . . . Support Vector Machines . . .

1987–. . . — redeneren, "machine learning" (Hidden Markov Modellen (HMM's), data mining, Bayesiaanse netwerken)

2001–. . . — beschikbaarheid van grote datasets . . . big data . . . data science . . . grote bedrijven & successen

2011–. . . Deep learning (Geoff Hinton et al.), Reinforcement learning, GPT-3 van OpenAI, . . .

In Hoofdstuk 27 van [RN] (in de derde druk Hoofdstuk 26) staat van alles over de filosofische grondslagen van de AI.

John Searle maakt onderscheid tussen de zwakke AI-hypothese (machines kunnen zich wellicht intelligent gedragen) en de sterke (machines kunnen denken). Vergelijk: "Kan een machine vliegen?" en "Kan een machine zwemmen?". En Alan Turing vond het een "polite convention that everyone thinks".

Mensen als Ray Kurzweil (The Age of Spiritual Machines, 1999) voorspellen een versmelting van mens en computer: de "Singularity".

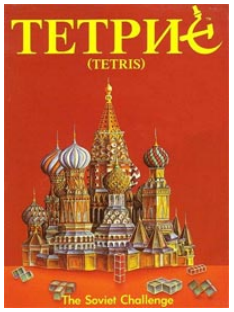
Ethiek?

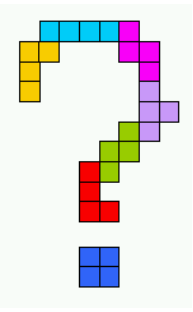
Positieve en negatieve verwachtingen!

Lees Hoofdstuk 1, p. 1–35 van [RN] (in de derde druk p. 1–30). En — voor de aardigheid — Hoofdstuk 27 (in de derde druk Hoofdstuk 26). En de teksten van Alan Turing en Jason Hutchens, zie de website.

Het huiswerk voor de volgende keer (14 februari 2024): lees Hoofdstuk 2, p. 36–62 van [RN] (in de derde druk p. 34–59) door, en bekijk in het bijzonder Figure 2.6.

Denk na over
Monte Carlo en Tetris.



Kunstmatige Intelligentie (AI)

Hoofdstuk 2 van Russell/Norvig = [RN]

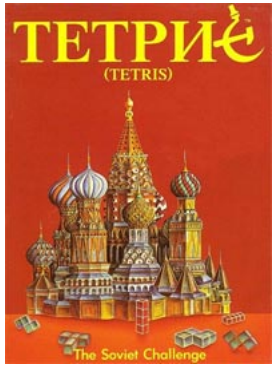
Intelligente agenten

voorjaar 2023

College 2, 14 februari 2024

www.liacs.leidenuniv.nl/~kosterswa/AI/agenten.pdf

Een paar opmerkingen over
Monte Carlo & Tetris:



- Werkcollege 15.2 heeft als thema "slim"
- Video's, C++ en voorbeeldcode
- En de verschillende strategieën, Monte Carlo en "slim"?
- Experimenten; grafieken, statistieken.
- Het verslag (in LaTeX), met wat `tips` → volgende week

`www.liacs.leidenuniv.nl/~kosterswa/AI/teet2024.html`

Een agent is iets wat zijn/haar omgeving (environment) waarneemt met behulp van sensoren en op deze omgeving werkt met actuatoren.
Een mens in de fysieke wereld heeft ogen en handen.

De waarnemingen die een agent gedaan heeft vormen zijn of haar percept sequence (rij met waarnemingen). De door de agent te kiezen actie kan in principe van die hele rij afhangen, maar niet op iets wat niet is waargenomen.
De agent functie beeldt rijtjes af op acties.

Wat rationeel is hangt af van:

- de performance maat (measure) die de mate van succes definieert;
- wat de agent weet van zijn/haar omgeving (voorkennis = "prior knowledge");
- de mogelijke acties;
- de percept sequence op dat moment.

Een ideale rationele agent moet voor iedere mogelijke percept sequence die actie uitvoeren waarvan verwacht wordt dat deze de performance maat maximaliseert, op basis van dat rijtje en alle ingebouwde kennis die de agent bezit.
NB Dit is iets anders dan alwetend zijn.

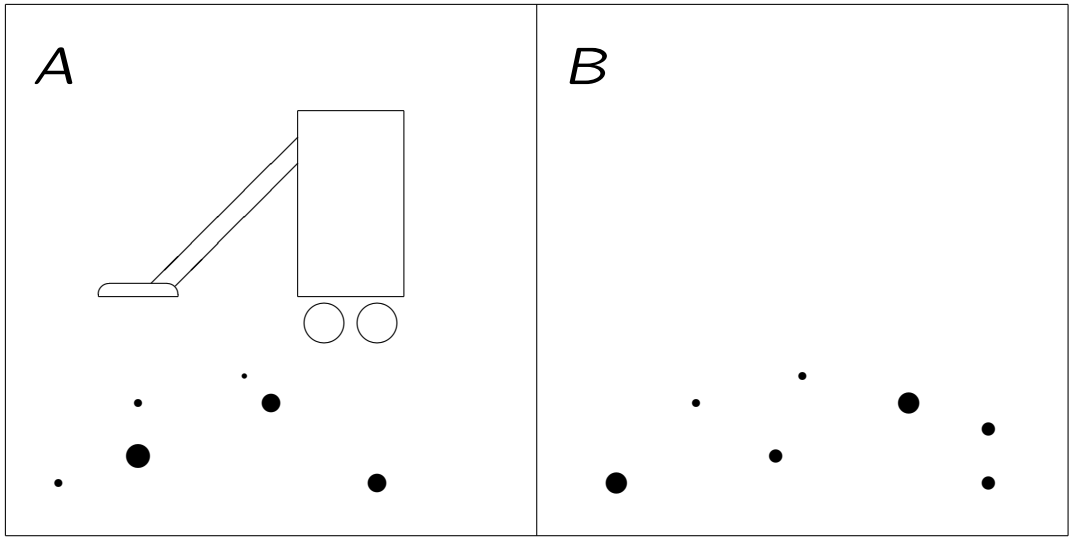
De ideale afbeelding van percept sequences naar acties specificeert welke actie de agent zou moeten nemen, gegeven een percept sequence. Dit levert een ideale agent.

Een agent is autonoom als zijn/haar gedrag wordt bepaald door zijn/haar *eigen* ervaring.

Als er geen aandacht is voor percepts is er geen sprake van autonomie.
Als een agent autonoom is, vertoont hij/zij doorgaans *lerend* gedrag — om te compenseren voor gedeeltelijke of foutieve voorkennis.

Voorbeeld: een altijd (zomer- en wintertijd) gelijk lopend horloge is niet per se autonoom. En een op afstand bedienbare "robot" al zeker niet.

De Stofzuiger-wereld ziet er als volgt uit:



Mogelijke acties: *Left* (stukje naar links), *Right* (stukje naar rechts), *Suck*, *Nothing*.
Eén waarneming (oftewel percept) is bijvoorbeeld [*A*,*Clean*] of [*B*,*Dirty*].

Een eenvoudige agent voor de Stofzuiger-wereld zou de volgende tabel kunnen benutten:

Percept sequence	Actie
$[A, Clean]$	*Right*
$[A, Dirty]$	*Suck*
$[B, Clean]$	*Left*
$[B, Dirty]$	*Suck*
$[A, Clean], [A, Clean]$	*Right*
$[A, Clean], [A, Dirty]$	*Suck*
...	...
$[A, Clean], [A, Clean], [A, Clean]$	*Right*
...	...

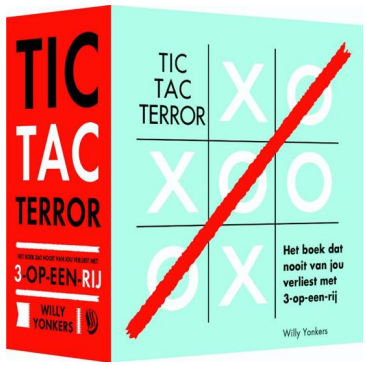
Dit levert een rationele agent op.

NB We nemen aan dat *Left*/*Right* een klein stukje beweegt (als je al helemaal rechts bent, gebeurt er bij *Right* niets).

Opzoektabel

Waarom gebruik je in het algemeen geen opzoektabel, met een complete opsomming van alle mogelijkheden?

- Zelfs voor een "simpele" schaak spelende agent heb je zo'n 35^{100} ($35 \approx$ aantal mogelijke zetten; $100 \approx$ lengte partij) regels nodig. Dus geheugenproblemen.
- Het zou erg lang duren een dergelijke tabel te vullen.
- Er is geen autonomie. Problemen met veranderende omgeving.
- Zelfs met leren erbij zou het oneindig lang (kunnen) duren.
- Boeit niet.



Als we een rationele agent willen ontwerpen, moeten we de task environment (taak-omgeving) specificeren.
Bijvoorbeeld, voor een automatische taxi-chauffeur:

P erformance maat: veiligheid, winst, bestemming, comfort, de wet, . . .

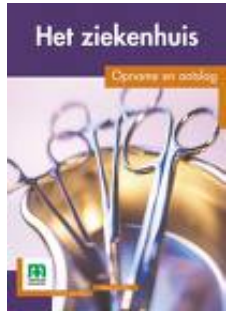
E nvironment (Omgeving): straten, verkeer, voetgangers, weer, . . .

A ctuatoren: stuur, remmen, gaspedaal, scherm, . . .

S ensoren: camera, meters, microfoon, GPS, toetsenbord, . . .

En voor een systeem ten behoeve van medische diagnose:

P erformance maat: gezonde patient, lage kosten, rechtzaken, . . .



E nvironment: patient, ziekenhuis, staf, . . .

A ctuatoren: vragen, onderzoeken, diagnoses, behandelingen, . . .

S ensoren: antwoorden patient, invoeren symptomen op toetsenbord, thermometer, . . .

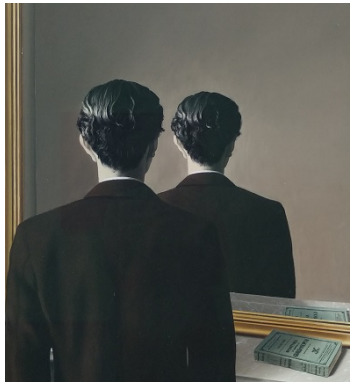
Probeer zelf eens de omgeving voor een agent ten behoeve van internet-winkelen te beschrijven.

Er zijn verschillende dimensies waarlangs je — na veel discussie — omgevingen kunt leggen:

- volledig observeerbaar $\leftrightarrow$ deels observeerbaar
- deterministisch $\leftrightarrow$ niet-deterministisch
  stochastisch $\approx$ niet-deterministisch met kansen
  strategisch: det., afgezien van acties andere agenten
- episodisch $\leftrightarrow$ sequentieel
- statisch $\leftrightarrow$ dynamisch
  semi-dynamisch: alleen score verandert met tijd
- discreet $\leftrightarrow$ continu
- enkele agent $\leftrightarrow$ multi-agent (competitief, cooperatief)

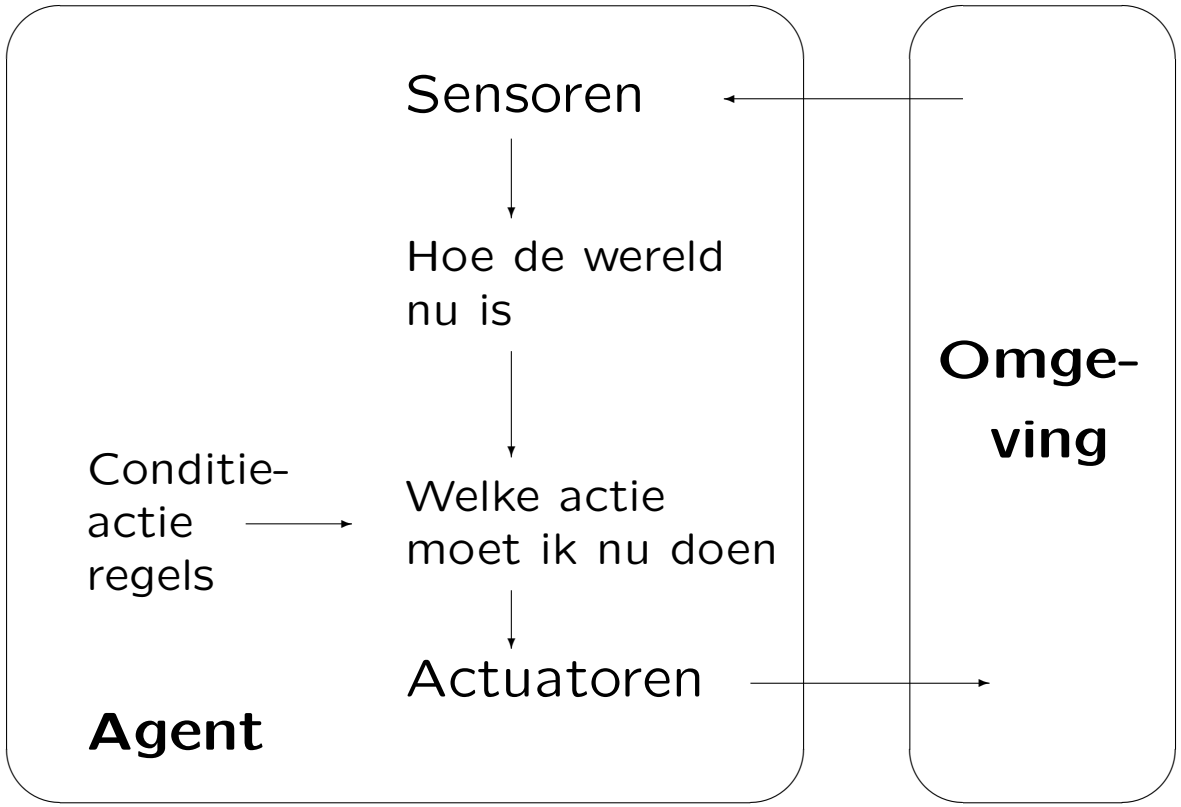
Daarnaast kan er van alles over de omgeving (on)bekend zijn (known $\leftrightarrow$ unknown). Een voorbeeld: ken je de spelregels van poker al, of leer je die tijdens het spelen?

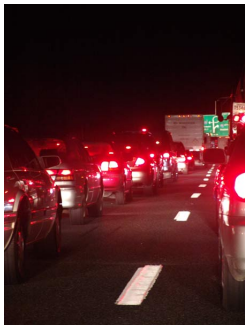
Dit is iets anders dan volledig observeerbaar $\leftrightarrow$ deels observeerbaar! Voorbeeld: bij patience ken je de spelregels, maar weet je niet alle kaarten (deels observeerbaar); bij een nieuw computerspel kun je alles zien, maar weet je soms nog niet hoe de "knoppen" werken (unknown).



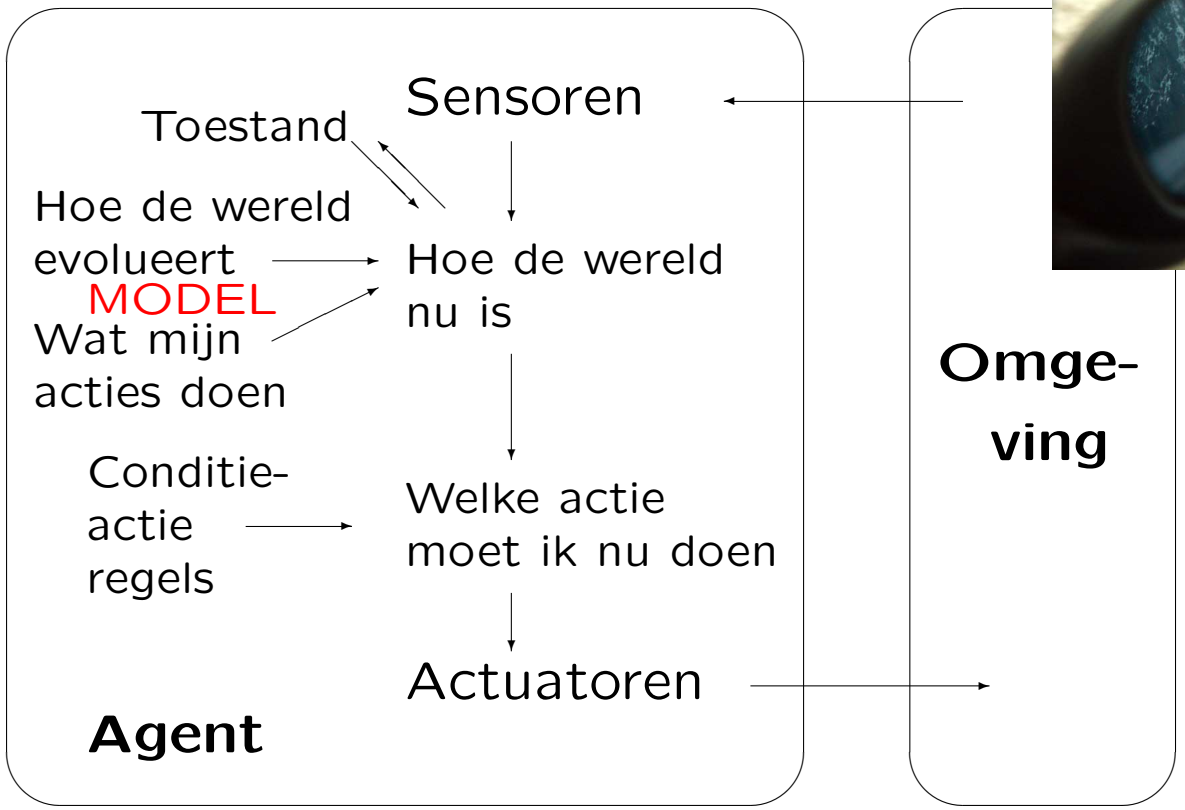
	Obs.	Det.	Epis.	Stat.	Disc.	Agt.
Puzzel	J	J	N	J	J	1
Schaak+klok	J	J	N	semi	J	> 1
Poker	N	J	N	J	J	> 1
Backgammon	J	N	N	J	J	> 1
Taxi rijden	N	N	N	N	N	> 1
Beeldanalyse	J	J	J	semi	N	1
Fabrieks-robot	N	N	J	N	N	1
WWW-winkel	N	$\pm$	N	semi	J	$1/> 1$

Volledig observeerbaar: alle *relevante* zaken gezien.
Deterministisch: volgende toestand wordt bepaald door huidige toestand en actie (gezien vanuit de agent).
Episodisch: onderling onafhankelijke "atomaire" episodes; als voorbeeld: een schaaktoernooi.
Sommige poker-varianten zijn wel stochastisch.



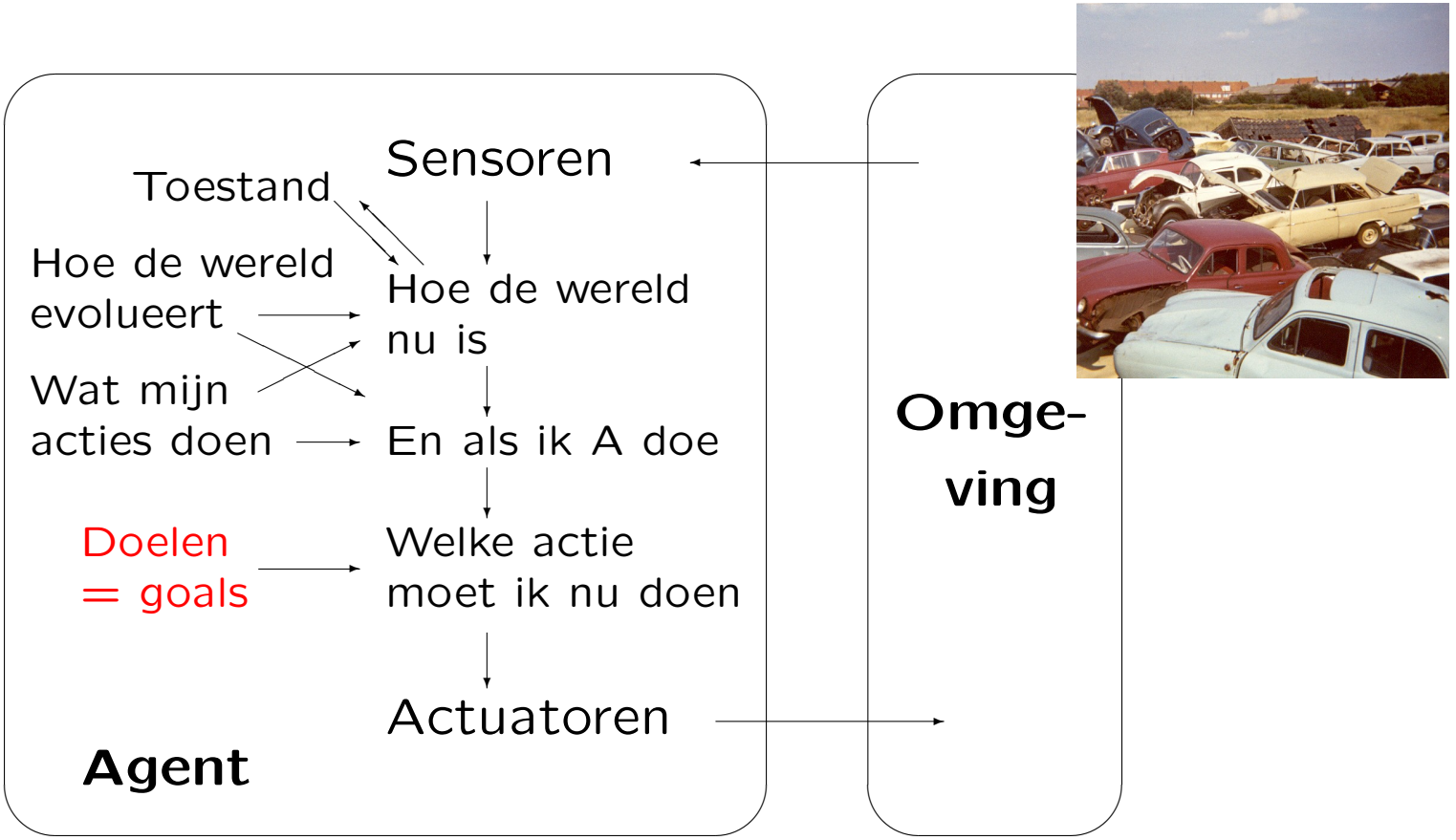


ALS auto_voor_je_remt DAN begin_zelf_te_remmen
Randomiseren helpt je soms uit oneindige loops.

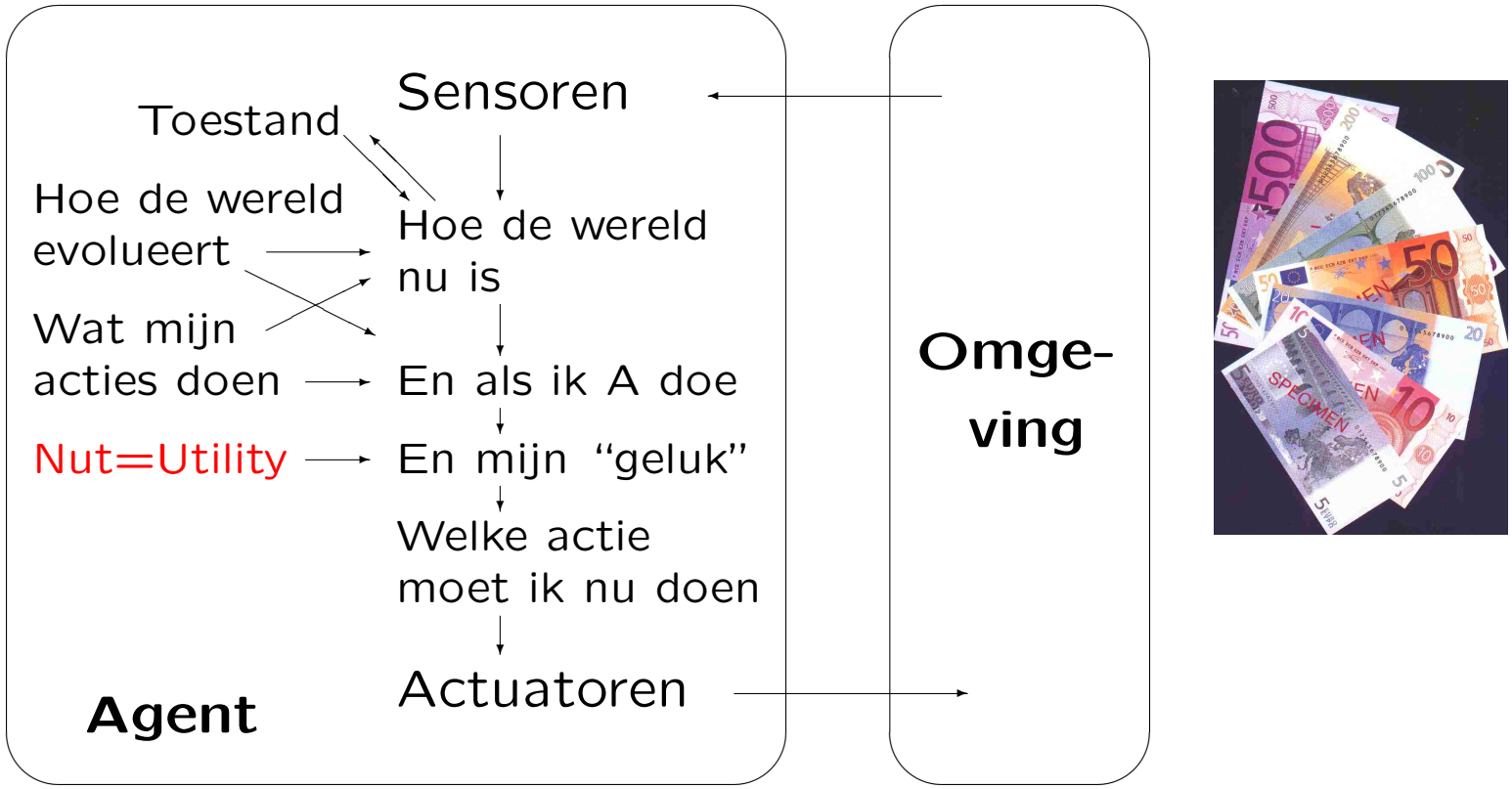


Wat zagen we zo-even in de spiegel?

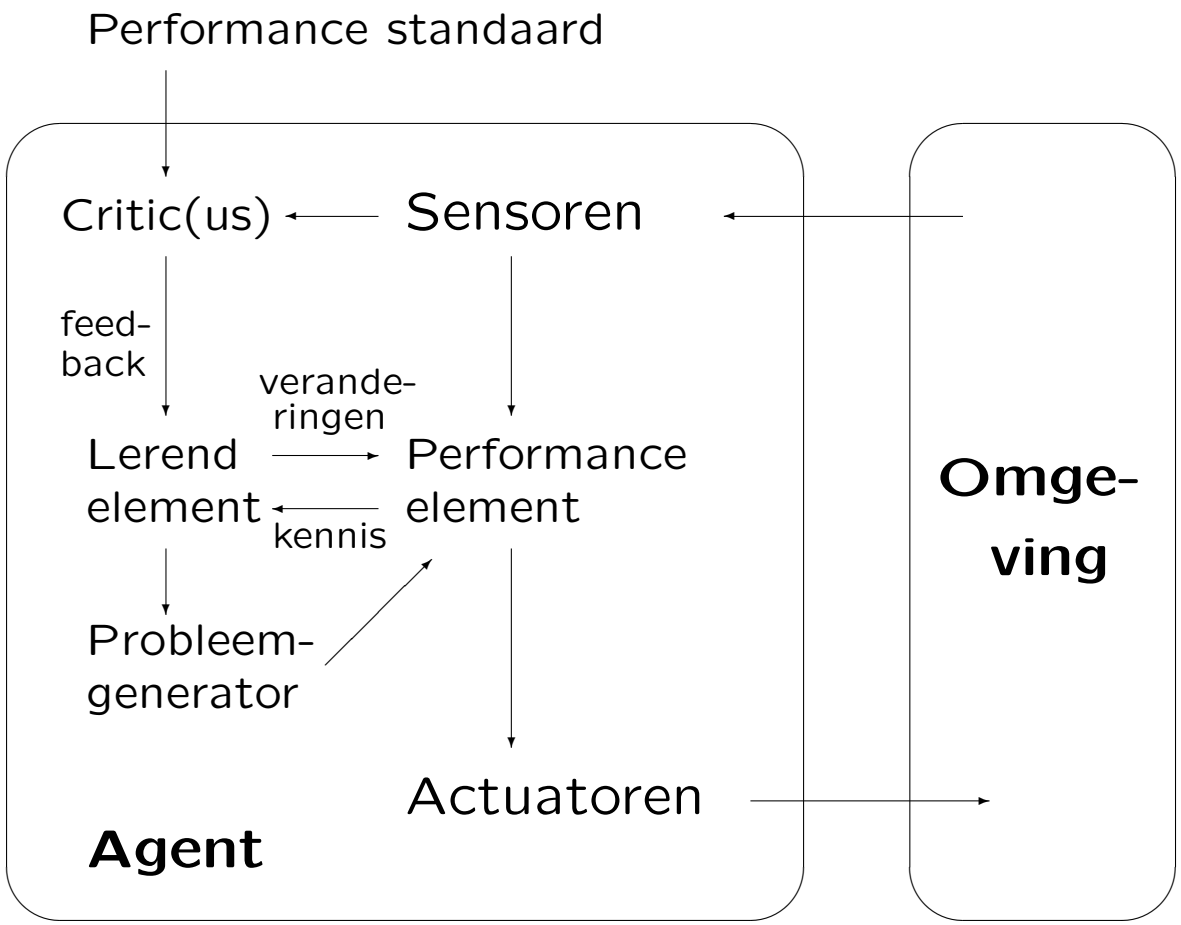
De agent heet wel model-gebaseerd.

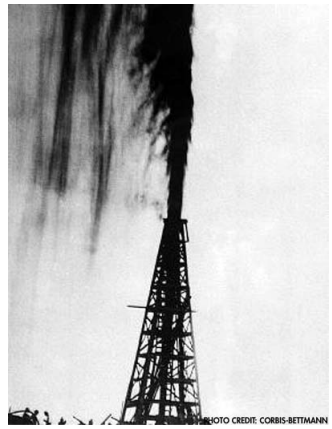


Waar moet de auto naar toe? (ook weer model-gebaseerd)



Hoe snel/veilig/duur/... wordt de bestemming bereikt?
De utility-functie "weegt" doelen en meet kansen.





De probleem-generator geeft exploratie ($\leftrightarrow$ exploitatie).

Het Belief-Desire-Intention model (BDI) is een door de filosoof/psycholoog Michael Bratman in de jaren 80 ontwikkeld schema om praktisch redeneren te kunnen begrijpen en analyseren.

De software-kant van het model bestudeert en programmeert agenten. Het model probeert het selecteren van plannen te scheiden van het uitvoeren daarvan, maar creëert zelf geen plannen.

Een BDI-agent heeft:

beliefs wat gelooft de agent over zichzelf en de wereld?

desires wat wil de agent graag?

een doel (goal) is een verlangen (desire) dat de agent actief najaagt, consistent met andere doelen

intentions wat heeft de agent gekozen?

de agent besluit voor een of meer plannen/acties

Een voorbeeld:

- je gelooft dat op 1 april college AI is
- je wilt graag AI halen, uitslapen, feesten
- je neemt je voor naar ieder college te gaan

Een programma voor een Eenvoudige reflex-agent (al dan niet met een op een model van de wereld gebaseerde toestand) is:

toestand ← Interpreteer_Input (*percept*)
regel ← Regel_Match (*toestand*,*regels*)
actie ← Regel_Actie[*regel*]

En in concreto voor de Stofzuiger:

if *status* = *Dirty* **then return** *Suck*
else if *locatie* = *A* **then return** *Right*
else return *Left*

Je kunt op verschillende nivo's naar componenten van agenten kijken (lees ook Rodney Brooks):

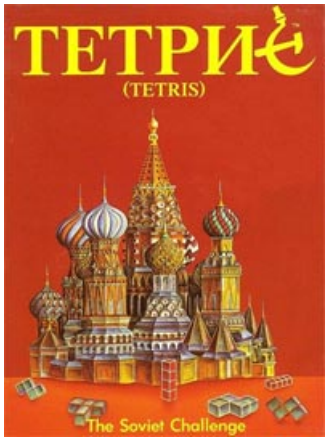
atomair geen interne structuur in de toestanden
zoeken in grafen, spel(l)en, . . .

opgedeeld = **factored** toestanden worden bepaald door variabelen met hun waardes
CSP's, propositie-logica, Bayesiaanse netwerken, . . .

gestructureerd zaken zijn met elkaar gerelateerd
eerste-orde logica, natuurlijke taal, . . .

Het huiswerk voor de volgende keer (21 februari 2024): lees Hoofdstuk 7 en 8, p. 208–224, p. 237–240 en p. 251–271 van [RN] door over het onderwerp Logische agenten.

Denk tevens aan de eerste opgave: Monte Carlo & Tetris; deadline: 28 februari 2024.



Kunstmatige Intelligentie (AI)

Hoofdstukken 7/8 van Russell/Norvig = [RN]

Logisch(e) redenerende agenten

voorjaar 2023

College 3, 21 februari 2024

www.liacs.leidenuniv.nl/~kosterswa/AI/logisch.pdf

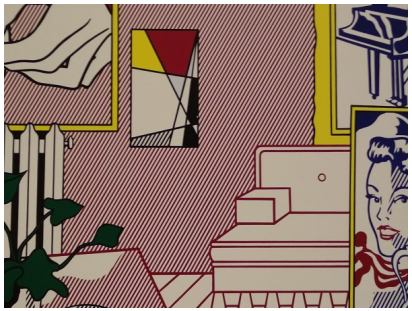
We bekijken eenvoudige propositie-logica voor agenten ([RN], Hoofdstuk 7, p. 208–224 en 237–240):

$$R_1 = \quad \neg S_{1,1} \Rightarrow \neg W_{1,1} \wedge \neg W_{1,2} \wedge \neg W_{2,1} \ ,$$

dit voor de eenvoudige "Wumpus-wereld".

Daarna gaan we over op eerste orde (predicaten-)logica ([RN], Hoofdstuk 8, p. 251–271):

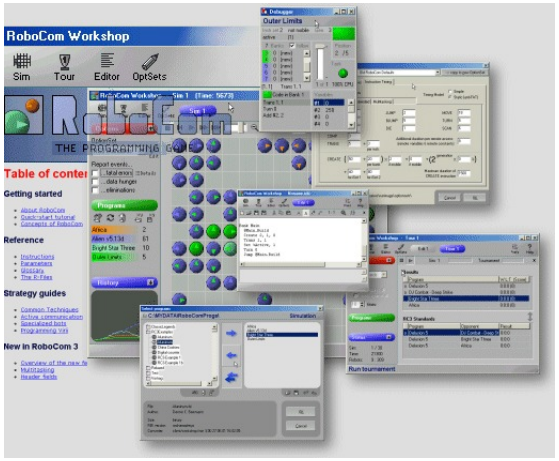
$$\forall s \ \ Breezy(s) \Leftrightarrow \exists r \ \ Adjacent(r, s) \wedge Pit(r)$$



Maar eerst Monte Carlo & Tetris (tips) en . . .

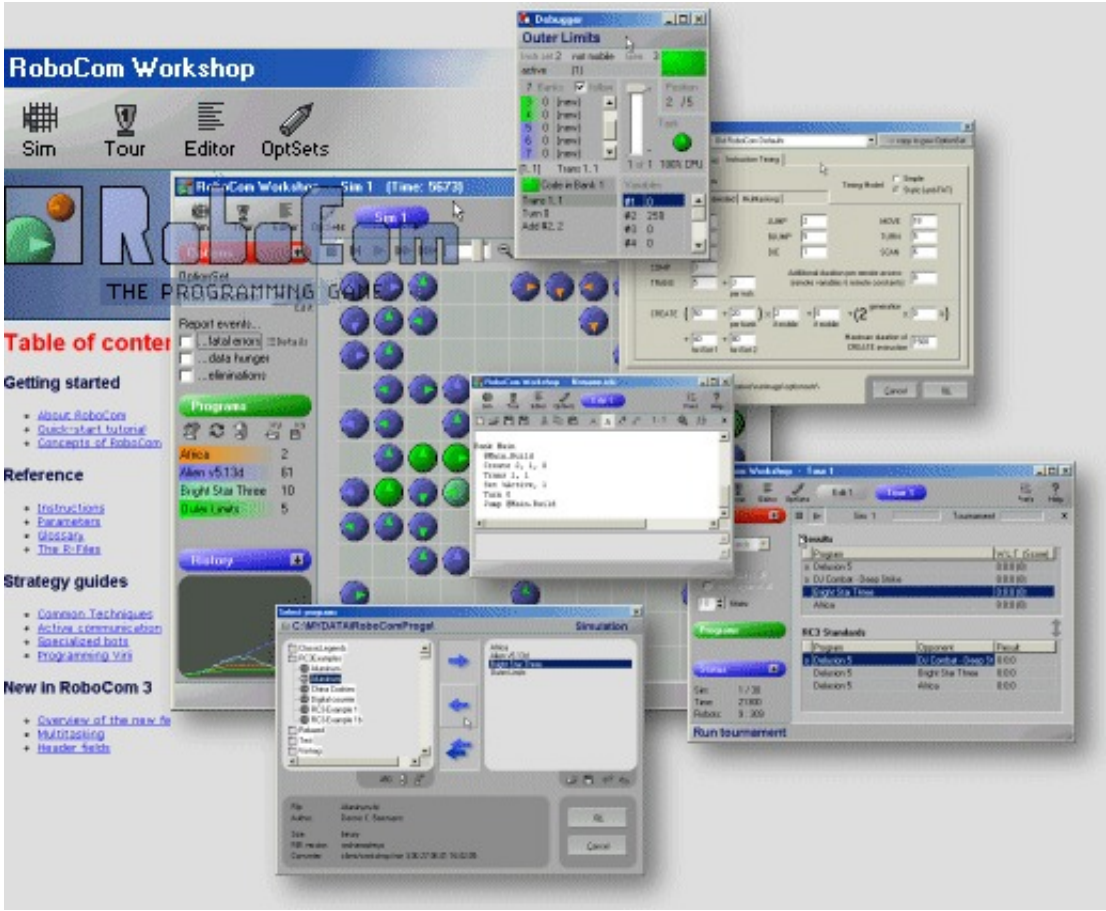
Er zijn allerlei robot-simulaties.
We gebruiken RoboCom, een opvolger van "CoreWar", van Dennis Bemmann: kleine robot-programma's vechten in een vierkant stuk computergeheugen met 18×18 vakjes (= velden). Vergelijk: "Robocode".

Software: RoboCom Workshop 3.1, voor Windows en Linux (`wine`).



Er is nog meer: uitgebreide instructieset, multitasking.

RoboCom — Software



www.liacs.leidenuniv.nl/~kosterswa/AI/robot2024.html

Een robot is een klein assembler-achtig programma dat in één van de 18×18 velden van het "speelveld" leeft. Dit veld is een "torus": links grenst aan rechts, boven aan onder. Een robot ziet één aangrenzend veld in de "kijkrichting": het reference field. Een robot kan nieuwe bewegende robots maken.

Er zijn drie instruction sets: basic (0; met `ADD`, `BJUMP`, `COMP`, `DIE`, `JUMP`, `MOVE`, `SET`, `SUB` en `TURN`), advanced (1; met `SCAN` en `TRANS` erbij) en super (2; met ook nog `CREATE` erbij).

Een robot heeft interne integer prive-variabelen `#1`, `#2`, ..., `#20`. De variabele `#Active` geeft aan of de robot actief is (waarde ≥ 1) of niet. En de constante `$Mobile` (0/1) is de mobiliteit, `$Banks` het aantal "banken" — zie verderop.

Een robot-programma is opgedeeld in maximaal 50 banken (≈ functies). Executie begint bij de eerste. Als je een bank uitloopt begin je weer bij de eerste bank: "auto-reboot".

Voorbeeldprogramma, met één bank:

```
; voorbeeldprogramma, een ";" duidt op commentaar
NAME DoetNietVeel

BANK Hoofdprogramma
  SET #3,7          ; variabele 3 wordt 7
  @EenLabel         ; definieer een label
  ADD #3,1          ; hoog variabele 3 met 1 op
  TURN 1            ; draai 90 graden rechtsom (0: linksom)
  JUMP @EenLabel    ; spring terug naar label
```

`ADD #a,b`	tel `b` bij `#a` op
`BJUMP a,b`	spring naar instructie `b` van bank `a`
`COMP a,b`	sla volgende instructie over als `a` = `b`: "if"
`CREATE a,b,c`	maak in reference field nieuwe robot met instruction set `a`, `b` banken en mobiliteit `c`
`DIE`	robot gaat dood
`JUMP a`	spring `a` verder (naar label `@Iets` mag ook)
`MOVE`	ga naar reference field
`SCAN #a`	bekijk reference field; `#a` wordt 0 (leeg), 1 (vijandige robot) of 2 (bevriende robot)
`SET #a,b`	`#a` krijgt waarde van `b`
`SUB #a,b`	trek `b` van `#a` af
`TRANS a,b`	kopieer eigen bank `a` naar de `b`-de bank van robot in reference field
`TURN a`	draai linksom als `a` = 0, anders rechtsom

Hierbij: `#a`: variabele; `a`, `b`, `c`: elk type.

Instructies kosten tijd, de ene meer dan de andere.

Van de eventuele robot op het reference field kun je de activiteit benaderen (en wijzigen!) via de "remote" `%Active`.

Als na een "auto-reboot" de eerste bank leeg blijkt, gaat de robot dood van "data-honger".

De beginrobot staat altijd stil (mobiliteit 0).

Er zijn allerlei speciale gevallen, nog meer instructies, . . .

De opgave bestaat uit het maken van een bot:

coöperatief Maak een vriendelijk bot, die als je met twee kopieën X en Y ervan begint, zo snel mogelijk het volgende patroon overlaat op een verder leeg veld:

$$\begin{matrix} X & X & X \\ X & Y & X \\ X & X & X \end{matrix}$$

Hoe vaak werkt het, wanneer en waarom? Wat gebeurt er bij twee of meer van de bots? PEAS? De "6"?

www.liacs.leidenuniv.nl/~kosterswa/AI/robot2024.html

```
; een klein robot-programma
NAME Mine

BANK Mine          ; eerste bank
  @Loop            ; label
  TURN 1           ; draai rechtsom
  SCAN #1          ; scan reference field
  COMP #1,1        ; een tegenstander?
  JUMP @Loop       ; nee, verder draaien
  SET %Active,0    ; ja, deactiveer tegenstander (eerder?)
  TRANS 2,1        ; en kopieer narigheid
  SET %Active,1    ; re-activeer
  ; auto-reboot

BANK Poison        ; tweede bank: narigheid
  DIE              ; vergif
```

RoboCom — Nog een voorbeeld

```
; nog een klein robot-programma
NAME Flooder/Shielder

BANK Flood        ; eerste bank
  @Loop           ; label
  TURN 0          ; draai linksom
  SCAN #5         ; scan reference field
  COMP #5,0       ; leeg?
  JUMP @Loop      ; nee, verder draaien
  CREATE 2,1,0    ; ja; creeer nieuwe robot
  TRANS 1,1       ; en kopieer jezelf
  SET %Active,1   ; activeer hem/haar
  ; auto-reboot
```

Een Knowledge base (KB) is een verzameling zinnen ("sentences") in een formele taal. Een KB is domein-specifiek. De waarheid van axioma's nemen we aan.
Een Inference engine gebruikt logisch goed-gefundeerde algoritmen om met behulp van de KB vragen te beantwoorden.

In de declaratieve aanpak vertel ("Tell") je zinnen aan de KB, en vraag ("Ask") je queries. Bij de procedurele aanpak vertaal je gewenst gedrag rechtstreeks in programma-code.

Je kunt op verschillende nivo's naar "knowledge based agenten" kijken:

knowledge (kennis) nivo de Niels Bohrweg verbindt Wassenaarseweg en Einsteinweg

logisch nivo link (Niels Bohrweg,Wass'weg,Einsteinweg)

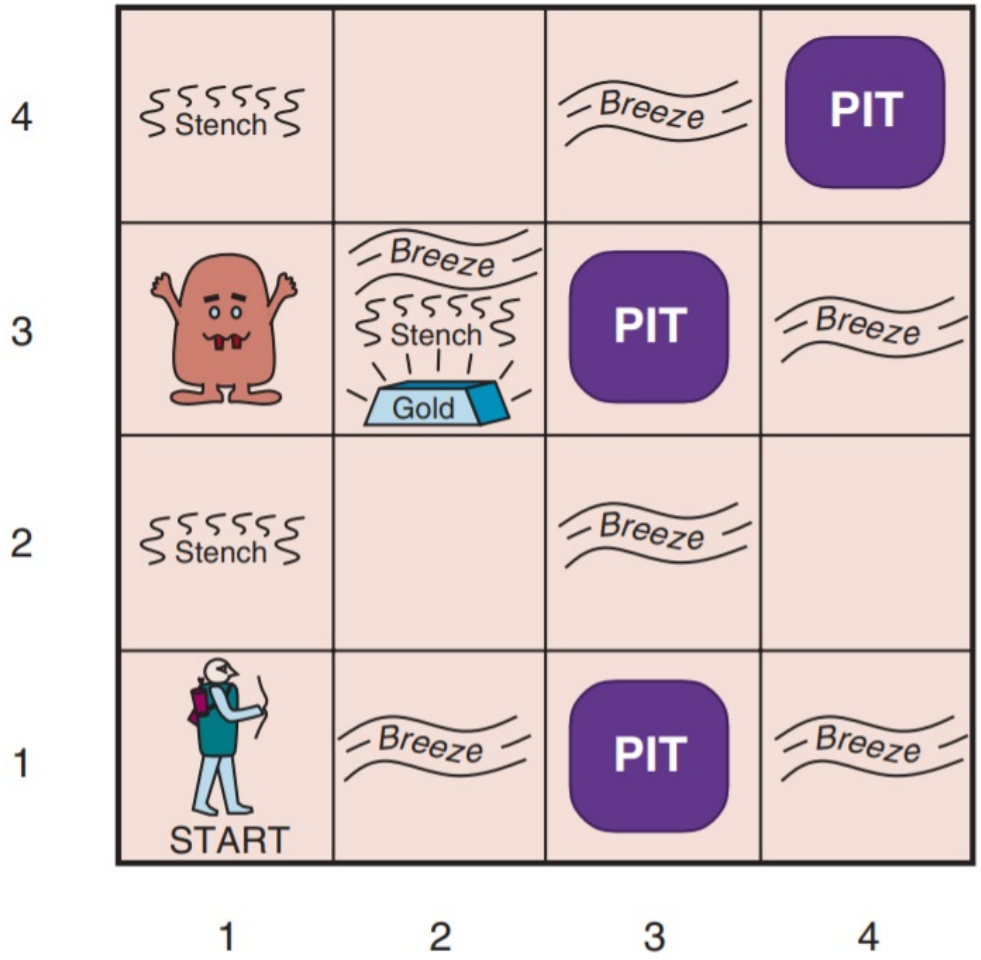
implementatie nivo ergens een 1 in een file

Een eenvoudige kennis-gebaseerde agent zet als volgt een *percept* (waarneming) om in een *actie*, met Knowledge base KB en tijd t (initieel 0):

function KBagent(*percept*)
  Tell(KB, MakePerceptSentence(*percept*, t))
  actie $\leftarrow$ Ask(KB, MakeActionQuery(t))
  Tell(KB, MakeActionSentence(*actie*, t))
  $t \leftarrow t + 1$
  return *actie*

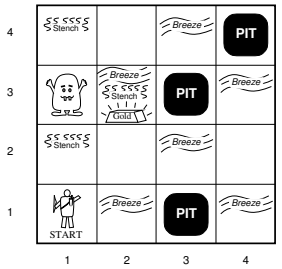
De KB kan aan het begin achtergrondkennis bevatten.

