Logisch programmeren 2012
Zoeken

Michael Moortgat

26 maart 2012

Inhoudsopgave

1 Toestandsruimte 3

2 Zoektechnieken 10

3 Depth-first search (df) 11

4 bf: breadth-first search 18

5 Zoeken met heuristiek: best first search 22

6 Samenvatting 29

1. Toestandsruimte

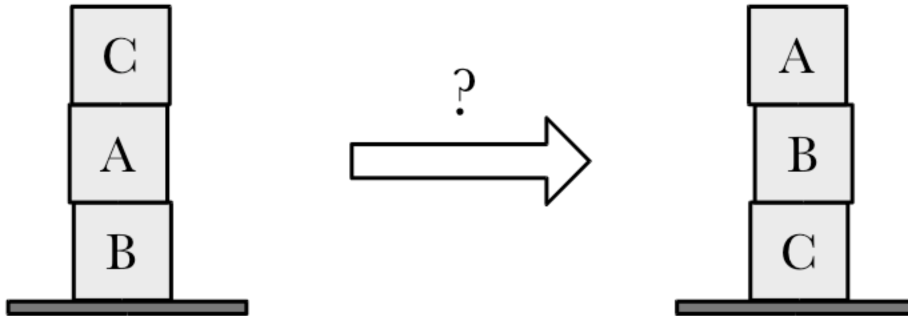
Veel AI problemen kunnen teruggebracht worden tot het zoeken van een pad in een

toestandsruimte

De toestandsruimte (Engels: state space) is een gerichte graaf:

- knopen: probleemsituaties
- boogjes: toegestane overgangen tussen toestanden (zetten)
- gegeven: begintoestand; eindtoestand(en)

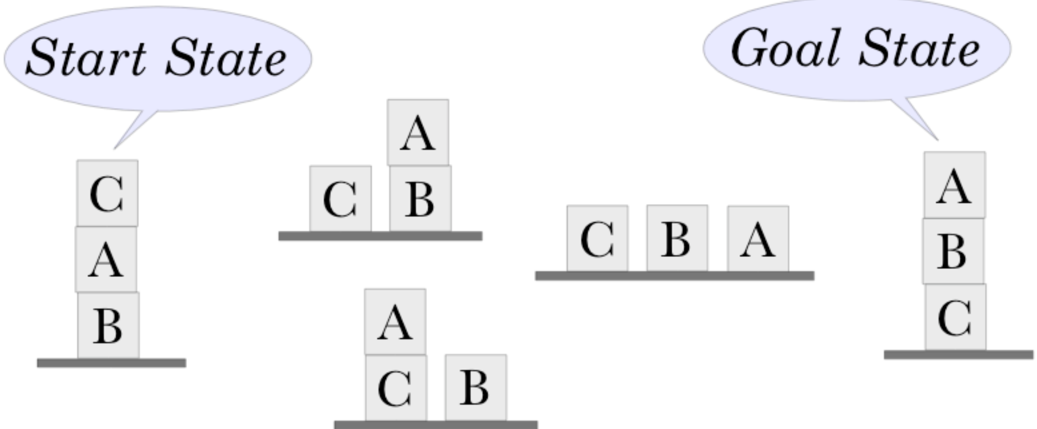
Voorbeeld: blokken wereld



Zoek een reeks overgangen die je van de begintoestand (*Start node*) naar de eindtoestand (*Goal node*) brengen

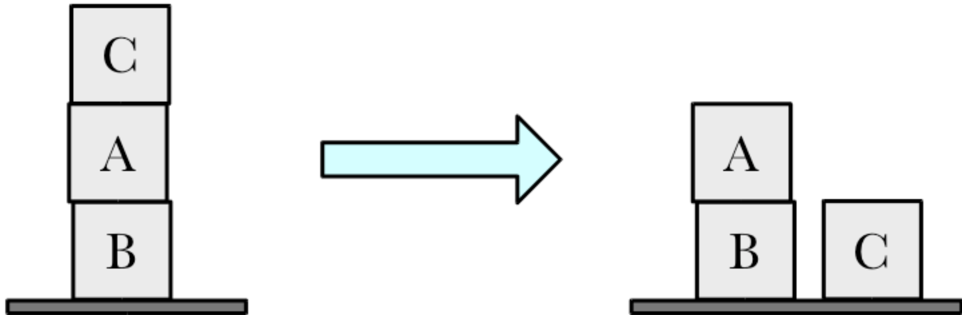
Toestanden

De toestanden zijn de verschillende legitieme bloksituaties:



