Kunstmatige Intelligentie (AI)

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

Probleemoplossen en zoeken

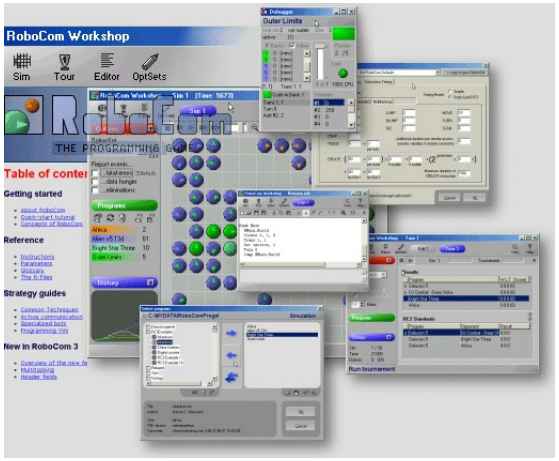
voorjaar 2021

College 4, 22 februari 2021

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

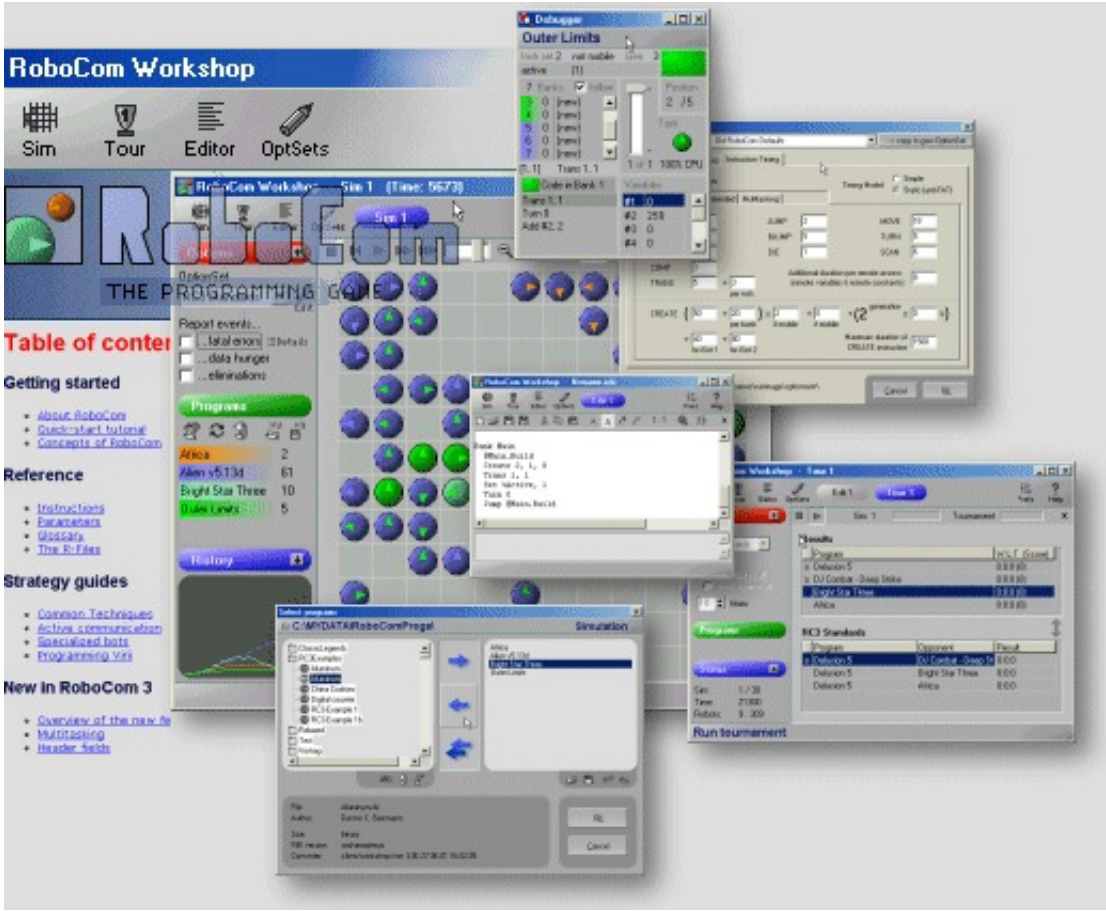
Er zijn allerlei robot-simulaties.
We gebruiken RoboCom, een opvolger van "CoreWar", van Dennis Bemmann: kleine robot-programma's vechten in een vierkant stuk computergeheugen met 18×18 vakjes (= velden). Vergelijk: "Robocode".