Als eenvoudig voorbeeld bekijken we de Wumpus wereld:



Wumpus-wereld — PEAS

De PEAS beschrijving van de Wumpus-wereld:



Performance maat: goud $+1000$, dood -1000, -1 per stap, -10 voor gebruik pijl

Environment: 4×4, start in $(1,1)$, random goud, naast (horizontaal, verticaal) Wumpus stinkt het ("stench"), naast put waait het, in het goudvakje glimt het, je kunt de Wumpus doden als je hem "ziet", slechts één pijl, je kunt goud oppakken in het goudvakje en loslaten waar je wilt, ...

Actuatoren: TurnLeft, TurnRight, Forward, Grab, Release, Climb (uit de wereld), Shoot

Sensoren: Stench/Smell, Breeze, Glitter, Bump (tegen een muur) en Scream (dode Wumpus) — vijf stuks

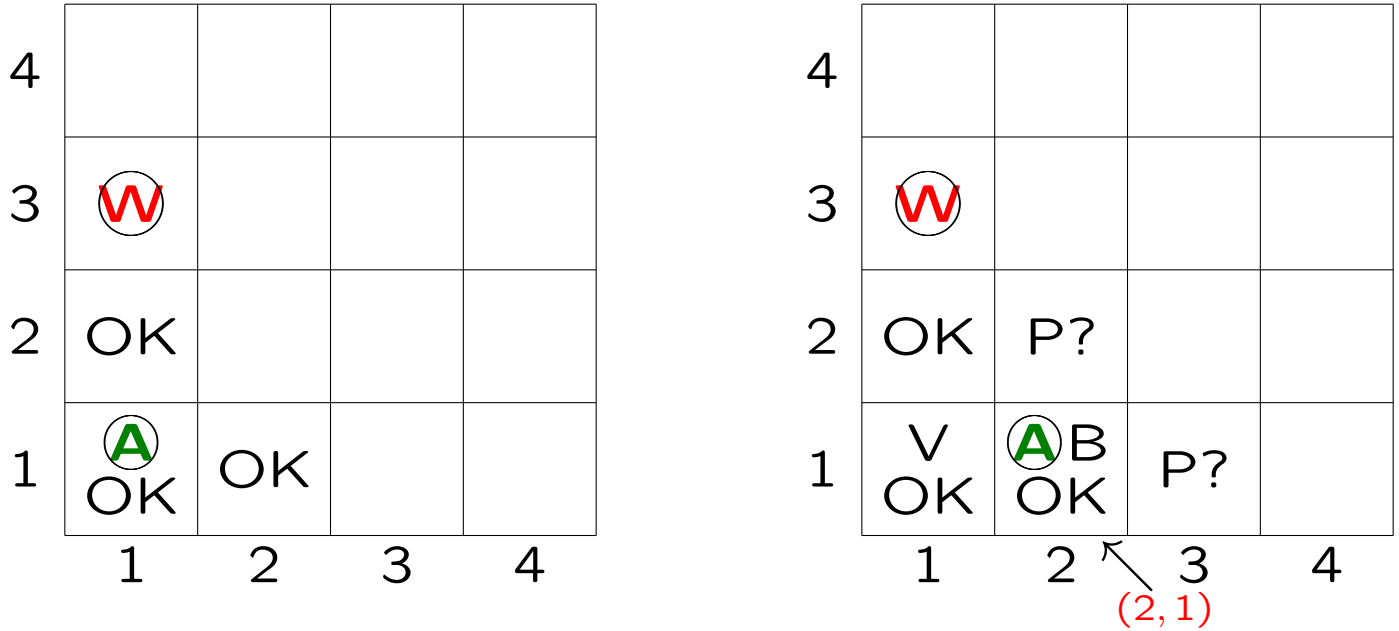
De Wumpus-wereld is:

– niet observeerbaar (deels: alleen locaal),
– deterministisch (“uitkomsten” liggen vast),
– niet episodisch, maar sequentieel,
– statisch (Wumpus, goud en putten staan stil),
– discreet en
– single-agent (de Wumpus hoort bij de omgeving).

Wumpus-wereld — acties 1

Breeze *Bump*
Stench ↓ *Glitter* ↓ *Scream*
↓ ↓ ↓

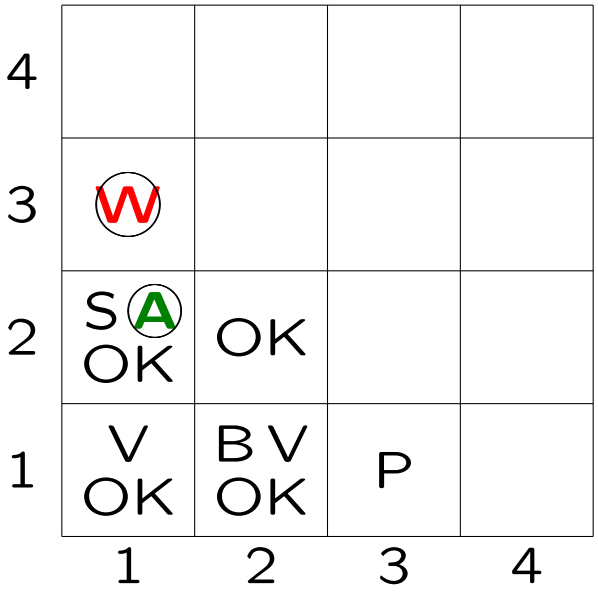
In het begin, na percept $[-, -, -, -, -]$ ($- = None$), en als je naar $(2, 1)$ bent gegaan, met percept $[-, Breeze, -, -, -]$:

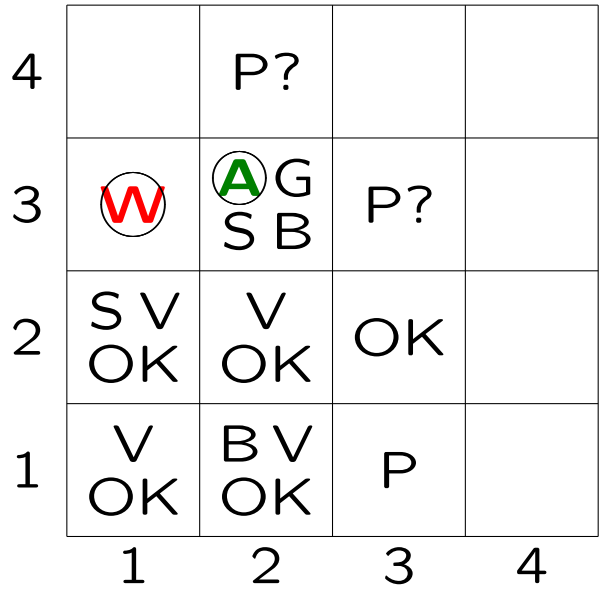


A = Agent; OK = veilige plek; V = bezocht; P = put;
B = Breeze; S = Stench; W = Wumpus; G = Glitter

Stench ↓ *Breeze* ↓ *Glitter* ↓ *Bump* ↓ *Scream* ↓

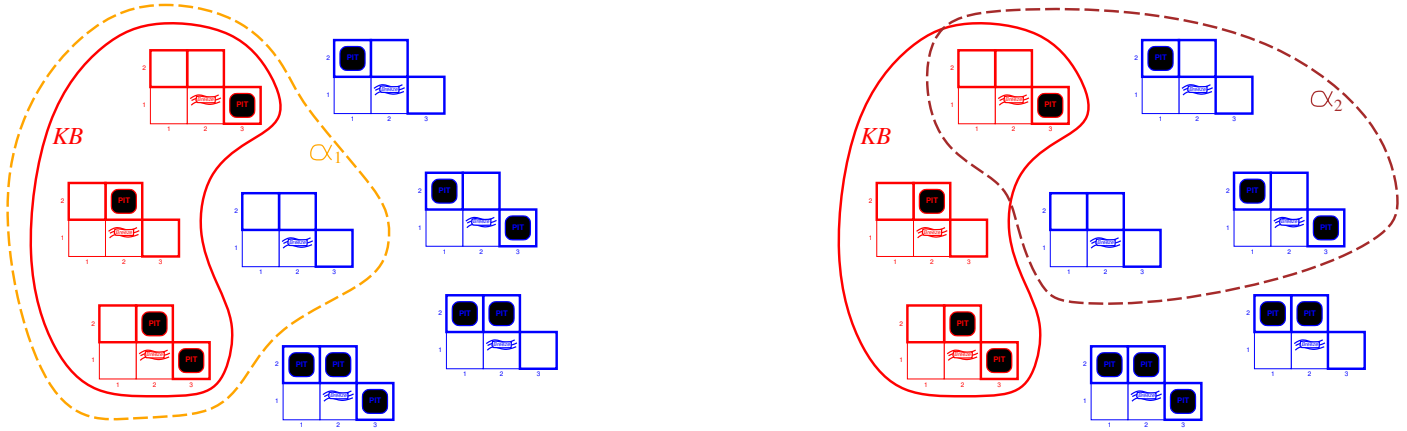
Twee stappen later, met percept $[Stench, -, -, -, -]$, in $(1,2)$ (via $(1,1)$), en als je in stap vijf via $(2,2)$ naar $(2,3)$ bent gegaan, met percept $[Stench, Breeze, Glitter, -, -]$:





A = Agent; OK = veilige plek; V = bezocht; P = put;
B = Breeze; S = Stench; W = Wumpus; G = Glitter

In de tweede situatie hebben we eerder niets in $(1,1)$ waargenomen en nu Breeze in $(2,1)$. We gaan alle "modellen checken", voor wat betreft naburige putten.



Er geldt dat $KB \models \alpha_1$, met $\alpha_1 =$ "Er is geen put in $(1,2)$", maar $KB \not\models \alpha_2$, met $\alpha_2 =$ "Er is geen put in $(2,2)$".

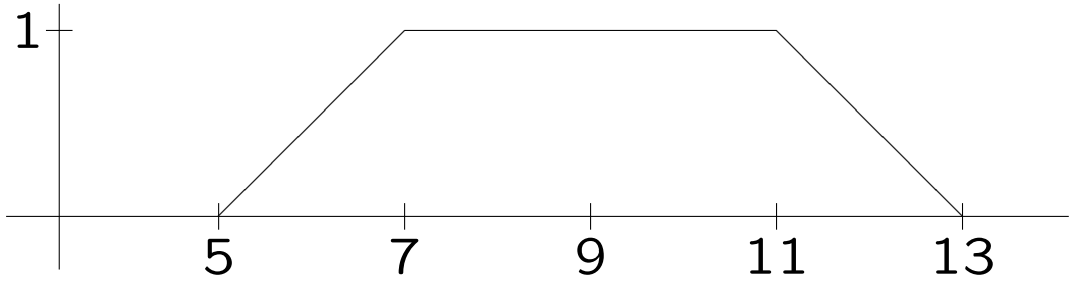
Een eenvoudige logica is de propositie-logica.

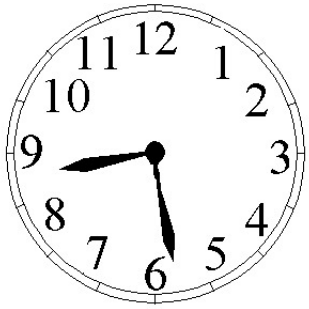
Syntax:

Sentence $\rightarrow$ AtomicSentence | ComplexSentence
AtomicSentence $\rightarrow$ $\mathbf{True}$ | $\mathbf{False}$ | P | Q | R | ...
ComplexSentence $\rightarrow$ (Sentence) |
Sentence Connective Sentence | ¬Sentence
Connective $\rightarrow$ $\wedge$ | $\vee$ | $\Rightarrow$ | $\Leftrightarrow$

Semantiek: de bekende waarheidstafels voor $\neg P$, $P \wedge Q$, $P \vee Q$, $P \Rightarrow Q$ en $P \Leftrightarrow Q$.

Ook mogelijk is fuzzy logic, waarbij in plaats van $\mathbf{False}/\mathbf{True}$ de variabelen reële waarden tussen 0 en 1 krijgen:

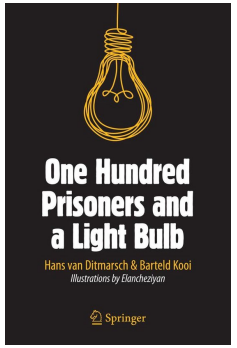




Hierboven staat het predicaat `ochtend` met bijvoorbeeld:

`ochtend`$(6) = 0.5$ en `ochtend`$(8) = 1.0$.

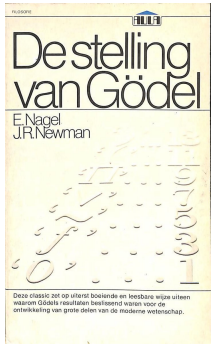
(En verder heb je ook nog temporele logica, en epistemische logica, . . .)



Voor de eenvoudige propositie-logica gelden de "gewone" inferentieregels, zoals:

$$\frac{\alpha \Rightarrow \beta, \alpha}{\beta} \qquad \text{(modus ponens)},$$

$$\frac{\alpha \vee \beta, \neg\beta}{\alpha} \qquad \text{(unit resolutie)},$$



AND-eliminatie, AND-introductie, OR-introductie, dubbele ontkenning, resolutie, ...

De stappen uit de inferentie vormen een bewijs. Als je ware ($\models$, "entailment", "volgen uit") beweringen afleidt ($\vdash$), is de inferentie "sound"; als je alle ware beweringen kunt afleiden is de inferentie "compleet". Zie college Logica. En voor liefhebbers: de Stelling van Gödel.

Voor de Wumpus-wereld introduceren we een grote hoeveelheid Booleaanse logische variabelen, zoals $S_{1,2}$: het stinkt op positie $(1,2)$ = er is Stench in $(1,2)$; en $\neg W_{1,2}$ betekent: er staat geen Wumpus op $(1,2)$.

En regels als:

$$R_1 = \quad \neg S_{1,1} \Rightarrow \neg W_{1,1} \wedge \neg W_{1,2} \wedge \neg W_{2,1}$$

$$R_2 = \quad S_{1,2} \Rightarrow W_{1,3} \vee W_{1,2} \vee W_{2,2} \vee W_{1,1}$$

enzovoorts.

Na de Wumpus of Gold gevonden te hebben moet je deze kennis overigens ook nog in acties vertalen.

Hoe vind je de Wumpus?

1. Modus ponens op $\neg S_{1,1}$ en $R_1 = \neg S_{1,1} \Rightarrow \neg W_{1,1} \wedge \neg W_{1,2} \wedge \neg W_{2,1}$ levert $\neg W_{1,1} \wedge \neg W_{1,2} \wedge \neg W_{2,1}$
2. AND-eliminatie: $\neg W_{1,1}$ $\neg W_{1,2}$ $\neg W_{2,1}$
3. Modus ponens op $\neg S_{2,1}$ en $\neg S_{2,1} \Rightarrow \neg W_{1,1} \wedge \neg W_{2,1} \wedge \neg W_{2,2} \wedge \neg W_{3,1}$ en daarna AND-eliminatie geeft: $(\neg W_{1,1})$ $(\neg W_{2,1})$ $\neg W_{2,2}$ $\neg W_{3,1}$
4. Modus ponens op $S_{1,2}$ en $R_2 = S_{1,2} \Rightarrow W_{1,3} \vee W_{1,2} \vee W_{2,2} \vee W_{1,1}$ geeft: $W_{1,3} \vee W_{1,2} \vee W_{2,2} \vee W_{1,1}$
5. Unit resolutie met $\alpha = W_{1,3} \vee W_{1,2} \vee W_{2,2}$ en $\beta = W_{1,1}$ levert: $W_{1,3} \vee W_{1,2} \vee W_{2,2}$
6. Unit resolutie met $\alpha = W_{1,3} \vee W_{1,2}$ en $\beta = W_{2,2}$ levert: $W_{1,3} \vee W_{1,2}$
7. Unit resolutie met $\alpha = W_{1,3}$ en $\beta = W_{1,2}$ geeft: $W_{1,3}$

Er zijn allerlei regels, zoals

$$W_{1,1} \vee W_{2,1} \vee \ldots \vee W_{4,4}$$

(er is minstens één Wumpus), en $\neg W_{2,3} \vee \neg W_{3,4}$ enzovoorts om maximaal één Wumpus af te dwingen. En een Wumpus stinkt, dus voor alle $1 \leq x, y \leq 4$ hebben we

$$S_{x,y} \Leftrightarrow W_{x,y} \vee W_{x,y-1} \vee W_{x,y+1} \vee W_{x+1,y} \vee W_{x-1,y}$$

(16 regels; langs de randen iets anders) — en ook zoiets voor het verband tussen Breeze en putten.

Nu moet je ook nog bijhouden waar je bent:

$$L^0_{1,1} \wedge \text{FacingEast}^0 \wedge \text{Forward}^0 \Rightarrow L^1_{2,1} \wedge \neg L^1_{1,1}$$

(met tijd-superscript; een "effect-axioma", zie later).

Er zijn dus allerlei problemen met deze eenvoudige predicaten-logica:

- zo krijg je veel te veel proposities (je mag geen variabelen als index gebruiken) en
- zijn er problemen met de tijd, die je eigenlijk ook als index wilt gebruiken ($L^0_{1,1}$: de agent is op tijdstip 0 op positie $(1,1)$).

Dus gaan we over naar eerste orde (predicaten-)logica, met objecten en relaties.

Een krachtiger logica is de eerste orde logica, met syntax:

Sentence $\rightarrow$ AtomicSentence | $\neg$Sentence
| Sentence Connective Sentence | (Sentence)
| Quantifier Variable,... Sentence
AtomicSentence $\rightarrow$ Predicate(Term,...) | Term = Term
Term $\rightarrow$ Function(Term,...) | Constant | Variable
Quantifier $\rightarrow \forall \mid \exists$
Connective $\rightarrow \wedge \mid \vee \mid \Rightarrow \mid \Leftrightarrow$
Constant $\rightarrow A \mid X_1 \mid$ John $\mid \ldots$
Variable $\rightarrow a \mid x \mid s \mid \ldots$
Predicate $\rightarrow$ Before | HasColor | ...
Function $\rightarrow$ Mother | LeftLegOf | ...

Enkele voorbeeldzinnen in eerste orde logica:

$$\forall x \ \text{Cat}(x) \Rightarrow \text{Mammal}(x)$$

$$\forall x \ [\text{Cat}(x) \vee (\exists x \ \text{Brother}(\text{Richard}, x))]$$

In deze laatste kun je de binnenste x beter door een y vervangen.
Een term zonder variabelen heet wel een grond-term.

In Prolog worden de constanten juist met kleine letters geschreven, en de variabelen met hoofdletters: `cat(felix).` (grond-term) en `cat(X) :- mammal(X),furry(X).`

Er zijn ook hogere orde logica's, waarin je bijvoorbeeld over functies kunt quantificeren/redeneren:

$$\forall f, g \ (f = g) \Leftrightarrow (\forall x \ f(x) = g(x))$$

(Overigens is "=" een predicaat Is.)

De speciale quantifier "$\exists!$": $\exists! x$ King(x) is equivalent met $\exists x$ King$(x) \wedge \forall y$ (King$(y) \Rightarrow x = y$)

En dan heb je nog de λ-calculus.
Een λ-expressie als $\lambda x, y \ x^2 - y^2$ werkt als volgt:

$$(\lambda x, y \ x^2 - y^2)(25, 24) = 25^2 - 24^2 = 49$$

Doorgaans gebruiken we eerste orde logica in een geschikt beperkt domein, bijvoorbeeld *familie*. Redelijke basis-predicaten zijn in dat geval Female, Spouse en Parent.

Je krijgt dan regels als:

$\forall m, c$ Mother$(m, c) \Leftrightarrow$ Female$(m) \wedge$ Parent(m, c)

$\forall w, h$ Husband$(h, w) \Leftrightarrow$ Male$(h) \wedge$ Spouse(h, w)

$\forall x$ Male$(x) \Leftrightarrow \neg$Female(x)

$\forall p, c$ Parent$(p, c) \Leftrightarrow$ Child(c, p)

$\forall g, c$ Grandparent$(g, c) \Leftrightarrow \exists p$ (Parent$(g, p) \wedge$ Parent(p, c))

$\forall x, y$ Sibling$(x, y) \Leftrightarrow x \neq y \wedge \exists p$ (Parent$(p, x) \wedge$Parent(p, y))

Dit zijn axioma's; er zijn ook stellingen, zoals

$\forall x, y$ Sibling$(x, y) \Leftrightarrow$ Sibling(y, x)

Prolog is in tegenstelling tot een procedurele/imperatieve programmeertaal als C++ een *declaratieve* taal. Na

```
koe(klara).           % een feit
dier(X) :- koe(X).    % en een regel, waarbij :- als betekent
```

kun je vragen

```
?- dier(klara).
true.
```

Het antwoord wordt gegeven door logische afleidingsregels los te laten op de database met feiten en regels.

Familierelaties, zoals boven, doen het goed in Prolog:

```
zwager(X,Y) :- man(X),brus(X,Z),getrouwd(Z,Y).
zwager(X,Y) :- man(X),getrouwd(X,Z),brus(Z,Y).
zwager(X,Y) :- man(X),getrouwd(X,Z),brus(Z,W),getrouwd(W,Y).
```

Hierbij betekent `brus` broer of zus ("sibling").
En recursie komt vaak handig van pas:

```
voorouder(X,Y) :- ouder(X,Y).
voorouder(X,Y) :- ouder(X,Z),voorouder(Z,Y).
```

We kunnen nu beter dan met de eenvoudige propositie-logica onze Wumpus-wereld beschrijven.

Een "percept-rijtje" op tijdstip 5 ziet er uit als ("Tell"):

$$Percept([Stench, Breeze, Glitter, None, None], 5).$$

We "Ask"-en een query als $\exists a\ BestAction(a, 5)$, en verwachten een lijstje als $\{a/Grab\}$ met "bindingen".

Er zijn eenvoudige "perceptie-regels", zoals:

$$\forall t, s, m, g, c\ Percept([s, Breeze, g, m, c], t) \Rightarrow Breeze(t)$$

En "reflex-gedrag" komt voort uit:

$$\forall t\ Glitter(t) \Rightarrow BestAction(Grab, t)$$

Hoe representeer je de omgeving?

Naast (tot en met hier toe) synchrone regels zijn er overigens ook diachrone (redeneren met tijd erbij; zie later).

Je krijgt regels als:

$\forall x, y, a, b \; Adjacent((x, y), (a, b)) \Leftrightarrow$
$(a, b) \in \{(x + 1, y), (x - 1, y), (x, y + 1), (x, y - 1)\}$
$\forall s, t \; At(Agent, s, t) \wedge Breeze(t) \Rightarrow Breezy(s)$

met coördinaten x, y, a, b, vakje s en tijd t; een vakje is een coördinatenpaar. $Breezy$ hangt niet van t (tijd) af.

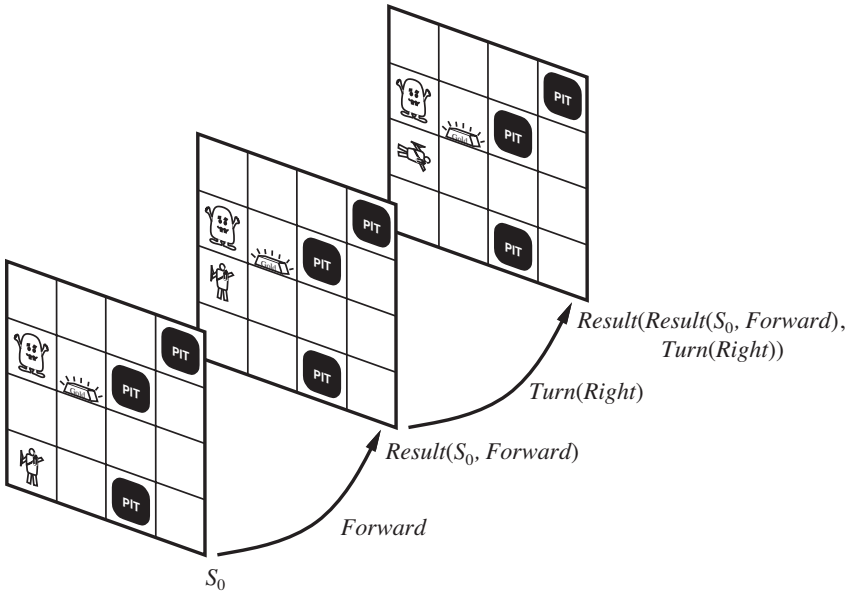
Diagnostische regels leiden van waargenomen effecten naar verborgen oorzaken ("koorts $\Rightarrow$ ziek"), causale regels net andersom ("ziek $\Rightarrow$ koorts"):

$$\forall s \; Breezy(s) \Rightarrow \exists r \; Adjacent(r, s) \wedge Pit(r) \qquad \text{(diagnostisch)}$$

$$\forall r \; Pit(r) \Rightarrow [\forall s \; Adjacent(r, s) \Rightarrow Breezy(s)] \qquad \text{(causaal)}$$

Systemen die redeneren met causale regels heten wel model-based reasoning systemen.

Situaties zijn: de beginsituatie, en dat wat je krijgt als je een actie op een situatie loslaat. De benadering die probeert veranderingen bij te houden heet situation calculus.



Bepaalde "eigenschappen" zijn niet afhankelijk van de tijd, en zijn eeuwig of atemporeel: $Wall((0,1))$.

Effect-axioma's beschrijven veranderingen ten gevolge van acties (met situaties s):

$$\forall s \; AtGold(s) \Rightarrow Holding(Gold, Result(Grab, s))$$

Frame-axioma's beschijven wat onveranderd blijft tijdens acties (met voorwerpen x en acties a):

$$\forall a, x, s \; Holding(x, s) \wedge a \neq Release \Rightarrow Holding(x, Result(a, s))$$

En samen in successor-state axioma's:

$$\forall a, s \; Holding(Gold, Result(a, s)) \Leftrightarrow$$
$$((a = Grab \wedge AtGold(s)) \vee (Holding(Gold, s) \wedge a \neq Release))$$

Dit lost het zogeheten representational frame problem op (te veel dingen waar een actie niets mee doet).

Het is nog weer een verhaal apart hoe je plannen maakt.

En hoe je het inferential frame problem aanpakt.

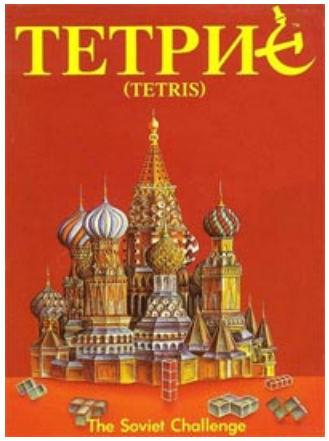
En hoe je voorkeuren verwoordt.

En dan zijn er nog possibility axioma's.

Enzovoorts.

Het huiswerk voor de volgende keer (28 februari 2024): lees Hoofdstuk 3, p. 63–84 van [RN] door (in de derde druk p. 64–109) over het onderwerp Probleemoplossen en zoeken.

Maak de eerste opgave over Monte Carlo & Tetris af — op woensdag 28 februari 2024, om 13:15 uur, is namelijk de deadline! Op de AI-website staat een voorbeeld LaTeX-file voor het verslag (`verslag.tex`) en wat tips (`opm.pdf`).



Kunstmatige Intelligentie (AI)

Hoofdstuk 3 (tot en met 3.4) van Russell/Norvig = [RN]

Probleemoplossen en zoeken

voorjaar 2024

College 4, 28 februari 2024

www.liacs.leidenuniv.nl/~kosterswa/AI/zoeken.pdf

Een probleemoplossende agent zoekt zijn weg naar een doel (goal), en gebruikt toegestane acties.

Er zijn vele soorten problemen (zie later), onder meer:

single-state — "weet alles"

multiple-state — beperkte kennis van de wereld

contingency — onvoorzien / onzekerheid; bij executie-fase opletten

exploration — experimenteren

In dit hoofdstuk is de probleemomgeving statisch, (doorgaans) volledig observeerbaar, discreet, deterministisch, sequentieel, en voor één agent.
Kortom, zo ongeveer de "eenvoudigste" soort omgeving: een standaard-omgeving.

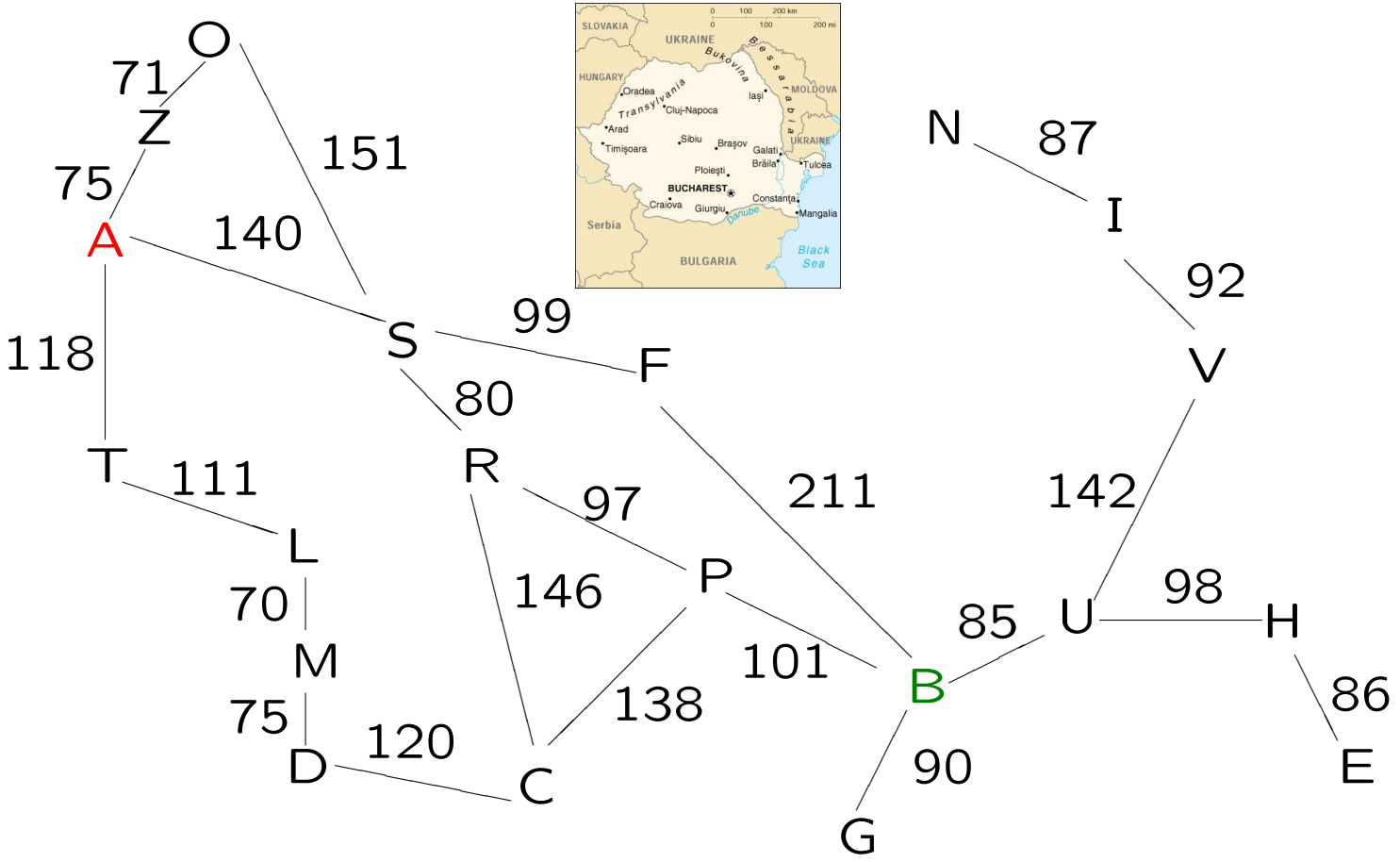
Het probleem heeft een toestand-actie-ruimte, zie ook het college Algoritmiek. Paden hierin (toestanden door acties verbonden) zijn mogelijke oplossingen.

Goed-gedefinieerde problemen hebben:

- één of meer begintoestanden (= initial states)
- één of meer doeltoestanden (= goal states)
- toegestane acties; per toestand x geeft de opvolger (successor) functie $S(x)$ een verzameling paren van het type $\langle$*actie*, *opvolger*$\rangle$, waarbij *actie* van x naar *opvolger* voert; soms gedefinieerd via "operatoren"
- een functie g die kosten aan paden toekent, en wel de som van de afzonderlijke stappen; de stapkosten voor actie a van toestand x naar toestand y zijn $c(x, a, y) \geq 0$

Voorbeeld: Roemenië

We gaan reizen in Roemenië, en maken allerlei abstracties:



We willen zo "snel" mogelijk van Arad naar Bucharest.

We hebben dus een abstractie gemaakt, een toestand is bijvoorbeeld: we bevinden ons in Arad, kortweg A.

Als we later onverhoopt opnieuw in Arad komen, zitten we weer in die zelfde toestand. Maar we hebben dan wel een heel pad afgelegd! Vaak slaan we dit soort zinloze paden over — maar bij het programmeren moeten we er wel op letten.

Als het probleem zou zijn "bezoek alle steden minstens één keer", bevatten de toestanden *wel* de hele tot dan toe afgelegde route.

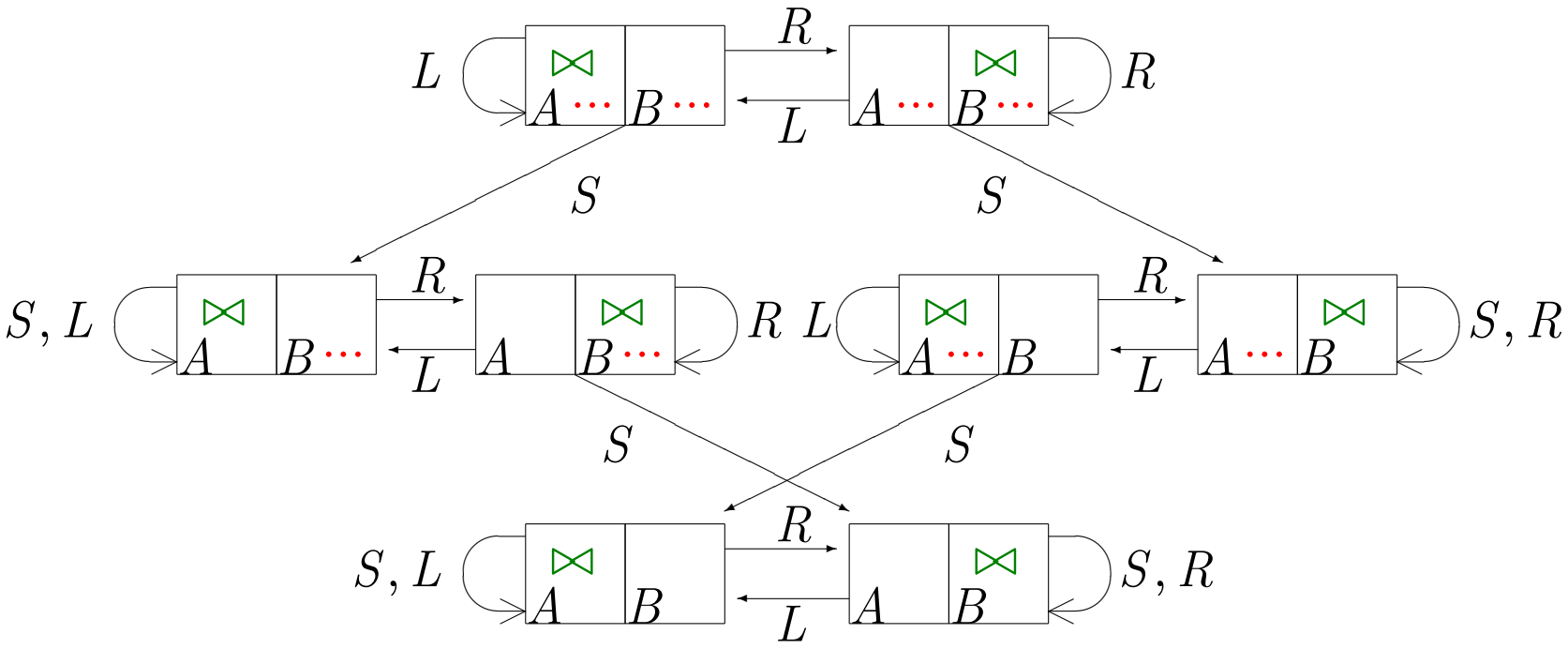
Voorbeeld: Stofzuiger-wereld — 1

Voor de vereenvoudigde Stofzuiger-wereld hebben we:

- acht toestanden: de stofzuiger bevindt zich in A (links) of B (rechts), A is vuil (*Dirty*) of schoon (*Clean*), B is vuil of schoon
- acties: $L = Left$, $R = Right$ (helemaal (!) naar rechts), $S = Suck$, $N = Nothing$
- doel-toestand: zowel A als B schoon
- elke stap kost 1

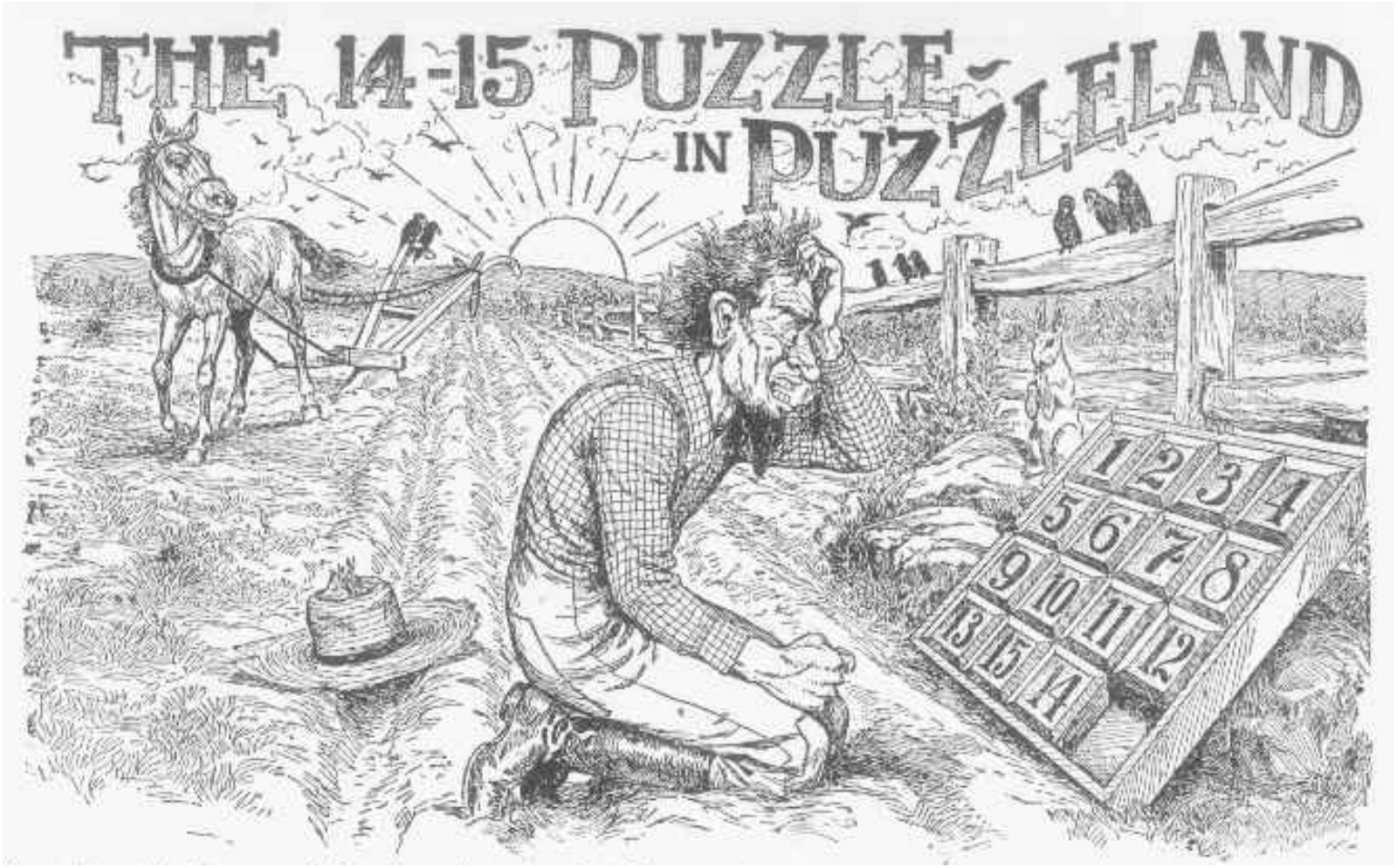
NB In de vorige versie gingen *Left* en *Right* een stukje naar links/rechts.

Voorbeeld: Stofzuiger-wereld — 2

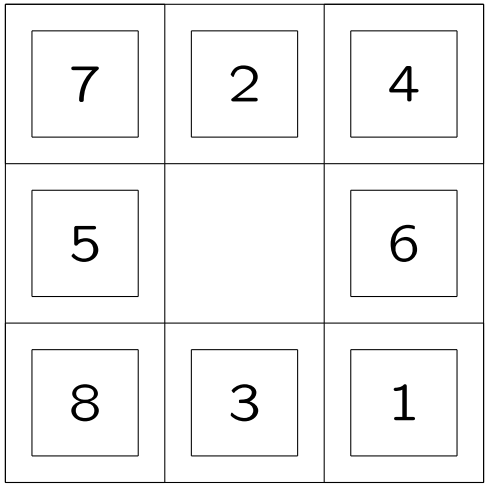


Legenda van het toestand-actie-diagram: stofzuiger: ⋈; stof (*Dirty*): ...; *L(eft)*; *R(ight)*; *S(uck)*. *N* is weggelaten.

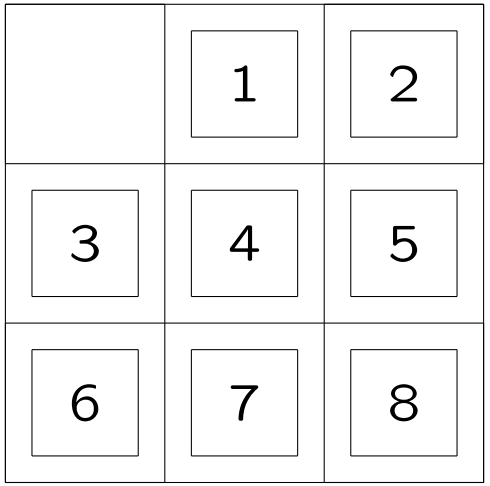
Voor zowel het single-state probleem als het multiple-state probleem is *R*,*S*,*L*,*S* een oplossing.


THE 14-15 PUZZLE
IN PUZZLELAND
1 2 3 4
5 6 7 8
9 10 11 12
13 15 14

Voorbeeld: De 8-puzzel



begintoestand



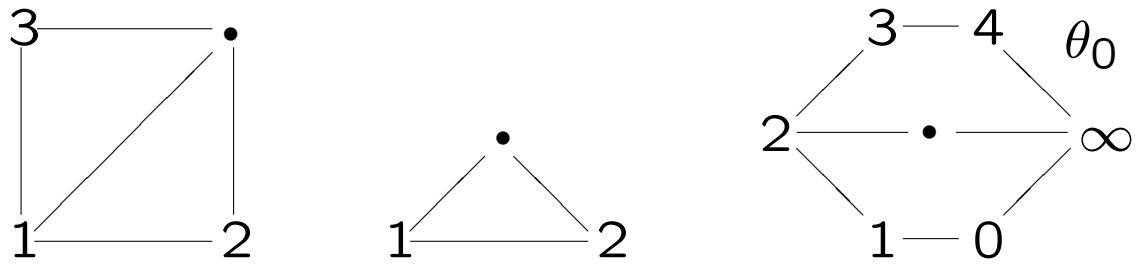
doel-toestand

Je mag een getal verschuiven naar een (horizontaal of verticaal) aangrenzende lege plek. Er zijn $9! = 362.880$ toestanden, waarvan de helft bereikbaar is vanuit een gegeven begintoestand. Internet: Sam+Loyd+fifteen.

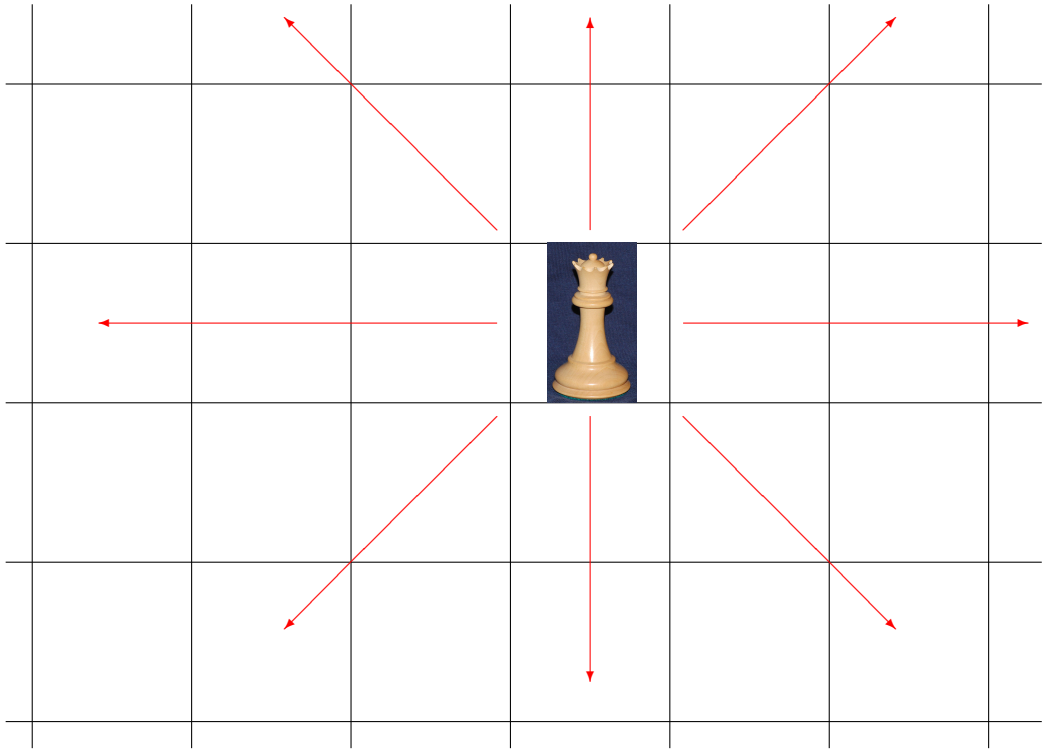
Voorbeeld: De 15- en 24-puzzel

De 15-puzzel (uitvinder: Noyes Chapman, 1880, en niet Sam Loyd) kan optimaal worden opgelost in enkele seconden, vanuit elke begintoestand. Er zijn, vanuit elke begintoestand, $16!/2 = 20.922.789.888.000/2$ toestanden bereikbaar — de helft van alle. De 24-puzzel, op een 5×5 bord, is door huidige computers binnen enkele uren op te lossen. Aantal toestanden: $25! \approx 10^{25}$.