Overgangen (transformaties, zetten)

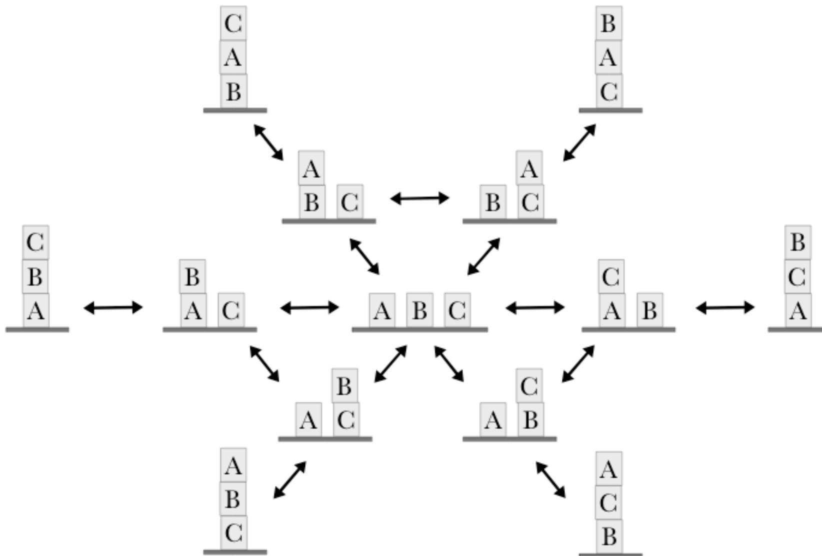
We beschikken over een stel zetten (transformaties) om van een gegeven toestand in een andere over te gaan:



1. Zet C op de tafel
2. - zet A op de tafel
   - zet A op C
   - zet C op A

De toestandsruimte als graaf

De toestanden en mogelijke overgangen tussen de toestanden kunnen worden weergegeven in de vorm van een gerichte graaf:



Samenvattend

De toestandsruimte voor een probleem krijgt de vorm van een gerichte graaf:

- de knopen corresponderen met toestanden
- de pijlen corresponderen met overgangen tussen toestanden
- aan een overgang kan desgewenst een gewicht worden toegekend

Daarnaast moeten we om een probleem op te lossen, kennis hebben van:

- de startsituatie
- de doelsituatie(s)

In prolog

Weergave van de toestandsruimte in prolog:

1. vind een geschikte representatie voor de toestanden
2. leg de toegestane transformaties vast (als s/2 feiten of regels)
3. geef aan wat de starttoestand is
4. geef condities voor de doeltoestand(en)

Prolog representatie: s(S1,S2) slaagt als er een transformatie is tussen een toestand `S1` en een toestand `S2`: S2 is de successor van S1.

Eventueel met toevoeging van extra info: gewicht, toegepaste zet, . . .

2. Zoektechnieken

Het vinden van een oplossing

- Vind een pad van de starttoestand naar een doeltoestand in een toestandsruimte

Zoekstrategieën

- depth-first search; iterative deepening
- breadth-first search
- best-first search

3. Depth-first search (df)

We zoeken een pad Sol van een gegeven knoop N naar een doelknoop:

basisgeval als N gelijk is aan de doelknoop, dan geldt Sol = [N]

recursie als N een kind N1 heeft,

en er loopt een pad Sol1 van N1 naar een doelknoop,

dan geldt Sol = [N|Sol1]

df in prolog

Uit Bratko, Prolog, Programming for AI.

```
% dfsolve( Node, Solution):
% Solution is a path (in reverse) between Node and a goal

dfsolve( Node, Solution)  :-
  depthfirst( [], Node, Solution).

% depthfirst( Path, Node, Solution):
%   extending the path to a goal gives Solution

depthfirst( Path, Node, [Node | Path] )  :-
   goal( Node).

depthfirst( Path, Node, Sol)  :-
  s( Node, Node1),
  depthfirst( [Node | Path], Node1, Sol).
```

Problemen voor df

- omdat je dezelfde toestand verschillende keren zou kunnen aandoen, is het mogelijk dat er nooit een oplossing wordt gevonden
- er zouden veel kortere oplossing kunnen zijn dan de eerst gevonden oplossing

Mogelijke verbeteringen:

- lusdetectie: check of kandidaattoestand niet al voorkomt in het pad
- dieptelimiet: voorkomt dat de df procedure een oneindig pad ingaat