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/robot2021.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 robot:

coöperatief Klonen/kinderen `X` en `Y` van twee dezelfde robots vormen samen na enige tijd één of twee "vierkanten":

```
X X
Y Y
```

Hoe vaak werkt het, wanneer en waarom? Wat gebeurt er bij drie in plaats van twee? PEAS? De "6"?

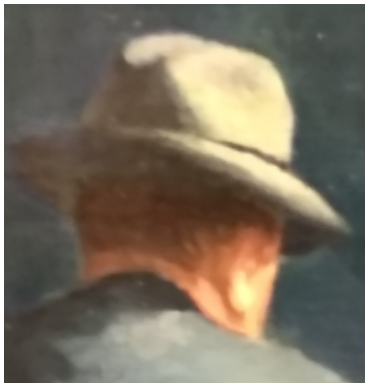
www.liacs.leidenuniv.nl/~kosterswa/AI/robot2021.html

RoboCom — Voorbeeld

```
; 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 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.

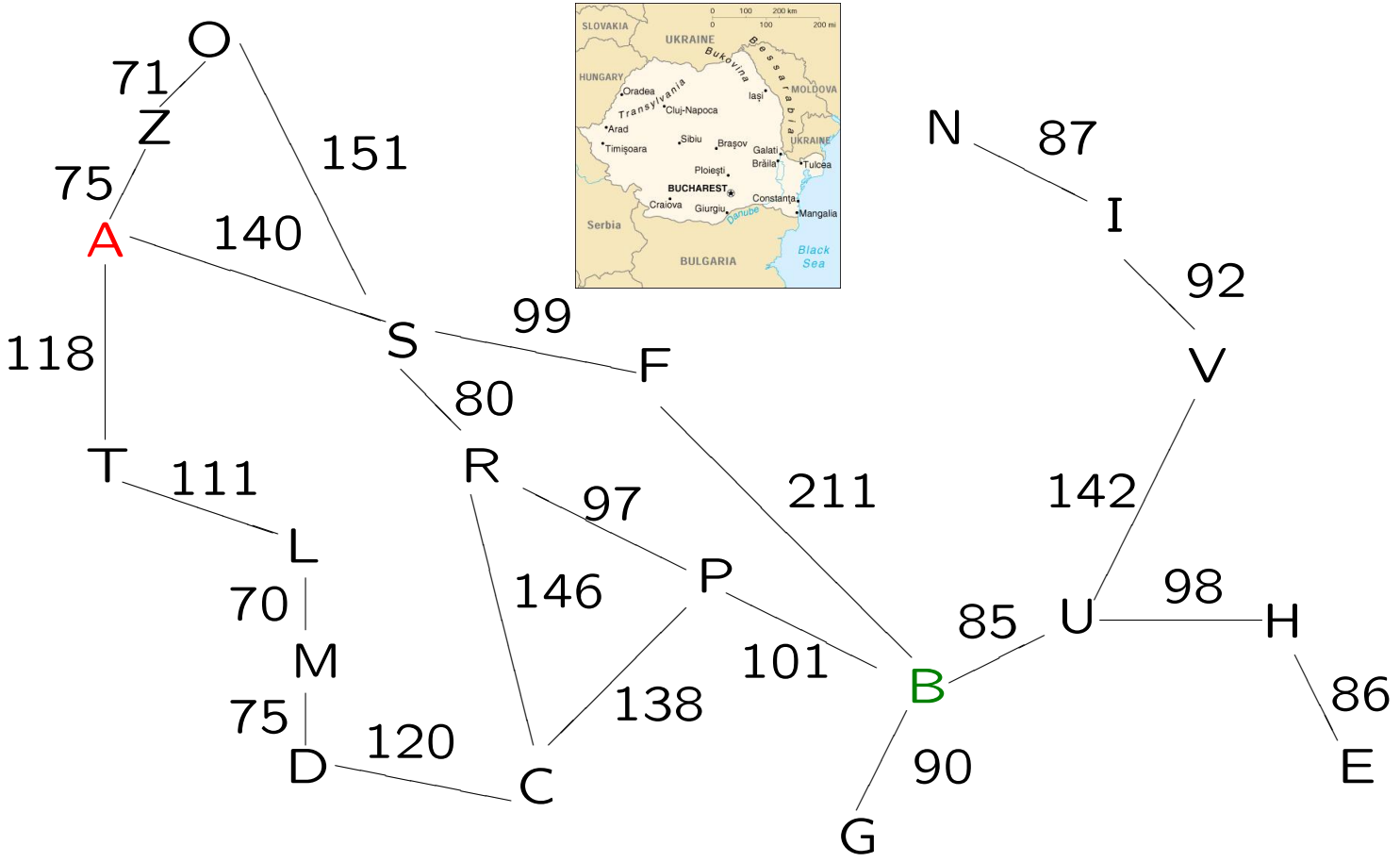
Het probleem heeft een toestand-actie-ruimte, zie ook het college Algoritmiek. Paden hierin (toestanden door acties verbonden) zijn mogelijke oplossingen.