Wilson heeft dit soort puzzels geclassificeerd (zie mathworld). De "Tricky Six Puzzle" (θ_0-graaf, rechts) heeft maar liefst 6 verschillende "samenhangscomponenten":



Schaken — De dame



www.liacs.leidenuniv.nl/~kosterswa/nqueens/

Voorbeeld: 8 dames — 1

Het bekende 8 dames probleem valt op verschillende manieren te beschrijven. Het gaat er om 8 (of n) dames op een 8 bij 8 (n bij n) schaakbord te zetten, zodanig dat geen dame een andere kan "zien" (= aanvalt). Een dame ziet een andere dame in dezelfde rij, kolom of diagonaal.

Doeltoestand: 8 "correcte" dames op een bord (92 mogelijkheden, waarvan 12 "unieke": $92 = 11 \times 8 + 1 \times 4$).

Beschrijving 1 ("incrementeel"):
Toestanden: elke opstelling van $0 \ldots 8$ dames op een bord.
Actie: dame ergens toevoegen.
Aantal te onderzoeken rijtjes: $64 \cdot 63 \cdots 57 \approx 3 \times 10^{14}$.

Beschrijving 2 ("incrementeel", zie Algoritmiek):
Toestanden: elke opstelling van 0...8 dames in de meest linker kolommen, waarbij geen dame een andere aanvalt.
Acties: zet een dame in de meest linkse lege kolom, zodanig dat deze niet wordt aangevallen door een eerdere dame.

Nu "slechts" 2057 mogelijke rijtjes te onderzoeken.
Algemeen, op een n bij n bord: $\geq \sqrt[3]{n!}$ (NB $\sqrt[3]{8!} \approx 16$).
Bewijs: zeg x mogelijke rijtjes. Dan $x \geq n(n-3)(n-6)\ldots$ (elke voorgaande kolom verbiedt hoogstens 3 posities in de huidige). Dus

$$\begin{aligned} x^3 &\geq n^3(n-3)^3(n-6)^3\ldots \\ &\geq n(n-1)(n-2)(n-3)\ldots = n! \end{aligned}$$

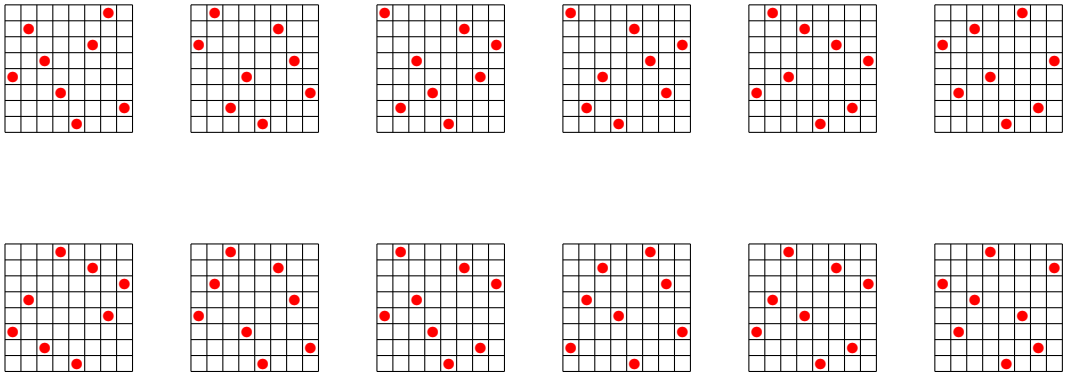
(In 2021 bewees Michael Simkin overigens dat er $\approx (0.143n)^n$ oplossingen zijn op het $n \times n$ bord.)

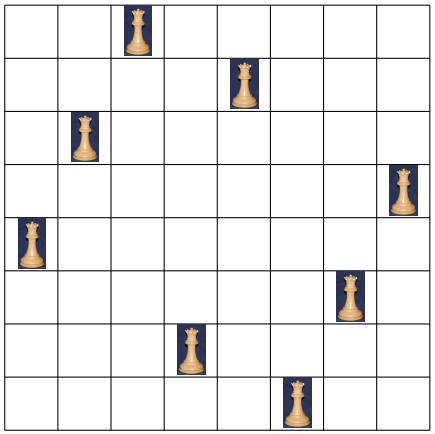
Voorbeeld: 8 dames — 3

Kortom: de juiste formulering maakt een groot verschil voor de grootte van de zoekruimte!

Beschrijving 3 (“complete-state”):
Toestanden: opstellingen van 8 dames, één in iedere kolom.
Acties: verplaats een aangevallen dame naar een andere plek in dezelfde kolom — zie ook later.

De oplossingen:





Voorbeeld: Een "echt" probleem

Een echt (reis)probleem uit de luchtvaart zou kunnen zijn:

- toestanden: locatie (vliegveld) en tijd ter plaatse
- opvolger-functie: de vanaf de betreffende locatie nog mogelijke vluchten
- doel-test: zijn we op een redelijke tijd ter bestemming?
- padkosten: geld, tijd, enzovoorts

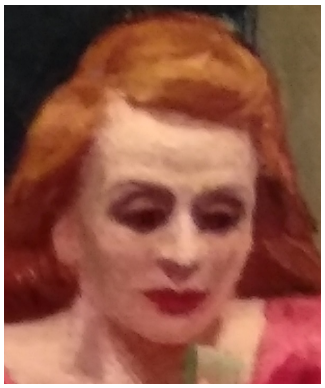
Tarieven kunnen uiterst complex zijn.
Let ook op eventuele alternatieven in geval er iets mis gaat.

Andere echte problemen: VLSI-ontwerp, robot-navigatie, TSP (Traveling Salesman Problem), ontwerp van eiwitten, internet-agenten, . . .

Als de agent precies weet in welke toestand hij is, en acties eenduidige resultaten hebben, spreken we van een single-state probleem. Deterministisch, volledig observeerbaar.

Als het niet volledig observeerbaar is: multiple-state of sensorloos of conformant. Je hebt dan belief toestanden: verzamelingen toestanden, waarbij je alleen maar weet in welke verzameling je zit.

In beide gevallen: oplossingen zijn *rijtjes*.



Als het probleem niet-deterministisch en/of gedeeltelijk observeerbaar is, hebben we een contingency probleem (onzekerheid). Als de onzekerheid wordt veroorzaakt door de acties van een andere agent: adversarial — zie Spel(l)en.

De oplossing is een *boom* of *policy*.

Als de toestandsruimte (= zoekruimte) onbekend is, spreken we van (online) exploratie.

Een zelfde probleem kan vele versies hebben!

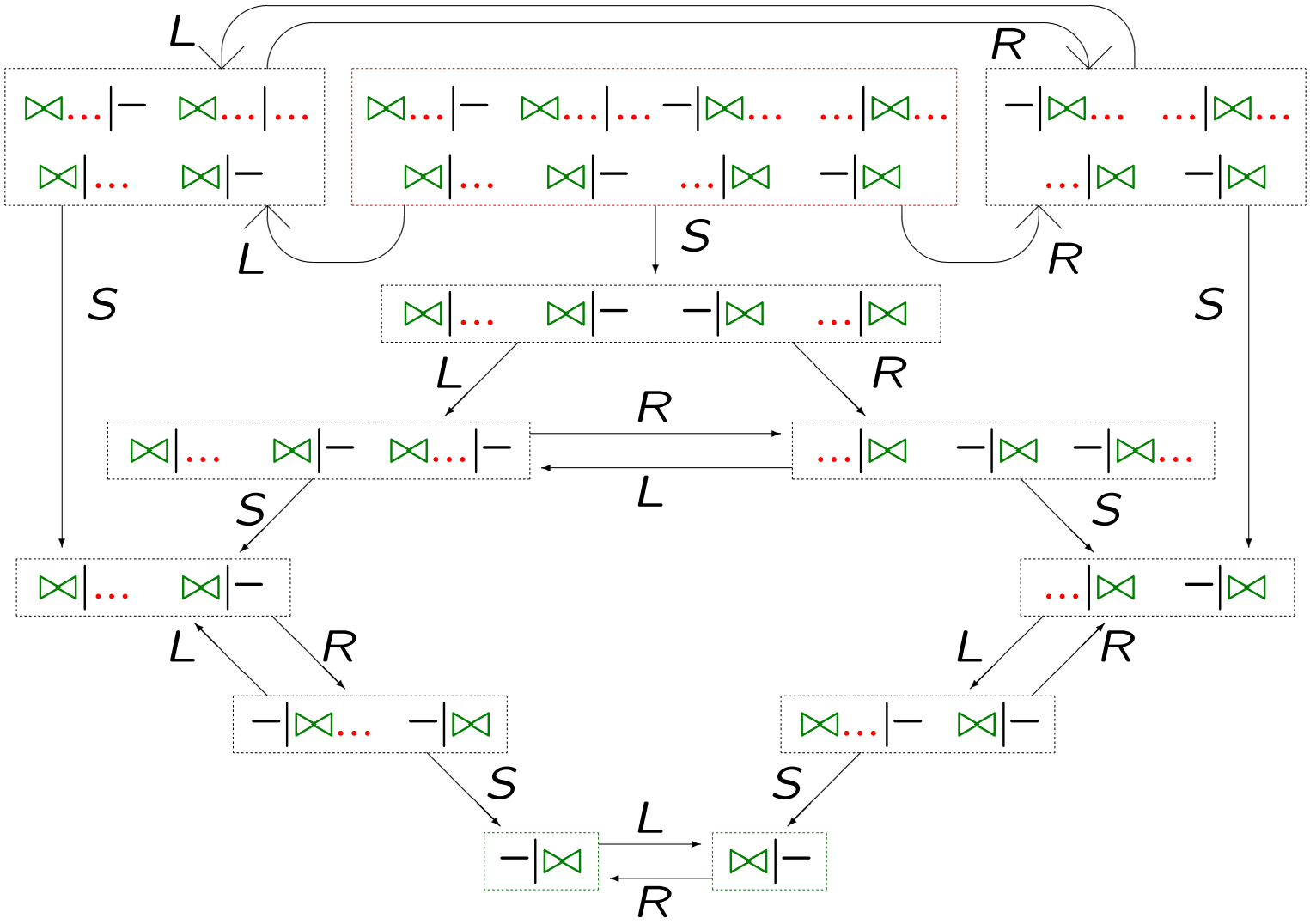
We noteren weer: stofzuiger: $\bowtie$; stof (*Dirty*): ...; *L(eft)*; *R(ight)*; *S(uck)*. En met een | de scheidsmuur; − is een schone ruimte (zonder stofzuiger).

Voor het speciale single-state probleem, beginnend in $\bowtie$|..., is *R*,*S* een oplossing.

Voor het conformant probleem (= multiple-state), ergens beginnend, is het rijtje *R*,*S*,*L*,*S* een oplossing. Merk op dat je na *R* zeker weet dat de stofzuiger in *B* is!

Voor het contingency-probleem, nu beginnend in $\bowtie$|??, en *Murphy-Suck* (maakt *soms* vuil), is *R*, **if** *Dirty* **then** *M* een oplossing (*M* = *Murphy-Suck*).

Stofzuiger — Multiple-state



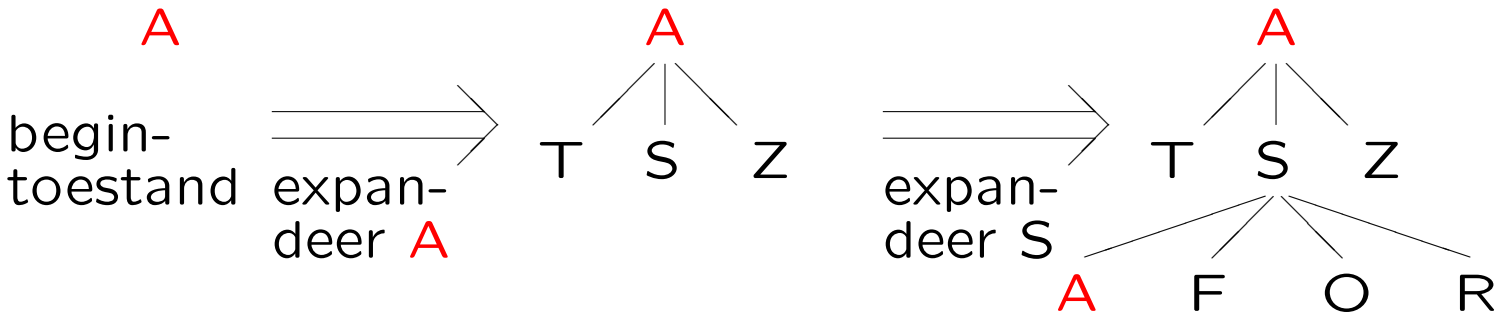
We hebben hier belief states, waaronder begin en doel.

Er zijn 12 vanuit de begintoestand bereikbare belief states = info(rmation) sets, die ieder bestaan uit één of meer fysieke toestanden. We zoeken een pad naar een belief state die geheel uit doel-toestanden bestaat.

De complete belief ruimte bestaat uit 2^S toestanden, als de fysieke toestandsruimte er S heeft; hier dus $2^8 = 256$.

Overigens hebben we flauwe acties/pijlen weggelaten.

Een zoekstrategie zegt je in welke volgorde de toestanden ge-expandeerd (= ontwikkeld) moeten worden. Door knopen te genereren wordt een zoekboom opgebouwd (niet te verwarren met de oorspronkelijke graaf!):



De kandidaten voor expansie vormen samen de frontier = grens. Ze worden doorgaans in een rij (queue) geordend. De plaats in de rij waar nieuwe elementen komen wordt bepaald door de zoekstrategie.

Ongericht zoeken

Bij ongeinformeerd of ongericht of blind zoeken hebben we geen extra informatie over de toestanden, afgezien van de probleemdefinitie.

Bij gericht zoeken, zie [RN] Hoofdstuk 3.5/3.6, gebruiken we wel meer informatie. Volgende keer!

Een knoop (= node) is een data-structuur met vijf componenten:

- de toestand uit de zoekruimte waarmee de knoop correspondeert (voorbeeld Roemenië: F)
- de ouderknoop, die deze knoop genereerde (S)
- de actie die is toegepast ($S \rightarrow F$)
- de padkosten van begintoestand tot nu toe (239)
- de diepte: het aantal stappen vanuit begintoestand tot nu toe (2)

We bekijken de volgende zoek-algoritmen:

- Breadth-First Search = BFS
- Uniform cost search = Dijkstra
- Depth-First Search = DFS
- Depth-limited search
- Iterative deepening depth-first search
- Bidirectional search

Het "generieke" algoritme in pseudo-taal is:

frontier $\leftarrow$ knoop met begintoestand
while true do
    if *frontier* $= \emptyset$ **then**
        return failure
    knoop $\leftarrow$ eerste uit *frontier*
    if *knoop* representeert een doeltoestand **then**
        return oplossing
    frontier $\leftarrow$ *frontier* met toegevoegd alle knopen uit
        de expansie van *knoop*

De strategie bepaalt dus de volgorde van toevoegen!
NB Controleren of je in een doeltoestand bent vindt pas plaats als je op het punt staat de knoop te expanderen.

Criteria voor zoekstrategieën

Er zijn vier belangrijke criteria om zoekstrategieën op te beoordelen:

compleetheid Vinden we gegarandeerd een oplossing — mits die er is?

tijdcomplexiteit Hoe lang duurt het?

ruimtecomplexiteit Hoe veel geheugen vergt het?

optimaliteit Vinden we de *optimale* oplossing, dus die met de laagste padkosten?

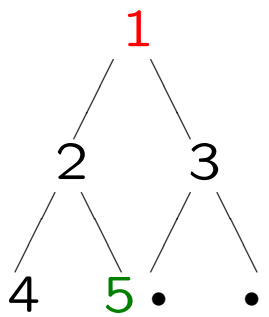
De complexiteit wordt vaak uitgedrukt in drie grootheden:

- de branching factor (vertakkingsgraad) b: het hoogste aantal "opvolgers" van een knoop, oftewel het grootste aantal kinderen

- de diepte d van het meest ondiepe ("shallowest") doel

- de maximale lengte m van een pad in de toestandsruimte

Voor Breadth-First Search (BFS) stop je gewoon de nieuwe kandidaten *achterin* de frontier — het is dus een echte rij: FIFO (First In First Out).



volgorde van
expanderen van
de knopen
$b = 2,\ d = 2,\ m = 2$
5 is doelknoop

BFS is compleet (als $b < \infty$), en optimaal — mits de padkosten-functie een niet-dalende functie is van de diepte van de knopen (en alleen daarvan afhangt).

Het maximaal aantal knopen dat ge-expandeerd is als je een oplossing op diepte d vindt is

$$1 + b + b^2 + \ldots + b^d = \frac{b^{d+1} - 1}{b - 1} \quad ,$$

waarbij b de vertakkingsgraad is. En er zijn er ongeveer b keer zoveel gegenereerd.

Tijd- en ruimtecomplexiteit: $O(b^d)$ of eventueel $O(b^{d+1})$.

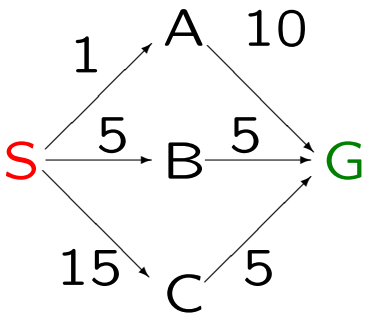
Het ruimteprobleem is het ergste. Bijvoorbeeld, voor $b = d = 10$, is 10^{11} tijdseenheden handelbaarder dan 10^{11} geheugen-eenheden. Maar toch gaat het met de tijd ook niet fijn.

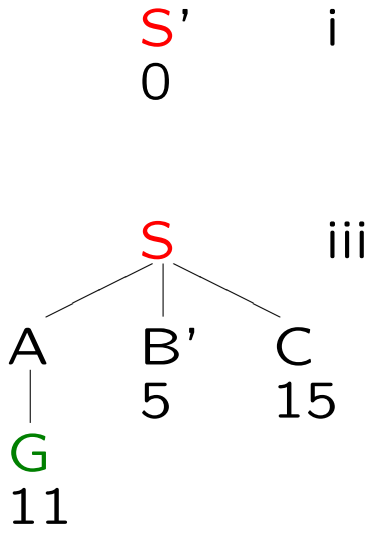
Uniform cost search — 1

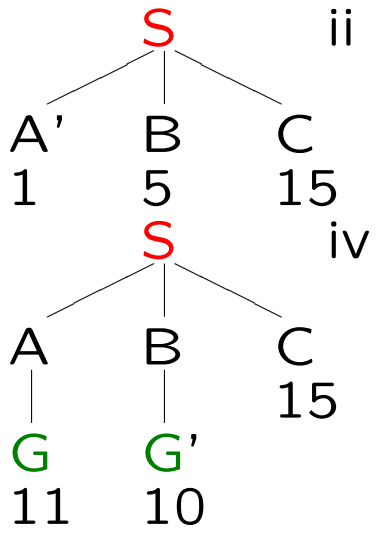
Bij Uniform cost search, in Europa ook bekend als Dijkstra's algoritme, wordt steeds als eerste de knoop met de *laagste padkosten* ge-expandeerd. De frontier is een rij, geordend op padkosten.

Compleet mits de kosten van iedere stap groter zijn dan een vaste $\epsilon > 0$, en dan ook optimaal.

Merk op dat BFS hetzelfde is als Uniform cost in de situatie dat alle takken kosten 1 hebben.



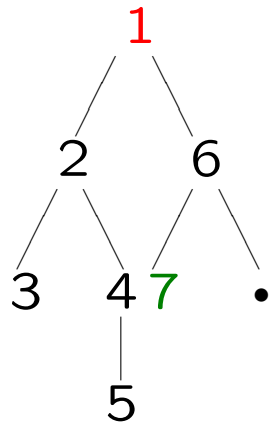




Er staat een ' bij de knoop die ontwikkeld wordt.
Let op: deze graaf is gericht (dus niet terug naar S)!

Stel dat C^* de kosten zijn van een optimale oplossing, en dat alle acties minstens $\epsilon > 0$ kosten. Dan is de worst case tijd- en ruimtecomplexiteit $O(b^{\lfloor C^*/\epsilon \rfloor + 1})$ (de oplossing zou wel eens op diepte $\approx C^*/\epsilon$ kunnen zitten). Als alle acties even veel, zeg ϵ, kosten geldt $C^*/\epsilon = d$, zie BFS.

Voor Depth-First Search (DFS) stop je gewoon de nieuwe kandidaten *voorin* de frontier: een stapel = stack, LIFO (Last In First Out).



volgorde van
expanderen van
de knopen
$b = 2,\ d = 2,\ m = 3$
7 is doelknoop

Bij maximum diepte m hebben we $O(bm)$ ruimte (het pad waarop je zit, inclusief "broers"; of, met handig backtracken, $O(m)$) nodig, en $O(b^m)$ tijd.

DFS is niet compleet en ook niet optimaal. Je moet het zeker niet gebruiken voor zoekbomen met grote (of zelfs oneindige) diepte.

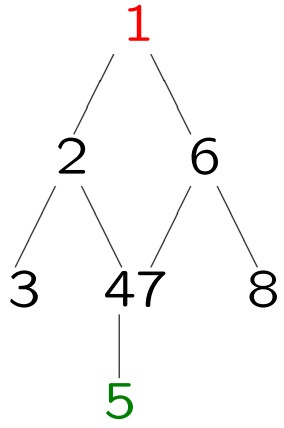
Voor Depth limited search doe je gewoon DFS, alleen houd je op bij een zekere (voorbedachte) diepte. Soms *weet* je dat er een oplossing van een zekere maximale diepte is, bijvoorbeeld dankzij de *diameter* van de zoekruimte. In Roemenië hebben we 20 steden, dus maximaal padlengte 19 (de echte diameter is overigens 9).

Je krijgt dan de goede eigenschappen van DFS, zonder gevaar oneindig ver af te dwalen, maar nog steeds: niet compleet!

Als ℓ de limiet op de diepte is, dan is de tijdcomplexiteit $O(b^\ell)$ en de ruimtecomplexiteit $O(b\ell)$.

Iterative deepening DFS

Voor Iterative deepening (depth-first) search combineer je alle ideeën uit het voorgaande. Je doet herhaald een depth limited search, net zolang tot je een oplossing hebt, waarbij je de limiet op de diepte steeds met 1 verhoogt.



limiet = 0: knoop 1

limiet = 1: knopen 1, 2 en 6

limiet = 2: knopen 1, 2, 3, 4, 6, 7 en 8

limiet = 3: knopen 1, 2, 3, 4 en 5

Als d weer de diepte van de meest "ondiepe" oplossing is, dan is de tijdcomplexiteit $O(b^d)$ en de ruimtecomplexiteit $O(bd)$. Bekeken knopen: $(d+1)1 + db + (d-1)b^2 + \ldots + b^d$.

Bij bidirectional search werk je tevens vanuit het doel terug: je hebt dus naast successors ook voorgangers = predecessors nodig.

De tijd- en ruimtecomplexiteit kunnen $O(b^{d/2})$ worden. Wel moet je de doorsnede van twee frontiers efficiënt bepalen.

Het kan lastig zijn predecessors te vinden. En soms heb je meer doelen . . .

Voorbeeld: analyse van een spel met een eindspel-database (checkers).

De verschillende zoekstrategieën

	tijd	ruimte	optimaal?	compleet?
BFS	$O(b^d)$	$O(b^d)$	Ja^e	Ja^a
Uniform cost	$O(b^{\lfloor C^*/\epsilon \rfloor + 1})$	$O(b^{\lfloor C^*/\epsilon \rfloor + 1})$	Ja	Ja^{a,c}
DFS	$O(b^m)$	$O(bm)$	Nee	Nee
Depth limited	$O(b^\ell)$	$O(b\ell)$	Nee	Ja^a, als $\ell \geq d$
Iter. deepen.	$O(b^d)$	$O(bd)$	Ja^e	Ja^a
Bidirectional	$O(b^{d/2})$	$O(b^{d/2})$	Ja^{e,f}	Ja^{a,f}

Hierbij: b = branching factor, d = diepte van oplossing, m = maximale zoekdiepte, ℓ = limiet op de diepte, ϵ = ondergrens kosten acties, C^* = kosten optimale oplossing. ^{a} als b eindig; ^{c} als stapkosten $\geq \epsilon > 0$, ^{e} als stapkosten alle gelijk, ^{f} met BFS in beide richtingen.

Het huiswerk voor de volgende keer (6 maart 2024): lees Hoofdstuk 3.5/3.6, p.84–100 van [RN] (in de derde druk p.120–130) door over het onderwerp Gericht zoeken.

Denk tevens aan de tweede opgave: Agenten & Robotica; deadline: woensdag 20 maart 2024.

Bestudeer de opgaven van:

www.liacs.leidenuniv.nl/~kosterswa/AI/opgaven1.pdf

www.liacs.leidenuniv.nl/~kosterswa/AI/opgaven2.pdf

Deze worden ook gemaakt op de sommenwerkcolleges, in het bijzonder op 7 maart: opgaven 2, 3, 7, 8, 14.

Kunstmatige Intelligentie (AI)

Hoofdstuk 3.5/3.6 van Russell/Norvig = [RN]

Gericht zoeken

voorjaar 2024

College 5, 6 maart 2024

www.liacs.leidenuniv.nl/~kosterswa/AI/gericht.pdf

Een informed zoekmethode gebruikt meer dan alleen de probleem-omschrijving, ten einde beter (gericht) te kunnen zoeken.

We bekijken best first methoden, die een knoop n kiezen met de laagste waarde voor een geschikte evaluatiefunctie $f(n)$. Deze gebruikt een gegeven heuristische functie $h(n)$ die de afstand vanuit knoop n tot het doel schat; altijd moet gelden dat $h(n) = 0$ voor doelknopen.

$\rightarrow$ Greedy best-first search probeert de afstand tot het doel zo snel mogelijk te verkleinen.
$\rightarrow$ A* ("A-ster") probeert de totale afstand/kosten zo laag mogelijk te houden.
$\rightarrow$ Om geheugen te sparen kun je IDA* gebruiken.

Roemenië

O
71
Z
151
75
A
140
S
99
F
118
80
T
111
R
97
211
142
L
P
V
70
146
98
85
U
H
M
101
B
86
75
138
D
120
C
90
E
G
N
87
I
92

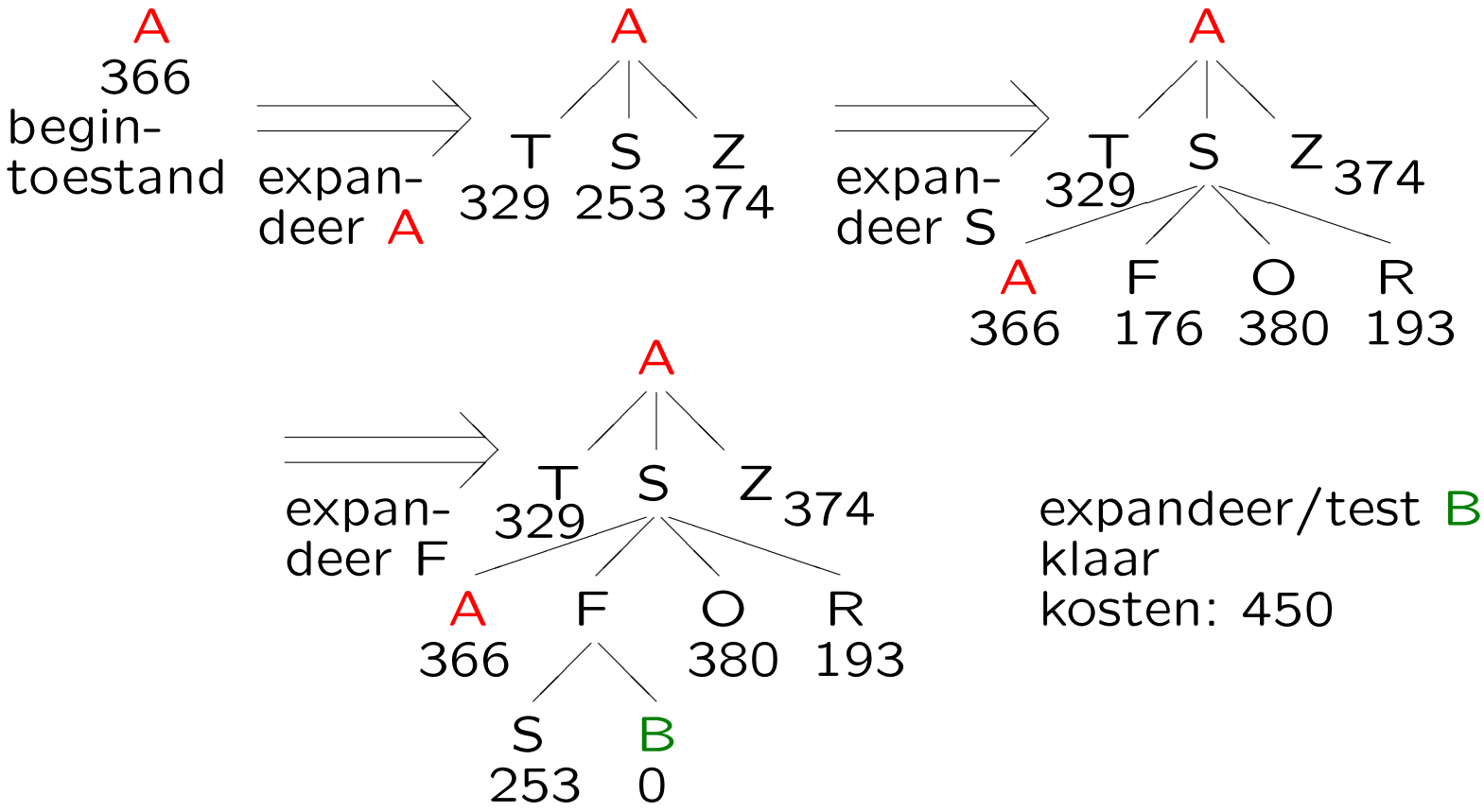
We bekijken weer de reis van A(rad) naar B(ucharest) in Roemenië.

We gebruiken als heuristiek h_{SLD}, de vogelvlucht afstand (straight line distance) tot het doel:

A	B	C	D	E	F	G	H	I	L
366	0	160	242	161	176	77	151	226	244
M	N	O	P	R	S	T	U	V	Z
241	234	380	100	193	253	329	80	199	374

Merk op dat $h_{SLD}(\text{B}) = 0$. En h_{SLD} is een *onderschatting* voor de echte afstand tot het doel.

Greedy best-first search: ontwikkel ("frontier"-)knoop n met de laagste $h(n)$, in dit geval $h = h_{SLD}$. Oftewel: $f(n) = h(n)$.



Greedy best-first search 2

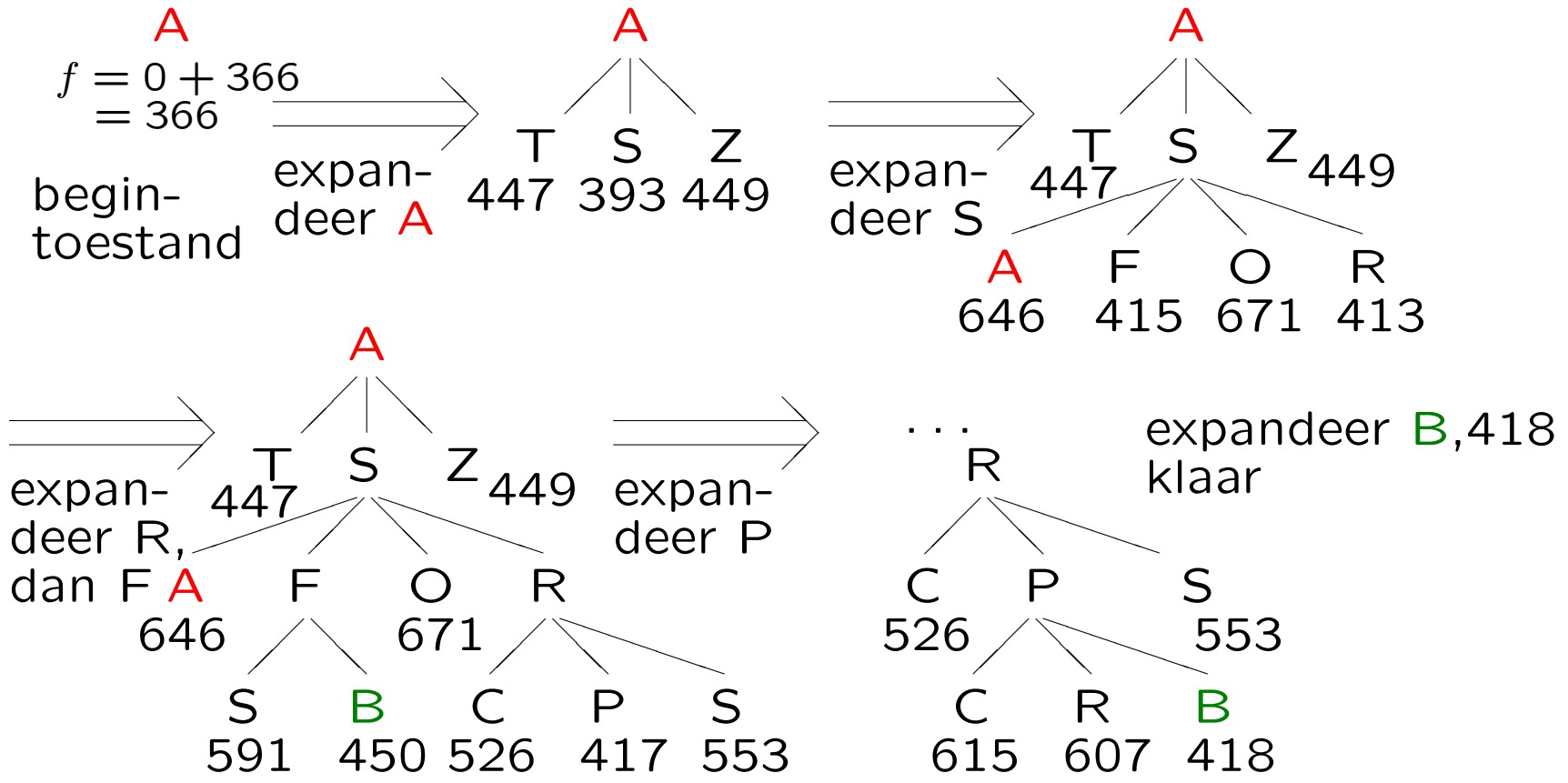
Greedy best-first search heeft tijd- en ruimtecomplexiteit $O(b^m)$, met b de vertakkingsgraad en m de maximale diepte van de zoekruimte.

Helaas: niet compleet (oneindige paden . . .) en niet optimaal (korter via R en P, zie straks)!

Beter idee: breng ook de verbruikte afstand in rekening.

Als $h(n) = 0$ voor alle n is dit hetzelfde als Uniform cost search = Dijkstra's algoritme.

Voor A* ("A-ster") definiëren we $f(n) = g(n) + h(n)$, een schatting voor het totale pad van begin naar doel.



A* — samengevat

Het A*-algoritme is dus als volgt:

> Expandeer steeds de (een) knoop uit de "frontier" met de laagste f-waarde, waarbij voor knoop n geldt: $f(n) = g(n) + h(n)$. Hierbij is $g(n)$ de exacte waarde van de reeds gemaakte kosten, en $h(n)$ de (toelaatbare) schatting voor de rest van de kosten. Stop als je een doelknoop expandeert — laten we aannemen dat dat ooit gebeurt.

Onder redelijke voorwaarden (zie straks) is A* compleet en optimaal; er zijn allerlei handige aanpassingen mogelijk. Let op "herbezoeken" van locaties.

Een heuristiek h heet admissibel = toelaatbaar als hij nooit de kosten naar het doel overschat.
Voorbeeld: h_{SLD}. Of (flauw): $h(n) = 0$ voor alle n.

Bewering: Als h admissibel is, is A* optimaal: A* vindt het "beste" doel — als A* er één ontdekt, tenminste.
Bewijs: Stel dat je een niet-optimaal doel G_2 op de "frontier" hebt. De kosten naar een optimaal doel zijn C^*, zeg. Dan: $f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^*$. Pak een knoop n op de "frontier", die op een pad naar een optimale oplossing zit (zo'n knoop is er). Dan: $f(n) = g(n) + h(n) \leq C^*$ (h admissibel). Dus $f(n) \leq C^* < f(G_2)$. Dus n wordt voor G_2 ge-expandeerd.

Een heuristiek h heet monotoon = consistent wanneer altijd — als je via actie a van knoop n naar knoop n' gaat — geldt $h(n) \leq c(n, a, n') + h(n')$. Dit is een soort driehoeks-ongelijkheid.

In dat geval zijn de waarden van f op ieder pad niet-dalend:

$$\begin{aligned} f(n') &= g(n') + h(n') = g(n) + c(n, a, n') + h(n') \\ &\geq g(n) + h(n) = f(n). \end{aligned}$$

En A* expandeert alle knopen met $f(n) < C^*$ en een of meer knopen met $f(n) = C^*$ (met C^* de kosten van een optimale oplossing) — mits dat er *eindig veel* zijn, en is dan dus ook compleet. Je krijgt een soort contourlijnen.

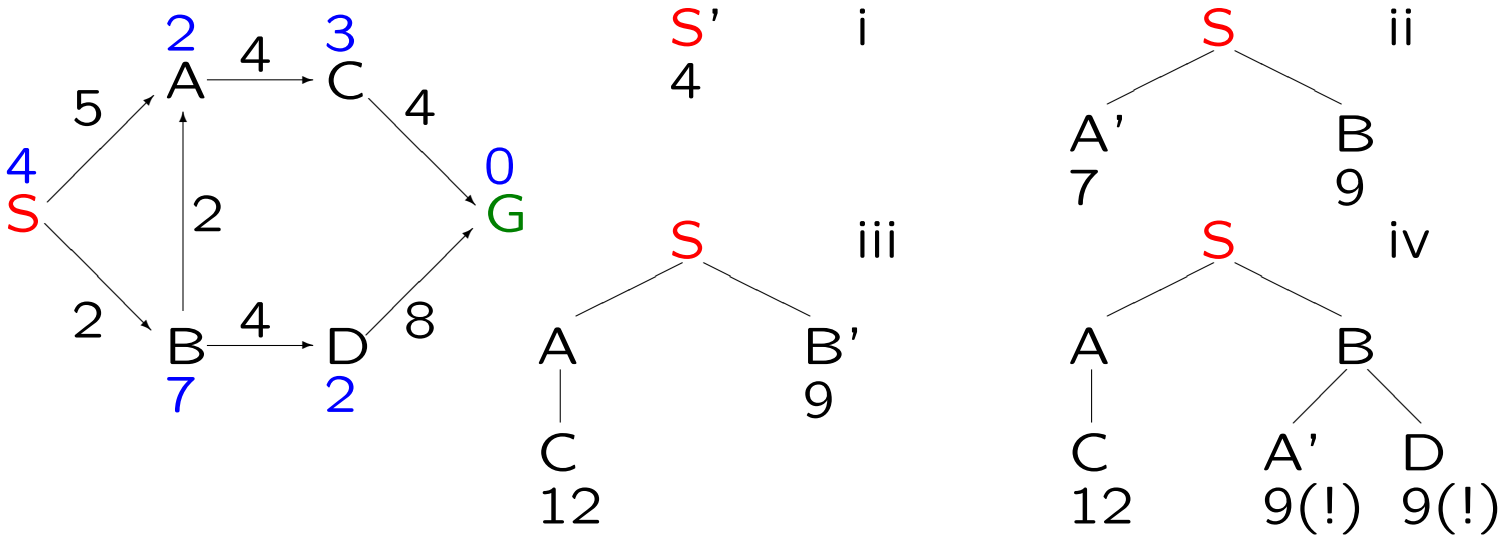
Stel dat de f-waarde daalt langs een zeker pad — de heuristiek is dan blijkbaar niet consistent. Wat kun je daaraan doen? Oplossing: de pathmax-vergelijking toepassen:

$$f(n') = \max(f(n), g(n') + h(n')).$$

Merk op: consistent impliceert admissibel (maar niet omgekeerd). Bewijs: met inductie naar de afstand tot het doel. OK als die 0 is. Stel > 0 voor zekere n. Kies de eerstvolgende n' op een kortste pad naar het doel; die heeft kleinere afstand tot het doel en dus (inductie) $h(n') \leq$ afstand tot het doel. Nu $h(n) \leq c(n, a, n') + h(n') \leq c(n, a, n') +$ afstand van n' tot het doel $=$ afstand van n tot het doel.

Consistent wordt wel "locaal optimistisch" genoemd, admissibel "globaal optimistisch".

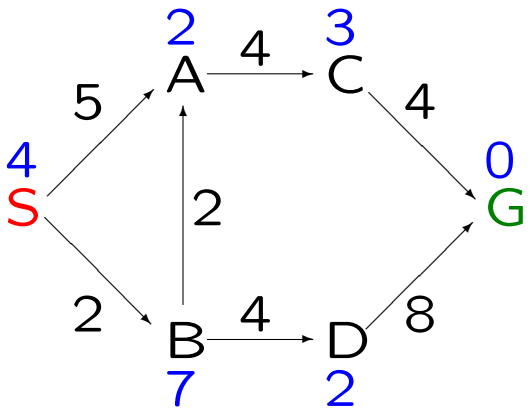
A* — Voorbeeld

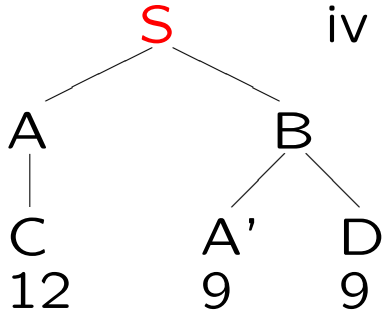


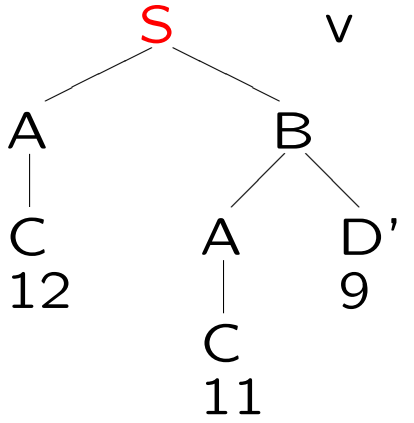
S is het Start-punt, G is de Goal (= doel).
Er staat steeds een ' bij de knoop die ontwikkeld wordt.

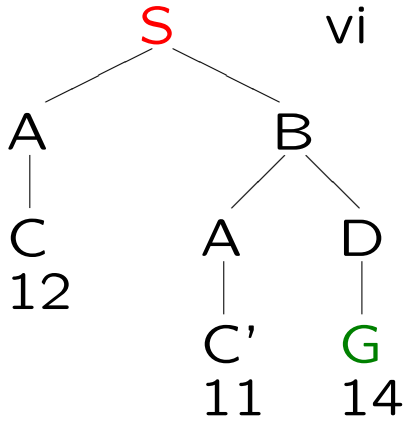
Merk op dat de heuristiek admissibel is — maar niet consistent, zie (!). Bij A' in iv krijg je eigenlijk $f = 4+2 = 6$, maar de ouder had 9. Dus, dankzij pathmax, 9. Idem D.

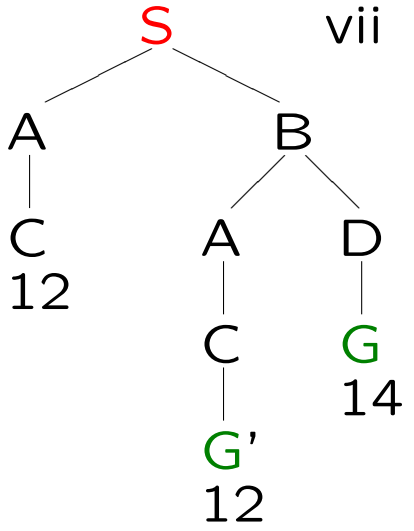
A* — Voorbeeld (vervolg)











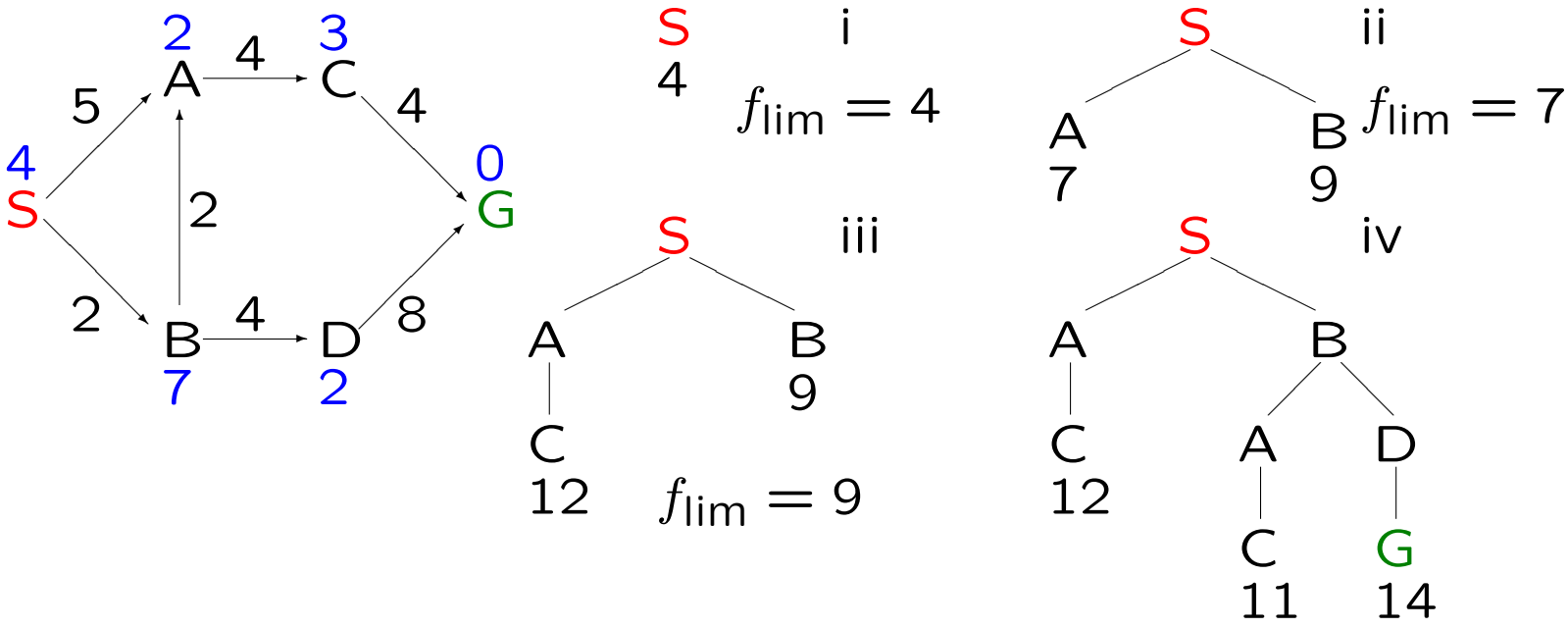
Bij iv mag je ook eerst D ontwikkelen,
en bij vii eventueel eerst C (G is een doel).
De kortste weg (S-B-A-C-G) is dus 12 lang.