Lusdetectie

```
depthfirst( Path, Node, Sol)  :-
  s( Node, Node1),
  \+ member( Node1, Path),                % Prevent a cycle
  depthfirst( [Node | Path], Node1, Sol).
```

df met vaste dieptelimiet

Bratko:

```
% depthfirst2( Node, Sol, Maxdepth):
% Sol is a path, not longer than Maxdepth, from Node to a goal

depthfirst2( Node, [Node], _)  :-
   goal( Node).

depthfirst2( Node, [Node | Sol], Maxdepth)  :-
   Maxdepth > 0,
   s( Node, Node1),
   Max1 is Maxdepth - 1,
   depthfirst2( Node1, Sol, Max1).
```

df met groeiende dieptelimiet

Het df zoekregime met vaste dieptelimiet is incompleet: misschien zit de gezochte oplossing dieper dan de gestelde maximale diepte.

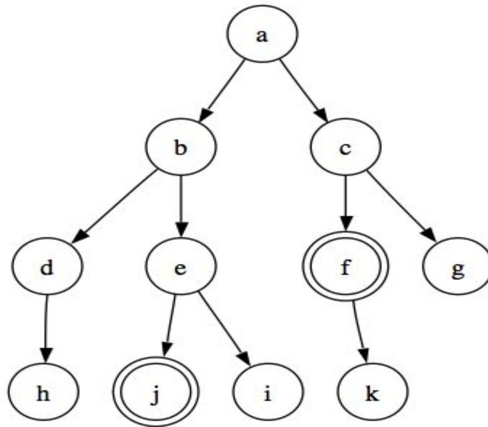
Een iterative deepening regime lost dit probleem op:

- start met df met lage limiet n
- als diepte n uitputtend is doorzocht: hoog limiet op tot $n + 1$

Iteratieve deepening vindt een pad naar een oplossing (als die er is . . .)

Je hebt zelf het iterative deepening regime geprogrammeerd bij de opgaven van vorige week.

Voorbeeld: df



- $\leadsto$ df verkent eerst de knopen verst van de startknoop verwijderd
- $\leadsto$ geen garantie voor het vinden van het kortste pad naar een oplossing

4. bf: breadth-first search

- bf verkent eerst knopen dichtst bij de startknoop
- vereist het bijhouden van een verzameling kandidaat paden

Voordelen en nadelen t.o.v. het df regime:

- bf vindt altijd eerst kortste pad naar oplossing; df niet
- bf kost meer ruimte (kandidaatpaden) dan df
- iteratief df combineert prettige eigenschappen van bf en naief df

Queues

Een geschikte datastructuur voor bf zoeken is de queue. Een queue (wachtrij) kan je modelleren met een Prolog term

q(N,Front,Back)

- Front,Back is een verschillijst paar
- N geeft de lengte van de wachtrij in successornotatie
- q(0,L,L) is de lege queue

Anders dan bij een lijst is een queue even makkelijk toegankelijk aan het begin als aan het eind.

Bewerkingen op queues

Maak een lege wachtrij:

```
empty_queue(q(0,L,L))
```

Haal het voorste element uit de wachtrij:

```
queue_cons(Voorste, q(N,L,R), q(s(N),[Voorste|L],R)).
```

Een lijst achteraan in de wachtrij zetten:

```
queue_append(Queue,Lijst,LangereQueue).
```

Een lijst vooraan aan een wachtrij toevoegen:

```
append_queue(Lijst,Queue,LangereQueue).
```

Definities vind je in de startcode voor Week 8.

Te verkennen knopen met queue

bf/3 bouwt een pad Path van een begintoestand Start naar een oplossing Final. De wrapper voor bf/5 voert twee controle argumenten toe: een wachtrij en een accumulator voor het te bouwen pad. De twee controle argumenten zijn initieel leeg.

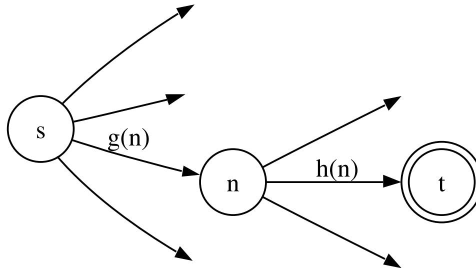
```
bf(Start, Goal, Path) :-
    empty_queue(Queue), bf(Start, Goal, Queue, [], Path).

bf(S, S, _, RevPath, Path) :- solution(S),
    reverse([S|RevPath], Path).

bf(S0, Goal, Queue0, Path0, Path) :-
    findall(n(S1,[S0|Path0]), s(S0,S1), Children),
    queue_append(Queue0, Children, Queue1),  % enqueue
    queue_cons(n(S2, Path2), Queue, Queue1),   % dequeue
    bf(S2, Goal, Queue, Path2, Path).
```