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.

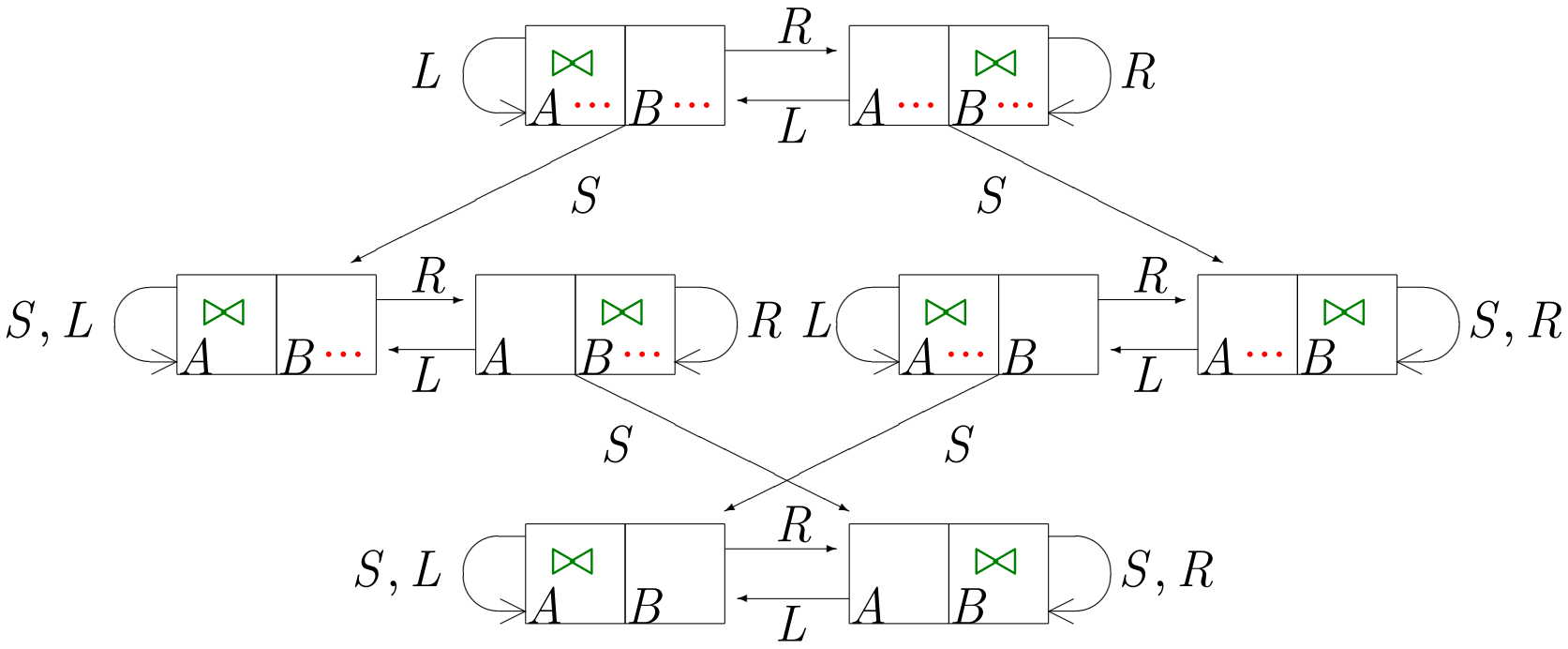
Voorbeeld: Stofzuiger-wereld — 1

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

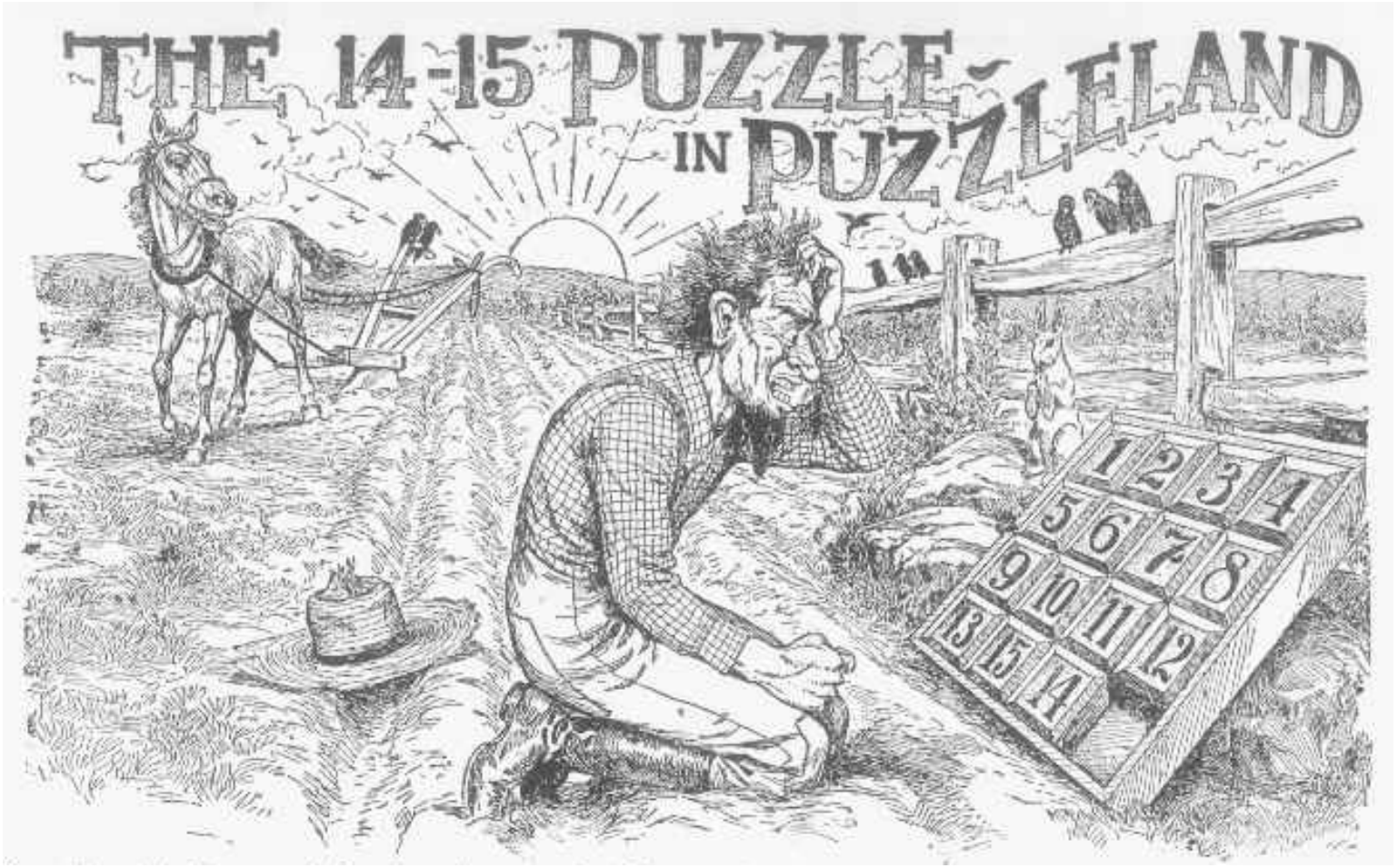
NB In de vorige versie gingen *Left* en *Right* een stukje naar links/rechts.

Voorbeeld: Stofzuiger-wereld — 2

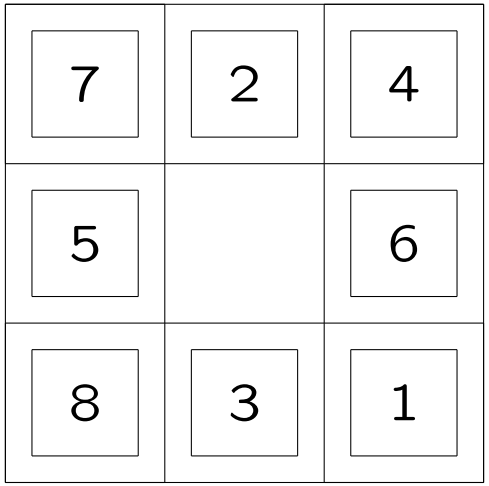


Legenda van het toestand-actie-diagram: stofzuiger: ⋈; stof (*Dirty*): ...; *L(eft)*; *R(ight)*; *S(uck)*. *N* is weggelaten.