De overgang van A* naar IDA* is analoog aan de overgang van DFS via Depth limited search naar Iterative deepening.

Bij IDA* is iedere stap een complete DFS-wandeling, waarbij je iedere knoop ontwikkelt die een f-waarde heeft hoogstens gelijk aan de laagste f-waarde f_{lim} die nog "open stond" uit de vorige stap. In de eerste doorgang is f_{lim} gelijk aan de f-waarde van de beginknoop, oftewel diens h-waarde.

Er zijn nog meer varianten: RBFS, MA*, SMA*, ... Deze laatste "vergeet" knopen met hoge f-waarden als het geheugen vol raakt.

IDA* — Voorbeeld



Na iedere stap (een DFS-wandeling) is de boom steeds weg!

Na stap iv wordt $f_{\text{lim}} = 11$, in de volgende stap 12 en wordt (deels) dezelfde boom als in stap vii van A* doorlopen, maar nu met ook het kind G,13 van de linker C erbij.

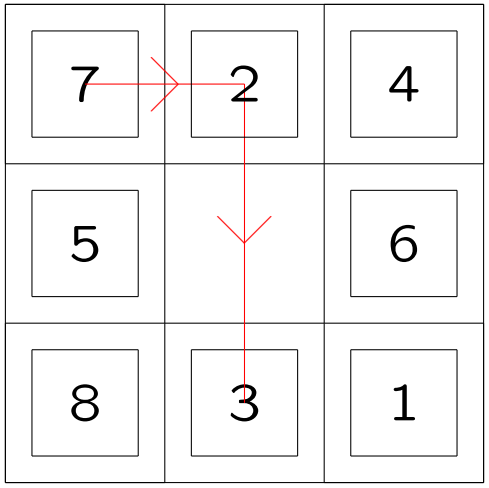
Kortom, wat is nu het *verschil* tussen A* en IDA*, Iterative Deepening A*?

Bij A* expandeer je steeds de (een) knoop met de laagste f-waarde uit de "frontier" die in het geheugen zit. De grootte van die "frontier" is ruwweg de "breedte" van de boom. Er is maar één boom in het spel.

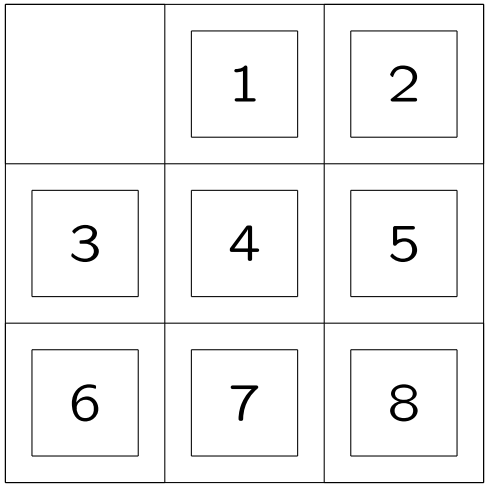
Bij IDA* maak je steeds een DFS-wandeling door een (bij iedere ronde groter wordende) boom; je loopt steeds tot en met de laagste open staande f-waarde uit de vorige ronde. Nu heb je altijd alleen een pad uit de boom in je geheugen, en dat heeft met "diepte" te maken.

Welke heuristiek? — 8-puzzel

Er zijn meestal verschillende heuristische functies mogelijk.



begin



doel

Voor h_1 nemen we het aantal "tegels" dat uit positie is; voor h_2 de som van de afstanden van de "tegels" tot hun uiteindelijke positie. Als afstand kiezen we de Manhattan-afstand (= city block distance). Hier: $h_1 = 8$ en $h_2 = 18$.

Een goede maat voor de kwaliteit van heuristieken is aan de hand van de effectieve vertakkingsgraad b^*. Stel dat A* N knopen genereert en een oplossing vindt op diepte d. Dan is b^* de vertakkingsgraad die een volledig gevulde boom met $N+1$ knopen van diepte d heeft. Dus:

$$N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d.$$

Als A* een oplossing vindt op diepte 5, met behulp van 52 knopen, geldt $b^* = 1{,}92$.

Hoe dichter b^* bij 1 zit, hoe beter (dan vind je "diepe" oplossingen).
Als het ware meet b^* hoe "breed" je zoekt.

Voor de 8-puzzel (gemiddeldes over 100 problemen; IDS = Iterative deepening search; d = diepte oplossing):

	aantal ge-exp. knopen			eff. vertakkingsgraad		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27

Als voor iedere knoop n geldt dat $h_2(n) \geq h_1(n)$, zeggen we: h_2 domineert h_1. A* met h_2 zal dan doorgaans minder knopen expanderen dan h_1 (zie ook Experiment), "want" elke knoop n met $h(n) < C^* - g(n)$ wordt ge-expandeerd.

Conclusie: kies altijd een heuristiek met zo hoog mogelijke waarden — mits admissibel (nooit overschatten).

Als $h_1,\ldots,h_m$ admissibele heuristieken zijn, domineert

$$h(n) = \max\{h_1(n),\ldots,h_m(n)\}$$

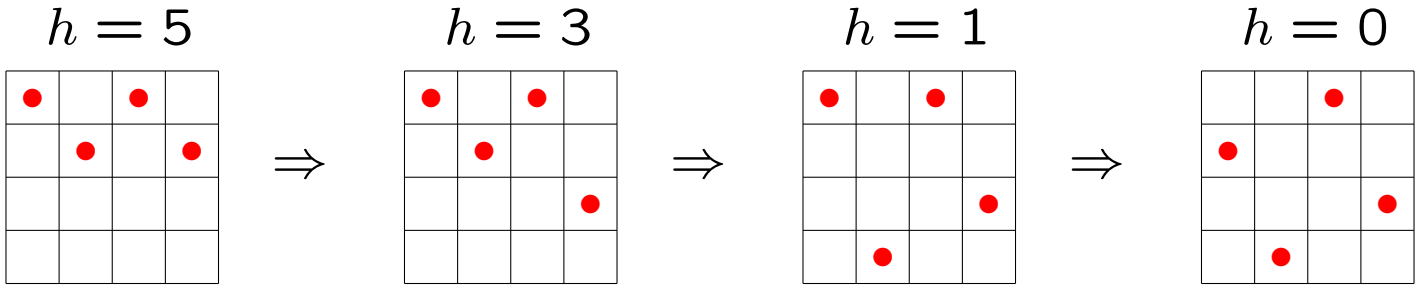
ze allemaal.

Een handige manier om een heuristiek te vinden is het exact oplossen van een vereenvoudigd (afgezwakt, relaxed) probleem. De optimale oplossing hiervan kost hooguit even veel als die van het oorspronkelijke probleem.
Voor de 8-puzzel: een "tegel" mag van A naar B bewegen mits A en B aangrenzend zijn *en* B leeg is. Dit zwakken we af tot:

- Een "tegel" mag van A naar B bewegen mits A en B aangrenzend zijn ($\Rightarrow h_2$)
- Een "tegel" mag van A naar B bewegen mits B leeg is
- Een "tegel" mag (altijd) van A naar B bewegen ($\Rightarrow h_1$)

Voor het dames-op-schaakbord probleem: een heuristiek h is het aantal ruziënde paren. Een actie is het verplaatsen van een dame in haar eigen kolom: op het n bij n bord $n(n-1)$ opvolgers.

Dit suggereert de volgende aanpak. Een hill-climber kiest random uit de beste opvolgers. Dit is een greedy = gretige strategie — met locale minima! Zie later: Complex zoeken.



Het A*-algoritme wordt onder andere gebruikt om NPC's ("non-player characters") te laten lopen in computerspellen, samen met Rodney Brooks' eindige automaten.

Er zijn nog allerlei geavanceerde zaken:

- pattern databases om exacte oplossingen van deelproblemen in op te slaan;
- (local) beam search houdt een k-tal beste oplossingen bij, die elkaar "stimuleren";
- en nog veel meer ... zie ook: Complex zoeken.

Het huiswerk voor de volgende keer (12 maart 2024): lees Hoofdstuk 5, p. 146–170 van [RN] door (in de derde druk p. 161–190) over Spellen door.

Denk tevens aan de tweede opgave: Agenten & Robotica; deadline: 22 maart 2024.

Bestudeer de opgaven van:

www.liacs.leidenuniv.nl/~kosterswa/AI/opgaven1.pdf

www.liacs.leidenuniv.nl/~kosterswa/AI/opgaven2.pdf

Deze worden ook gemaakt op de sommenwerkcolleges, in het bijzonder op 7 maart: opgaven 2, 3, 7, 8, 14.

Kunstmatige Intelligentie (AI)

Hoofdstuk 5 van Russell/Norvig = [RN]

Spel(l)en

voorjaar 2024

College 6 en 7, 13 en 20 maart 2024

www.liacs.leidenuniv.nl/~kosterswa/AI/spellen.pdf

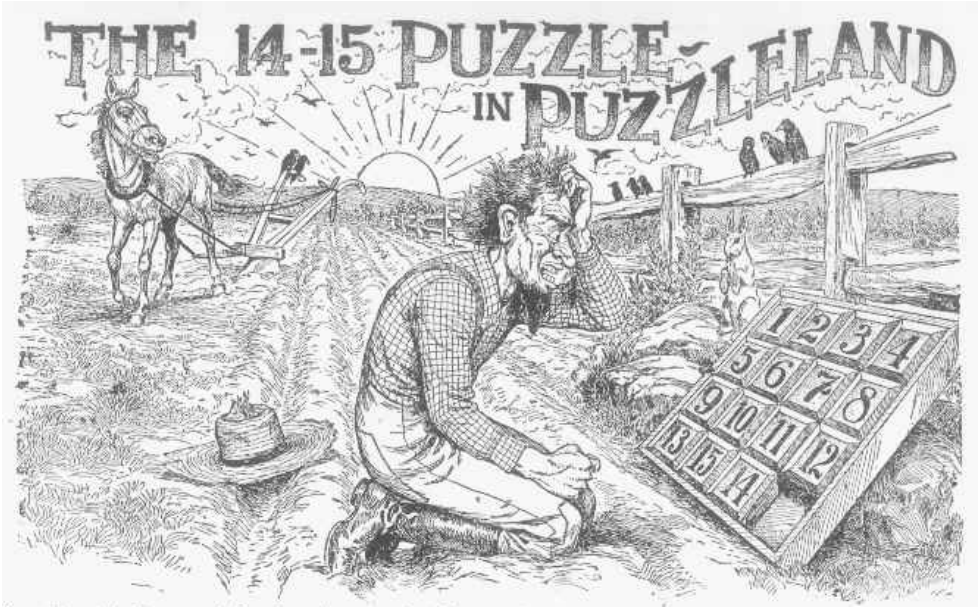
Spellen geven aanleiding tot zeer complexe zoekproblemen, bijvoorbeeld bij schaken, Go, vier-op-een-rij. Extra problemen zijn contingency (onzekerheid), kansen, . . .
We hadden evaluatie-functies nodig om niet-eindtoestanden te beoordelen, bijvoorbeeld bij schaken: pion 1, paard/loper 3, toren 5, koningin 9, veiligheid koning, . . .

We willen prunen: niet alles doorrekenen. We bekijken met name het minimax-algoritme van Von Neumann (1928) en het α-β-algoritme uit 1956/58.

Nu: deep learning, . . .

Leuk om te lezen: H.J. van den Herik, J.W.H.M. Uiterwijk en J. van Rijswijck, Games solved: Now and in the future, Artificial Intelligence 134 (2002) 277–311.

De derde programmeeropgave gaat over schuifpuzzels als de 15-puzzel die we met A* en IDA* in C++ willen programmeren.

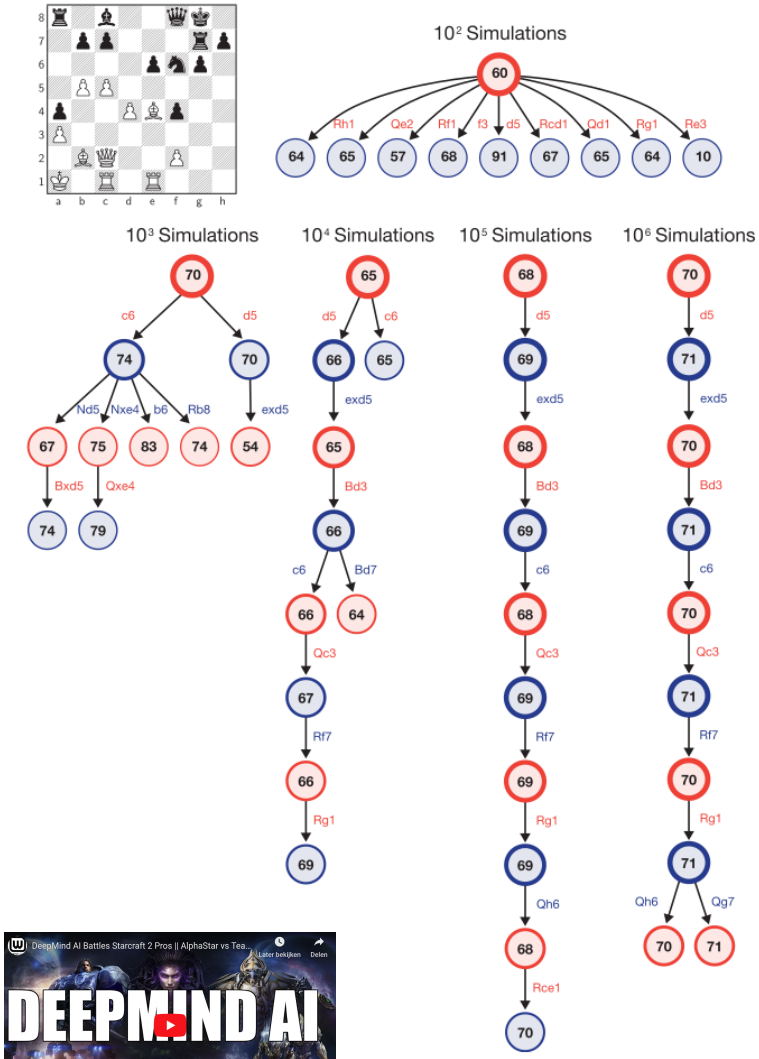


www.liacs.leidenuniv.nl/~kosterswa/aster2024.html

Vergeet de sommen niet, zoals opgave 8e over A* en IDA*.

Deep Blue vs. Garry Kasparov, 1997

Marion Tinsley (1927–1995) was de beste menselijke checkers-speler (dammen op een schaakbord) ooit.
In 2007 werd door Jonathan Schaeffer bewezen dat de beginspeler altijd minstens remise kan halen.



December 2018

AlphaZero

Silver et al.

Science 362, 1140–1144

RESEARCH

COMPUTER SCIENCE

A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play

David Silver[1,2*†], Thomas Hubert[1*], Julian Schrittwieser[1*], Ioannis Antonoglou[1], Matthew Lai[1], Arthur Guez[1], Marc Lanctot[1], Laurent Sifre[1], Dharshan Kumaran[1], Thore Graepel[1], Timothy Lillicrap[1], Karen Simonyan[1], Demis Hassabis[1†]

The game of chess is the longest-studied domain in the history of artificial intelligence. The strongest programs are based on a combination of sophisticated search techniques, domain-specific adaptations, and handcrafted evaluation functions that have been refined by human experts over several decades. By contrast, the AlphaGo Zero program recently achieved superhuman performance in the game of Go by reinforcement learning from self-play. In this paper, we generalize this approach into a single AlphaZero algorithm that can achieve superhuman performance in many challenging games. Starting from random play and given no domain knowledge except the game rules, AlphaZero convincingly defeated a world champion program in the games of chess and shogi (Japanese chess), as well as Go.

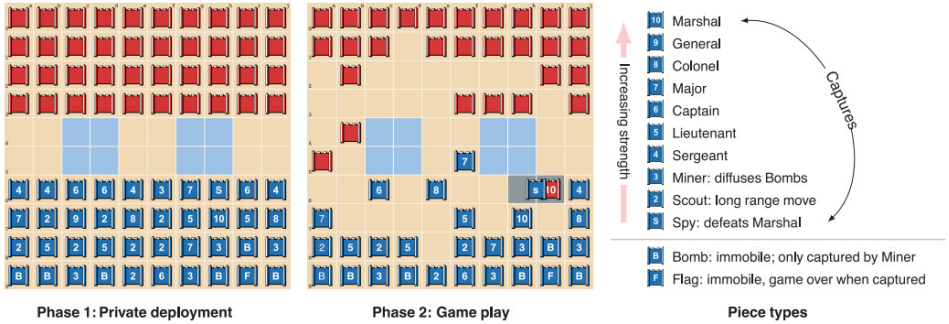
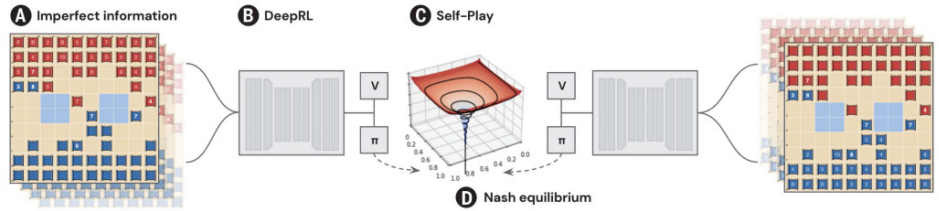


Fig. 1. Stratego is a two-player board game in which players try to capture the opponent's flag. Initially, the players secretly deploy 40 pieces of diverse strengths on the board. Then, they take turns moving pieces, possibly encountering an opponent piece that reveals both piece identities, and then the weaker piece is removed. Two lakes (indicated in blue) cannot be crossed by any piece. The complete rules are defined by the International Stratego Federation.



Replicator dynamics: $\frac{d}{d\tau}\pi^i_\tau(a^i) = \pi^i_\tau(a^i)\left[Q^i_{\pi_\tau}(a^i) - \sum_{b^i}\pi^i_\tau(b^i)Q^i_{\pi_\tau}(b^i)\right]$

Reward transformation: $r^i(\pi^i, \pi^{-i}, a^i, a^{-i}) = r^i(a^i, a^{-i}) - \eta\log\left(\frac{\pi^i(a^i)}{\pi^i_{\text{reg}}(a^i)}\right) + \eta\log\left(\frac{\pi^{-i}(a^{-i})}{\pi^{-i}_{\text{reg}}(a^{-i})}\right)$

breaking news

↓ link

RESEARCH

MACHINE LEARNING

Mastering the game of Stratego with model-free multiagent reinforcement learning

Julien Perolat*†, Bart De Vylder*†, Daniel Hennes, Eugene Tarassov, Florian Strub, Vincent de Boer‡, Paul Muller, Jerome T. Connor, Neil Burch, Thomas Anthony, Stephen McAleer, Romuald Elie, Sarah H. Cen, Zhe Wang, Audrunas Gruslys, Aleksandra Malysheva, Mina Khan, Sherjil Ozair, Finbarr Timbers, Toby Pohlen, Tom Eccles, Mark Rowland, Marc Lanctot, Jean-Baptiste Lespiau, Bilal Piot, Shayegan Omidshafiei, Edward Lockhart, Laurent Sifre, Nathalie Beauguerlange, Remi Munos, David Silver, Satinder Singh, Demis Hassabis, Karl Tuyls*†

We introduce DeepNash, an autonomous agent that plays the imperfect information game Stratego at a human expert level. Stratego is one of the few iconic board games that artificial intelligence (AI) has not yet mastered. It is a game characterized by a twin challenge: It requires long-term strategic thinking as in chess, but it also requires dealing with imperfect information as in poker. The technique underpinning DeepNash uses a game-theoretic, model-free deep reinforcement learning method, without search, that learns to master Stratego through self-play from scratch. DeepNash beat existing state-of-the-art AI methods in Stratego and achieved a year-to-date (2022) and all-time top-three ranking on the Gravon games platform, competing with human expert players.

Perolat et al., Science 378 (2022) 990–996: Mastering the game of Stratego . . .

Deep neural nets, reinforcement learning, selfplay.

Kunnen computers denken? Diplomacy!

Spellen kunnen als volgt worden ingedeeld:

	deterministisch	kans
perfecte informatie	schaken, dammen, checkers, Go, othello	monopoly, backgammon
onvolledige informatie	zeeslag, mastermind	bridge, poker, scrabble

En dan is er nog onderscheid in het aantal spelers.
Met name over schaken is veel gepubliceerd, waaronder allerlei anecdotes.
De ultieme uitdaging is/was het spel Go. En Diplomacy?

Er zijn verschillende strategieën om (een benadering van) de "speltheoretische waarde" van een spel (met perfecte informatie = volledig observeerbaar; nulsom = zero-sum: goed voor de één is even slecht voor de ander) te bepalen.

Shannon (1950) onderscheidt drie types:

type A reken alles tot en met zekere diepte door, en gebruik daar een evaluatie-functie

type B reken soms verder door (als het onrustig is: "quiescence"); gebruik heuristische functie om dit te sturen

type C doelgericht menselijk zoeken

A en B zijn $\approx$ "brute-force", C is meer "knowledge-based".

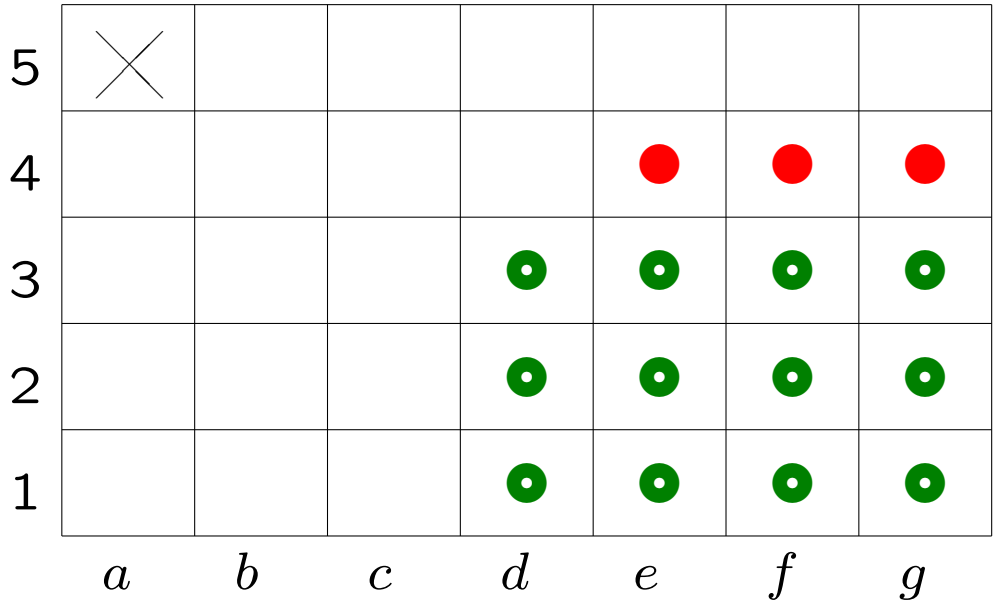
Er worden soms drie soorten oplossingen van spellen onderscheiden:

ultra-zwak de speltheoretische waarde van de beginstand is bekend: "je kunt vier-op-een-rij winnen"

zwak idem, en een optimale strategie is bekend (begin in middelste kolom, . . . , zie later)

sterk in elke legale positie is een optimale strategie bekend

Het spel Chomp wordt gespeeld met een rechthoekige reep chocola, waar de spelers om de beurt een stuk rechtsonder afhappen (een blokje en alles hier onder/rechts van). Wie het (vergiftigde) blokje linksboven eet, heeft verloren.



eerste hap: $d3$

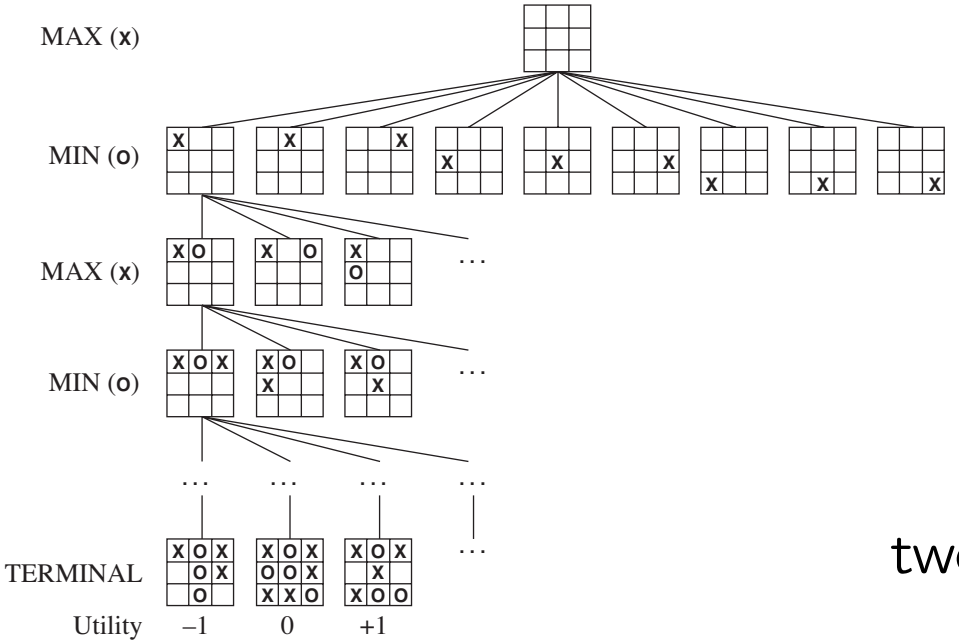
tweede hap: $e4$

Bewering: de beginnende speler bij Chomp kan altijd winnen! Immers, als je door het blokje rechtsonder te nemen kunt winnen is het goed. Als dat niet zo is, heeft de tegenstander blijkbaar een voor hem winnende "tegen-hap". Die kun je dan zelf als eerste doen, en dus daarmee winnen! Dit argument heet strategy stealing.

Kortom: het spel Chomp is ultra-zwak opgelost, de echte winnende zet weten we niet . . .

Voor bijvoorbeeld 2×2 en 2×3 Chomp "wint" het blokje rechtsonder (algemener voor vierkanten: neem het vakje rechts onder het vergiftigde, en dan "spiegelen"); bij 3×4 Chomp het blokje uit de middelste rij, derde kolom.

Voorbeeld spelboom



Boter, kaas en eieren

twee spelers, om en om:

MAX (X) en MIN (O); X begint

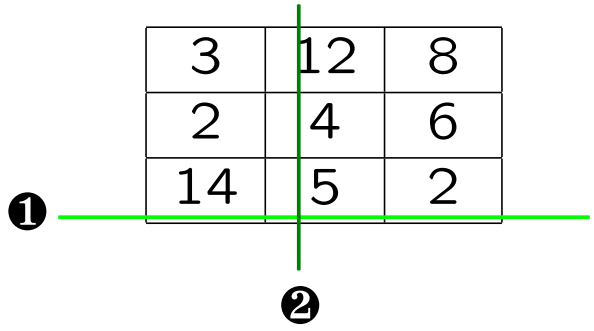
5478 toestanden

Boter, kaas en eieren is een voorbeeld van een tweepersoons deterministisch nulsom-spel met volledige informatie, waarbij de spelers om de beurt een "legale zet" doen. MAX moet een strategie vinden die tot een winnende eindtoestand leidt, ongeacht wat MIN doet. De strategie moet elke mogelijke zet van MIN correct beantwoorden.

Een utility-functie (= payoff-functie) geeft de waarde van eindtoestanden. Hier is dat: $-1/0/1$; bij backgammon is dat: $-192 \ldots +192$.

Uit symmetrie-overwegingen kunnen veel toestanden = posities worden wegbezuinigd. En een "actie" heet nu een "zet".

Maxi en Mini spelen het volgende eenvoudige spel: Maxi wijst eerst een (horizontale) rij aan, en daarna kiest Mini een (verticale) kolom:



Bijvoorbeeld: Maxi ❶ kiest rij 3, daarna kiest Mini ❷ kolom 2; dat levert einduitslag 5.

Maxi wil graag een zo groot mogelijk getal, Mini juist een zo klein mogelijk getal.

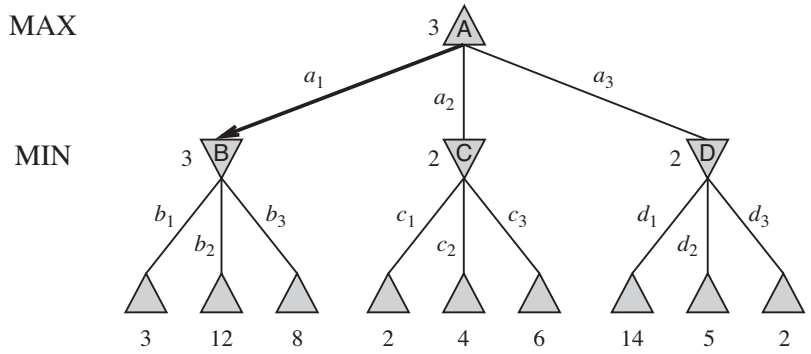
Hoe spelen we dit spel zo goed mogelijk?

Als Maxi rij 1 kiest, kiest Mini kolom 1 (levert 3); als Maxi rij 2 kiest, kiest Mini kolom 1 (levert 2); als Maxi rij 3 kiest, kiest Mini kolom 3 (levert 2). Dus kiest Maxi rij 1!

3	12	8
2	**?**	**?**
14	5	2

Nu merken we op dat de analyse (het minimax-algoritme) hetzelfde verloopt als we niet eens weten wat onder de twee vraagtekens zit. Het α-β-algoritme onthoudt als het ware de beste en slechtste mogelijkheden, en kijkt niet verder als dat toch nergens meer toe kan leiden (zie verderop).

In boomvorm:



Het minimax-algoritme is "recursief": neem in bladeren de evaluatie-functie, in MAX-knopen het maximum van de kinderen, in MIN-knopen het minimum van de kinderen. MAX- en MIN-knopen wisselen elkaar af.

Bovenstaande boom is één zet (= move) diep, oftewel twee ply.

function $MaxWaarde(toestand)$
    if $eindtoestand$ **then return** $Utility(toestand)$
    $waarde \leftarrow -\infty$
    for s **in** $Opvolgers(toestand)$ **do**
        $waarde \leftarrow \max(waarde, MinWaarde(s))$
    return $waarde$

function $MinWaarde(toestand)$
    if $eindtoestand$ **then return** $Utility(toestand)$
    $waarde \leftarrow +\infty$
    for s **in** $Opvolgers(toestand)$ **do**
        $waarde \leftarrow \min(waarde, MaxWaarde(s))$
    return $waarde$

Compleet: als de boom eindig is (bij schaken dankzij speciale regels)

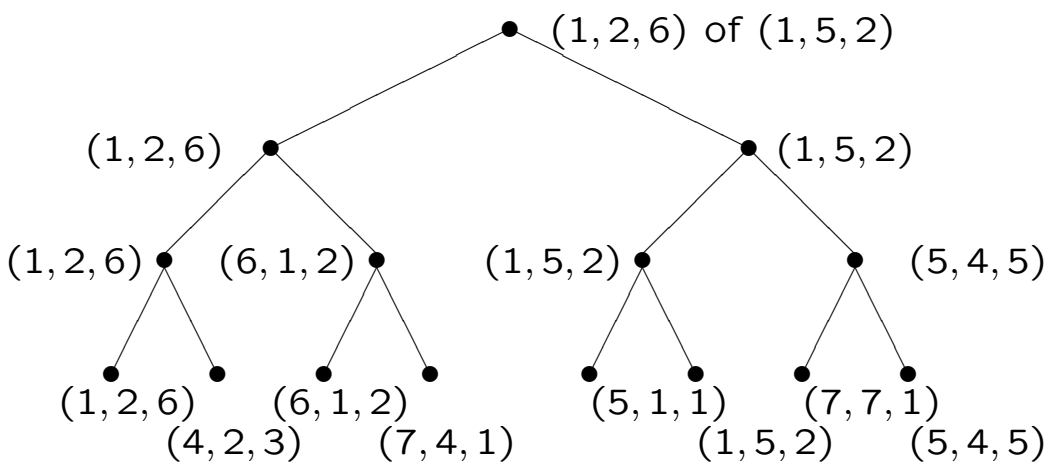
Optimaal: tegen een "optimale" tegenstander

Tijdscomplexiteit: $O(b^m)$ (b is vertakkingsgraad, m diepte van de boom)

Ruimtecomplexiteit: $O(bm)$ (bij depth-first exploratie)

Bij schaken: $b \approx 35$, $m \approx 100$; een exacte oplossing is dus heeeeeeeeeel ver weg.

Met drie spelers (A, B en C) heb je per knoop drie evaluatie-waardes (A wil de eerste zo hoog mogelijk, etcetera):



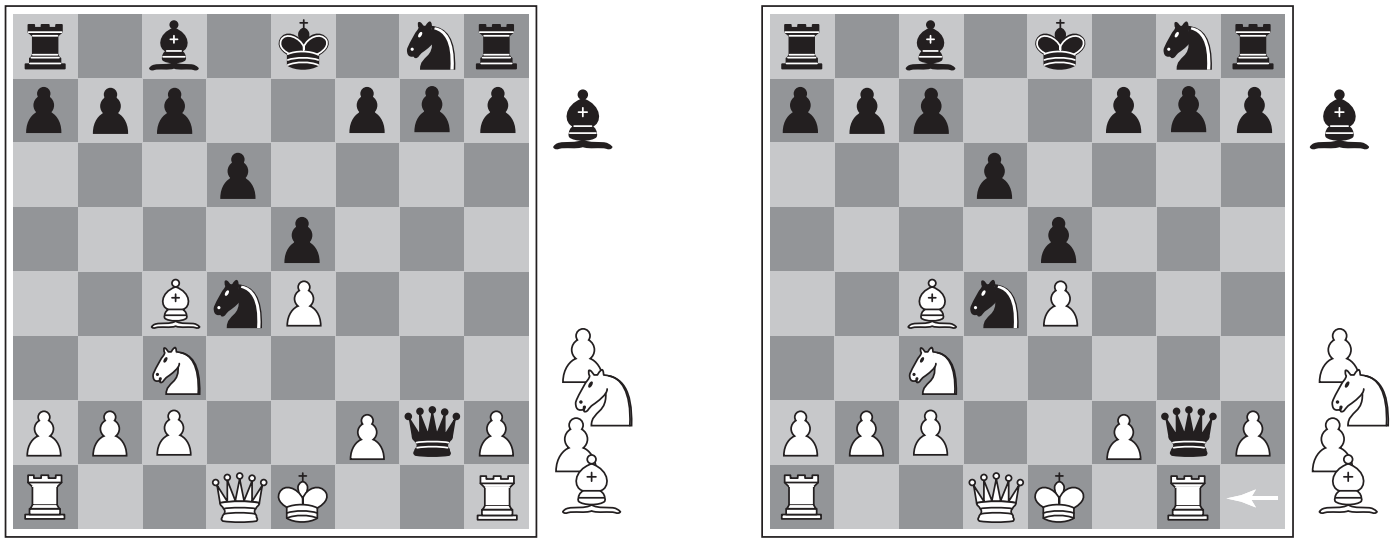
A aan zet

B aan zet

C aan zet

Voor twee spelers is één getal voldoende (bij een nulsomspel is de winst van de een het verlies van de ander).

En hoe zit het bij Diplomacy? En poker?



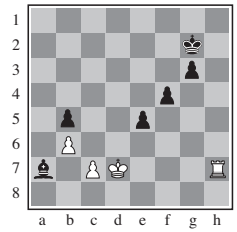
(a) White to move (b) White to move

Voor schaken is de evaluatie-functie meestal een gewogen som van allerlei kenmerken (“features”): 9 maal (aantal witte dames − aantal zwarte dames) + . . .

De functie moet eenvoudig te berekenen zijn, de kans op winnen aangeven, en kloppen met de utility-functie op eindtoestanden.

Bij cut-off search vervang je in het minimax-algoritme de test op eindtoestanden door een "cut-off test" — en een evaluatie-functie aldaar.

Bij schaken, met $b = 35$ en $b^m \approx 10^6$, krijg je $m = 4$. En 4-ply vooruit kijken is hopeloos ($\approx$ amateur). Men denkt: 8-ply $\approx$ PC of schaakmeester, en 12-ply is wereldtop-nivo (computer of mens).



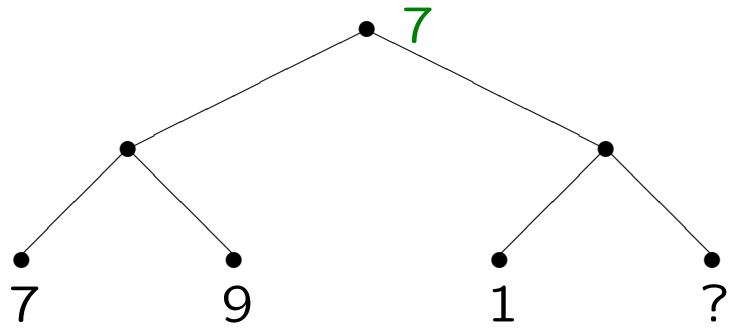
BTM

Je hebt te maken met:

– horizon-effect soms wordt een noodzakelijke slechte zet uit beeld geduwd door extra (minder slechte) zetten;

– quiescence in een onrustige periode moet je langer doorrekenen.

Het basisidee van het α-β-algoritme is het volgende: om de minimax-waarde in de wortel van onderstaande boom te berekenen is de waarde rechts onderin niet van belang — en die subboom kun je prunen (snoeien). Immers, het linker kind van de wortel is 7, en het rechterkind moet dus hoger dan 7 zijn wil het nog invloed uitoefenen. Nu is het (dankzij de 1) hoogstens 1, en zijn we klaar: waarde 7.

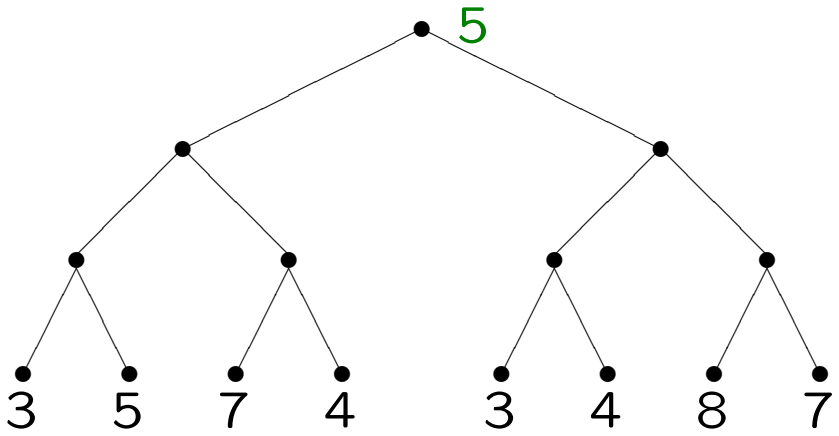


MAX aan zet

MIN aan zet

function $MaxWaarde(toestand, \alpha, \beta)$
  if $toestand$ is eindtoestand **then return** $Utility(toestand)$
    (of cut-off test, en gebruik evaluatie-functie)
  for s **in** $Opvolgers(toestand)$ **do**
    $\alpha \leftarrow \max(\alpha, MinWaarde(s, \alpha, \beta))$
    if $\alpha \geq \beta$ **then return** β
  return α

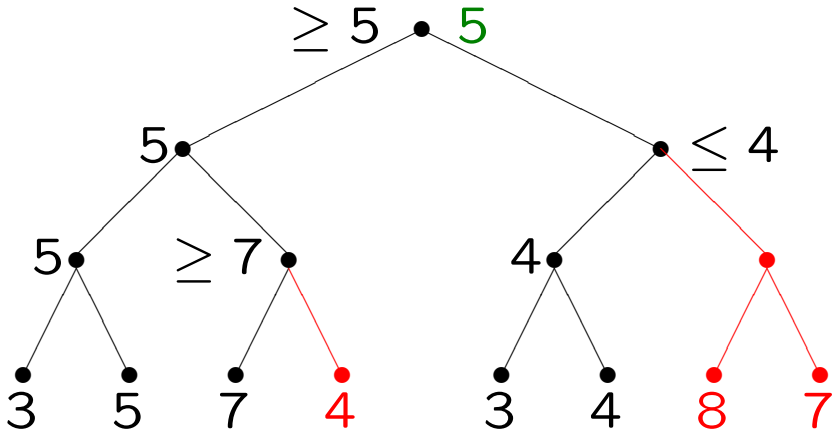
Analoog de functie $MinWaarde$, zie boek (voor een iets andere formulering). Normaal geldt bij aanroep $\alpha < \beta$; α geeft de beste (hoogste) waarde die MAX op het huidige pad kon bereiken, en β voor MIN. De variabelen α en β zijn locaal. Buitenste aanroep: $MaxWaarde(huidigetoestand, -\infty, +\infty)$.



MAX aan zet

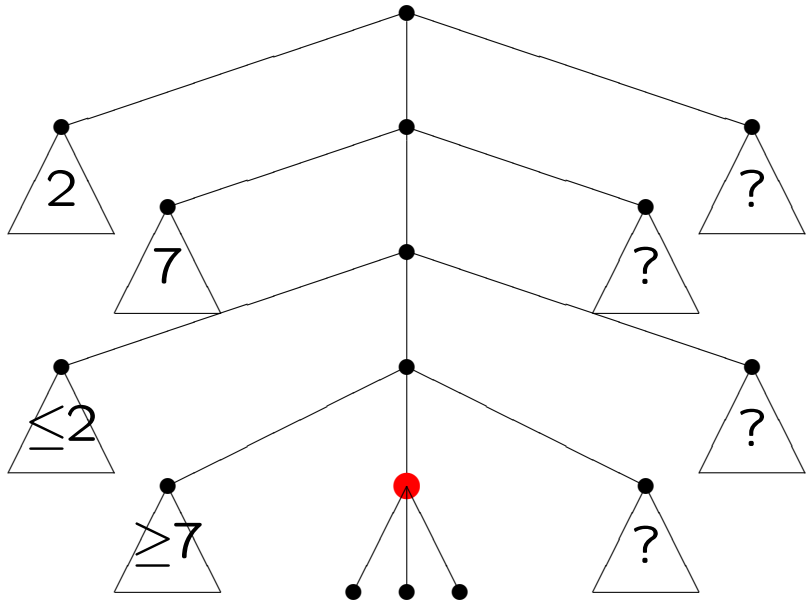
MIN aan zet

MAX aan zet



rood wordt

gepruned



MAX aan zet

MIN aan zet

MAX aan zet

MIN aan zet

MAX aan zet

Bij MAX-knoop ● geldt: $\alpha = 2$ en $\beta = 7$. Op het pad naar die knoop kan MAX al 2 afdwingen, en MIN 7. Als een kind van ● waarde 8 heeft, hoeven de volgende kinderen niet meer bekeken te worden, en krijgt ● waarde 7 (de β).

Er geldt: α-β-pruning levert exact hetzelfde eindresultaat in de wortel als "gewoon" minimax.

De effectiviteit hangt sterk af van de volgorde waarin de kinderen (zetten) bekeken worden.

Met "perfecte ordening" bereik je tijdscomplexiteit $O(b^{m/2})$ (m is de diepte van de boom), dus je kunt effectief de zoekdiepte verdubbelen.

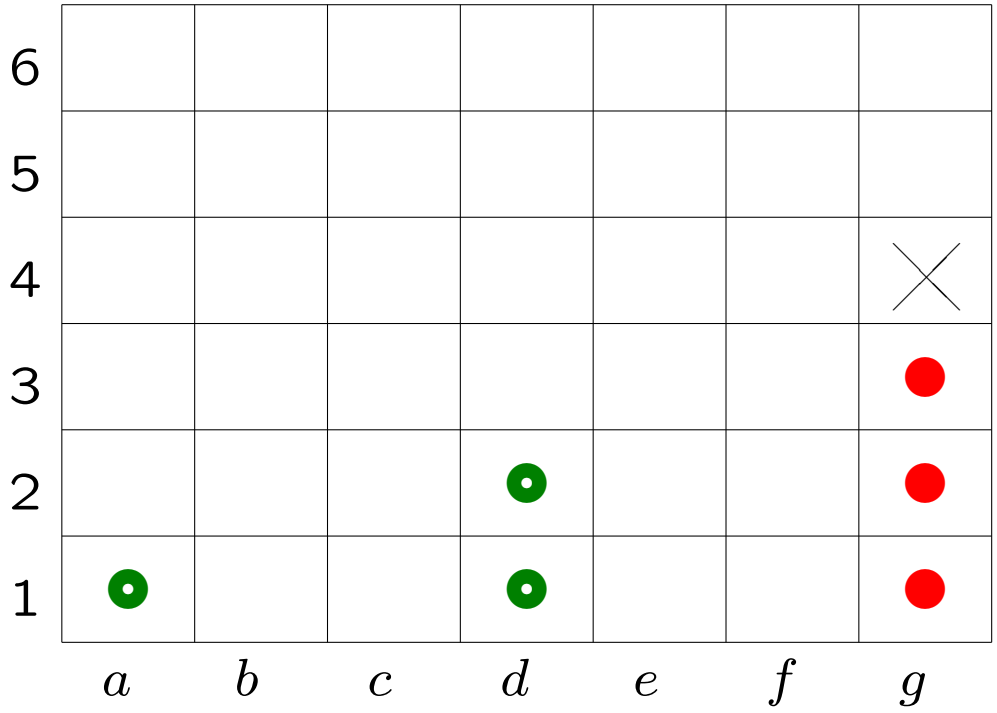
Het is dus van belang heuristieken te hebben om je zetten te ordenen. Enkele voorbeelden:

null-move bekijk eerst voor de tegenstander goede zetten (sla je eigen zet in gedachten even over)

killer als een zet ergens een snoeiing teweeg brengt, doet hij dat elders wellicht ook

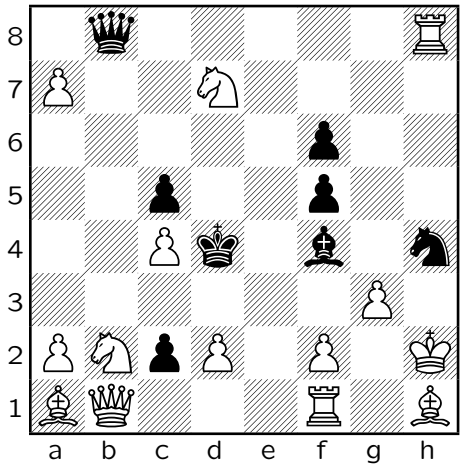
conspiracy-number $\approx$ aantal kinderen dat van waarde moet veranderen (samenzweren) om ouder van waarde te laten veranderen (voor stijgen MAX-knoop is maar één kind nodig, voor stijgen MIN-knoop zijn alle kinderen nodig)

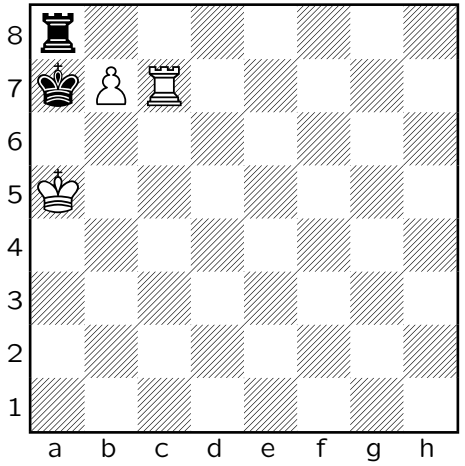
tabu-search onthoud aantal (zeer) slechte zetten

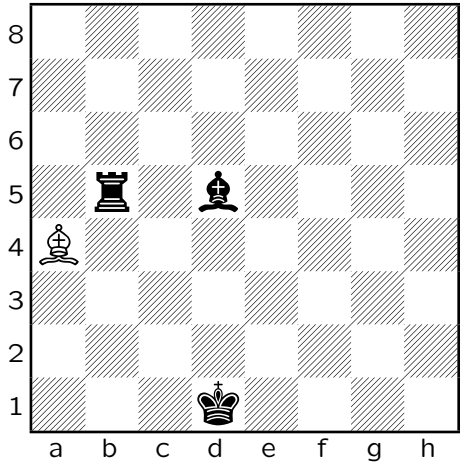


Vier-op-een-rij

Met groen aan de beurt, is $g4$ zowel voor de null-move- als de killer-heuristiek de aangewezen zet.
De voor groen winnende (begin)serie: $d1! - d2 - d3! - d4 - d5! - b1 - b2$ (een ! betekent: unieke winnende zet).