5. Zoeken met heuristiek: best first search

- df en bf beschouwen alle kandidaatknopen als even beloftevol voor verkenning
- een best first regime gebruikt een heuristiek om uit kandidaatknopen de meest beloftevolle te selecteren voor verkenning
- de heuristiek gebruikt probleemspecifieke informatie

De evaluatiefunctie

$f(n)$ schat de prijs ('lengte') van het beste pad van s naar t via n. $g(n)$ is de som van de prijs van de overgangen van s naar n (al verkend); $h(n)$ is de heuristische functie, en schat de minimale lengte van het onverkende stuk van n naar doel t.



$$f(n) = g(n) + h(n)$$

Geoorloofde heuristische functies

Zij $h^*(n)$ de lengte van het werkelijk kortste pad van n naar een doelknoop. Een heuristische functie h heet geoorloofd als voor alle toestanden n geldt dat

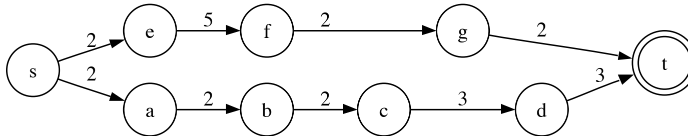
$$0 \leq h(n) \leq h^*(n)$$

Het A^* zoekalgoritme $=$ best first met evaluatiefunctie $f(n) = g(n) + h(n)$ met geoorloofde h. A^* is een heuristisch geoorloofd algoritme:

- het algoritme termineert voor elke begintoestand;
- het vindt een doeltoestand op minimale afstand van de begintoestand

Voorbeeld

Optimale route vinden:



Geoorloofde $h(n)$ is hier afstand in vogelvlucht naar doel, bijvoorbeeld

	a	b	c	d	e	f	g
$h(n)$	5	4	4	3	7	4	2

en dus $f(n) = g(n) + h(n)$

	a	b	c	d	e	f	g	t
$f(n)$	7	8	10	12	9	11	11	11!

Priority queues (heaps)

Voor best first zoeken gebruiken we de priority queue (of: heap). Een heap is een wachtrij waarin elk element voorzien is van een prioritering. Elementen worden uit de wachtrij gehaald in oplopende orde van prioritering (laagst is best).

De gebruikte SWI datastructuur (uit library(heaps)):

heap(H,Size)   met H een binaire boom t(El,Prioriteit,L,R)

Enkele bewerkingen:

`empty_heap`(*?Heap*)

`get_from_heap`(*?Heap0, ?Priority, ?El, -Heap*)

haalt element met beste waarde uit een heap

`merge_heaps`(*+Heap0, +Heap1, -Heap*)

voegt twee heaps samen tot een nieuwe heap

Best first met priority queue

Het algoritme bestf/4 zoekt een pad Path van Start naar Final met minimale kostprijs Cost.

```
bestf(Start, Final, Cost, Path) :-
    empty_heap(Heap),
    bestf(Start, Final, Heap, 0, Cost, [], Path).
```

De wrapper voegt een priority queue Heap toe voor de nog te exploreren knopen, en accumulatoren voor het te bouwen pad en zijn totale kostprijs.

De controle argumenten worden geïnitialiseerd met de gepaste nulwaarde.

Best first met priority queue (vervolg)

```
bestf(S, S, _, Cost, Cost, RevPath, Path) :-
    reverse([S|RevPath], Path).

bestf(S0, S, Heap0, G0, Cost, Path0, Path) :-
    findall(F-n(S1,G,[[C],S0|Path0]),
        (   s(S0,S1,C),  % overgang met kostprijs C
            G is C+G0,   % pad tot bij S1
            h(S1,S,H),   % heuristiek
            F is G+H),   % evaluatiefunctie
                Children),
    list_to_heap(Children,ChildrenHeap),
    merge_heaps(ChildrenHeap,Heap0,Heap1),
    get_from_heap(Heap1,_,n(S2, G2, Path2), Heap),
    bestf(S2, S, Heap, G2, Cost, Path2, Path).
```

6. Samenvatting

De drie zoekalgoritmen maken een onderscheid tussen twee sets van knopen:

- Open: de nog te exploreren knopen
- Closed: de al geëxploreerde knopen waaruit het pad naar de oplossing wordt gebouwd

Ze verschillen in de manier waarop de Open set wordt beheerd:

- depth first: stack
- breadth first: queue
- best first: priority queue