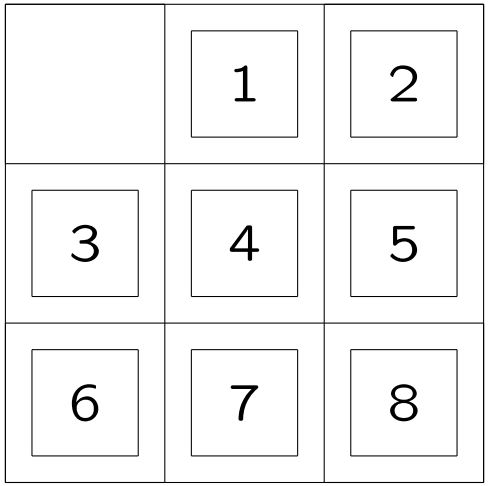
Voor zowel het single-state probleem als het multiple-state probleem is *R*,*S*,*L*,*S* een oplossing.


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



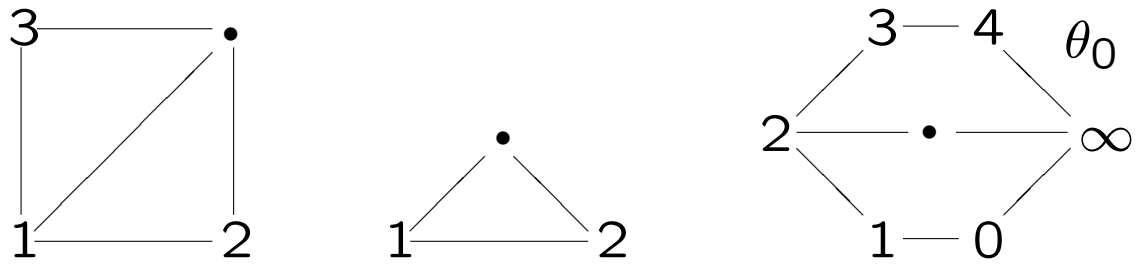
doel-toestand

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.

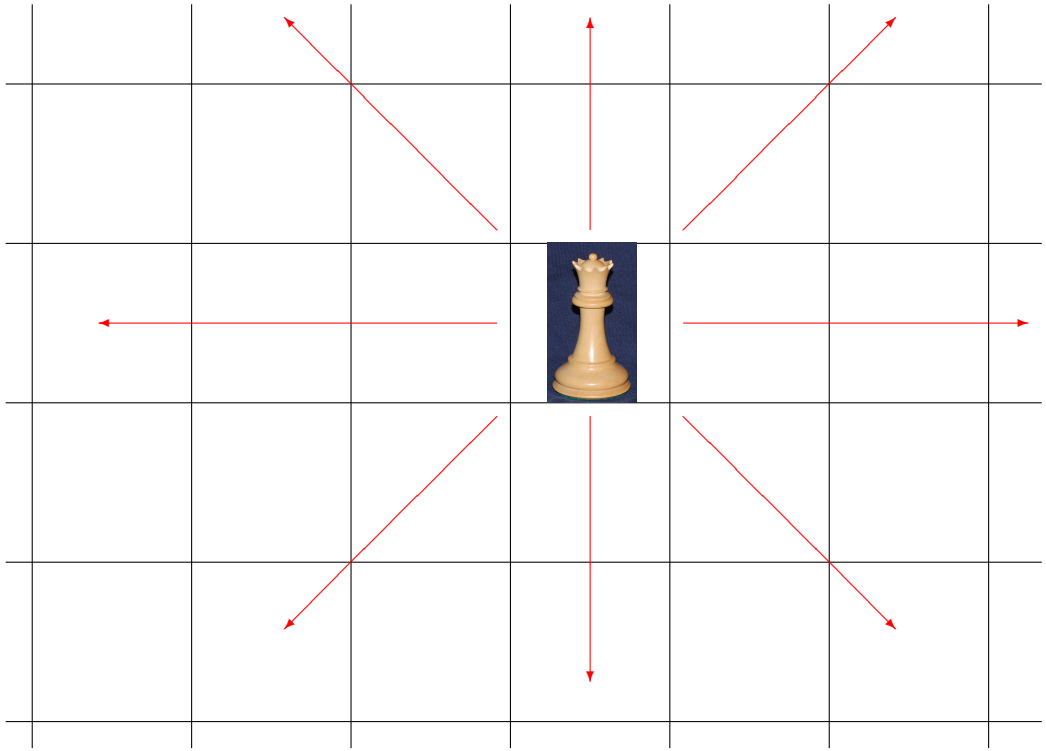
Voorbeeld: De 15- en 24-puzzel

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

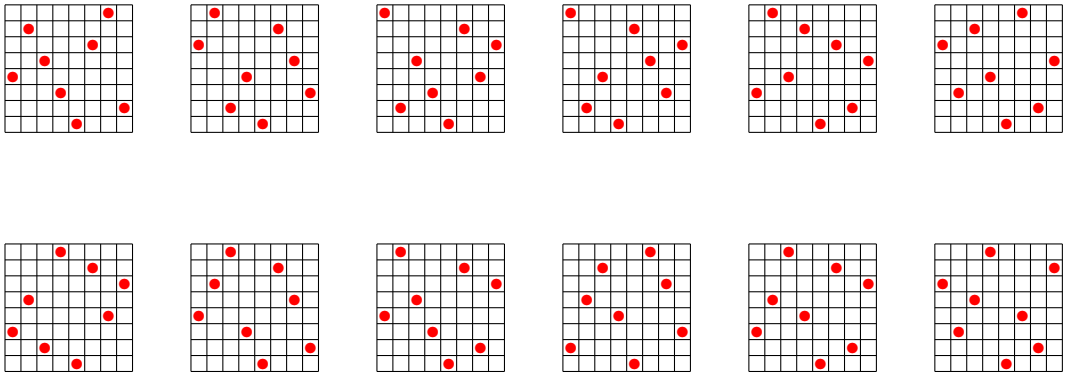