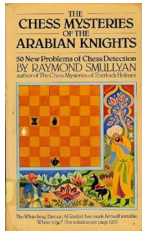


Babson task (R×h4)

Wit: mat in 1

Waar staat witte koning?

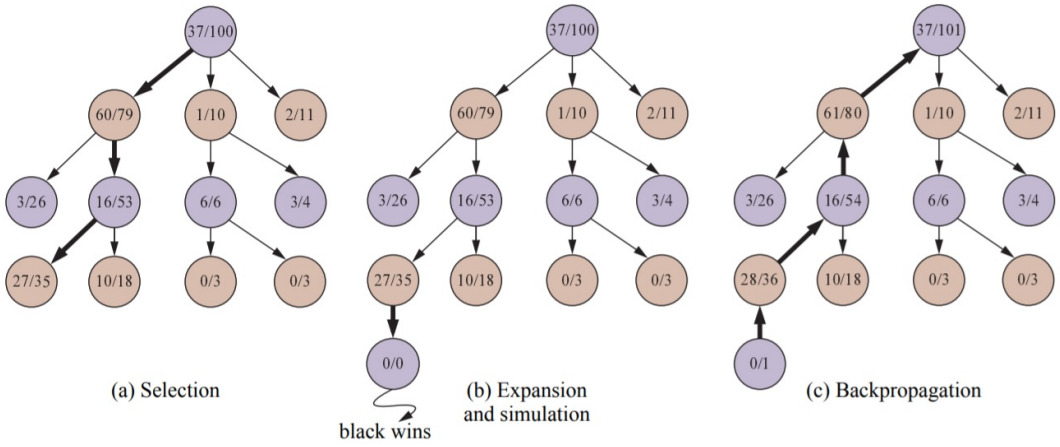
Leonid Yarosh,
Tim Krabbé



Raymond Smullyan
retrograde analyse

Bij pure Monte Carlo search bekijken we voor ieder mogelijke zet een playouts aantal random vervolgpartijen, en kiezen de (= een) zet met de hoogste (gemiddelde) uitkomst. Zie de eerste programmeeropgave.

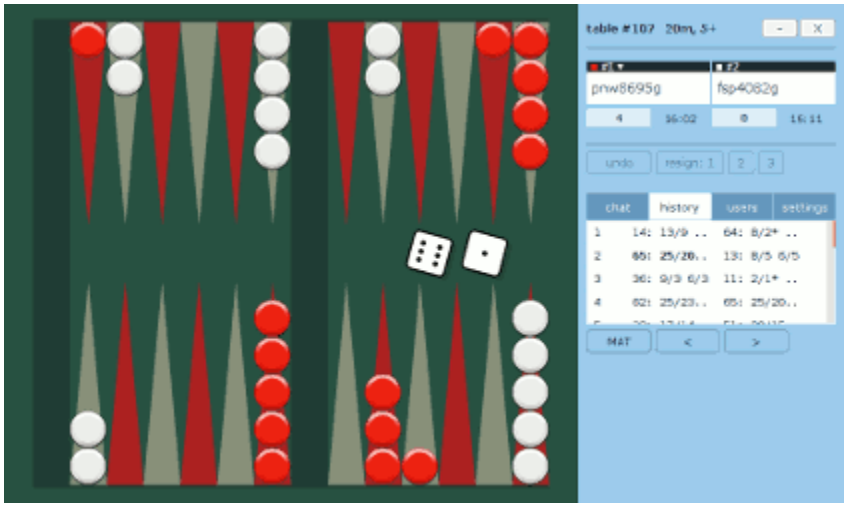
Bij Monte Carlo Tree Search (MCTS) kies je herhaald een kind (a) tot je een nieuwe knoop moet maken (b1), speel dan random uit (b2), en propageer de waardes terug naar de wortel (c). Je kiest hierbij steeds een kind met UCT-selectie (*): "upper confidence bounds applied to trees".



exploitatie exploratie

(*) $UCB1(n) = U(n)/N(n) + C \cdot \sqrt{\log(N(\text{parent}(n)))/N(n)}$ is de kans bij knoop n. Je speelt echt het meest bezochte kind n (hoogste $N(n)$); $U(n)$ is diens utility. Vaak $C = \sqrt{2}$.

Backgammon

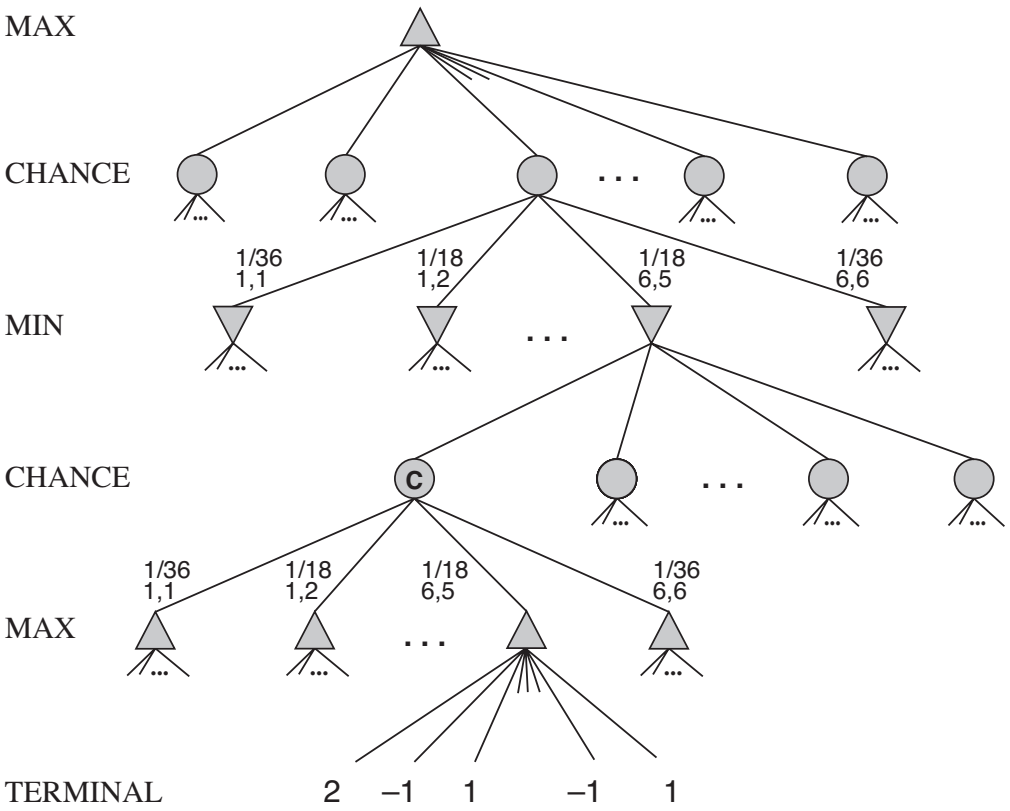


niet-deterministisch

gooi 2 dobbelstenen

en kies toegestane zet

Bij backgammon krijgen we een spelboom als:



MAX kiest

dobbel

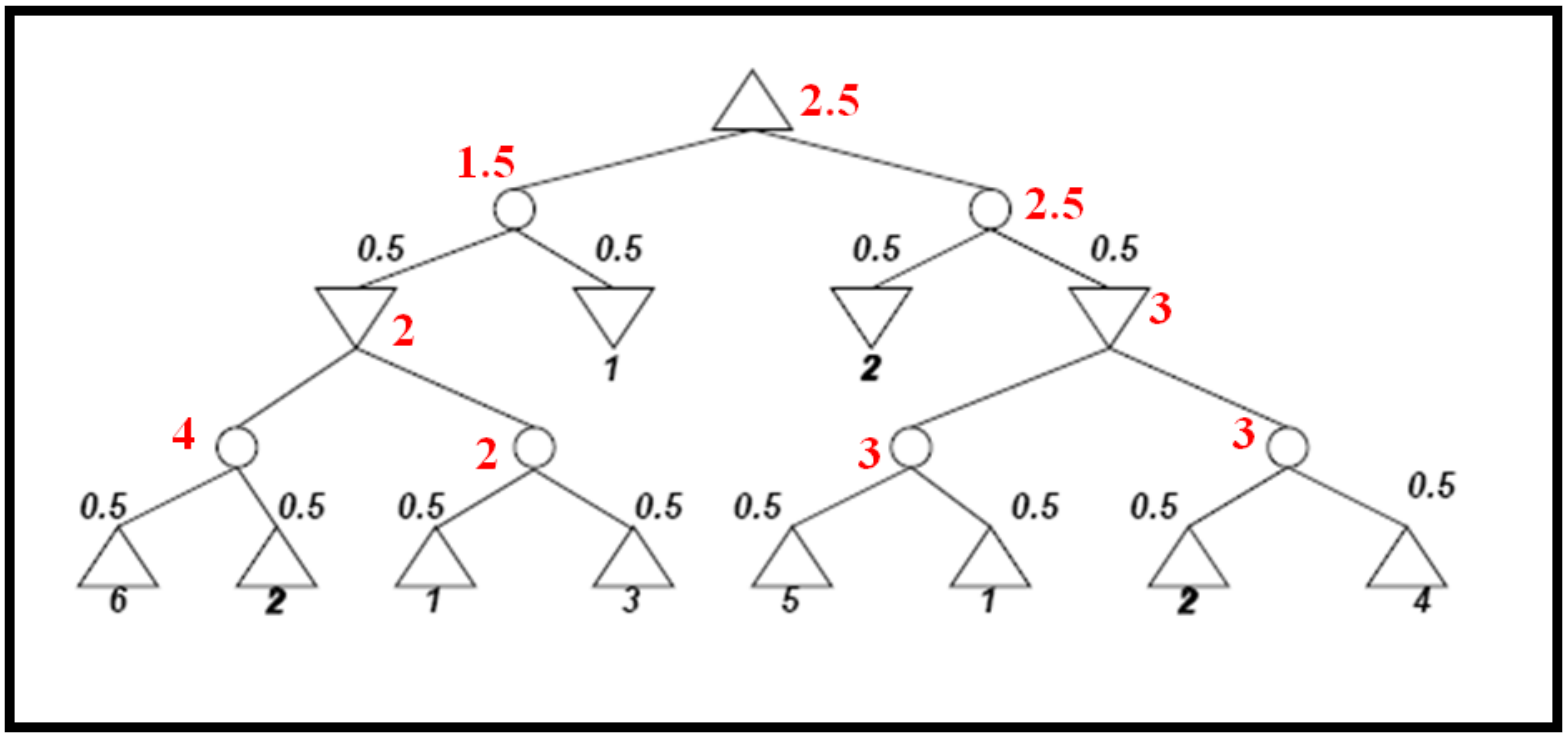
MIN kiest

dobbel

MAX kiest

eindtoestand

Een eenvoudig voorbeeld van een spel met een eerlijke munt:



Expecti-minimax — algemeen

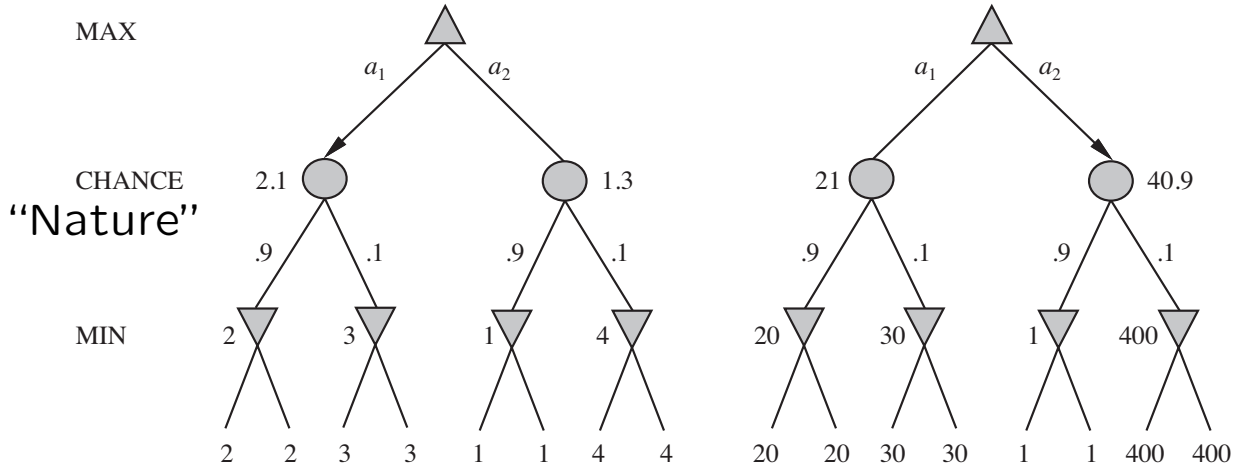
In het algemeen krijg je in een kansknoop n voor de expecti-minimax-waarde $ExpectiMinimax(n)$ een formule als:

$$\sum_{s \in Opvolgers(n)} P(s) \cdot ExpectiMinimax(s),$$

waarbij $P(s)$ de kans op s is.

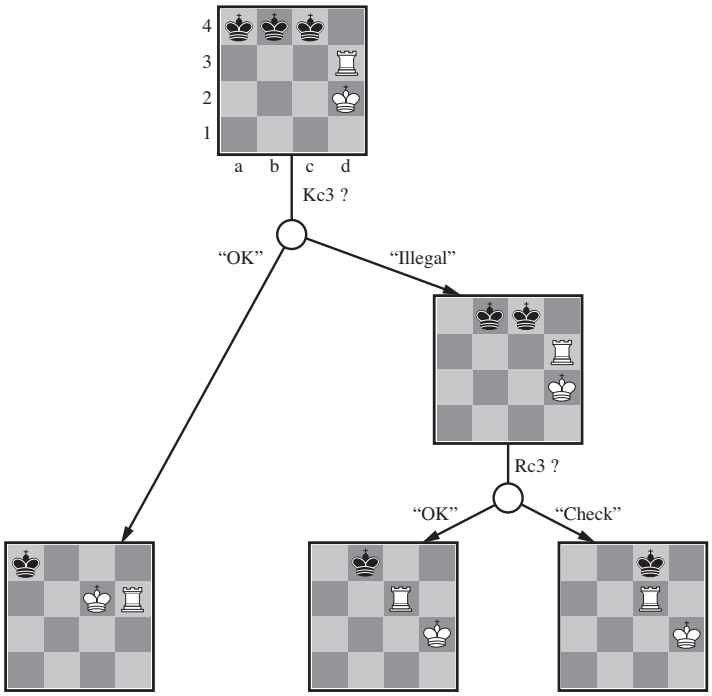
Soms is het gemiddelde "beter" dan de "exacte" minimax-waarde. Stel je voor dat je (= MAX) moet kiezen uit een MIN-knoop met kinderen 99, 1000, 1000 en 1000, en een MIN-knoop met kinderen 100, 101, 102 en 103 ...

In geval van kansknopen moet je beter op de evaluatie-functie letten!



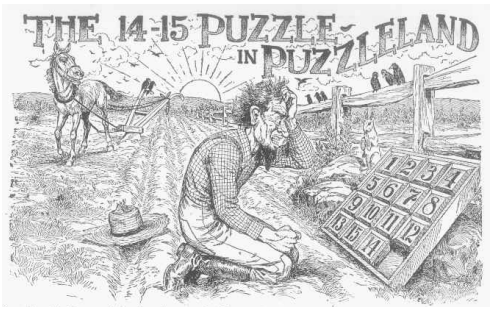
Links, met bladeren 1, 2, 3, 4, is de "linker" zet het beste, rechts, met bladeren 1, 20, 30, 400, de "rechter" zet.
De evaluatie-functie moet proportioneel zijn met de "winst".

Een deels observeerbaar spel is Kriegspiel: schaak waarbij de speler alleen de eigen stukken ziet, en een scheidsrechter zetten beoordeelt (“OK”, “Verboden”, “Schaak”, . . .).



Let ook op de “belief states”.

De derde programmeeropgave gaat over de 15-puzzel die we met A* en IDA* (verbeter!) in C++ programmeren. Mag je ook “schuin” schuiven? En dezelfde getallen?



www.liacs.leidenuniv.nl/~kosterswa/aster2024.html

```
./aster2024 drie.txt 25 50000 1 2 2 100 0 123
```

`drie.txt`: invoerfile, 25: max_lengte, 50000: geheugen, 1: oplosbaarheid, 2: heuristiek, 2: methode (3: IDA*), 100: aantal_spellen, 0: printen, 123: randomseed

De eerstvolgende keer zijn we nog met spellen bezig. Het huiswerk voor de daaropvolgende keer (3 april 2024): lees Hoofdstuk 6, p. 180–199 van [RN] door (in de derde druk p. 202-223) over het onderwerp Constrained Satisfaction Problemen.

Denk aan de tweede opgave: Agenten & Robotica; deadline: 22 maart 2024.
Werk daarna aan de derde opgave: A*.

En voor de liefhebbers: Combinatorial Game Theory.

Vergeet de "opgaven" = "sommen" niet, zie
www.liacs.leidenuniv.nl/~kosterswa/AI/opgaven1.pdf

Kunstmatige Intelligentie (AI)

Hoofdstuk 6 van Russell/Norvig = [RN]

Constrained Satisfaction Problemen (CSP's)

voorjaar 2024

College 8, 3 april 2024

www.liacs.leidenuniv.nl/~kosterswa/AI/csps.pdf

Bij Constraint Satisfaction Problemen (CSP's) gaat het om problemen waarbij de mogelijke toestanden worden gedefinieerd door toekenningen aan variabelen X_i (met waarden uit een domein D_i; $1 \leq i \leq n$); de doeltest is een verzameling constraints die aangeven welke combinaties van waarden voor deelverzamelingen van de variabelen zijn toegestaan. Dit zijn dus speciale zoekproblemen.

Standaard voorbeeld: kleur een landkaart met een beperkt aantal kleuren, zodat aangrenzende landen verschillende kleuren hebben.

Voorbeeld: Australië

Variabelen:

$X_1 = WA$, $X_2 = NT$,
$X_3 = Q$, $X_4 = NSW$,
$X_5 = V$, $X_6 = SA$
and $X_7 = T$.

Domeinen:

$D_i = \{rood, groen, blauw\}$
voor $i = 1, 2, \ldots, 7$.

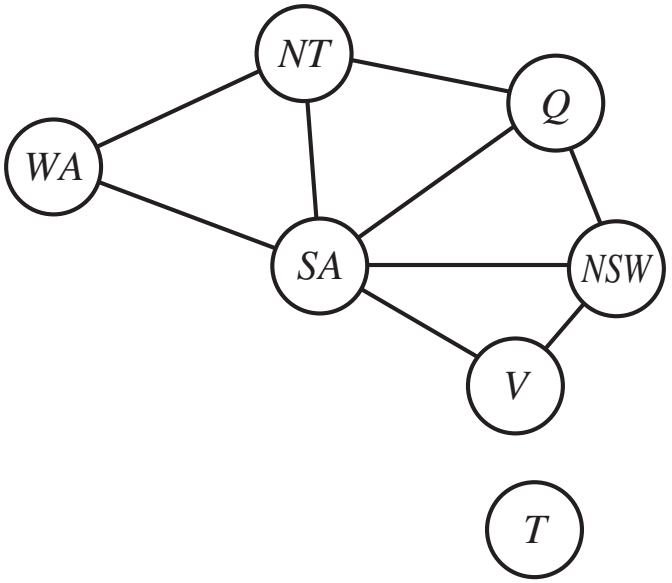
Constraints:

aangrenzende gebieden moeten
verschillende kleuren hebben,
oftewel: $WA \neq NT$, ... (als de "taal" dat toelaat)
oftewel: $(WA, NT) \in \{(rood, groen), (groen, blauw), \ldots\}, \ldots$

Constraint graaf

In een binair CSP heeft elke constraint betrekking op maximaal twee variabelen.

De constraint graaf heeft de variabelen als knopen, en de takken laten de constraints zien.



Tasmanië vormt overigens
een onafhankelijk deelprobleem.

Discrete variabelen:

Eindige domeinen, bijvoorbeeld Booleaanse CSP's, waaronder het NP-volledige SAT (Satisfiability); als alle domeinen $\leq d$ elementen hebben, zijn er $O(d^n)$ volledige toekenningen.

Oneindige domeinen, bijvoorbeeld job-scheduling; noodzaak voor speciale "constraints-taal": $StartJob_1 + 5 \leq StartJob_3$; lineaire constraints: oplosbaar, niet-lineair: onbeslisbaar.

Continue variabelen:

Bijvoorbeeld tijden voor waarnemingen met de Hubble-telescoop; lineaire constraints oplosbaar in polynomiale tijd met Lineair Programmeren.

Soorten constraints

unaire constraints hebben betrekking op één variabele, bijvoorbeeld $SA \neq groen$; je kunt ze wegwerken door de domeinen aan te passen

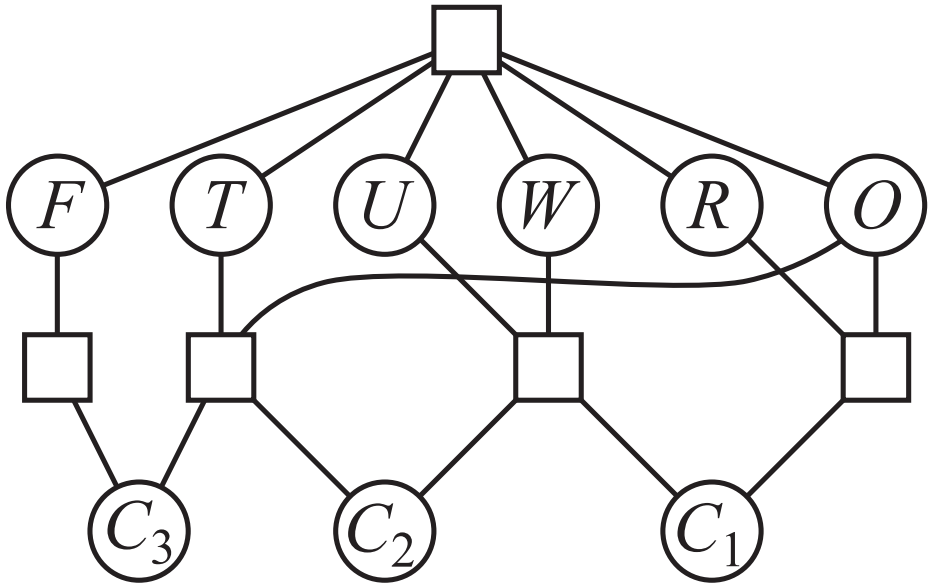
binaire constraints hebben betrekking op paren variabelen, bijvoorbeeld $SA \neq WA$

hogere orde constraints hebben betrekking op 3 of meer variabelen, bijvoorbeeld "rekenpuzzels" (zie straks)

voorkeuren — soft constraints, bijvoorbeeld $groen$ is beter dan $rood$ $\longrightarrow$ "constrained optimization problems"

Voorbeeld — Rekenpuzzel

$$\begin{array}{r} T\ W\ O \\ +\ T\ W\ O \\ \hline F\ O\ U\ R \end{array}$$



Variabelen: $F, T, U, W, R, O, C_1, C_2, C_3$ hypergraaf

Domeinen: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Constraints: $F \neq T$, $F \neq U$, ... $(Alldiff\,(F, T, U, W, R, O))$

$O + O = R + 10 \cdot C_1$, $W + W + C_1 = U + 10 \cdot C_2$,

$T + T + C_2 = O + 10 \cdot C_3$, $C_3 = F$

Voorbeeld — Echte problemen

- Toekenningsproblemen: wie geeft welke les?
- Roosterproblemen: welke les wordt wanneer en waar gegeven?
- Configuratie van hardware
- Spreadsheets
- Logistieke problemen

NB Vaak zijn de variabelen *reëelwaardig*.

De meest voor de hand liggende ("incrementele") aanpak is de volgende. Toestanden worden gedefinieerd door de tot dan toe toegekende waarden. Begintoestand: de "lege" toekenning $\emptyset$. Doeltest: de huidige toekenning is compleet (alle n variabelen OK). Opvolger-functie: geef waarde aan "vrije" variabele, zó dat er geen conflicten optreden.

Elke oplossing zit op diepte n; we kunnen dus DFS gebruiken. Het pad is irrelevant. Helaas: de vertakkingsgraad b op nivo ℓ is $(n-\ell)d$ (d is de (maximale) domeingrootte), en we krijgen een boom met $n! \cdot d^n$ bladeren ... terwijl er slechts d^n complete toekenningen zijn. Dit komt door commutativiteit: of je eerst aan X_1 toekent of eerst aan X_2 maakt niet uit!

We kunnen ons (dus) beperken tot toekenningen aan één variabele per knoop. In de wortel heb je dan d (de grootte van het domein) mogelijkheden in plaats van nd — als je tenminste één van de n variabelen hebt weten te kiezen. Dus vertakkingsgraad $b = d$ en er zijn (zoals verwacht) d^n bladeren.

Backtracking is depth-first search voor CSP's, met toekenningen aan enkele variabelen.

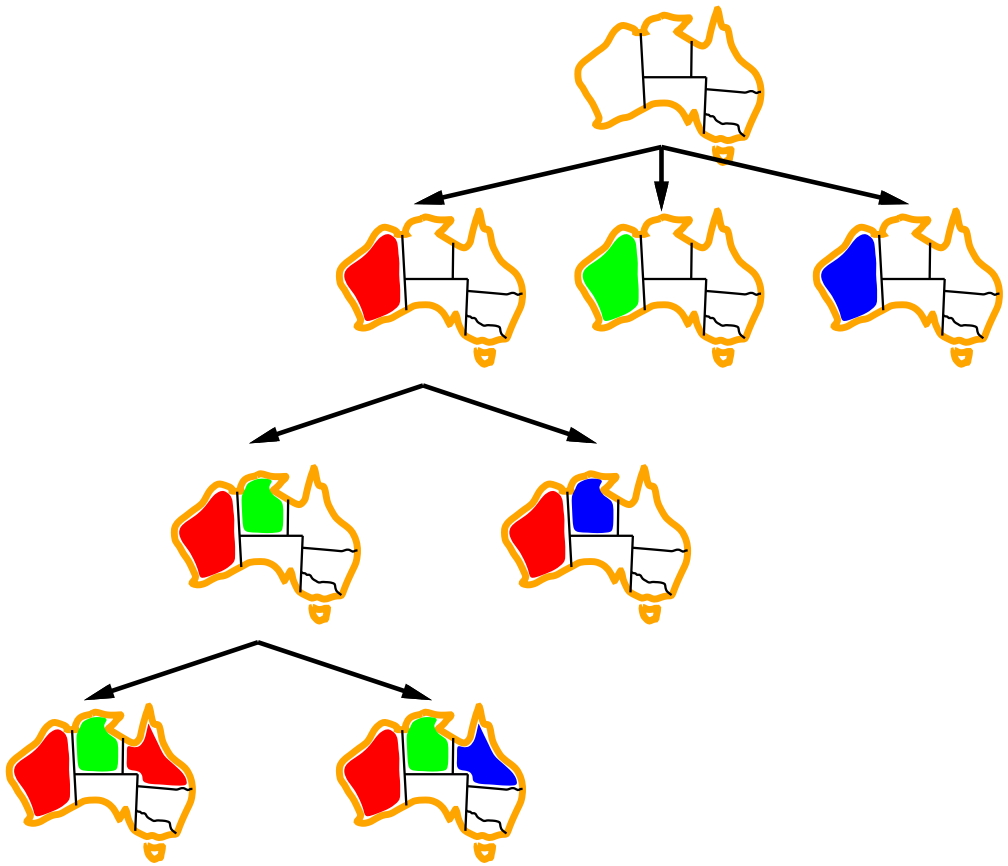
Dit is *het* basisalgoritme voor CSP's. Er zijn allerlei verbeteringen mogelijk, zoals we zullen zien.

Backtracking levert de volgende recursieve functie op:

function RecBack ($reeds, csp$)
  if $reeds$ is compleet **then return** $reeds$
  $var \leftarrow$ KiesVrijeVar ($vars, reeds, csp$)
  for each $waarde$ **in** Waardes (var, csp)
    if $waarde$ is consistent met $reeds$
        volgens Constraints[csp] **then**
      $res \leftarrow$ RecBack ($\{var = waarde\} \cup reeds, csp$)
      if $res \neq failure$ **then return** res
  return $failure$

Hierbij is $reeds$ de verzameling van de reeds toegekende variabelen en hun waarden ($\{WA = rood, T = blauw\}$).

Backtracking — Voorbeeld

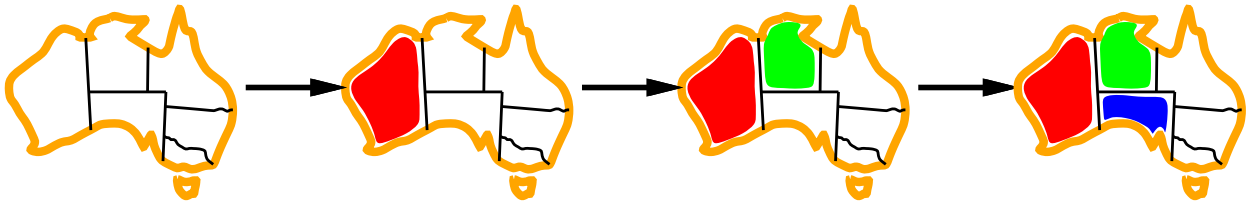


Vragen

Drie hoofdvragen zijn:

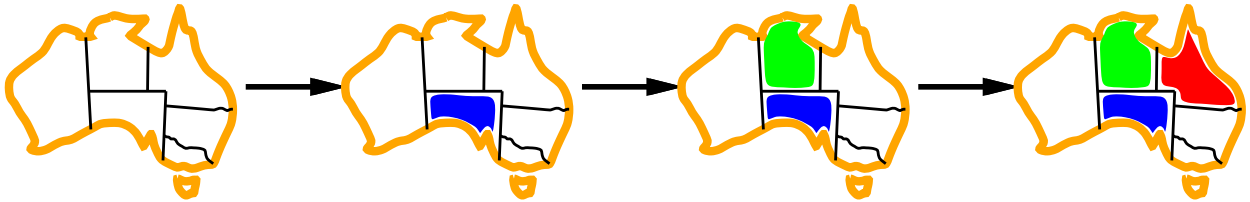
- Welke variabele moet ik eerst doen, en welke waarde kan ik hem geven?
- Wat zijn de implicaties van de huidige toekenningen voor de nog niet toegekende variabelen?
- Als een pad faalt, hoe kun je dan dit zelfde probleem in de toekomst vermijden?

De Minimum Remaining Values (MRV) heuristiek (= Most Constrained Variable) kiest de variabele met de minste toegestane waarden, en kleurt in de derde stap *SA* (en wel blauw), want *SA* heeft nog slechts één mogelijke kleur:



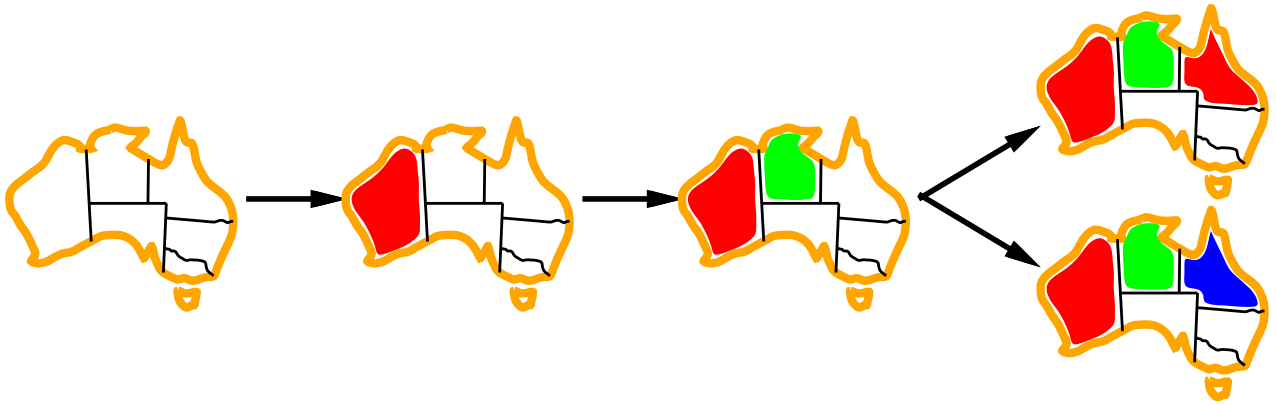
"Welke variabele moet ik kiezen?"

De Most Constraining Variable (MCV) heuristiek (= degree-heuristiek) kiest de variabele met de meeste constraints op de overblijvende variabelen, en kleurt in het begin *SA*:



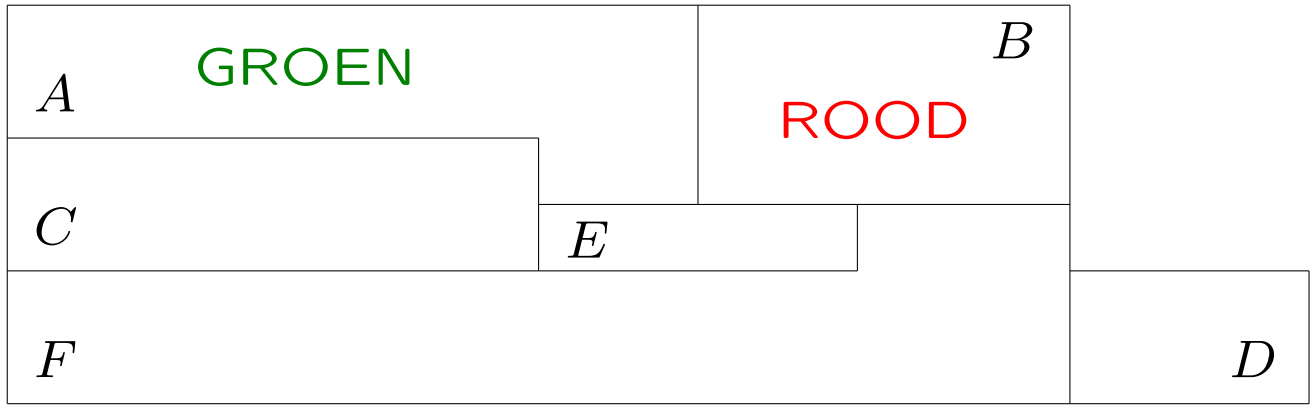
"Welke variabele moet ik kiezen?"

De Least Constraining Value (LCV) heuristiek kiest, gegeven een variabele, de waarde die de meeste waarden voor de overblijvende variabelen mogelijk laat, en kleurt in de derde stap Q (als je die variabele dus al gekozen hebt) rood zodat SA nog één mogelijkheid heeft (bij blauw geen!):



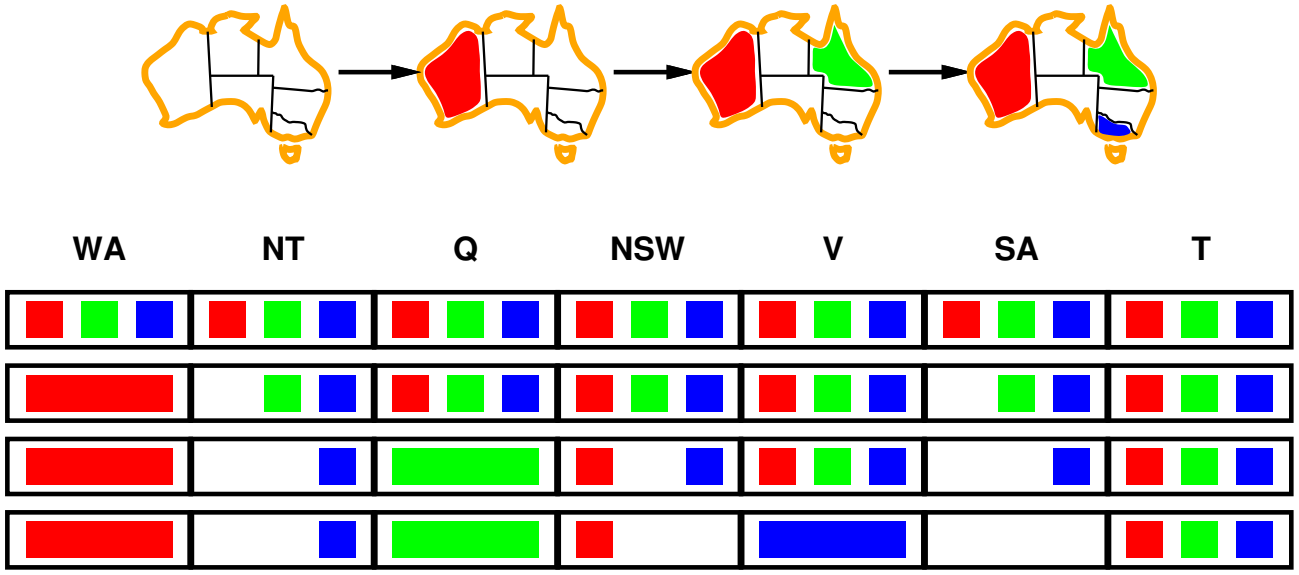
"Welke waarde moet ik kiezen?"

De drie voorgaande heuristieken samengevat in een voorbeeld, weer met drie kleuren:



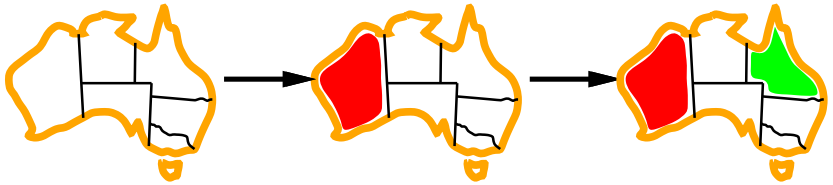
MRV (Minimum Remaining Values): kleur E nu (en wel blauw)
MCV (Most Constraining Variable): kleur *nu* F (of wellicht, als je anders telt, E); kleur *als eerste* E of F (veel buren)
LCV (Least Constraining Value): *als* je nu C wilt kleuren, dan met rood

Bij forward checking houd je de nog toegestane waarden bij voor de nog niet toegekende ("vrije") variabelen. Je kunt stoppen zodra er een variabele is zonder toegestane waarden.



Dit gaat goed samen met de MRV.

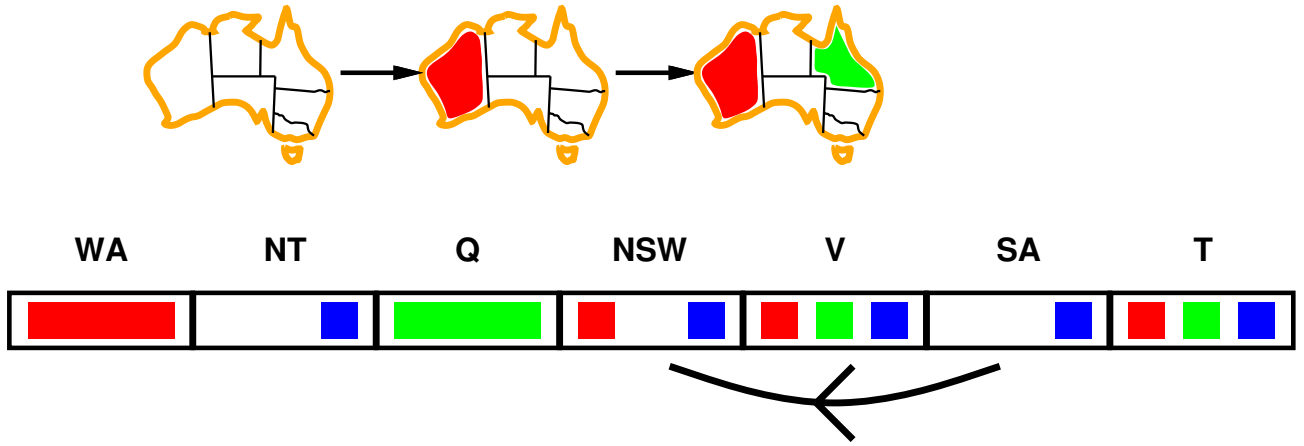
Forward checking ontdekt niet alle inconsistenties:



NT en *SA* kunnen niet beide blauw zijn!

Constraint propagatie probeert herhaald locaal constraints te forceren.

De eenvoudigste propagatie-vorm maakt pijlen ("arcs") consistent: $X \rightarrow Y$ heet consistent als voor elke waarde van X er nog minstens één toegestane waarde van Y is.



Deze pijl is consistent: als *SA* blauw is, kan *NSW* nog rood zijn. Andersom niet: als *NSW* blauw is, kan *SA* niet netjes gekleurd worden. En tussen *NT* en *SA* is geen (consistente) pijl: klaar!

Arc consistency — Algoritme

Er bestaat een arc consistency algoritme (AC-3) dat meer doet dan Forward checking, zie boek. Het wordt herhaald toegepast. Per gecheckte pijl worden andere pijlen, die door het eventueel verwijderen van foute waarden inconsistent dreigen te worden, ook weer gecheckt.

Complexiteit: er zijn $O(n^2)$ pijlen, elk wordt hooguit d keer op de agenda gezet, en de check op consistentie kost $O(d^2)$, bij elkaar $O(n^2 d^3)$.
Je kunt niet "alles" in polynomiale tijd detecteren (want 3-SAT is NP-volledig — zie het college Complexiteit!).

Er zijn allerlei gespecialiseerde CSP-solvers, en zelfs "Constraint Programming". En "SAT-solvers", zie verderop.

Voorbeeld — Sudoku

Een Sudoku is een CSP. We hebben in het 9×9-geval 81 variabelen $\{X_{11}, X_{12}, \ldots, X_{19}, X_{21}, \ldots, X_{99}\}$, alle met domein $D = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Voor de reeds gevulde vakjes bestaat het domein uit het gegeven getal. De constraints eisen dat alle rijen, kolommen en de negen 3×3-blokken precies alle getallen uit D bevatten: 27 keer $Alldiff$.

De MRV-heuristiek kiest een vakje waar het minste aantal waarden nog mogelijk is.
De MCV-heuristiek kiest een vakje dat met zoveel mogelijk andere nog "open" vakjes interfereert.
De LCV-heuristiek kiest een waarde (gegeven een vakje) die het meeste overlaat voor de andere vakjes.

Vergelijk: Japanse puzzels (Nonogrammen), Minesweeper.

Sudoku oplossen

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

Met arc consistency kun je afleiden dat E6 een 4 moet zijn. En I6 moet een 7 zijn.

Om een SAT-solver te kunnen gebruiken, moet het probleem eerst als Satisfiability (SAT) probleem worden geformuleerd, zie het college Complexiteit.

Gegeven zijn Booleaanse variabelen x_1, x_2, x_3, x_4 (of meer), en verder "clausules" als $C_1 = x_2 \vee \overline{x_3} \vee x_4$ en $C_2 = \overline{x_1} \vee x_3$. (Hierbij heten x_3 en $\overline{x_3} = \neg x_3$ "literals".) De vraag is of $C_1 \wedge C_2$ waargemaakt kan worden, hier bijvoorbeeld door $x_1 = x_3 =$ `false` en $x_2 = x_4 =$ `true`. Dit heet wel CNF: Conjunctive Normal Form.

Het beslissingsprobleem SAT is "NP-volledig", dus lastig!

Een SAT-solver probeert op efficiënte wijze een SAT-probleem op te lossen, zie MiniSAT, Lingeling, . . .

Aan de basis ligt doorgaans het DPLL-algoritme (Davis-Putnam-Logemann-Loveland) uit 1962:

- "unit propagation" behandelt clausules met één literal
- "pure literal elimination" handelt literals af die alleen "true of false" voorkomen
- "splitting rule" probeert een variabele true/false te maken, met behulp van backtracking

Vele verbeteringen: Conflict-Driven Clause Learning, . . .

Sudoku, met $s_{xyz} = \texttt{true} \Leftrightarrow$ plek (x, y) bevat getal z:

- There is at least one number in each entry:
  $\bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigvee_{z=1}^{9} s_{xyz}$
- Each number appears at most once in each row:
  $\bigwedge_{y=1}^{9} \bigwedge_{z=1}^{9} \bigwedge_{x=1}^{8} \bigwedge_{i=x+1}^{9} (\neg s_{xyz} \vee \neg s_{iyz})$
- Each number appears at most once in each column:
  $\bigwedge_{x=1}^{9} \bigwedge_{z=1}^{9} \bigwedge_{y=1}^{8} \bigwedge_{i=y+1}^{9} (\neg s_{xyz} \vee \neg s_{xiz})$
- Each number appears at most once in each 3x3 sub-grid:
  $\bigwedge_{z=1}^{9} \bigwedge_{i=0}^{2} \bigwedge_{j=0}^{2} \bigwedge_{x=1}^{3} \bigwedge_{y=1}^{3} \bigwedge_{k=y+1}^{3} (\neg s_{(3i+x)(3j+y)z} \vee \neg s_{(3i+x)(3j+k)z})$
  $\bigwedge_{z=1}^{9} \bigwedge_{i=0}^{2} \bigwedge_{j=0}^{2} \bigwedge_{x=1}^{3} \bigwedge_{y=1}^{3} \bigwedge_{k=x+1}^{3} \bigwedge_{l=1}^{3} (\neg s_{(3i+x)(3j+y)z} \vee \neg s_{(3i+k)(3j+l)z})$.

In the extended encoding, the resulting CNF formula will have 11,988 clauses, apart from the unit clauses representing the pre-assigned entries. From these clauses, 324 clauses are nine-ary and the remaining 11,664 clauses are binary. The nine-ary clauses result from the four sets of at-least-

- There is at most one number in each entry:
  $\bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigwedge_{z=1}^{8} \bigwedge_{i=z+1}^{9} (\neg s_{xyz} \vee \neg s_{xyi})$
- Each number appears at least once in each row:
  $\bigwedge_{y=1}^{9} \bigwedge_{z=1}^{9} \bigvee_{x=1}^{9} s_{xyz}$
- Each number appears at least once in each column:
  $\bigwedge_{x=1}^{9} \bigwedge_{z=1}^{9} \bigvee_{y=1}^{9} s_{xyz}$
- Each number appears at least once in each 3×3 sub-grid:
  $\bigvee_{z=1}^{9} \bigwedge_{i=0}^{2} \bigwedge_{j=0}^{2} \bigwedge_{x=1}^{3} \bigwedge_{y=1}^{3} s_{(3i+x)(3j+y)z}$.

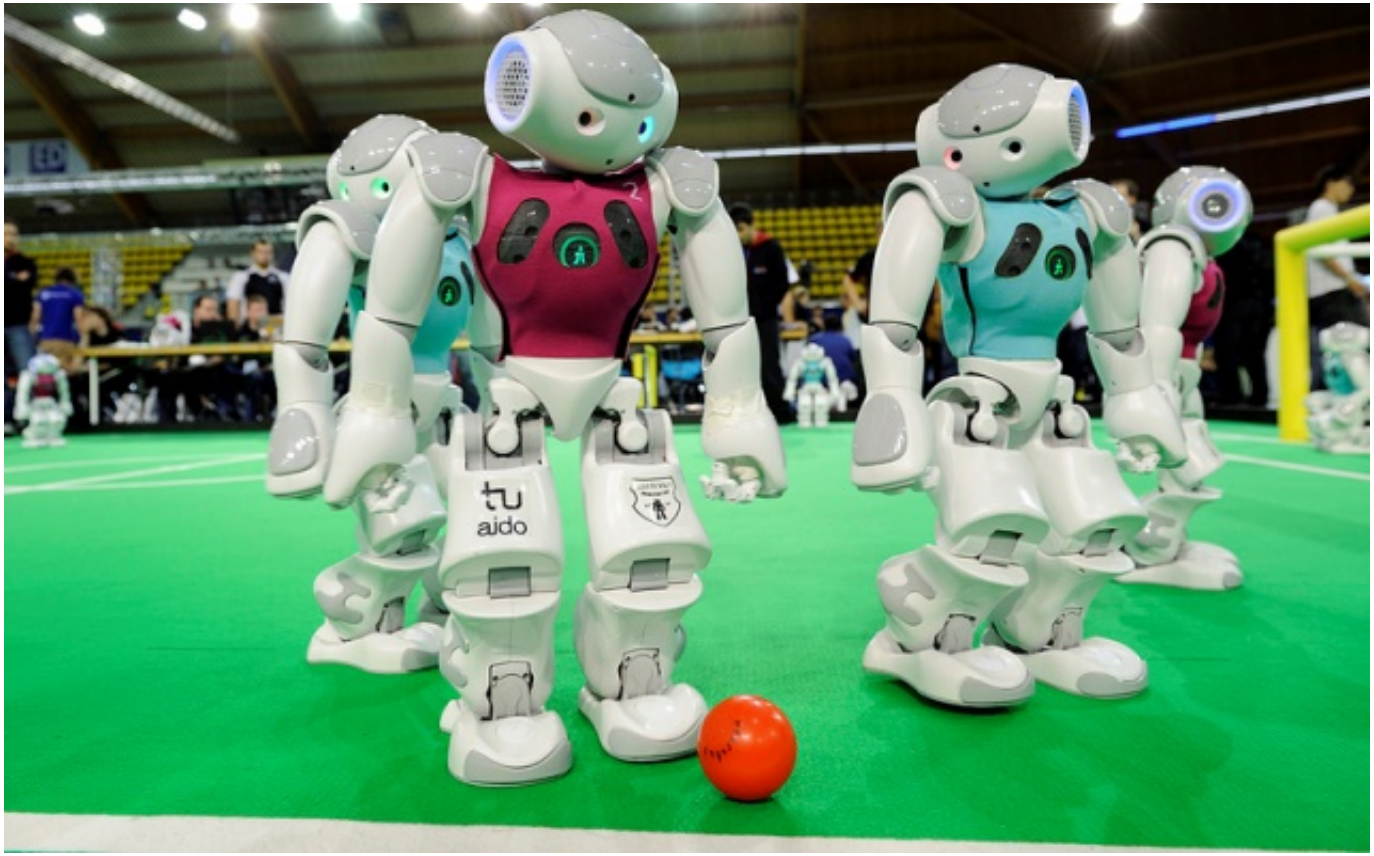
Uit: I. Lynce, J. Ouaknine, Sudoku as a SAT problem, ISAIM 2006.

Technieken als hill-climbing en Simulated Annealing werken met "complete" toestanden: alle variabelen hebben een waarde.

Voor CSP's moet je toestanden met "geschonden" constraints toestaan, en operatoren maken die variabelen van waarde wijzigen.

Je kunt random een "foute" variabele kiezen, en (met de min-conflicts heuristiek) die waarde kiezen die de minste constraints schendt.
Voorbeeld: dames op schaakbord, zie elders.

Ook mogelijk: Genetische algoritmen, zie later.



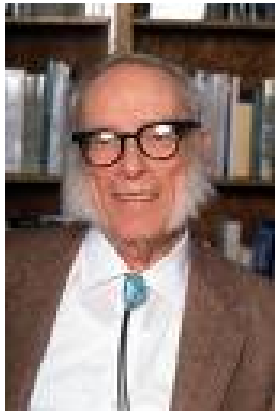
www.robocup.org

☰ = geen tentamenstof

Een robot is een "actieve, kunstmatige agent wiens omgeving de fysieke wereld" is. Het woord stamt uit 1921 (of eerder), en is gemaakt door de Tsjechische broers Čapek. En softbots: RoboCom, internet programma's.

Van de science fiction schrijver Isaac Asimov (auteur van "I, Robot") zijn de drie (later vier) wetten van de robotica:

1. Een robot mag een mens geen kwaad doen.
2. Een robot moet menselijke orders gehoorzamen (tenzij dat tegen 1. ingaat).



3. Een robot moet zichzelf beschermen (tenzij dat tegen 1. of 2. ingaat).

Een leuke robotsimulatie, van Michael Genesereth en Nils Nilsson, is de volgende.

Bedenk een taak, bijvoorbeeld een toren maken van een paar gekleurde blokken. Stel je nu een "robot" voor die uit *vier mensen* bestaat:

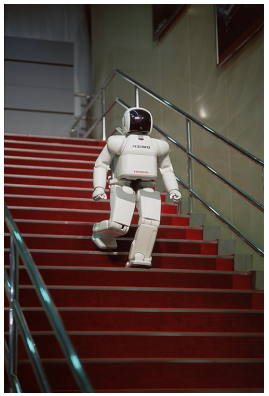
Brein krijgt input van Ogen, maar kan zelf niet zien; geeft opdrachten aan Handen; maakt plannen

Ogen weet niet wat het doel is

Handen, Links en Rechts voeren simpele opdrachten uit; ze zijn geblinddoekt

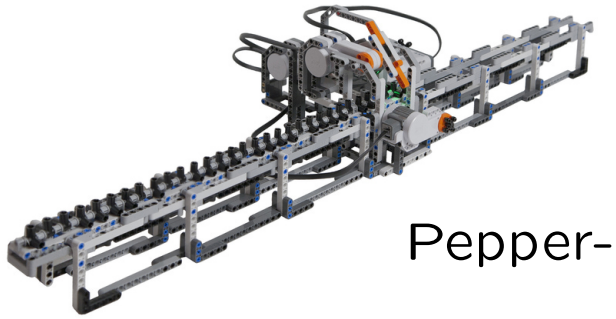
Curiosity

Sony Aibo



Hiroshi Ishiguro's robot (en hijzelf) "uncanny valley"

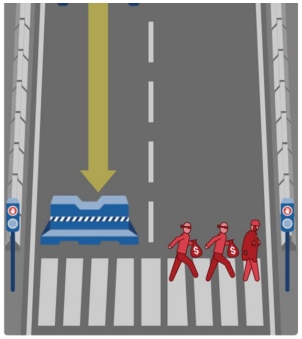
Honda ASIMO

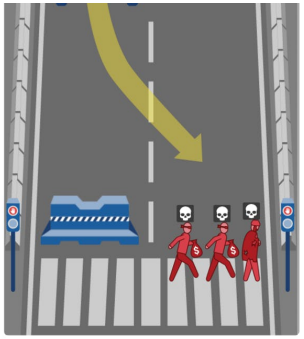


Lego Turing-machine

Pepper-robot van Softbank (2016)

Momenteel doen zelfrijdende systemen het erg goed.





↑computer-zien, kennis, de wet, ethiek(†)↑, . . .

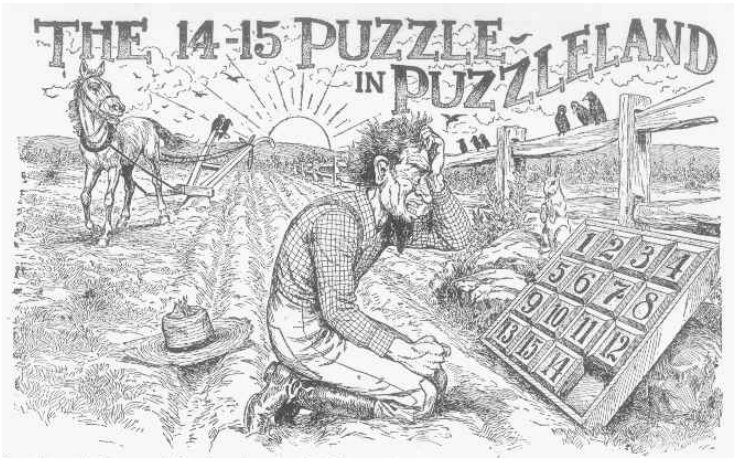
Udacity en Coursera ⟶ MOOC's over zelfrijdende auto's
(†) zie Michael Sandel's video's op YouTube

De subsumption architectuur is een idee van Rodney Brooks van MIT uit 1986, zie zijn artikel (via de AI-website).

Een robot bestaat hier uit verschillende *finite state machines* met klokken. Deze worden netjes gesynchroniseerd en gecombineerd. Zo vermijd je problemen met gigantische configuratieruimtes. Ze worden ook gebruikt om NPC's ("non-player characters") in computerspellen te modelleren.

Het werkt goed voor één kleine taak, maar wat er gebeurt is soms lastig te snappen. Zie je hiervan iets terug bij RoboCom?

De derde programmeeropgave gaat over puzzels en A*:



www.liacs.leidenuniv.nl/~kosterswa/AI/aster2024.html

Het huiswerk voor de volgende keer (10 april 2024): lees Hoofdstuk 19.1/4, p. 651–672 en p. 677–679 van [RN] door (in de derde druk p. 693–758 en p. 768–776) over het onderwerp Leren.

Maak (ook tijdens het werkcollege van donderdag 4 april 2024) in het bijzonder de sommen 11, 16 en 17 van

www.liacs.leidenuniv.nl/~kosterswa/AI/opgaven1.pdf

Er zijn video's met uitwerkingen.

Voor Robotica (☰ = geen tentamenstof) lees Hoofdstuk 26 van [RN] (in de derde druk Hoofdstuk 25).

Denk tevens aan de derde opgave: A*; deadline woensdag 17 april 2024.

Kunstmatige Intelligentie (AI)

Hoofdstuk 19.1/4 van Russell/Norvig = [RN]

Leren

voorjaar 2024

College 9, 10 april 2024

www.liacs.leidenuniv.nl/~kosterswa/AI/leren.pdf

Er zijn vele soorten leren:

supervised leren zowel input als juiste bijbehorende output zijn beschikbaar voor het leren (bijvoorbeeld ID3)

reinforcement leren het juiste antwoord wordt niet verteld, maar er is beloning/straf voor betere/slechtere antwoorden (bijvoorbeeld Genetische algoritmen)

unsupervised leren niets bekend over juiste antwoorden; probeer *patronen* te vinden (Data mining); clustering

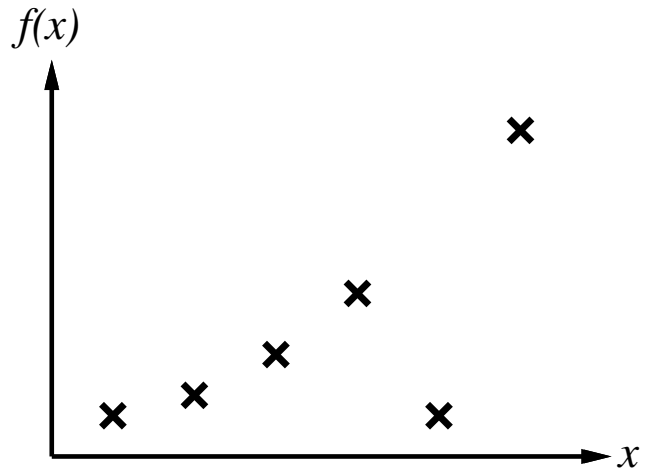
Sommigen onderscheiden ook statistische leermethoden, zoals Neurale netwerken en technieken voor Bayesiaanse netwerken. Samengevat: het gaat om Machine Learning.

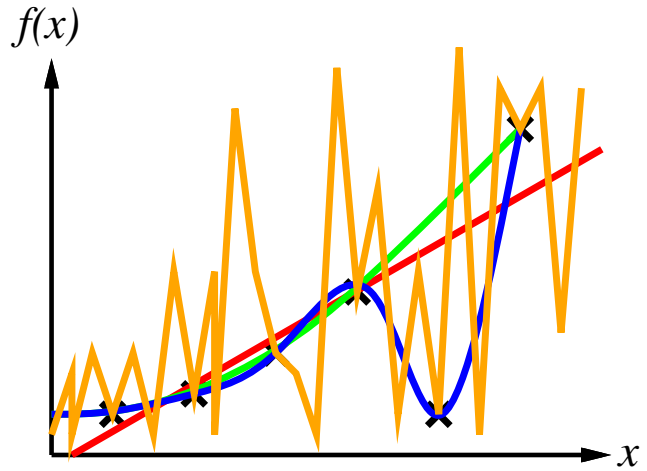
Zuiver inductief leren (inductie) is het zo goed mogelijk leren van een doelfunctie f op grond van voorbeeldparen $(x, f(x))$. We proberen een hypothese h te vinden die zoveel mogelijk op f lijkt: $h \approx f$, gegeven een trainingsset.

Voorbeeld: Neurale netwerken.

Dit is een simplificatie van het echte leren: zo gebruiken we geen voorkennis, nemen een observeerbare deterministische omgeving aan, en hebben voorbeelden nodig.

Een goede hypothese generaliseert, dat wil zeggen: kan ook nieuwe voorbeelden correct afhandelen.





Een bias is een voorkeur voor de ene hypothese boven de andere. Ockham's razor (ook: Occam) geeft de voorkeur aan eenvoudiger hypothesen (bij vergelijkbare prestatie).

±1300

Beslis(sings)bomen (decision trees) proberen uit een aantal voorbeelden een ja/nee beslissingsalgoritme af te leiden. Het is een classificatie-algoritme. Zie ook het college Data mining!

Elke interne knoop van de boom correspondeert met een test van de waarde van een van de eigenschappen (attributen) van de voorbeelden, en de takken naar de kinderen corresponderen met de mogelijke waarden.
Voor de classificatie van een nieuw voorbeeld v wandel je, beginnend bij de wortel, door de boom; je kiest steeds het kind dat correspondeert met v's waarde van het betreffende attribuut. Een blad bevat de Booleaanse waarde die je moet retourneren als je dat blad bereikt: de klasse.

Voorbeeld: Restaurant

Bij het bezoek aan een plaatselijk restaurant moeten we op grond van 10 attributen = features beslissen of we blijven wachten tot er plaats voor ons is (de "target" WW = WillWait). Er zijn 12 voorbeelden (Examples):

Ex	*Attributes*										*Target*
	Alt	Bar	Fri	Hun	Pat	Price	Rain	Res	Type	Est	WW
X_1	T	F	F	T	Some	$$$	F	T	French	0–10	T
X_2	T	F	F	T	Full	$	F	F	Thai	30–60	F
X_3	F	T	F	F	Some	$	F	F	Burger	0–10	T
X_4	T	F	T	T	Full	$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	$$$	F	T	French	>60	F
X_6	F	T	F	T	Some	$$	T	T	Italian	0–10	T
X_7	F	T	F	F	None	$	T	F	Burger	0–10	F
X_8	F	F	F	T	Some	$$	T	T	Thai	0–10	T
X_9	F	T	T	F	Full	$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	$$$	F	T	Italian	10–30	F
X_{11}	F	F	F	F	None	$	F	F	Thai	0–10	F
X_{12}	T	T	T	T	Full	$	F	F	Burger	30–60	T

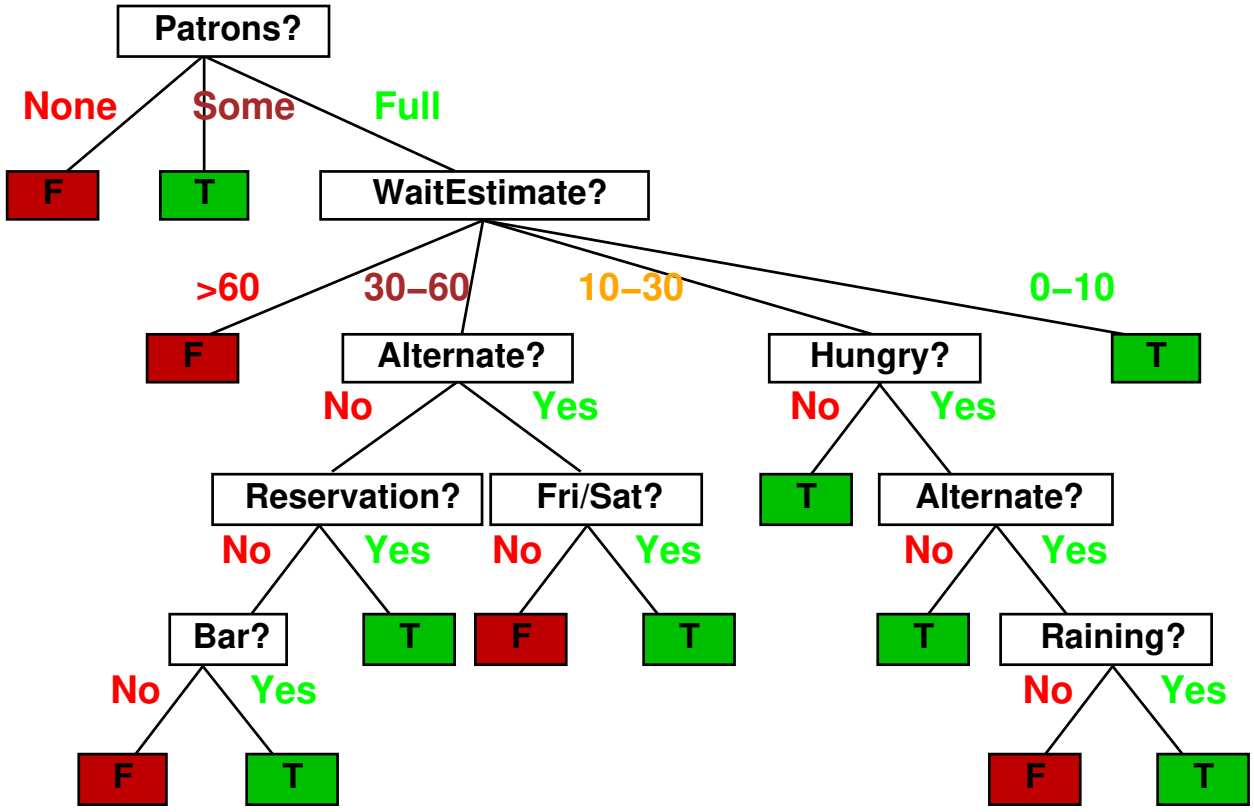
Restaurant — Attributen

We hebben 10 attributen:

Alt: is er dichtbij een alternatief?
Bar: is er een bar?
Fri: is het vandaag vrijdag/zaterdag of niet?
Hun: hebben we honger?
Pat: aantal klanten (“patrons”)
Price: prijs
Rain: regent het nu?
Res: hebben we een reservering gemaakt?
Type: soort restaurant
Est: door de eigenaar geschatte wachttijd

Restaurant — Boom

Een mogelijke beslissingsboom is:



Probleem

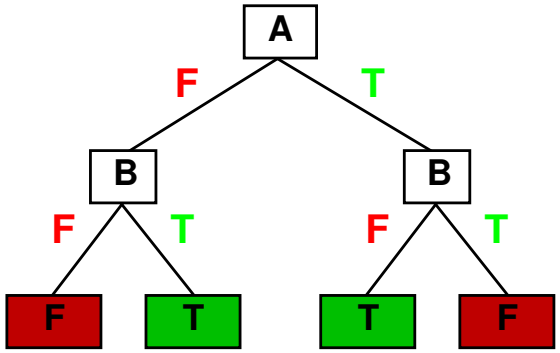
Het probleem is nu om bij gegeven voorbeelden (de trainingsset), positieve zowel als negatieve, een beslissingsboom te vinden die zo goed mogelijk bij de voorbeelden past en (Ockham) zo eenvoudig mogelijk is.

Er kunnen voorbeelden zijn die elkaar tegenspreken! En de boom hoeft ook niet alle attributen te gebruiken . . .

J. Ross Quinlan bedacht hiervoor in 1986 de methode ID3 ("Iterative Dichotomiser 3"), in 1993 verbeterd tot C4.5 en J48 — zoals het in het gratis data mining pakket Weka heet.

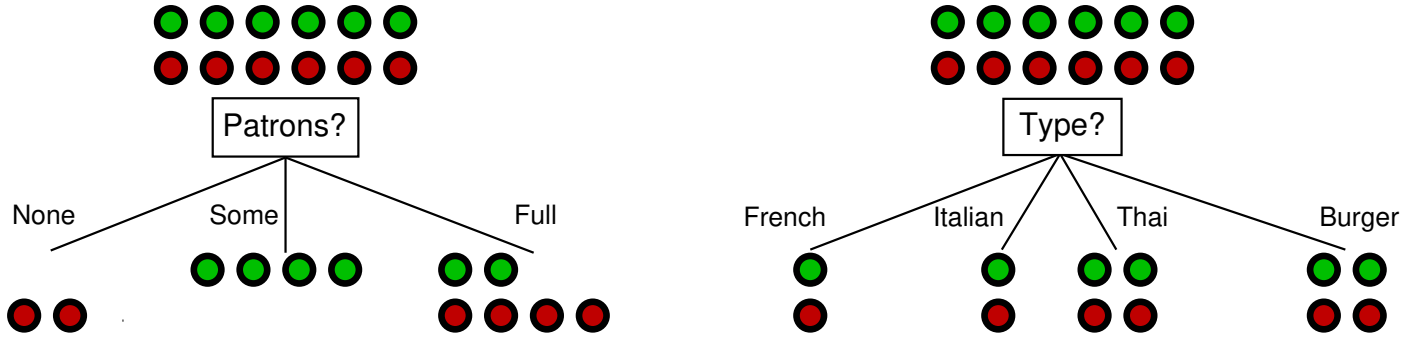
Nog een eenvoudig voorbeeld, een beslissingsboom voor het XOR-probleem:

A	B	A xor B
F	F	F
F	T	T
T	F	T
T	T	F



Je kunt dus voor ieder voorbeeld een pad in de boom maken, maar dat ziet er in het algemeen niet uit en generaliseert niet goed (en het gaat ook niet als voorbeelden elkaar tegenspreken).

Het idee van ID3 is als volgt. We moeten eerst het wortel-attribuut vinden, het attribuut waar we als eerste op splitsen: de splijtende vraag die je als eerste stelt.



Het lijkt er op dat het attribuut Pat(rons; aantal klanten) beter schift dan Type. (Dat maken we straks preciezer.) Dus kiezen we dat, en zo verder.

Het recursieve ID3-algoritme werkt als volgt.
Stel we zitten in een knoop, en hebben de verzameling V van alle voorbeelden die in die knoop terecht komen. Dan zijn er vier gevallen:

1. Geen voorbeelden meer ($V = \emptyset$)? Geef defaultwaarde, bijvoorbeeld "majority-value" van ouderknoop.

2. Alle voorbeelden dezelfde classificatie? Geef die.

3. Geen attributen meer? "Majority-value" van V.

4. Bepaal "beste" attribuut (hoe? — zie straks), splits daarop, en ga recursief verder met de kinderen.

Het ruwe idee van Shannon's informatietheorie is: hoe meer we verrast worden door een mededeling, hoe meer "inhoud" die mededeling had. De entropie kun je zien als de verwachte verrassing.

Als we een eerlijke munt hebben (kans op kop is 0.5) geeft een uitslag van een muntworp 1 bit "informatie".
Als de munt 0.99 kans op kop heeft (en dus 0.01 op munt) is dat (gemiddeld) 0.08 bits "informatie".

Stel dat een random-variabele X de volgende kansverdeling heeft: waarde A heeft kans 1/2, B kans 1/4 en C en D beide kans 1/8. Hoeveel ja/nee vragen heb je gemiddeld nodig om achter de waarde van X te komen? Dat zijn er (als je eerst naar A vraagt, dan (eventueel) naar B en dan (eventueel) naar C): $1/2 \cdot 1 + 1/4 \cdot 2 + 1/4 \cdot 3 = 1.75$.

Voor de entropie $H(X)$ van de kansverdeling X geldt (per definitie; $\log_2$ is logaritme met grondtal 2, $\log_2 2 = 1$):

$$\begin{aligned} H(X) &= -\sum_p p \log_2 p \\ &= -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{4}\log_2\frac{1}{4} - \frac{1}{8}\log_2\frac{1}{8} - \frac{1}{8}\log_2\frac{1}{8} = 1.75 \end{aligned}$$

En als je "optimaal" wilt coderen? Nu: A als 1, B als 01, C als 001 en D als 000, met gemiddelde lengte weer 1.75.

Voor ID3 wordt dit: stel we hebben p positieve en n negatieve voorbeelden in een knoop. "Vooraf" is de entropie:

$$-\frac{p}{p+n}\log_2\frac{p}{p+n}-\frac{n}{p+n}\log_2\frac{n}{p+n}\ ,$$

het gemiddeld aantal bits nodig om een voorbeeld te classificeren. (Voor het restaurant, met $p=n=6$: 1 bit.)

Stel we splitsen op een zeker attribuut a met ν mogelijke waarden, waarbij p_i positieve en n_i negatieve gevallen waarde i hebben ($1\leq i\leq\nu$). Dan is de entropie "achteraf":

$$\sum_{i=1}^{\nu}\frac{p_i+n_i}{p+n}\cdot\left\{-\frac{p_i}{p_i+n_i}\log_2\frac{p_i}{p_i+n_i}-\frac{n_i}{p_i+n_i}\log_2\frac{n_i}{p_i+n_i}\right\}\ ,$$

het verwachte aantal bits nog nodig voor classificatie — als we op a hebben gesplitst.

We kiezen nu het attribuut dat de maximale entropiewinst boekt. In ons restaurant-voorbeeld: vooraf

$$-\frac{6}{12}\log_2\frac{6}{12}-\frac{6}{12}\log_2\frac{6}{12}=1;$$

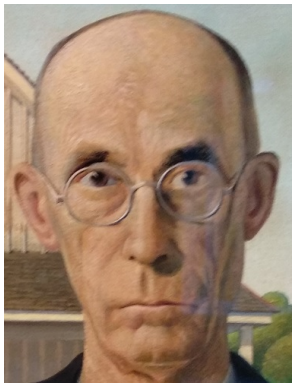
en achteraf, bij splitsen op Pat:

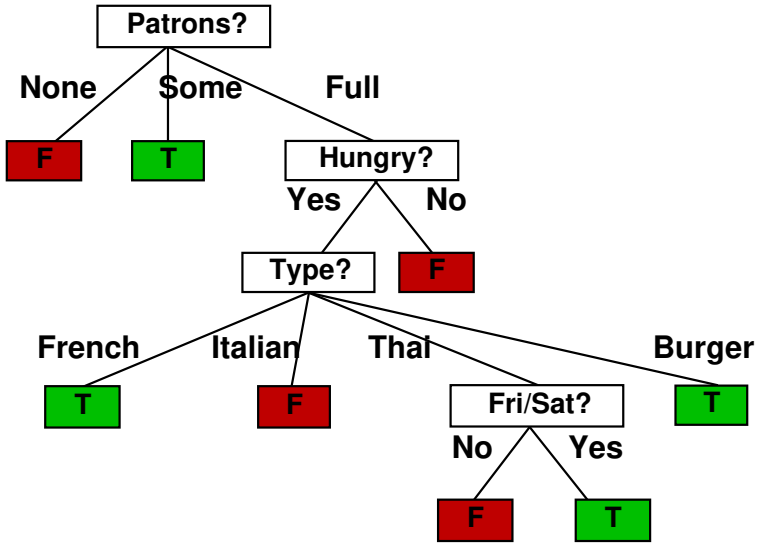
$$\frac{2}{12}\cdot 0+\frac{4}{12}\cdot 0+\frac{6}{12}\cdot\{-\frac{2}{6}\log_2\frac{2}{6}-\frac{4}{6}\log_2\frac{4}{6}\}=0.459$$

(de zogeheten "gain" is $1-0.459=0.541$; interpreteer $0\ \log_2 0$ als 0) en bij Type: nog steeds 1, met "gain" $1-1=0$.
Kies dus het attribuut met de hoogste "gain": Pat.

ID3-boom

De uiteindelijke ID3-boom is:





Let op: ID3 is een gretig algoritme, en levert niet altijd de beste boom op!

In het algemeen gebruik je een trainingsset, testset en validatieset, en "cross-validatie" . . . zie Neurale netwerken.

Er zijn allerlei problemen, waaronder noise (positief en negatief voorbeeld met dezelfde attribuut-waarden), irrelevante attributen (de dag als voorspeller voor een dobbelsteenworp), continue attribuutwaarden (temperatuur), missende waarden, enzovoorts. In C4.5, de opvolger van ID3, wordt hier rekening mee gehouden. En het maakt de boom kleiner: "pruning".

Een voorbeeld. Duidelijk is dat meerwaardige attributen vaak onterecht gekozen worden, bijvoorbeeld de datum of het identificatienummer bij ons restaurant-voorbeeld. Daarom wordt de entropiewinst vaak gedeeld door

$$-\sum_{i=1}^{\nu} \frac{p_i + n_i}{p + n} \log_2 \frac{p_i + n_i}{p + n} .$$

Nog een voorbeeld: Tennis

Bekijk de volgende dataset met trainings-voorbeelden:

Dag	Weer	Temp	Vochtigheid	Wind	Tennis?
D1	zon	warm	hoog	zwak	N
D2	zon	warm	hoog	sterk	N
D3	bewolkt	warm	hoog	zwak	J
D4	regen	gem	hoog	zwak	J
D5	regen	koud	normaal	zwak	J
D6	regen	koud	normaal	sterk	N
D7	bewolkt	koud	normaal	sterk	J
D8	zon	gem	hoog	zwak	N
D9	zon	koud	normaal	zwak	J
D10	regen	gem	normaal	zwak	J
D11	zon	gem	normaal	sterk	J
D12	bewolkt	gem	hoog	sterk	J
D13	bewolkt	warm	normaal	zwak	J
D14	regen	gem	hoog	sterk	N

We berekenen de "gain" van de vier mogelijke attributen:

Weer	Temp	Vochtigheid	Wind
0.246	0.029	0.151	0.048

Dus kiezen we voor Weer in de wortelknoop. Nu zijn er 5 dagen met Weer = zon, 2 positief en 3 negatief. De "gain" van Temp is hier $\{-\frac{3}{5}\log_2\frac{3}{5}-\frac{2}{5}\log_2\frac{2}{5}\}-\{\frac{2}{5}\cdot 0+\frac{2}{5}\cdot 1+\frac{1}{5}\cdot 0\} = 0.970-0.4 = 0.570$, maar die van Vochtigheid is het beste: 0.970 (voor Wind: 0.019), dus die kiezen we daar.

Uiteindelijk vinden we de beslissingsboom die hoort bij:
(Weer = zon $\wedge$ Vochtigheid = normaal) $\vee$
(Weer = bewolkt) $\vee$ (Weer = regen $\wedge$ Wind = zwak)

Elke Booleaanse functie kun je als een beslissingsboom schrijven. De majority functie (zie Neurale netwerken) vergt een erg grote boom, evenals de "parity functie" (1 precies als een even aantal inputs 1 is). Een beslissingsboom is soms goed, soms niet.

Er zijn 2^{2^n} verschillende waarheidstafels met n Booleans, en dus evenveel mogelijke beslissingsbomen. Voor $n = 6$ zijn dat er 18.446.744.073.709.551.616. Dit is het totaal aantal mogelijke hypothesen.

Een beslissingsboom met maar één knoop heet een decision stump.

Zie ook PAC (probably approximately correct) learning.

Het leren van hypothesen kun je ook als volgt doen. Voor ons restaurant-voorbeeld komen de voorbeelden een voor een binnen als logische zinnen: $\mathsf{Alt}(X_1) \wedge \neg\mathsf{Bar}(X_1) \wedge \ldots$
We zoeken een hypothese die de classificatie $\mathsf{WillWait}(X)$ of $\neg\mathsf{WillWait}(X)$ goed "voorspelt".

Onze ID3-boom stelt de volgende hypothese voor:

$$\begin{aligned}\forall x\ \mathsf{WillWait}(x) \Leftrightarrow\ & \mathsf{Pat}(x, \mathsf{Some}) \\ \vee\ & (\mathsf{Pat}(x, \mathsf{Full}) \wedge \mathsf{Hun}(x) \wedge \mathsf{Type}(x, \mathsf{French})) \\ \vee\ & (\mathsf{Pat}(x, \mathsf{Full}) \wedge \mathsf{Hun}(x) \wedge \mathsf{Type}(x, \mathsf{Thai}) \\ & \qquad \wedge\ \mathsf{Fri}(x)) \\ \vee\ & (\mathsf{Pat}(x, \mathsf{Full}) \wedge \mathsf{Hun}(x) \wedge \mathsf{Type}(x, \mathsf{Burger}))\end{aligned}$$

Na het zien van X_1, positief, met Alt(X_1) true, zou onze hypothese kunnen zijn

$$H_1 = \forall x \text{ WillWait}(x) \Leftrightarrow \text{Alt}(x)$$

Na X_2, negatief, met Alt(X_2) true (dus een false positive) gaan we specialiseren naar (bijvoorbeeld):

$$H_2 = \forall x \text{ WillWait}(x) \Leftrightarrow \text{Alt}(x) \wedge \text{Pat}(x, \text{Some})$$

Na de false negative X_3 generaliseren naar (bijvoorbeeld):

$$H_3 = \forall x \text{ WillWait}(x) \Leftrightarrow \text{Pat}(x, \text{Some})$$

Denk aan het kloppend houden van X_1 en X_2. Na X_4:

$$H_4 = \forall x \text{ WillWait}(x) \Leftrightarrow \text{Pat}(x, \text{Some}) \vee (\text{Pat}(x, \text{Full}) \wedge \text{Fri}(x))$$

Current best hypothesis

Je kunt dus één hypothese onderhouden, die je aanpast bij nieuwe voorbeelden: de current best hypothesis.

Het algoritme is:

- bij "false positive": specialiseren (voeg condities toe)
- bij "false negative": generaliseren (laat condities weg)

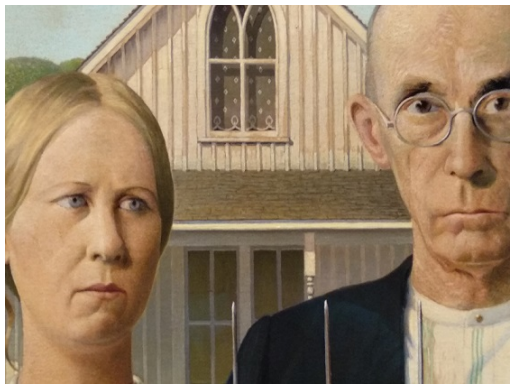
Problemen: herhaald erg veel controleren, lastige heuristieken, veel backtracking noodzakelijk.

Least commitment search

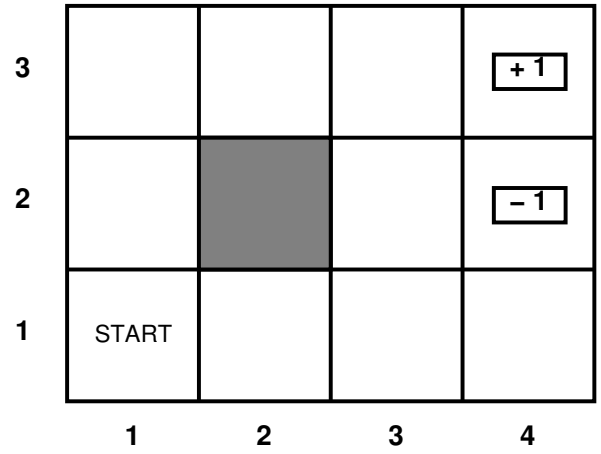
Bij least commitment search onderhoudt je een partieel geordende verzameling van toegestane hypothesen (de version space), met twee grensverzamelingen bestaande uit de meest algemene (de G-verzameling; "general") en de meest specifieke (de S-verzameling) hypothesen.

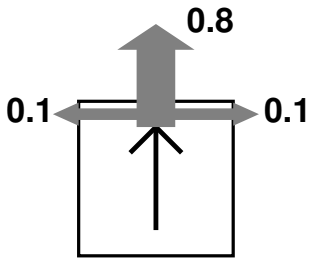
Kom je een false positive tegen voor een $S_i \in S$, dan gooi je S_i er uit. En bij een false negative vervang je S_i door redelijke generalisaties. Analoog bij G.

Je komt wel voor ingewikkelde problemen te staan.



Een voorbeeld van een Markov Decision Process (MDP):





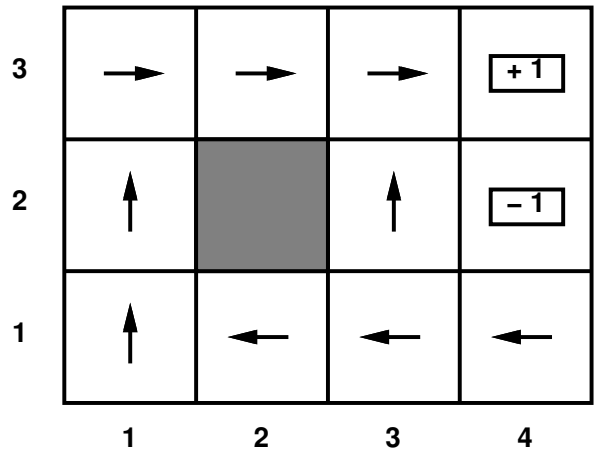
Het transitiemodel $T(s, a, s')$ (rechts) geeft de kans dat actie a vanuit toestand s resulteert in toestand s'. De beloning (reward) voor een niet-doel toestand s is $R(s) = -0.04$.

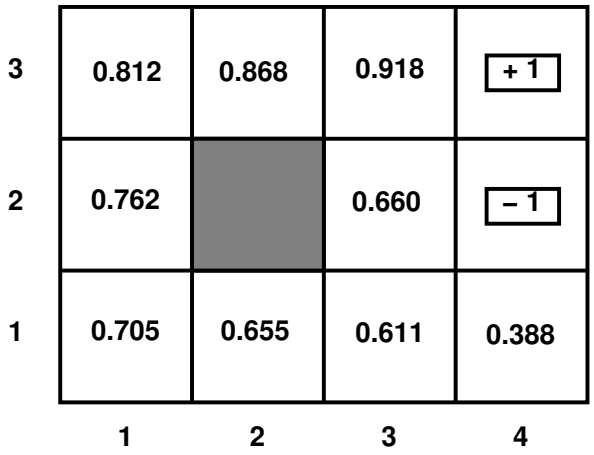
Een policy π vertelt in elke toestand s wat te doen: $\pi(s)$.

De utility(-functie) U is de "som van de opgedane rewards".

Een optimale policy π^* geeft de hoogste verwachte utility.

Een optimale policy met bijbehorende utilities:





De verwachte utility hangt dus af van de policy:

$$U^{\pi}(s) = E\{\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid s = s_0, \pi\}$$

Hier is $0 < \gamma \leq 1$ de discount factor; $\pi^*(s)$ is de actie a met maximale $\sum_{s'} T(s, a, s') U^{\pi^*}(s')$.

De (optimale) utilities $U(s) = U^{\pi^*}(s)$ voldoen aan de Bellman vergelijking (1957):

$$U(s) = R(s) + \gamma \max_a \sum_{s'} T(s, a, s')\, U(s')$$

In bijvoorbeeld vakje $(3, 3)$ geldt (als $\gamma = 1$):

$$0.918 = -0.04 + 1 \cdot (0.1 \cdot 0.918 + 0.8 \cdot 1 + 0.1 \cdot 0.660)$$

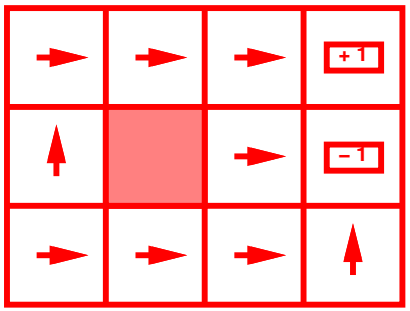
Je kunt de vergelijking(en) benaderend oplossen met:

$$U_{i+1}(s) = R(s) + \gamma \max_a \sum_{s'} T(s, a, s')\, U_i(s')$$

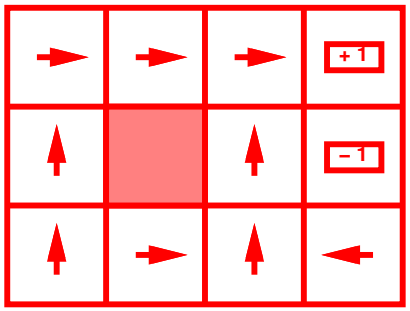
Dit heet value iteration; ook bestaat policy iteration.

Verschillende optimale policy's ☰

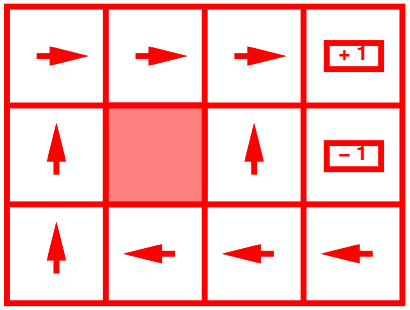
Afhankelijk van de waarde $r = R(s)$ van de reward in niet-doel toestanden heb je verschillende optimale policy's:



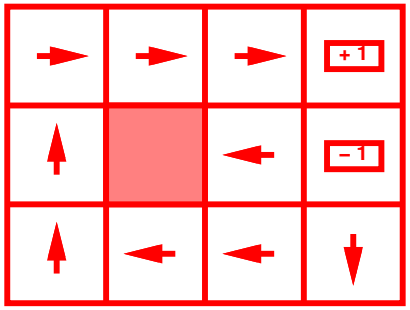
r in [−∞ ; −1.6284]



r in [−0.4278 ; −0.0850]



r in [−0.0480 ; −0.0274]



r in [−0.0218 ; 0.0000]

Bij direct utility estimation bekijk je grote aantallen trials, en schat op grond daarvan:
$(1,1)_{-0.04} \to (1,2)_{-0.04} \to (1,3)_{-0.04} \to (1,2)_{-0.04}$
$\to (1,3)_{-0.04} \to (2,3)_{-0.04} \to (3,3)_{-0.04} \to (4,3)_{+1}$
levert schatting $0.84 = (0.80 + 0.88)/2$ voor $U^\pi((1,3))$.

Bij Temporal Difference (TD) leren gebruik je de leerregel
$U^\pi(s) \leftarrow U^\pi(s) + \alpha\,(R(s) + \gamma U^\pi(s') - U^\pi(s))$
als je $s \to s'$ ziet (α is de leersnelheid). Als je denkt dat $U^\pi((1,3)) = 0.84$ en $U^\pi((2,3)) = 0.92$, en je ziet $(1,3) \to (2,3)$, verwacht je $U^\pi((1,3)) = -0.04 + 1 \cdot U^\pi((2,3))$!

Bij Q-leren probeer je $Q(a,s)$, de utility als je a doet in toestand s, te leren; $U(s) = \max_a Q(a,s)$. We kunnen

$$Q(a,s) = R(s) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q(a',s')$$

leren via (als $s \xrightarrow{a} s'$)

$$Q(a,s) \leftarrow Q(a,s) + \alpha\,(R(s) + \gamma \max_{a'} Q(a',s') - Q(a,s))$$

Het huiswerk voor de volgende keer (17 april 2024): lees Hoofdstuk 21, p. 750–775 van [RN] door (in de derde druk p. 727–737) over het onderwerp Deep learning.

Voor meer gevanceerde onderwerpen (Ξ = geen tentamenstof) lees Hoofdstuk 17 (over MDP's) en Hoofdstuk 22 (Reinforcement learning) van [RN].

Werk aan de derde opgave: A*; deadline: 17 april 2024.