$$\begin{aligned} x^3 &\geq n^3(n-3)^3(n-6)^3\ldots \\ &\geq n(n-1)(n-2)(n-3)\ldots = n! \end{aligned}$$

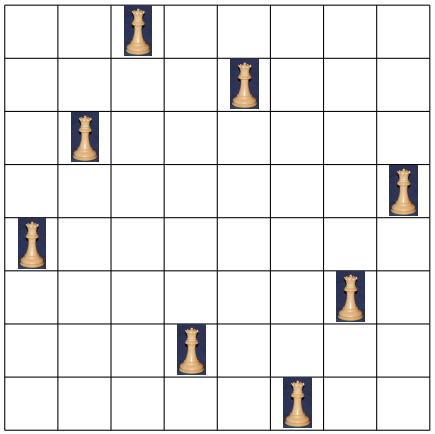
Voorbeeld: 8 dames — 3

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.

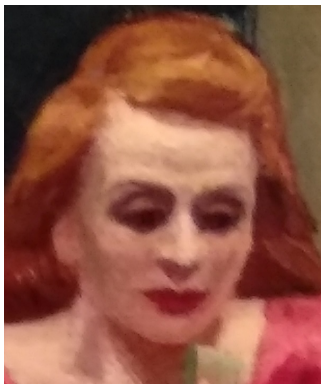
Andere echte problemen: VLSI-ontwerp, robot-navigatie, TSP (Traveling Salesman Problem), ontwerp van eiwitten, internet-agenten, . . .

Soorten problemen — 1

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*.



Soorten problemen — 2

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!

Stofzuiger-wereld — versies

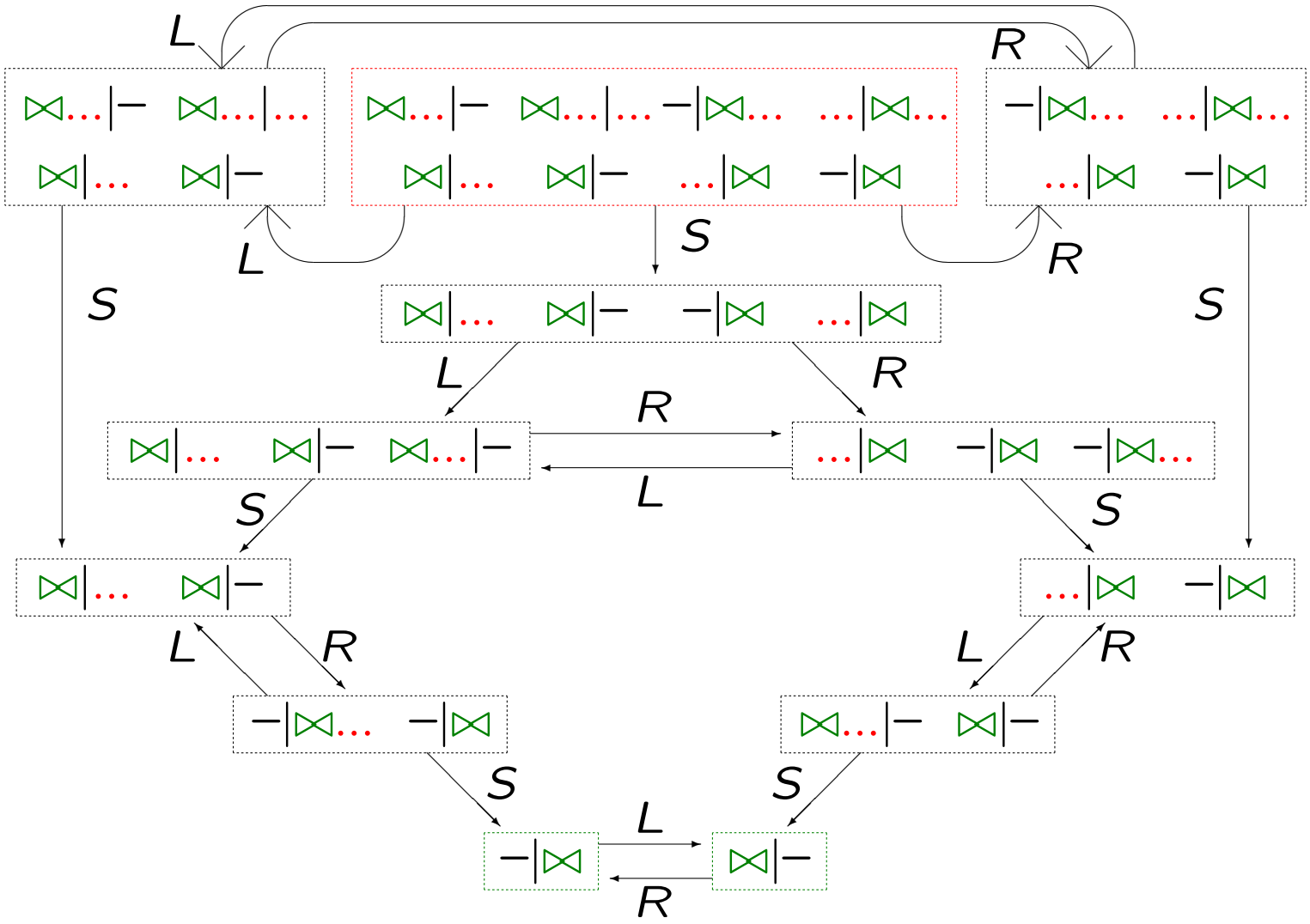
We noteren weer: stofzuiger: $\bowtie$; stof (*Dirty*): ...; *L(eft)*; *R(ight)*; *S(uck)*. En met een | de scheidsmuur; − is een schone ruimte (zonder stofzuiger).

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



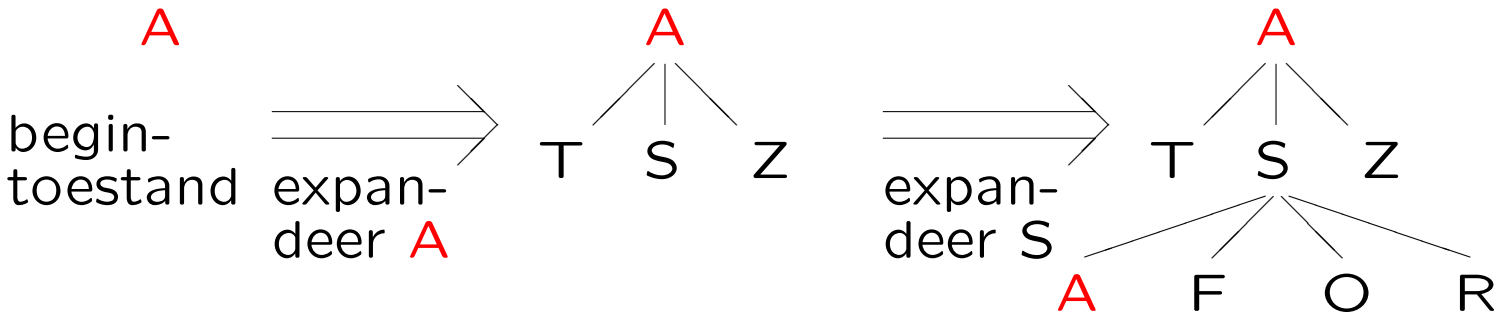
We hebben hier belief states, waaronder begin en doel.