Kunstmatige Intelligentie (AI)

Hoofdstuk 21 van Russell/Norvig = [RN]

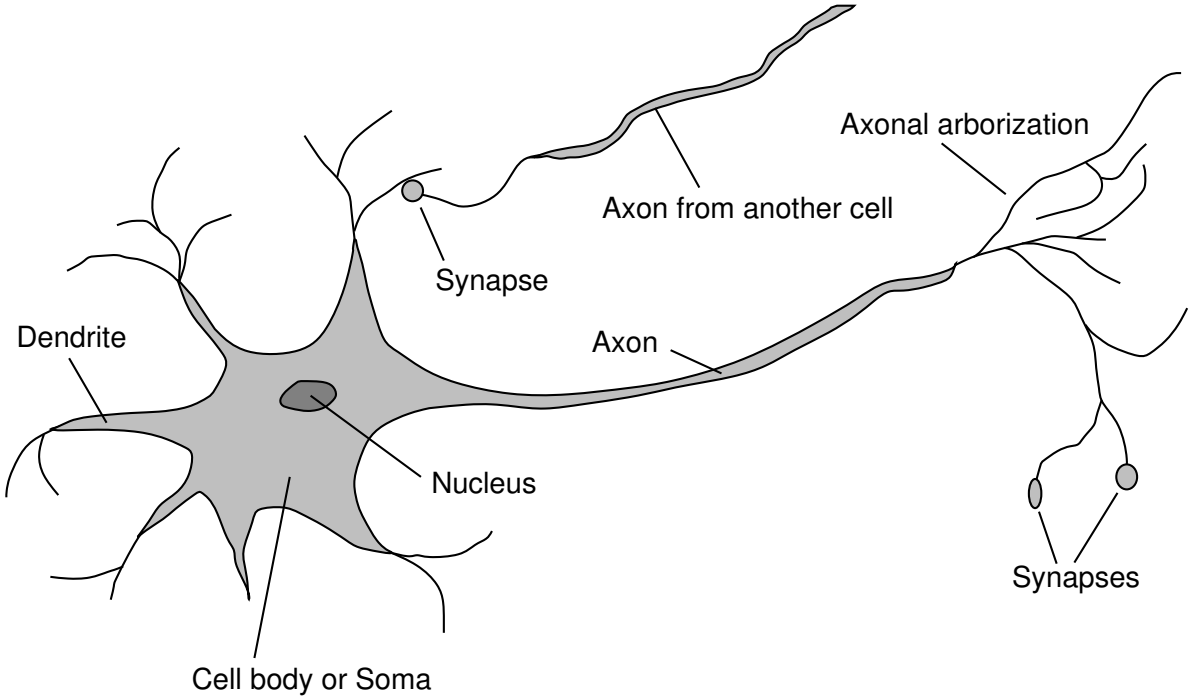
Deep learning, Neurale netwerken

voorjaar 2024

College 10 en 11, 17 en 24 april 2024

www.liacs.leidenuniv.nl/~kosterswa/AI/neuraal.pdf

De menselijke hersenen bestaan uit 10^{11} neuronen (grootte $\approx$ 0.1 mm; meer dan 20 types), die onderling met axonen (lengte $\approx$ 1 cm) en synapsen verbonden zijn:



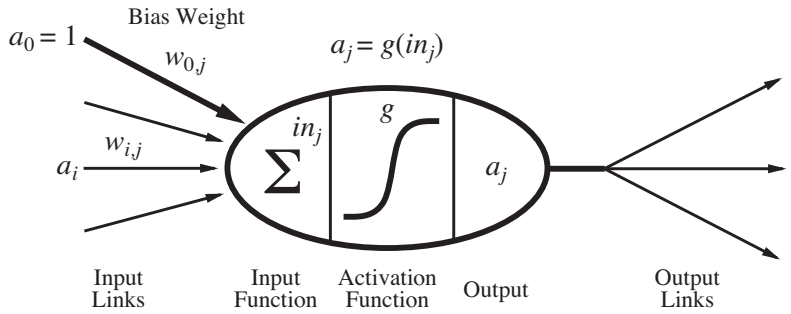
Signalen worden doorgegeven via een nogal gecompliceerde electro-chemische reactie.

Als de electrische potentiaal van het cellichaam een zekere drempelwaarde haalt, wordt een puls/actie-potentiaal/spike-train op het axon gezet.

Er zijn excitatory (verhogen potentiaal) en inhibitory (verlagen potentiaal) synapsen.

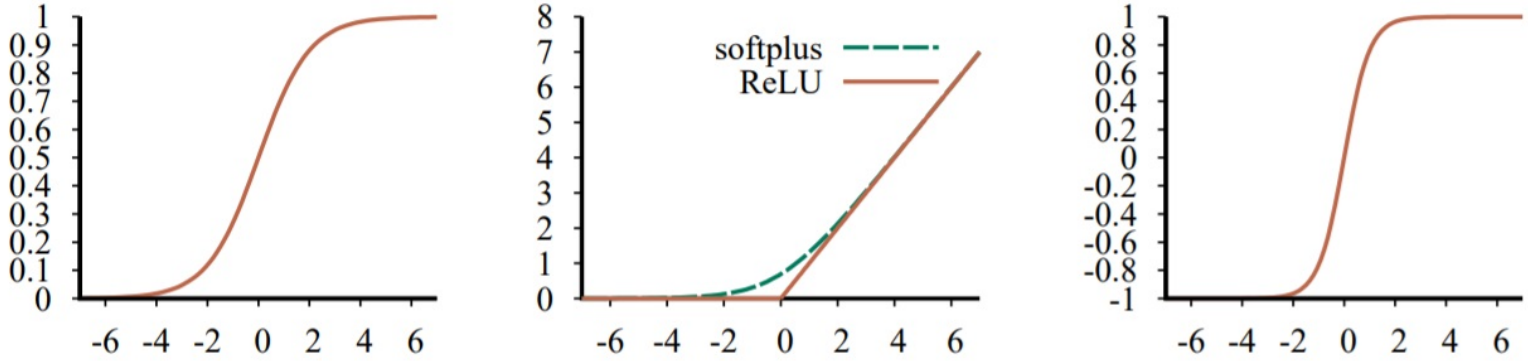
Ooit imiteerden/modelleerden Neurale netwerken menselijke hersenen. Of andersom?

We modelleren de neuronen door middel van eenheden (units) die we ook weer neuronen noemen:



Er geldt: de input van neuron j is $in_j = \sum_i W_{i,j}\ a_i$ (de gewogen som van de inputs), waarbij de a_i's de activaties bij de inkomende verbindingen (inputs) zijn, en $W_{i,j}$ hun gewichten ($W_{i,j}$ weegt de verbinding van neuron i naar neuron j). De activatie (output) van neuron j is $a_j = g(in_j)$, met g de activatie-functie.

Veelgebruikte activatie-functies zijn:



sigmoid$(x) = \sigma(x) = 1/(1 + e^{-\beta x})$ (vaak $\beta = 1$)

ReLU$(x) = \max(0, x)$ en softplus$(x) = \log(1 + e^x)$

$\tanh(x) = (e^{2x} - 1)/(e^{2x} + 1) = 2\sigma(2x) - 1$

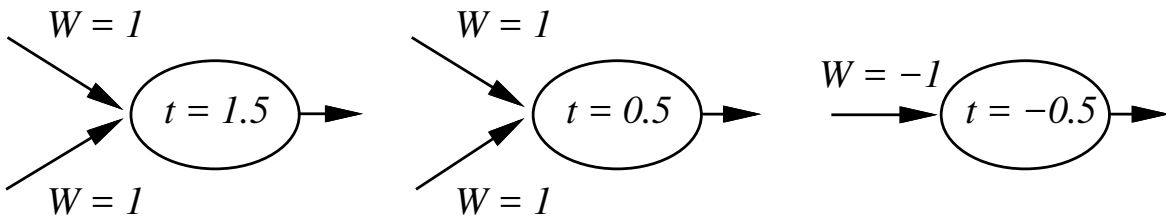
ReLU = rectified linear unit; sigmoid = logistic

De activatie-functies "kantelen" bij 0; De drempelwaarde t binnen de neuronen kan "gesimuleerd" worden door een extra (constante) activatie -1 (of $+1$) in een zogeheten bias-knoop 0 met gewicht $W_{0,j} = t$ op de verbinding van neuron 0 naar neuron j:

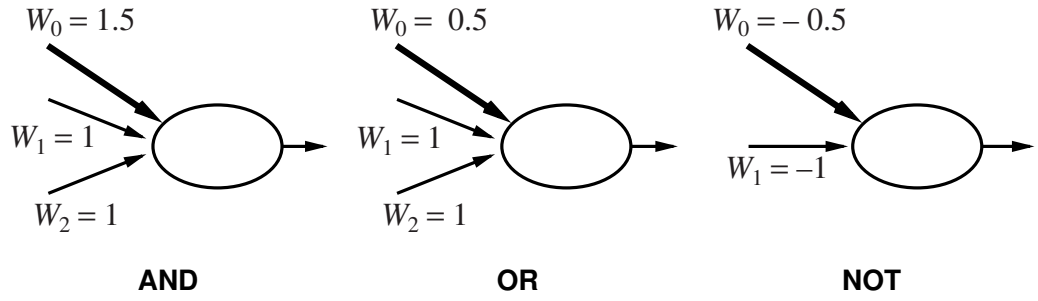
$$\sum_{i=1}^{n} W_{i,j}\ a_i > t \Leftrightarrow \sum_{i=0}^{n} W_{i,j}\ a_i > 0$$

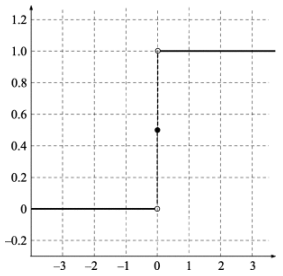
Zo behandelen we drempelwaardes en gewichten uniform.

Alle Booleaanse functies kunnen gerepresenteerd worden door netwerken met geschikte neuronen:



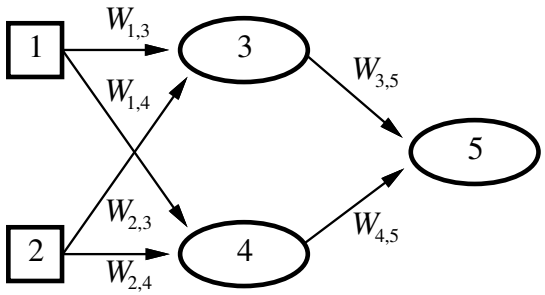
En mét bias-knopen:





Met activatie-functie $\mathtt{Y}(x) = 1$ als $x \geq 0$; 0 als $x < 0$.

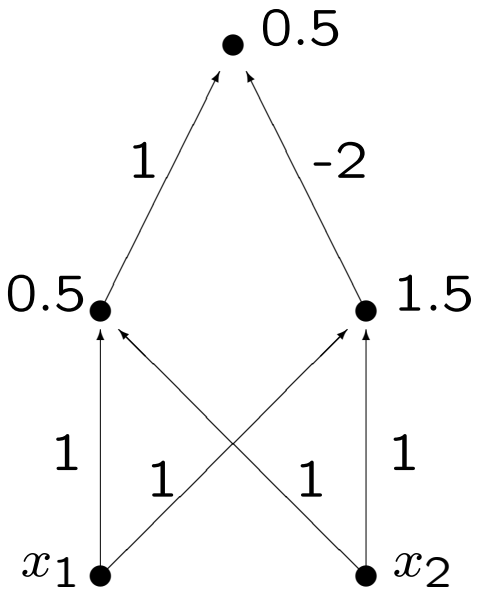
Een feed-forward netwerk heeft gerichte takken en geen cykels (in tegenstelling tot een recurrent netwerk). Vaak is zo'n netwerk in lagen georganiseerd:

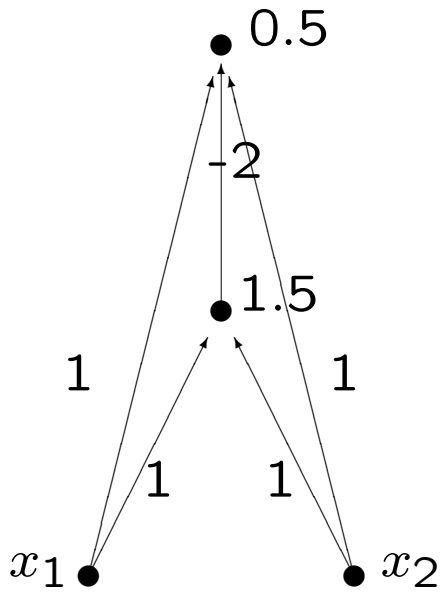


Dit netwerk heeft twee input-knopen 1 en 2, twee verborgen (= hidden) knopen 3 en 4, en één uitvoer-knoop 5. Het representeert de volgende functie:

$$\begin{aligned} a_5 &= g_5\left(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4\right) \\ &= g_5\left(W_{3,5} \cdot g_3\left(W_{1,3} \cdot x_1 + W_{2,3} \cdot x_2\right)\right. \\ &\quad \left. + W_{4,5} \cdot g_4\left(W_{1,4} \cdot x_1 + W_{2,4} \cdot x_2\right)\right) \end{aligned}$$

Voorbeelden:





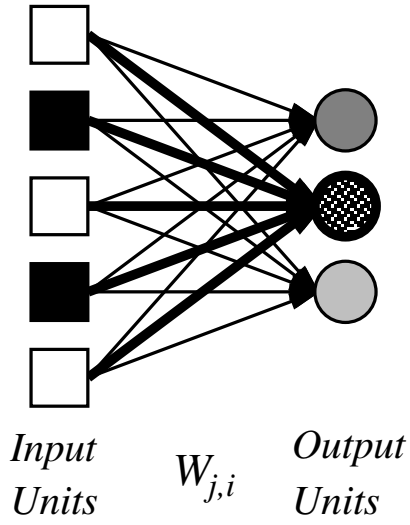
De drempels staan naast de knopen.
Het linker feed-forward netwerk representeert de `XOR`-functie, de verborgen units zijn in feite een `OR` en een `AND`.
Het rechter netwerk representeert dezelfde functie, maar is niet "conventioneel" (geen lagen).

Een feed-forward netwerk zonder verborgen neuronen heet een perceptron. Een meerlaags (= multi-layer) netwerk heeft één of meer verborgen lagen, en alle pijlen van laag ℓ gaan naar laag $(\ell + 1)$.

Met één (voldoende grote) laag met verborgen units kunt je elke continue functie benaderen, en met twee lagen zelfs elke discontinue functie (Cybenko).

De output van een netwerk hangt af van de parameters, de gewichten. Bij *te veel* parameters is er gevaar voor overfitting: het netwerk generaliseert dan niet goed.

In een perceptron (geen verborgen units!) zijn de uitvoerknopen onafhankelijk:



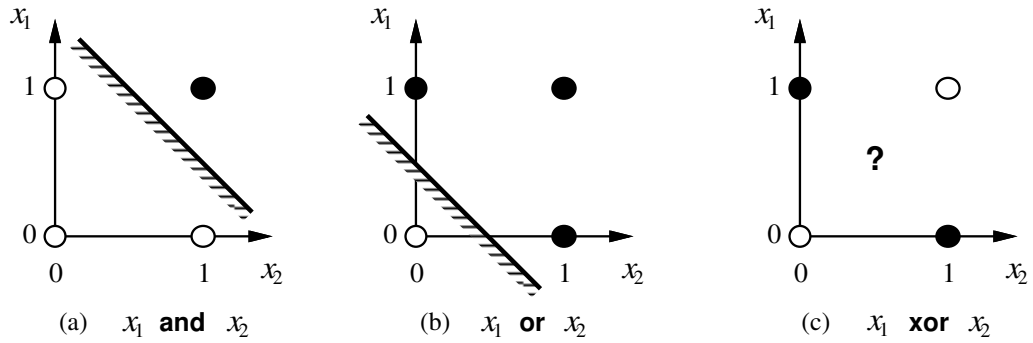
Voor een enkele output-unit geldt dat de uitvoer 1 is indien $\sum_j W_j x_j = W_0 x_0 + W_1 x_1 + \ldots + W_n x_n \geq 0$ en anders 0. Hierbij is $(x_1, \ldots, x_n)$ de invoer en $x_0 = -1$ de bias-knoop. We gebruiken dus weer even de Heaviside/stap-functie

$$\mathtt{Y}(x) = 1 \text{ als } x \geq 0;\ 0 \text{ als } x < 0$$

Wat kan een perceptron?

De majority functie (uitvoer 1 $\Leftrightarrow$ meer dan de helft van de n inputs is 1) kan eenvoudig gemaakt worden: kies $W_0 = n/2$ en $W_j = 1$ voor $j = 1, 2, \ldots, n$.

De vergelijking $-W_0 + W_1 x_1 + \ldots + W_n x_n \geq 0$ laat zien dat je precies Booleaanse functies kunt maken die lineair te scheiden zijn, de `XOR`-functie dus niet:



(a) x_1 **and** x_2 (b) x_1 **or** x_2 (c) x_1 **xor** x_2

Gegeven genoeg trainings-voorbeelden, kan een perceptron elke Booleaanse lineair te scheiden functie leren. Een (hyper)vlak scheidt positieve en negatieve voorbeelden.

Rosenblatt's algoritme uit 1957/60 werkt als volgt:

$$W_j \longleftarrow W_j + \alpha \cdot x_j \cdot \text{Error}$$

met Error = correcte uitvoer − net-uitvoer en $\alpha > 0$ de leersnelheid (= learning rate). De correcte uitvoer heet wel de target, het doel.
Als Error = 1 en $x_j = 1$, wordt W_j ietsje opgehoogd, in de hoop dat de net-uitvoer hoger wordt.

We willen een perceptron x_1 AND x_2 leren, met $\alpha = 0.1$:

	$x_0 x_1 x_2$	W_0	W_1	W_2	uitvoer	target	Error
1	-1 1 1	-0.160	-0.606	-0.217	0.000	1.000	1.000
2	-1 1 0	-0.260	-0.506	-0.117	0.000	0.000	0.000
3	-1 0 1	-0.260	-0.506	-0.117	1.000	0.000	-1.000
4	-1 1 0	-0.160	-0.506	-0.217	0.000	0.000	0.000
5	-1 1 0	-0.160	-0.506	-0.217	0.000	0.000	0.000
6	-1 1 1	-0.160	-0.506	-0.217	0.000	1.000	1.000
7	-1 0 0	-0.260	-0.406	-0.117	1.000	0.000	-1.000
8	-1 0 0	-0.160	-0.406	-0.117	1.000	0.000	-1.000
...							
70	-1 1 0	0.140	0.194	0.183	1.000	0.000	-1.000
71	-1 1 1	0.240	0.094	0.183	1.000	1.000	0.000
...							

Probleem: wanneer stop je?

We kunnen in plaats van de discontinue functie $\mathtt{Y}$ ook een differentieerbare activatie-functie g gebruiken in de knopen. Een vergelijkbaar leeralgoritme gaat dan als volgt.

Zij $E = \frac{1}{2}\text{Error}^2 = \frac{1}{2}\Big(y - g(\sum_{j=0}^{n} W_j x_j)\Big)^2$ met y de target. Met behulp van gradient descent bepalen we in welke richting de fout het snelst *stijgt*:

$$\frac{\partial E}{\partial W_j} = \text{Error} \cdot \frac{\partial \text{Error}}{\partial W_j} = -\text{Error} \cdot g'(\text{in}) \cdot x_j$$

met $\text{in} = \sum_{j=0}^{n} W_j x_j$. Dus leerregel (let op de extra $-$, we willen de fout laten *dalen*; leersnelheid $\alpha > 0$):

$$W_j \longleftarrow W_j + \alpha \cdot \text{Error} \cdot g'(\text{in}) \cdot x_j \quad .$$

Een kunstmatig voorbeeld: stel we proberen de uitvoer $y = 5$ te bereiken met de functie $-W_1 + 4W_2$, en we zitten in $(W_1, W_2) = (1, 1)$; de gok is dus 3. En E is gelijk aan:

$$E = \frac{1}{2}\Big(5 - (-W_1 + 4W_2)\Big)^2 \text{ dus } \frac{1}{2}(5-3)^2 = 2$$

We rekenen uit

$$\left.\frac{\partial E}{\partial W_1}\right|_{(W_1,W_2)=(1,1)} = \left.(5 - (-W_1 + 4W_2))\right|_{(W_1,W_2)=(1,1)} = 2$$

$$\left.\frac{\partial E}{\partial W_2}\right|_{(W_1,W_2)=(1,1)} = \left.-4(5 - (-W_1 + 4W_2))\right|_{(W_1,W_2)=(1,1)} = -8$$

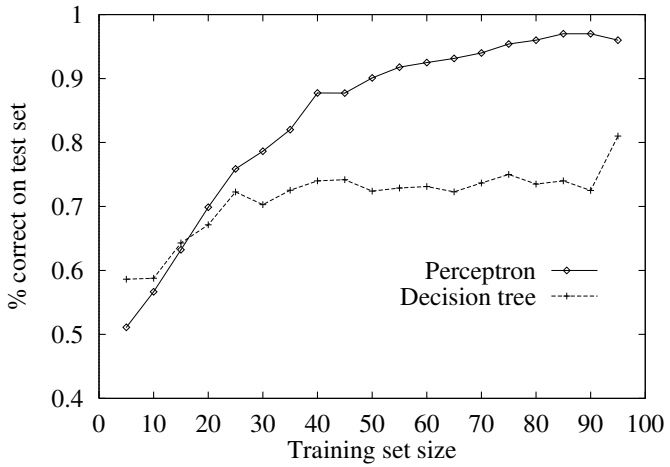
en passen aan (met kleine $\alpha > 0$; let op de $-$):

$$W_1 \longleftarrow 1 - 2\alpha, \quad W_2 \longleftarrow 1 + 8\alpha$$

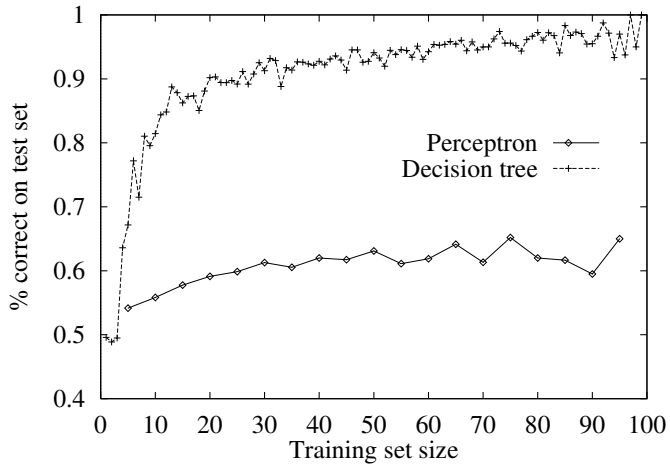
Dus W_2 wordt meer (omhoog) aangepast!

Perceptron — vergelijking

Voor een lineair te scheiden majority probleem (11 inputs) is het perceptron veel beter dan een techniek als beslissingsbomen (decision trees, zie Data Mining en ook het vorige college "Leren"):



Maar voor een meer complex probleem als het "restaurant" is het omgekeerd:



Er zijn allerlei keuzes om invoer en uitvoer te coderen. Je kunt bijvoorbeeld locaal coderen: stel dat een variabele 3 waarden kan hebben: Geen, Gemiddeld en Veel; dit kun je dan in één knoop coderen als 0.0, 0.5 en 1.0 respectievelijk. Maar waarschijnlijk beter met drie aparte knopen, en dan respectievelijk als 1–0–0, 0–1–0 en 0–0–1: gedistribueerd coderen = one-hot encoding.

Je kunt zelfs bij de foutmaat ervoor zorgen dat je waarden als 1–1–0.9 niet fijn vindt, en daar van weg trainen met een extra laag.
Oftewel: we willen kansen ≥ 0 die samen 1 zijn:

$$\text{softmax}(x_1, \ldots, x_n) = \left(\frac{e^{x_1}}{\sum_{i=1}^{n} e^{x_i}}, \ldots, \frac{e^{x_n}}{\sum_{i=1}^{n} e^{x_i}} \right)$$

Hoe verloopt nu het leren bij Neurale Netwerken, het aanpassen van de gewichten, kortom: de training?

Je begint met een random initialisatie van de gewichten. Vervolgens biedt je één voor één voorbeelden aan uit een zogeheten trainingsset. Deze voorbeelden moeten in een willekeurige volgorde staan. Steeds geef je een invoer, en met behulp van de juiste uitvoer pas je volgens je trainings-algoritme de gewichten aan.

Je gaat net zolang door totdat (bijvoorbeeld) de fout op een apart gehouden validatieset niet meer daalt — of zelfs gaat stijgen (overtraining).

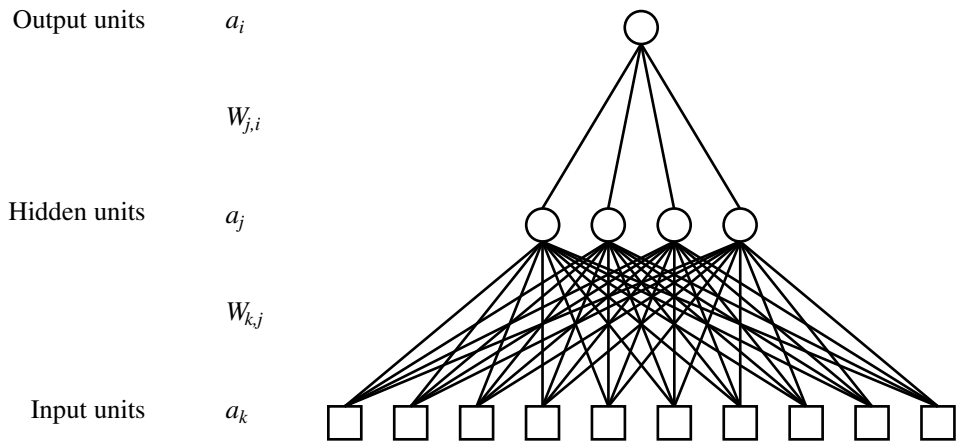
Je rapporteert tot slot over het gedrag op een (weer apart gehouden) testset.

Bij elk leer-algoritme kun je cross-validation gebruiken om overfitting tegen te gaan.

Bij "k-fold cross-validation" (vaak $k = 5$ of $k = 10$) draai je k experimenten, waarbij je steeds een $1/k$-de deel van de data apart zet om als testset te gebruiken — en de rest als trainingsset. De testset is dus steeds een ander random gekozen deel!

Als $k = n$ met n de grootte van de dataset, heet deze techniek wel "leave-one-out". En dan heb je ook nog "ensemble leren", AdaBoost, enzovoorts. Zie verder het bachelorcollege Data Mining. En Statistiek.

We kijken nu naar een meerlaags neuraal netwerk. Meestal zijn alle lagen onderling *volledig* verbonden.



De notatie is ietwat dubbelzinnig: zit $W_{1,2}$ op twee plaatsen? Je kunt of knopen doornummeren (zie sheet 8), of meerdere W's hanteren, of hopen dat er geen misverstand ontstaat. We gebruiken index i voor de uitvoerlaag, j voor de verborgen laag en k voor de invoerlaag. De a_k's zijn de input(s) — wat we eerder x_k noemden.

Met

$$\begin{aligned} a_5 &= g(in_5) = g(W_{3,5}\, a_3 + W_{4,5}\, a_4) \\ &= g(W_{3,5}\, g(W_{1,3}\, a_1 + W_{2,3}\, a_2) \\ &\quad + W_{4,5}\, g(W_{1,4}\, a_1 + W_{2,4}\, a_2)) \end{aligned}$$

geldt

$$\frac{\partial a_5}{\partial W_{3,5}} = g'(in_5) \cdot a_3$$

en

$$\frac{\partial a_5}{\partial W_{1,3}} = g'(in_5) \cdot W_{3,5} \cdot g'(in_3) \cdot a_1 \quad .$$

Hierbij: $in_5 = W_{3,5}\, a_3 + W_{4,5}\, a_4$ en $in_3 = W_{1,3}\, a_1 + W_{2,3}\, a_2$.
In dit geval hebben we dus doorgenummerd.

Het meest bekende leerschema voor Neurale Netwerken is back-propagation (1969, 198?), waarbij we — nadat een invoer een uitvoer heeft opgeleverd — de fout teruggeven (propageren) van uitvoerlaag richting invoerlaag.

Definieer Error_i als de fout in de i-de uitvoer (dat wil zeggen: i-de target minus i-de uitvoer van het netwerk, dus $y_i - a_i$). De leerregel is dan net als eerder:

$$W_{j,i} \longleftarrow W_{j,i} + \alpha \cdot a_j \cdot \Delta_i \quad \text{met} \quad \Delta_i = \text{Error}_i \cdot g'(\text{in}_i)$$

voor gewichten $W_{j,i}$ van knoop j in de verborgen laag naar knoop i in de uitvoerlaag.

Voor gewicht $W_{k,j}$ van de k-de invoerknoop naar knoop j in de verborgen laag is de leerregel:

$$W_{k,j} \longleftarrow W_{k,j} + \alpha \cdot a_k \cdot \tilde{\Delta}_j \quad \text{met} \quad \tilde{\Delta}_j = g'(\text{in}_j) \sum_i W_{j,i} \Delta_i$$

Hierbij geldt:

α is de leersnelheid

$a_k = x_k$ is de activatie van de k-de invoerknoop

in_j is de gewogen invoer voor de j-de verborgen knoop

$W_{j,i}$ is het gewicht op de verbinding tussen de j-de verborgen knoop en de i-de uitvoerknoop

Δ_i is de i-de "uitvoer-delta", zie de vorige sheet

We leiden de leerregel met gradient descent af. De gradient wijst in de richting van de sterkste toename.

De fout per voorbeeld (x, y) is $E = \frac{1}{2}\sum_i (y_i - a_i)^2$, waarbij de y_i's samen de target y vormen. Definieer $\text{Error}_i = y_i - a_i$.

Dan geldt voor het gewicht $W_{j,i}$ op de verbinding van de j-de verborgen knoop naar de i-de uitvoerknoop:

$$\frac{\partial E}{\partial W_{j,i}} = -(y_i - a_i) \cdot \frac{\partial a_i}{\partial W_{j,i}} = -(y_i - a_i) \cdot g'(\text{in}_i) \cdot a_j$$

We hebben namelijk $a_i = g(\text{in}_i) = g(\sum_\ell W_{\ell,i} a_\ell)$.

Dus wordt de leerregel:

$$W_{j,i} \longleftarrow W_{j,i} + \alpha \cdot a_j \cdot \Delta_i \quad \text{met} \quad \Delta_i = \text{Error}_i \cdot g'(\text{in}_i)$$

We hebben $E = \frac{1}{2}\sum_i (y_i - a_i)^2$, $a_i = g(\sum_\ell W_{\ell,i} a_\ell)$ en $a_j = g(\text{in}_j) = g(\sum_q W_{q,j} a_q)$, en dus:

$$\begin{aligned}\frac{\partial E}{\partial W_{k,j}} &= -\sum_i (y_i - a_i)\frac{\partial a_i}{\partial W_{k,j}} = -\sum_i (y_i - a_i) g'(\text{in}_i) W_{j,i} \frac{\partial a_j}{\partial W_{k,j}} \\ &= -\sum_i \Delta_i \cdot W_{j,i} \cdot g'(\text{in}_j) \cdot a_k = -a_k \cdot \tilde{\Delta}_j\end{aligned}$$

Hier hebben we gedefinieerd $\tilde{\Delta}_j = g'(\text{in}_j)\sum_i W_{j,i}\Delta_i$ en $\Delta_i = (y_i - a_i) g'(\text{in}_i)$, met leerregel $W_{k,j} \longleftarrow W_{k,j} + \alpha \cdot a_k \cdot \tilde{\Delta}_j$.

In diepe netwerken gebruikt men automatisch differentiëren om dit soort regels af te leiden.

Voor de sigmoide $\sigma(x) = 1/(1 + e^{-\beta x})$ geldt overigens dat $\sigma'(x) = \beta\, \sigma(x)(1 - \sigma(x))$.

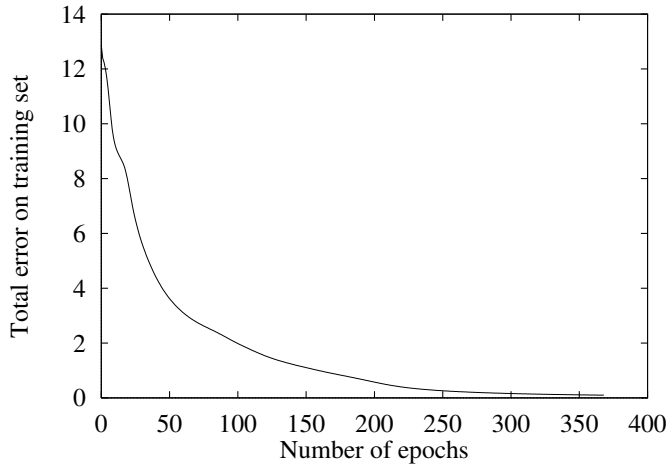
Het back-propagation-algoritme voor een netwerk met één verborgen laag gaat nu als volgt:

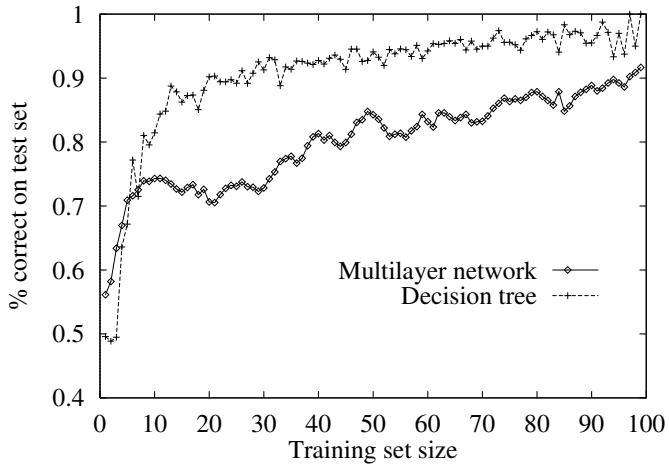
repeat
  for each e **in** trainingsset **do**
    maak de a_k's gelijk aan de x_k's
    bereken de a_j's en de a_i's (outputs)
    bereken de Δ_i's en de $\tilde{\Delta}_j$'s
    update de $W_{j,i}$'s en de $W_{k,j}$'s
until netwerk "geconvergeerd"

Let erop dat de gewichten goed random (?) geinitialiseerd worden. En bied de voorbeelden in random volgorde aan.

hint-sheet

Respectievelijk een trainings-curve en een leercurve:

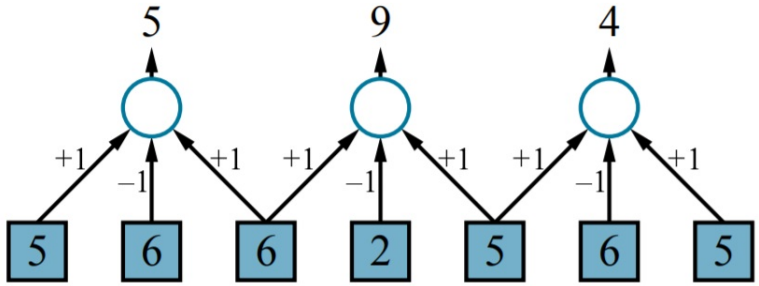




Een epoch is een ronde waarin alle (of één) voorbeelden uit de trainingsset één keer één voor één in random volgorde door het netwerk gegaan zijn. Soms heb je ∞ veel voorbeelden.

Er bestaat naast deze incrementele benadering ook een batch-benadering.

Deep learning staat volop in de belangstelling: netwerken met zeer veel lagen (50–100) van verschillende types. De architectuur is heel complex. Het werkt erg goed bij beeldherkenning (met pixels).
Men gebruikt Convolutional Neural Networks (CNN's): groepjes neuronen in een laag hebben dezelfde gewichten (dat is goed voor ruimtelijke invariantie, een oog in een plaatje ziet er overal hetzelfde uit): een "kernel".



We berekenen output z_i ($i = 1, 3, 5$) via $\sum_{j=1}^{\ell} k_j x_{j+i-(\ell+1)/2}$ uit invoer $(x_0, \ldots, x_6)$. Hier is $\ell = 3$ de grootte van de kernel $k = [+1, -1, +1]$, en stride (= stap) $s = 2$.

Vaak worden pooling layers (met vaste kernels) gebruikt. Bijvoorbeeld max pooling bepaalt het maximum van ℓ pixels.

Het werkt met speciale hardware (GPU's, TPU's) die efficiënt CNN's kan doorrekenen (de T is van "Tensor" = meer-dimensionaal array).

Zie verder Residual networks, . . .

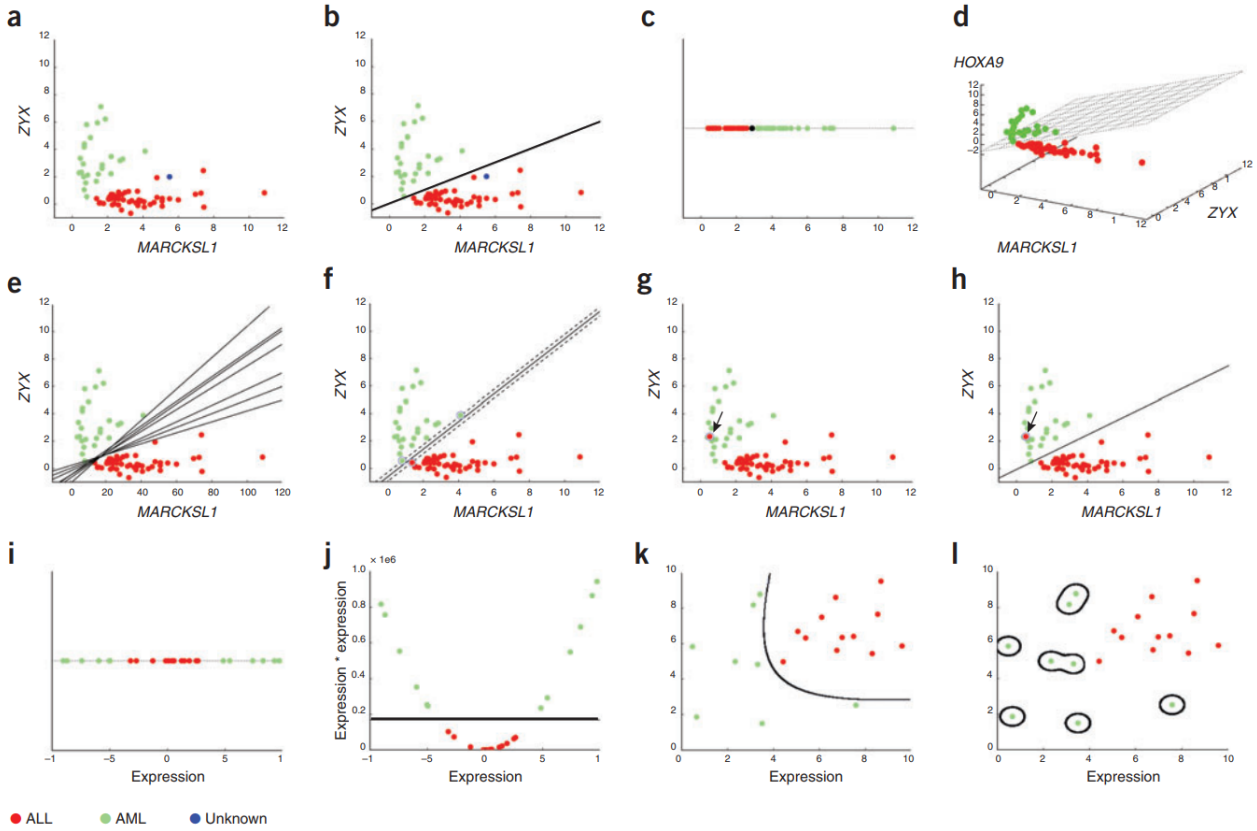
Zie ook `Keras`, TensorFlow, Theano, . . .

Nog meer soorten

Er zijn nog vele andere soorten netwerken, zoals

- Hopfield netwerken (associatief geheugen)
- Kohonen's Self Organizing Maps (SOM's)
- Support Vector Machines (SVM's), kernel machines
- Long short-term memory (LSTM) netwerken: spraakherkenning, tijdreeksen, . . .
- . . .

Een Support Vector Machine (SVM) probeert de invoerdata zo in een hoger dimensionale ruimte te leggen, dat klassen lineair te scheiden worden.



Uit: W.S. Noble, What is a Support Vector Machine? Nature Biotechnology 24, 1565–1567 (2006).

Er zijn allerlei uitbreidingen. Zo kun je regularisatie bijvoorbeeld in de vorm van weight decay toepassen (laat gewichten in de buurt van 0 verdwijnen), momentum (= impuls) toevoegen (onthoud vorige wijziging), . . . , en nu dus Deep learning.

Neurale netwerken worden overal ingezet: voor waterhoogtes, beurskoersen, Backgammon, sonarbeelden, beeldherkenning, datacompressie, . . .

Je blijft last houden van locale extremen. En de vanishing gradient?

Zie verder het mastercollege Neurale Netwerken.

Maak voor de vierde programmeeropgave een Neuraal Netwerk (met back-propagation) om eenvoudige functies te voorspellen: (X)OR en AND en . . . pokerhanden.

www.liacs.leidenuniv.nl/~kosterswa/AI/nn24.html

De eerstvolgende keer zijn we nog met Deep learning bezig. Het huiswerk voor de daaropvolgende keer (woensdag 1 mei 2024): lees Hoofdstuk 4.1, p.110–119 van [RN] door (in de derde druk p. 126–129) over het onderwerp Locaal zoeken en Optimalisatie.

Denk aan de "sommen":

www.liacs.leidenuniv.nl/~kosterswa/AI/opgaven1.pdf
www.liacs.leidenuniv.nl/~kosterswa/AI/opgaven2.pdf

Zie ook de video's met uitwerkingen op

www.liacs.leidenuniv.nl/~kosterswa/AI/

Werk aan de vierde opgave: Neurale netwerken; deadline: woensdag 15 mei 2024.

Kunstmatige Intelligentie (AI)

Hoofdstuk 4.1 van Russell/Norvig = [RN]

Locaal zoeken en Optimalisatie

voorjaar 2024

College 12, 1 mei 2024

www.liacs.leidenuniv.nl/~kosterswa/AI/genetisch.pdf

Als het pad naar de oplossing er niet toe doet (zoals bij het dames-probleem), kunnen we local search technieken gebruiken. Je verbetert dan in elke stap één of meer huidige oplossingen. Bijkomend voordeel: je hebt altijd een complete (redelijke) oplossing — robuust.

Enkele problemen hiermee:

- locale maxima
- plateau's
- "heuvelruggen" (= ridges)

We zullen in het bijzonder Genetische algoritmen bekijken.

We combineren de hill-climber (= gradient descent) met random bewegingen als volgt tot Simulated Annealing:

for $tijd \leftarrow 1$ **to** ∞ **do**
  $Temperatuur \leftarrow schedule[tijd]$
  if $Temperatuur = 0$ **then return** $huidig$
  $volgende \leftarrow$ random opvolger van $huidig$
  $\Delta E \leftarrow waarde[volgende] - waarde[huidig]$
  if $\Delta E > 0$ **then** $huidig \leftarrow volgende$
  else $huidig \leftarrow volgende$ met kans $e^{\Delta E \,/\, Temperatuur}$

Het algoritme is geinspireerd op natuurkundige ideeën en het Metropolis-algoritme uit 1953.

Er zijn allerlei soorten evolutionaire algoritmen (deel van het vakgebied natural computing, waar ook bijvoorbeeld DNA-computing onder valt, en misschien zelfs quantum computing) die ideeën uit de evolutietheorie gebruiken, zoals evolution strategies. Zie ook de speciale colleges Natural Computing en Evolutionary Algorithms.

Wij bekijken hier de wat meer traditionele Genetische algoritmen (GAs).

Genetische algoritmen (GAs) zijn een vorm van reinforcement leren, waarbij gewerkt wordt met beloningen en straffen in plaats van met de "goede antwoorden".

Een GA heeft een populatie met kandidaat-oplossingen (individuen of chromosomen) voor het betreffende probleem. Deze worden met elkaar (en zichzelf) gecombineerd tot hopelijk betere oplossingen. Essentieel is een fitness-functie die de kwaliteit van oplossingen beoordeelt.

Het wordt lastiger als het om multi-objective optimization gaat, met meerdere doelen. Je krijgt dan Pareto-fronten.

Het algemene schema van een GA ziet er zo uit:

Initialiseer populatie
Evalueer populatie
while not *klaar* **do**
    Selecteer ouders
    Genereer met crossover kinderen (recombineer)
    Muteer kinderen
    Evalueer kinderen
    Bepaal nieuwe populatie
return beste element uit populatie

Maar wanneer ben je klaar?

Er zijn vele varianten. Twee hoofdstromen zijn:

generationeel de kinderen vervangen de ouders: er komt steeds een nieuwe generatie (populatie)

steady state de nieuwe populatie bestaat uit de besten van ouders en kinderen

In het generationele geval behoudt een elitair algoritme een stel beste van de ouders. En die kun je ook weer laten verouderen: hun fitness daalt met hun leeftijd (ag(e)ing).

Maar de belangrijkste variatie zit in de genetische operatoren:

selectie hoe kies je kandidaten voor reproductie?

crossover hoe combineer je ouders?

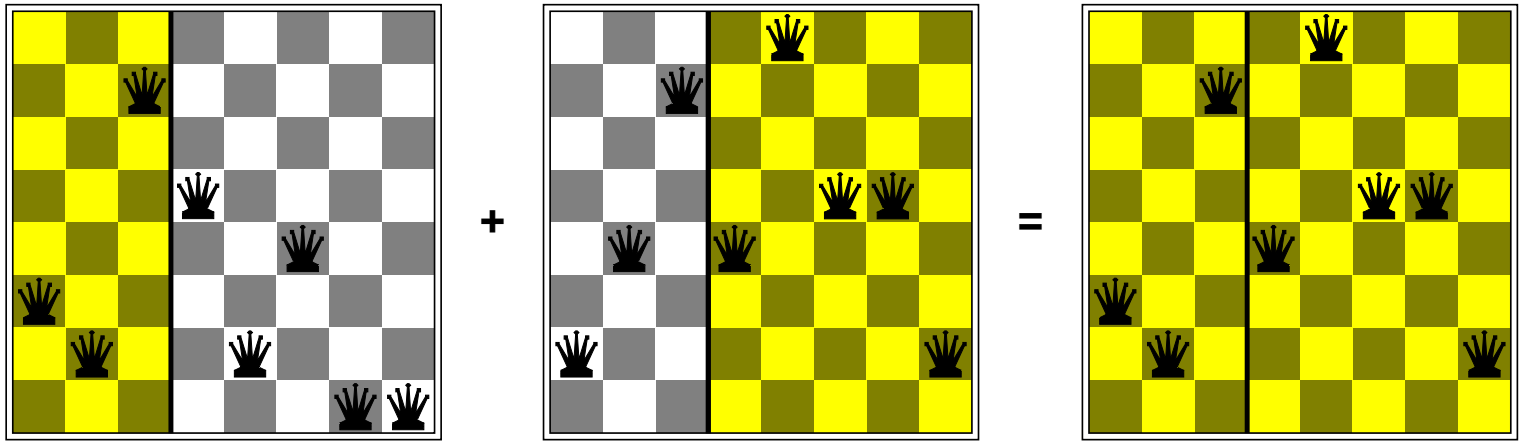
mutatie hoe wijzig je individuen op een zinvolle manier?

Vaak wordt selectie met een roulette-wiel gedaan. De kans dat een individu gebruikt wordt (bijvoorbeeld voor reproductie) is evenredig met zijn/haar fitness.

Heb je n individuen met gemiddelde fitness $\bar{f}$, dan wordt een individu met fitness f getrokken met kans $f/(n * \bar{f})$. Je draait aan een roulette-wiel waarvan de grootte van de taartpunten proportioneel is met de fitness.