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?

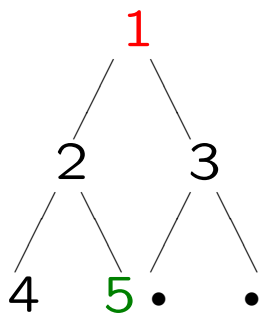
ruimtecomplexiteit Hoe veel geheugen vergt het?

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 $O(b^{d+1})$.

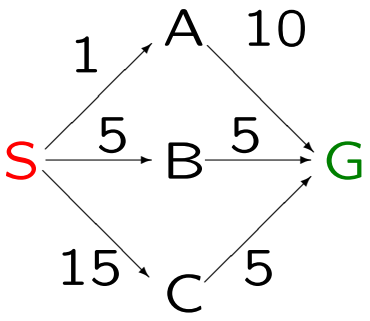
Het ruimteprobleem is het ergste. Bijvoorbeeld, voor $b = d = 10$, is 10^{11} tijdseenheden handelbaarder dan 10^{11} geheugen-eenheden. Maar toch gaat het met de tijd ook niet fijn.

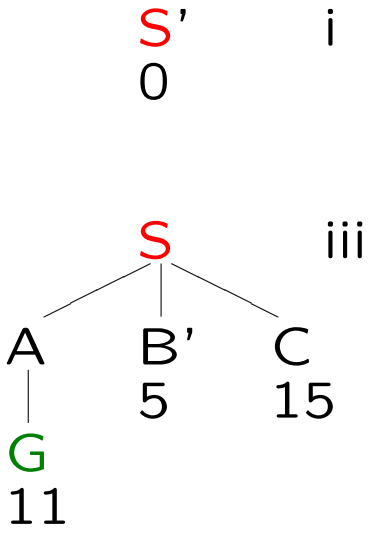
Uniform cost search — 1

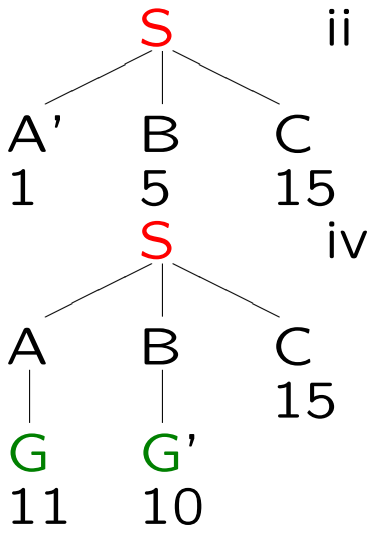
Bij Uniform cost search, in Europa ook bekend als Dijkstra's algoritme, wordt steeds als eerste de knoop met de *laagste padkosten* ge-expandeerd. De frontier is een rij, geordend op padkosten.

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.



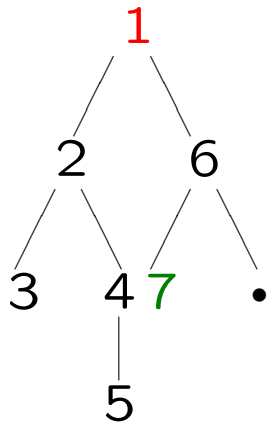




Er staat een ' bij de knoop die ontwikkeld wordt.
Let op: deze graaf is gericht (dus niet terug naar S)!

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.

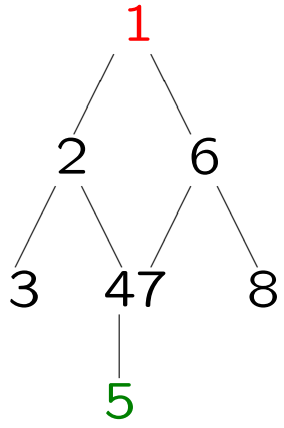
Voor Depth limited search doe je gewoon DFS, alleen houd je op bij een zekere (voorbedachte) diepte. Soms *weet* je dat er een oplossing van een zekere maximale diepte is, bijvoorbeeld dankzij de *diameter* van de zoekruimte. In Roemenië hebben we 20 steden, dus maximaal padlengte 19 (de echte diameter 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 (1 maart 2021): 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: 23 maart 2021.

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 week 5, 9, 11 en 14.
In het bijzonder in week 5: opgaven 2, 3, 7, 8, 14.