Een alternatief is rank-based selectie, waar alleen de volgorde van de fitness-waarden bepalend is.

Bij crossover wil je twee (of meer) ouders combineren. Voor de hand ligt de eerste helft van de ene ouder en de tweede helft van de andere ouder te nemen:



Wat gaat hier fout als je langs de rijen knipt?

De volgende soorten crossover worden vaak gebruikt:

single-point crossover (als hiervoor)

$$\left.\begin{array}{l}\underline{11101}001000\\ 00001\underline{010101}\end{array}\right\} \Rightarrow \begin{array}{l}11101010101\\ 00001001000\end{array}$$

two-point crossover

$$\left.\begin{array}{l}\underline{11}101001\underline{000}\\ 00\underline{00101}0101\end{array}\right\} \Rightarrow \begin{array}{l}11001011000\\ 00101000101\end{array}$$

uniform crossover

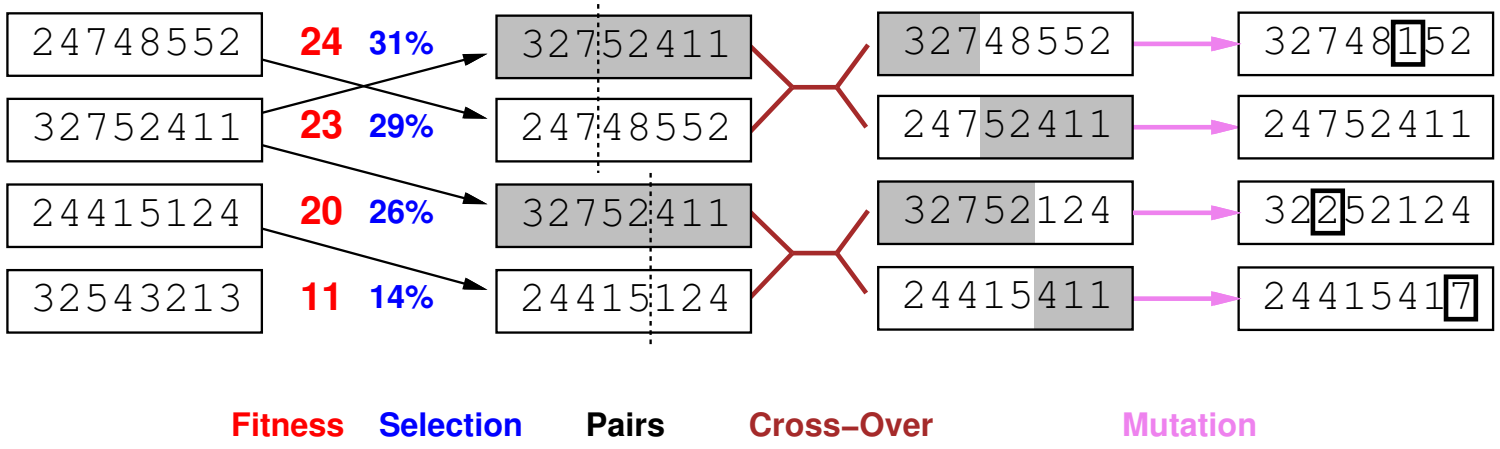
$$\left.\begin{array}{l}\underline{1}11\underline{0}1\underline{0}0\underline{0}\\ 0\underline{00}0\underline{1}\underline{0}\underline{1}01\end{array}\right\} \Rightarrow \begin{array}{l}10001000100\\ 01101011001\end{array}$$

Bij mutatie wordt elke bit van een individu met een zekere (kleine) kans, de mutation rate, omgeklapt. Een andere optie is om van elk individu een random bit om te klappen.

Meestal wordt mutatie (en ook crossover) speciaal gebouwd voor een probleem. Je kunt bijvoorbeeld ook random bits omwisselen, en dan het aantal enen constant houden — als dat zinvol lijkt.

En er zijn ook nog reparatie-operatoren, die een willekeurige bitstring ombouwen naar een potentiële oplossing, bijvoorbeeld met een "hill-climber".

Een voorbeeld-stap in een GA is:



Hier hebben we overigens geen bits, maar getallen in de individuen. Dit kun je zien als genen. De string heet soms wel het genotype, dat wat hij voorstelt het phenotype.

Resteren nog twee problemen: de representatie (hoe stop je kandidaat-oplossingen in een string?) en de fitness- of evaluatie-functie (hoe waarderen we die individuen?).

Soms is het zo dat iedere string kan voorkomen, soms niet. Soms zijn reparatie-operatoren nodig, soms niet.

Meestal is de fitness-functie de som van een aantal componenten. Harde constraints geven grote (of juist lage) bijdragen. Er kunnen zoals gezegd zelfs meerdere fitness-functies gebruikt worden: "multi-objective optimization".

Een beroemd probleem is het Traveling Salesperson/man Problem (TSP): in een gegeven complete graaf G, met gewichten = afstanden op de takken, moeten we een gesloten pad met minimale lengte vinden dat alle knopen aandoet.

Je kunt bijvoorbeeld de volgende pad-representatie gebruiken: (5 1 7 8 9 4 6 2 3) betekent dat je van 5 naar 1 naar 7 naar . . . naar 3 naar 5 gaat.

En de fitness-functie is de totale lengte van het pad.

Er zijn vele mogelijkheden voor crossover bij TSP, zoals:

partially mapped crossover (PMX)

$$\left.\begin{array}{l}(1\ 2\ 3\,|\,4\ 5\ 6\ 7\,|\,8\ 9)\\(4\ 5\ 2\,|\,1\ 8\ 7\ 6\,|\,9\ 3)\end{array}\right\} \Rightarrow \begin{array}{l}(4\ 2\ 3\,|\,1\ 8\ 7\ 6\,|\,5\ 9)\\(1\ 8\ 2\,|\,4\ 5\ 6\ 7\,|\,9\ 3)\end{array}$$

order crossover (OX)

$$\left.\begin{array}{l}(1\ 2\ 3\,|\,4\ 5\ 6\ 7\,|\,8\ 9)\\(4\ 5\ 2\,|\,1\ 8\ 7\ 6\,|\,9\ 3)\end{array}\right\} \Rightarrow \begin{array}{l}(2\ 1\ 8\,|\,4\ 5\ 6\ 7\,|\,9\ 3)\\(3\ 4\ 5\,|\,1\ 8\ 7\ 6\,|\,9\ 2)\end{array}$$

cycle crossover (CX)

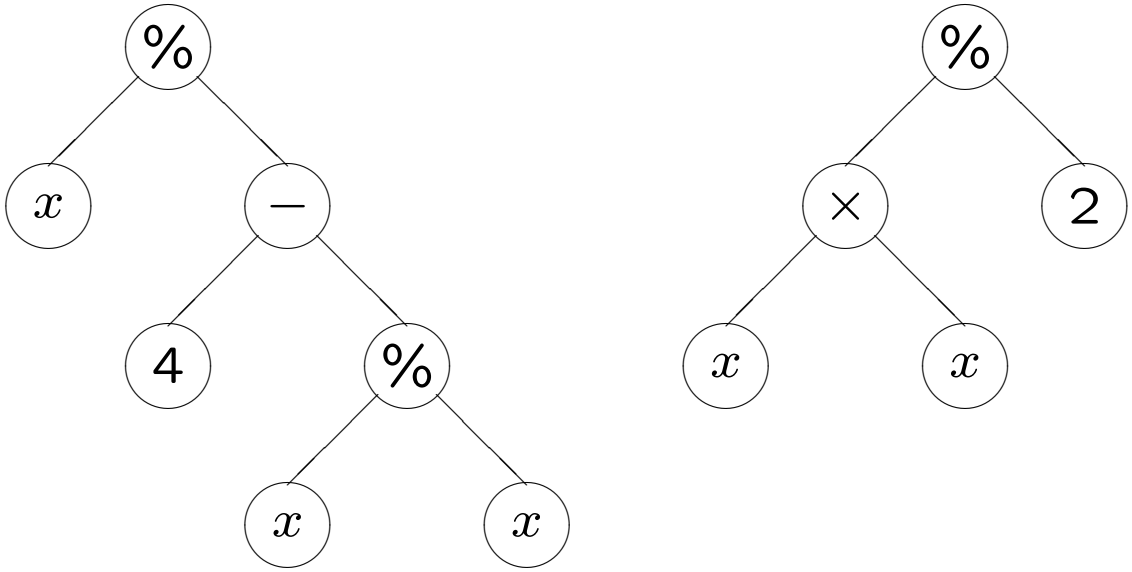
$$\left.\begin{array}{l}(1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9)\\(4\ 1\ 2\ 8\ 7\ 6\ 9\ 3\ 5)\end{array}\right\} \Rightarrow \begin{array}{l}(1\ 2\ 3\ 4\ 7\ 6\ 9\ 8\ 5)\\(4\ 1\ 2\ 8\ 5\ 6\ 7\ 3\ 9)\end{array}$$

In Genetic Programming (GP; John Koza, 1992) manipuleer je bomen (oftewel programma's) in plaats van strings.

Een voorbeeld. Stel je wilt de functie $f(x) = x^2/2$, gegeven door 10 paren (0.000,0.000), (0.100,0.005), ..., (0.900,0.405), maken via bomen met knopen x, $+$, $-$, $\times$, % ("protected division"), -5, -4, ..., 4, 5.

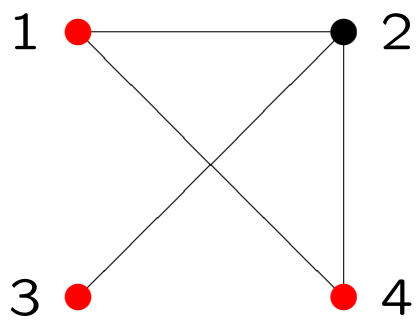
We kunnen een populatiegrootte van 600 nemen, crossoverkans 90%, mutatiekans 5%, en toernooi-selectie (kies er steeds random 4, en neem daarvan de beste; die 4 geeft "selection pressure"). Crossover: wissel twee willekeurige sub-bomen, mutatie: vervang willekeurige sub-boom door een random-boom.

Tijdens generatie 0 heb je bijvoorbeeld de linkerboom, die de functie $f_0(x) = x/3$ voorstelt. Tijdens generatie 3 zou je de boom rechts kunnen krijgen, die $f_3(x) = x^2/2$ voorstelt.



Stel je moet een Genetisch algoritme maken dat zo goed mogelijk een ongerichte graaf zodanig inkleurt dat zo weinig mogelijk aangrenzende knopen dezelfde kleur hebben, en dat nummers van buren met dezelfde kleur zo veel mogelijk verschillen. Hoe kies je representatie en operatoren? Invoer (het probleem, links) en mogelijke uitvoer:

```
4 2
0 1 0 1
1 0 1 1
0 1 0 0
1 1 0 0
```

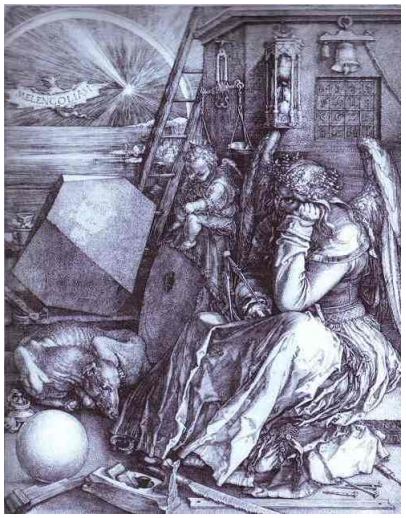


Dat betekent 4 knopen, 2 kleuren.

We moeten nu een Genetisch algoritme maken dat probeert een $n \times n$ magisch vierkant te fabriceren. Dit bestaat uit een vierkante opstelling van de n^2 positieve gehele getallen $1, 2, 3, \ldots, n^2$, waarbij alle rijen en kolommen (en wellicht ook nog andere combinaties) sommeren tot hetzelfde getal.

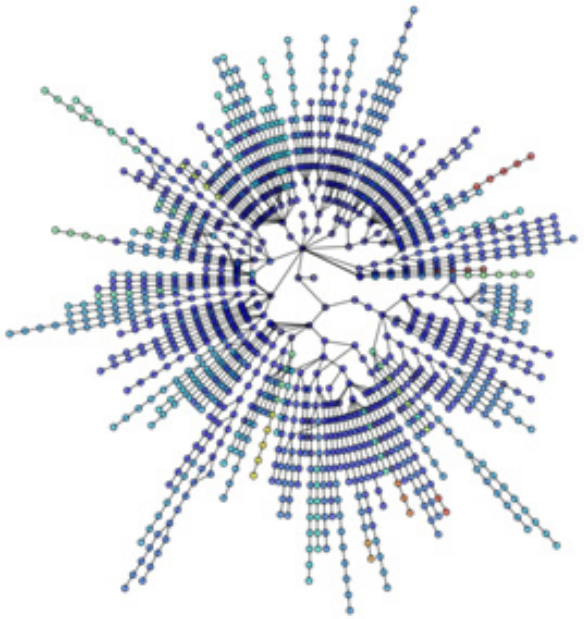
8	1	6
3	5	7
4	9	2

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1



Hoe doe je dat?

En nog een voorbeeld: maak een Genetisch algoritme dat probeert een graaf zo goed mogelijk te tekenen: zo weinig mogelijk snijdende takken, en afstanden lijkend op de “echte”.



Hoe maak je een Genetisch algoritme dat probeert een 4×4 en een 9×9 Sudoku op te lossen?

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

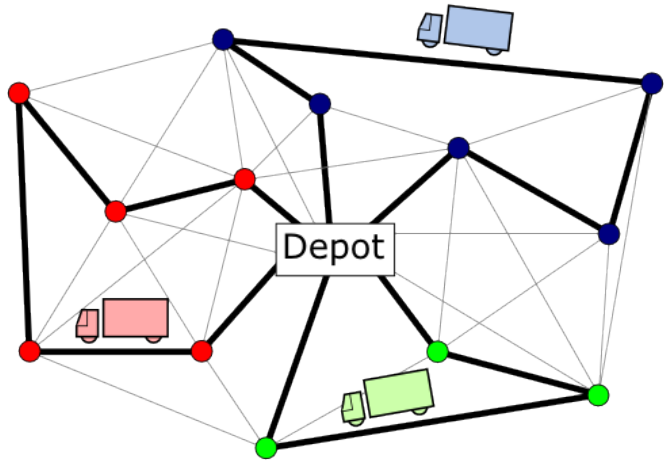
rijen, kolommen
en 3×3 "blokken"
bevatten $1, 2, \ldots, 9$

Hoe maak je een Genetisch algoritme dat probeert een 4×4 en een 9×9 Sudoku op te lossen?

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

bron: Wikipedia

Ontwerp een Genetisch algoritme dat zo goed mogelijk probeert een routerings-probleem op te lossen.



Mutatie? Crossover?

Multi-objective optimization? Pareto-front?

Het huiswerk voor de volgende keer (woensdag 8 mei 2024): lees Hoofdstuk 13 en 14, p. 385–402 en p. 412–436 van [RN] door (in de derde druk p. 480–499 en p. 510–529) over het onderwerp Onzekerheid en Bayesiaanse netwerken.

Tijdens het werkcollege van 2 mei worden opgaven gemaakt, in het bijzonder 10, 19, 23, 27 en 29 van:

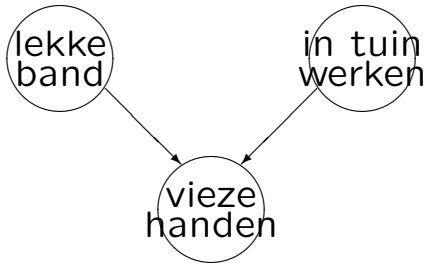
www.liacs.leidenuniv.nl/~kosterswa/AI/opgaven1.pdf
www.liacs.leidenuniv.nl/~kosterswa/AI/opgaven2.pdf

Denk na over de vierde opgave: Neurale netwerken; deadline: woensdag 15 mei 2024.

Kunstmatige Intelligentie (AI)

Hoofdstuk 12 en 13 van Russell/Norvig = [RN]
Bayesiaanse netwerken

voorjaar 2023
College 13, 8 mei 2024



www.liacs.leidenuniv.nl/~kosterswa/AI/bayesnet.pdf

We gaan nu eenvoudige "systemen" bekijken waarin kansen een rol spelen, in het bijzonder Bayesiaanse netwerken waarin vele (on)afhankelijkheden gelden.

Basisbouwsteen is de uit de kansrekening bekende regel van Bayes.

We gebruiken dus veelvuldig de regel van Bayes (te bewijzen via $P(A,B) = P(A \text{ en } B) = P(A \wedge B) = P(A)P(B|A)$):

$$P(B|A) \stackrel{\text{def}}{=} \text{kans op } B \text{ gegeven } A = \frac{P(A|B)P(B)}{P(A)}$$

Een eenvoudige toepassing is als volgt. Stel, met $S \stackrel{\text{def}}{=}$ stijve nek en $M \stackrel{\text{def}}{=}$ meningitis:

$$P(S|M) = 0.5 \quad P(M) = 1/50000 \quad P(S) = 1/20$$

(die laatste twee zijn de "prior probability"). Dan:

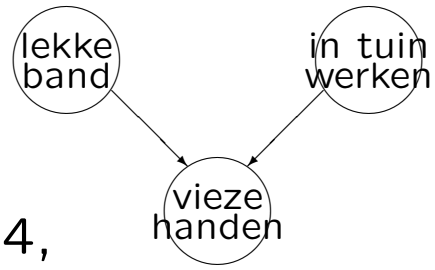
$$P(M|S) = \frac{P(S|M)P(M)}{P(S)} = \frac{0.5 \times 1/50000}{1/20} = 0.0002 \ .$$

Diagnostische kennis $P(M|S)$ gedraagt zich minder stabiel dan causale kennis $P(S|M)$ (denk maar aan M-epidemie!).
NB: als A en B onafhankelijk zijn: $P(A,B) = P(A)P(B)$.

Voorbeeld Bayes

lekke band = `true`

$\Downarrow$



oorzaak

$\Downarrow$?

gevolg

Stel dat vandaag geldt $P(\text{lek}) = 0.1$
(en dus $P(\neg\text{lek}) = 0.9$), $P(\text{tuin}) = 0.4$,
$P(\text{vies} \mid \text{lek}, \text{tuin}) = 0.8$, $P(\text{vies} \mid \neg\text{lek}, \text{tuin}) = 0.7$,
$P(\text{vies} \mid \text{lek}, \neg\text{tuin}) = 0.4$ en $P(\text{vies} \mid \neg\text{lek}, \neg\text{tuin}) = 0.0$.

Bereken $P(\text{vies} \mid \text{tuin}) = P(\text{vies} \mid \text{tuin}, \text{lek})P(\text{lek}) + P(\text{vies} \mid \text{tuin}, \neg\text{lek})P(\neg\text{lek}) =$
$= 0.8 \cdot 0.1 + 0.7 \cdot 0.9 = 0.71$, en dan met Bayes:

$$P(\text{tuin} \mid \text{vies}) = P(\text{vies} \mid \text{tuin})P(\text{tuin}) \,/\, P(\text{vies}) \approx 0.71 \cdot 0.4 \,/\, 0.3 \approx 0.9.$$

Gebruik: $P(\text{vies}) = P(\text{vies} \mid \text{lek}, \text{tuin})P(\text{lek})P(\text{tuin}) + P(\text{vies} \mid \neg\text{lek}, \text{tuin})P(\neg\text{lek})P(\text{tuin})$
$+ P(\text{vies} \mid \text{lek}, \neg\text{tuin})P(\text{lek})P(\neg\text{tuin}) + P(\text{vies} \mid \neg\text{lek}, \neg\text{tuin})P(\neg\text{lek})P(\neg\text{tuin}) \approx 0.3.$

Ook interessant: $P(\text{lek} \mid \text{vies}, \text{tuin})$. En een pijl van "lekke band" naar "in tuin werken"? En is het een "Noisy-OR"?

Met $W \stackrel{\text{def}}{=}$ whiplash kunnen we de relatieve "likelihood" van meningitis en whiplash, gegeven een stijve nek, berekenen:

$$\frac{P(M|S)}{P(W|S)} = \frac{P(S|M)P(M)}{P(S|W)P(W)} = \frac{0.5 \times 1/50000}{0.8 \times 1/1000} = \frac{1}{80} \ ,$$

wetende dat $P(S|W) = 0.8$ en $P(W) = 1/1000$.

En soms hoef je niet alles te weten:

$$P(M|S) = \alpha P(S|M)P(M) \quad P(\neg M|S) = \alpha P(S|\neg M)P(\neg M) \ ,$$

terwijl $P(M|S)+P(\neg M|S) = 1$: normalisatie, met uiteraard

$$\alpha = 1/P(S) = 1/\{P(S|M)P(M) + P(S|\neg M)P(\neg M)\} \ .$$

Bij de tandarts geldt, met $G \stackrel{\text{def}}{=}$ gaatje; $K \stackrel{\text{def}}{=}$ kiespijn:

$$P(G|K) = P(G)\ \frac{P(K|G)}{P(K)}\ .$$

En met $H \stackrel{\text{def}}{=}$ "haakje blijft hangen" erbij:

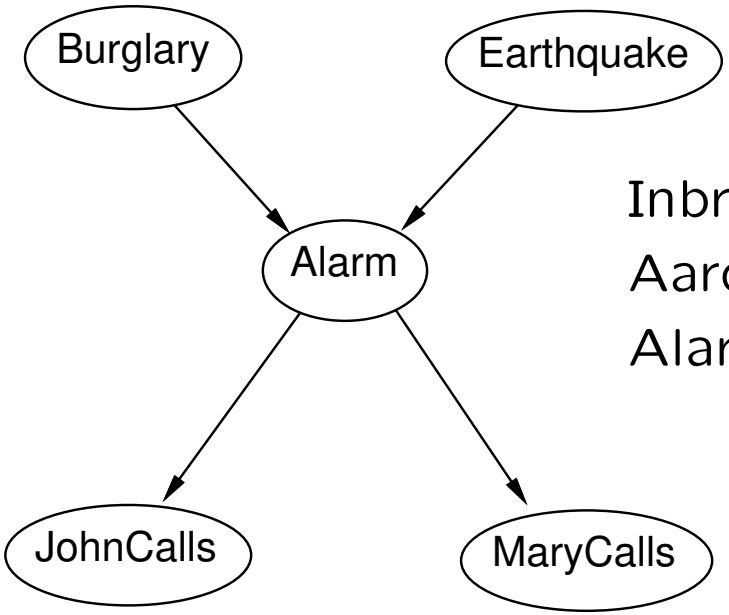
$$P(G|K \wedge H) = P(G|K)\ \frac{P(H|K \wedge G)}{P(H|K)} = P(G)\ \frac{P(K|G)}{P(K)}\ \frac{P(H|K \wedge G)}{P(H|K)}\ .$$

Een redelijke *aanname* (voorwaardelijke onafhankelijkheid) zoals $P(H|K \wedge G) = P(H|G)$ (of equivalent: $P(K|G \wedge H) = P(K|G)$) maakt het Bayesiaans updaten wat "eenvoudiger" voor de tandarts:

$$P(G|K \wedge H) = P(G)\ \frac{P(K|G)}{P(K)}\ \frac{P(H|G)}{P(H|K)}\ ,$$

waarbij je de noemers nog kunt "weg-normaliseren".

Ons standaardvoorbeeld van een Bayesiaans netwerk (oftewel probabilistisch netwerk = belief netwerk = knowledge map), afkomstig van Judea Pearl, is:

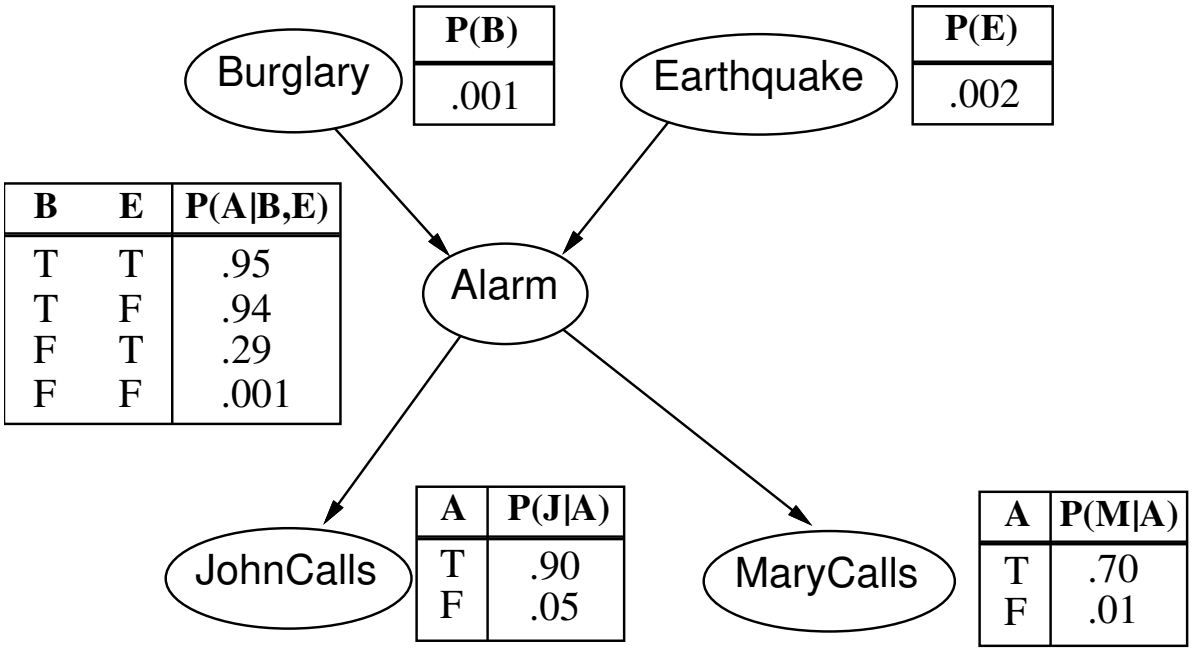


Inbreker kan Alarm af laten gaan
Aardbeving kan Alarm af laten gaan
Alarm kan Mary en/of John
aanzetten tot bellen

Een Bayesiaans netwerk is een graaf zonder cykels met:
1. random variabelen als knopen,
2. pijlen tussen knopen (die directe invloed aangeven),
3. conditional probability tables (CPT's) die het gezamenlijk effect van al hun ouders aangeven op de kinderen.

In ons voorbeeld is er *geen directe invloed* van Earthquake op MaryCalls — en dus geen pijl (andersom ook niet). Die invloed is er op zich wel, maar loopt geheel via Alarm: $P(M|A,E)(= P(M|A \wedge E)) = P(M|A)$: MaryCalls is voorwaardelijk onafhankelijk van Earthquake, gegeven Alarm. Als Mary ook rechtstreeks zou reageren op Earthquake, moest er een pijl bij.

Opnieuw ons Bayesiaanse netwerk, nu met CPT's:



Let er op dat de pijlen als het ware van oorzaak naar gevolg lopen. We hebben 10 getallen nodig om dit systeem te beschrijven, in plaats van $32 - 1 = 31$.

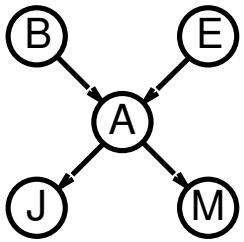
In een Bayesiaans netwerk moet altijd gelden dat

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i \,|\, \text{Parents}(X_i)) \ ,$$

als we definiëren $P(x_1, \ldots, x_n) = P(X_1 = x_1, \ldots, X_n = x_n)$ en Parents(X_i) = de verzameling van knopen die een pijl naar X_i hebben.

Een voorbeeld (met j = (JohnCalls = true), . . .):

$$\begin{aligned} P(j \wedge m \wedge a \wedge \neg b \wedge \neg e) &= P(j|a)P(m|a)P(a|\neg b \wedge \neg e)P(\neg b)P(\neg e) \\ &= 0.90 \times 0.70 \times 0.001 \times 0.999 \times 0.998 \\ &= 0.00063 \ . \end{aligned}$$



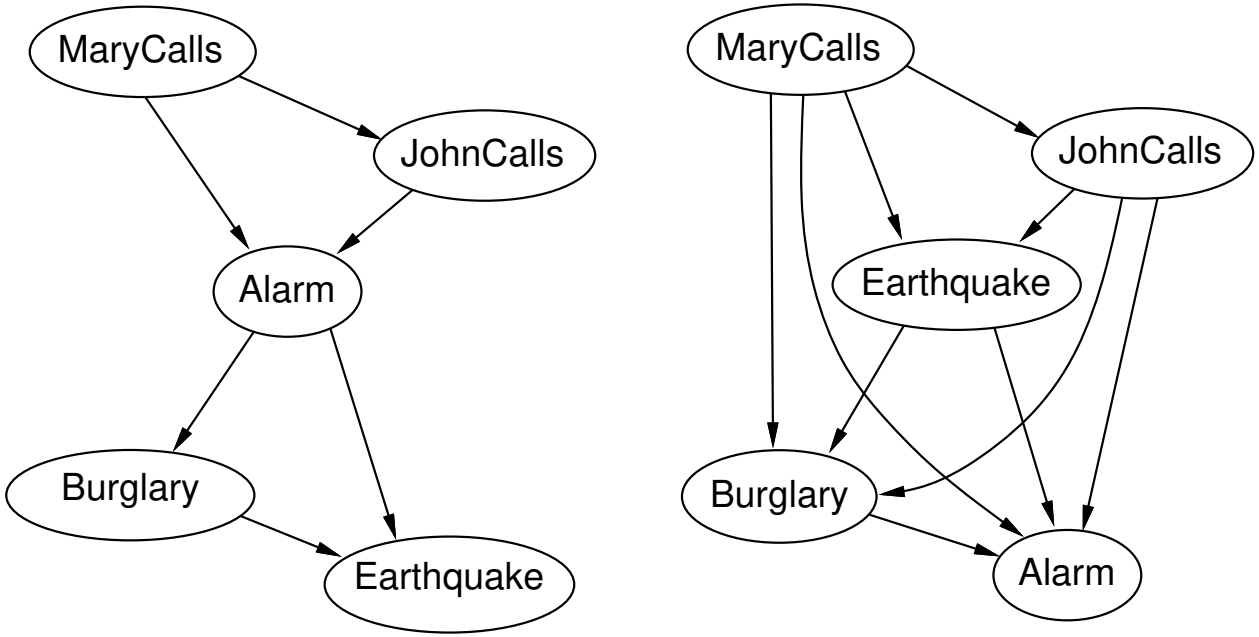
Hoe construeer je zo'n Bayesiaans netwerk?

1. Kies relevante variabelen die het domein beschrijven.
2. Orden ze "verstandig": $X_1,\ldots,X_n$.
3. Kies zo lang het kan de volgende overgebleven variabele X_i, maak daarvoor een knoop in het netwerk, en maak $\mathsf{Parents}(X_i) \subseteq \{X_{i-1},\ldots,X_1\}$ zo klein mogelijk met

$$P(X_i \,|\, X_{i-1},\ldots,X_1) = P(X_i \,|\, \mathsf{Parents}(X_i)) \ ,$$

de voorwaardelijke onafhankelijkheids eigenschap; definieer tot slot de CPT (conditional probability table) voor X_i.

De volgorde maakt veel uit voor het uiteindelijke netwerk:



Volgorde van toevoegen links: M, J, A, B, E, rechts: M, J, E, B, A (met 31 "parameters"!). Begin dus liever met oorzaken . . .

Grootte CPT's

Algemeen: neem aan dat elke Booleaanse variabele rechtstreeks wordt beinvloed door maximaal k andere, dan heb je — als er n knopen zijn — aan $n \cdot 2^k$ getallen voldoende voor de CPT's.
De volledige "joint" gebruikt er $2^n - 1$ (-1 omdat de getallen tot 1 sommeren).

Met $n = 20$ en $k = 5$ is dat 640 respectievelijk 1 miljoen.

Als je *meer* weet kun je ze soms efficiënter opslaan, zeker in het geval van deterministische knopen.

In "gewone" logica geldt:

$$\text{Koorts} \Leftrightarrow \text{Verkouden} \vee \text{Griep} \vee \text{Malaria}$$

Neem nu aan dat:
(1) elke oorzaak heeft een onafhankelijke kans het effect Koorts te veroorzaken;
(2) alle oorzaken zijn gegeven (voeg eventueel een "leak node" toe);
(3) dat wat (bijvoorbeeld) Verkouden ervan weerhoudt Koorts te veroorzaken is onafhankelijk van dat wat Griep verbiedt Koorts te veroorzaken.

Samen heet dit wel een Noisy-OR relatie.

We krijgen dan een volgende tabel:

Verkouden	Griep	Malaria	P(Koorts)	P(¬Koorts)
F	F	F	0.0 (2)	1.0
F	F	T	0.9 (1)	0.1
F	T	F	0.8 (1)	0.2
F	T	T	0.98	$0.02 = 0.2 \cdot 0.1$ (3)
T	F	F	0.4 (1)	0.6
T	F	T	0.94	$0.06 = 0.6 \cdot 0.1$ (3)
T	T	F	0.88	$0.12 = 0.6 \cdot 0.2$ (3)
T	T	T	0.988	$0.012 = 0.6 \cdot 0.2 \cdot 0.1$ (3)

Alleen de drie rode getallen hoef je te onthouden, de rest volgt hier uit.

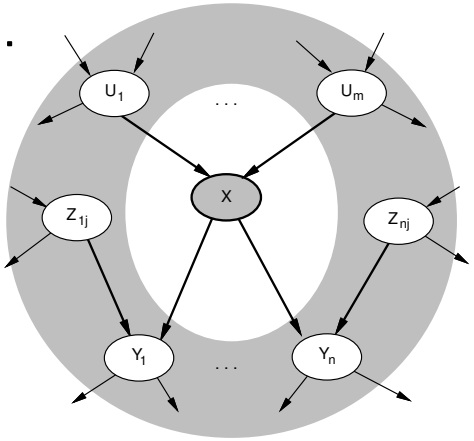
Bij de Naive Bayes classifier (liever: model; zie Data mining) neem je iets soortgelijks aan.

In ons Tennis-voorbeeld zou je kunnen veronderstellen:

$$\begin{aligned} &P(\text{Weer} = \text{zonnig} \wedge \text{Temp} = \text{koud} \mid \text{Tennis} = \text{Ja}) = \\ &\quad P(\text{Weer} = \text{zonnig} \mid \text{Tennis} = \text{Ja}) \\ &\quad \times P(\text{Temp} = \text{koud} \mid \text{Tennis} = \text{Ja}) \end{aligned}$$

Hiermee kun je "eenvoudig" classificeren. Als namelijk het weer zonnig, temperatuur koud, vochtigheid hoog en wind sterk is, kun je een hogere kans voor het effect Tennis = Nee (benaderend) uitrekenen dan voor Tennis = Ja.

Het berekenen van voorwaardelijke kansen in een Bayesiaans netwerk (inferentie) is in het algemeen niet eenvoudig. We willen "P(Query | Evidence)" weten: we hebben informatie over zekere "Evidence" variabelen, en zijn geinteresseerd in zekere "Query" variabelen.



In het algemeen heb je te maken met de zogeheten Markov blanket: ouders, kinderen en co-ouders van kinderen.

Er worden vier soorten inferentie onderscheiden:

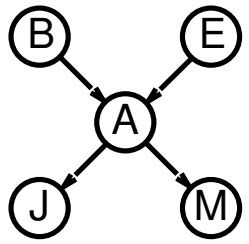
diagnostisch van effect naar oorzaak: $P(b|j) = 0.016$

causaal van oorzaak naar effect (met de pijlen mee):
$P(j|b) = 0.86$
(via $P(j|b) = P(j|a)P(a|b) + P(j|\neg a)P(\neg a|b)$ en
$P(a|b) = P(a|b,e)P(e) + P(a|b,\neg e)P(\neg e)$)

intercausaal ("explaining away") tussen oorzaken van gemeenschappelijk effect: $P(b|a \wedge e) = 0.003$

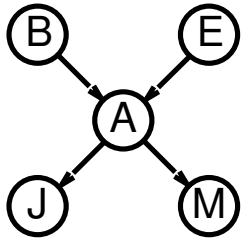
mixed overig: $P(a|j \wedge \neg e) = 0.03$

Waarbij weer j = (JohnCalls = true), ...



We zullen als voorbeeld van een diagnostische afleiding $P(b|j)$ berekenen:

Stap 1: $P(b|j) = P(b|a)P(a|j) + P(b|\neg a)P(\neg a|j)$
Stap 2: $P(b|a) = P(a|b)P(b)/P(a)$ (Bayes)
Stap 3: $P(a) = P(a|b,e)P(b)P(e) + P(a|\neg b,e)P(\neg b)P(e) +$
$P(a|b,\neg e)P(b)P(\neg e) + P(a|\neg b,\neg e)P(\neg b)P(\neg e) = 0.0025$
Stap 4: $P(a|b) = P(a|b,e)P(e) + P(a|b,\neg e)P(\neg e) = 0.94$
Stap 5: $P(b|a) = 0.376$ en $P(b|\neg a) = 0.00006$
Stap 6: $P(j) = P(j|a)P(a) + P(j|\neg a)P(\neg a) = 0.052$
Stap 7: $P(a|j) = P(j|a)P(a)/P(j) = 0.043$ en
$P(\neg a|j) = 0.957$
Stap 8: $P(b|j) = 0.016$



We kunnen $P(b|j)$ ook op een andere manier berekenen.

Omdat $P(b|j) = P(b,j)/P(j)$ en $P(\neg b|j) = P(\neg b,j)/P(j)$ is het voldoende $P(b,j)$ en $P(\neg b,j)$ te bepalen, en daarna te "normaliseren" — of $P(j)$ te gebruiken.

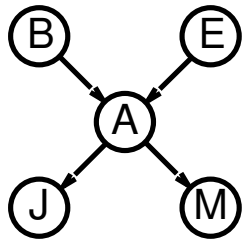
$$P(b,j) = P(b)\{P(e)\{P(a|b,e)P(j|a) + P(\neg a|b,e)P(j|\neg a)\} + P(\neg e)\{P(a|b,\neg e)P(j|a) + P(\neg a|b,\neg e)P(j|\neg a)\}\} ,$$

wat na invullen gelijk blijkt aan 0.00085.

Analoog: $P(\neg b,j) = \ldots = 0.05129$.

En dus $P(j) = P(b,j) + P(\neg b,j) = 0.05214$.

Tot slot: $P(b|j) = 0.00085/0.05214 = 0.016$.

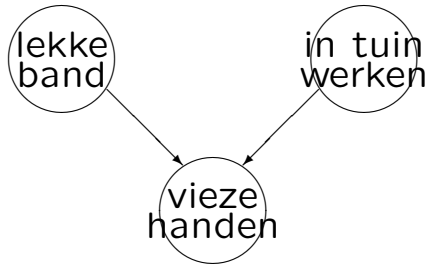


Het is blijkbaar lastig! Enkele vuistregels:

- werk toe naar "causaal"
- gebruik de uitgebreide regel van Bayes, met overal $|C$:

$$P(A|B,C) = P(B|A,C)P(A|C) \,/\, P(B|C)$$

Voorbeeld:



$$P(\text{lek} \,|\, \text{vies}, \text{tuin}) = \frac{P(\text{vies} \,|\, \text{lek}, \text{tuin})P(\text{lek} \,|\, \text{tuin})}{P(\text{vies} \,|\, \text{tuin})}$$

waarbij $P(\text{lek} \,|\, \text{tuin}) = P(\text{lek})$ en
$P(\text{vies} \,|\, \text{tuin}) = P(\text{vies} \,|\, \text{tuin}, \text{lek})P(\text{lek}) + P(\text{vies} \,|\, \text{tuin}, \neg\text{lek})P(\neg\text{lek})$

In netwerken waarbij knopen met meerdere paden verbonden zijn, is exacte inferentie nog moeilijker. In het algemeen is inferentie in een Bayesiaans netwerk zelfs NP-volledig!

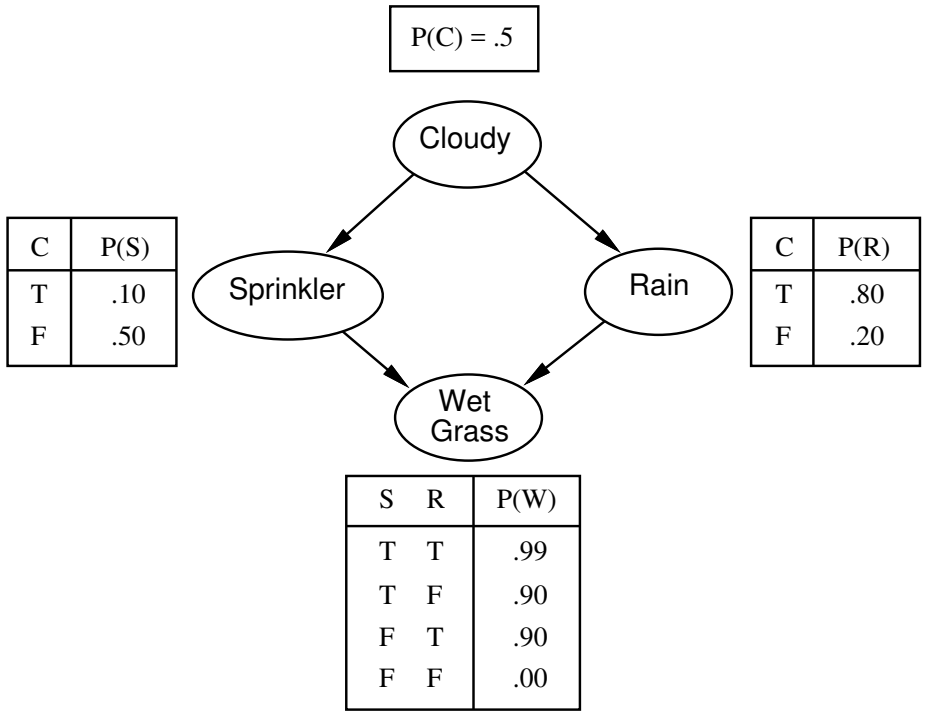
Je kunt verschillende methoden gebruiken:

clustering = join tree methoden: bouw het netwerk om naar één zonder verschillende paden: een "polytree"

conditioning methoden: maak kopieën van het netwerk door vaste waardes te kiezen voor lastige variabelen

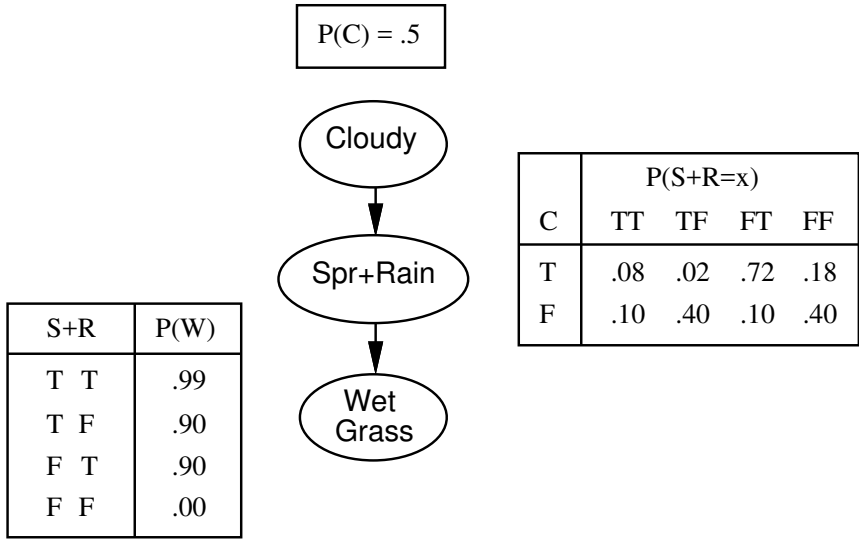
sampling methoden: doe simulaties om een benaderend antwoord te krijgen — veel statistiek dus

We bekijken het volgende "multiply connected" netwerk:



Hier zijn 9 getallen nodig. De onderste knoop voldoet overigens aan de "Noisy-OR" relatie.

Bij de join tree methode voegen we nu de knopen Sprinkler en Rain samen tot één mega-knoop met een grote CPT, en krijgen dan een "polytree":



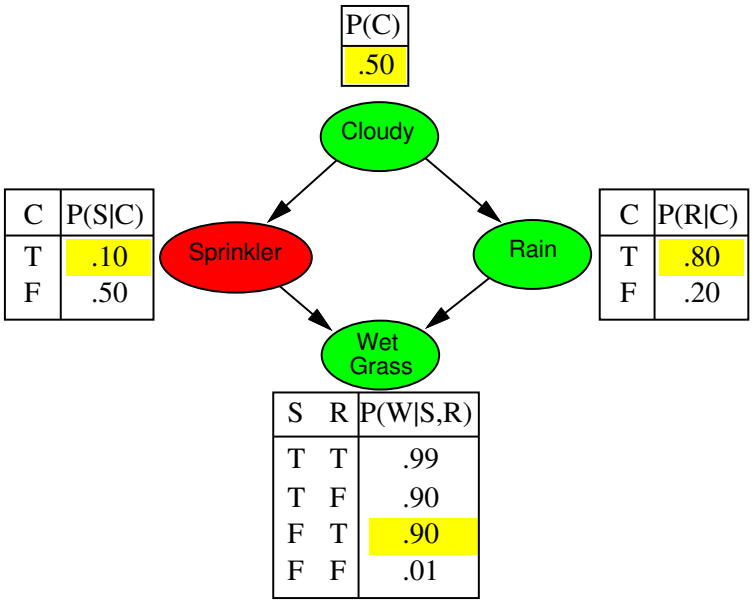
C	P(S+R=x) TT	TF	FT	FF
T	.08	.02	.72	.18
F	.10	.40	.10	.40

S+R	P(W)
T T	.99
T F	.90
F T	.90
F F	.00

Hier zijn 11 getallen nodig (de 12e en 13e volgen hieruit). In het algemeen explodeert dit aantal.

Je kunt ook (conditioning) twee netwerken maken, één voor Cloudy = true en één voor Cloudy = false: "polytrees" gelabeld met kans 0.5. Cloudy vormt de "cutset".

Of gaan "samplen":



In de laatste week, op woensdag 15 mei 2024, kijken we naar het geheel en naar een oud tentamen: 7 juni 2023. Zie ook het oude tentamen van 2 juni 2014, met (video's van) uitwerkingen. Er is dus geen college meer op 22 mei, en geen werkcollege op 23 mei.

Denk aan de opgaven:

www.liacs.leidenuniv.nl/~kosterswa/AI/opgaven2.pdf

Maak in het bijzonder 30, 31, 33, 13 en 14, tijdens het werkcollege van 16 mei 2024.

Het tentamen is op donderdag 13 juni 2024, 9:00–12.00 uur, in het Sportcentrum; het hertentamen is op maandag 8 juli 2024, 9.00–12.00 uur. Vergeet niet je aan te melden!

Zie verder www.liacs.leidenuniv.nl/~kosterswa/AI/