Logisch programmeren 2012
Zoeken

Michael Moortgat

28 maart 2012

Inhoudsopgave

1 Toestandsruimte 3

2 Zoektechnieken 9

1. Toestandsruimte

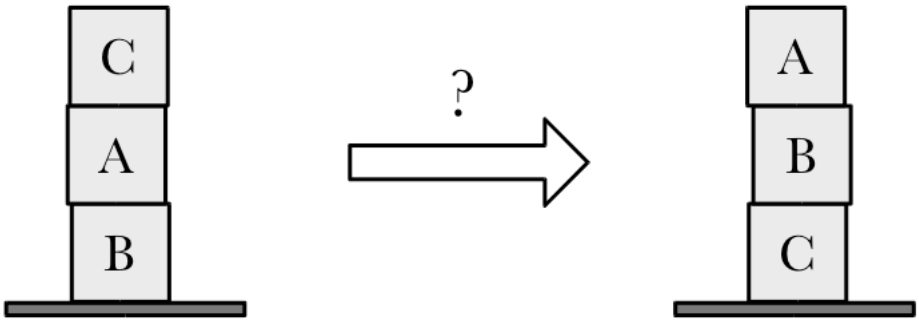
Veel AI problemen kunnen teruggebracht worden tot het zoeken van een pad in een

toestandsruimte

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

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

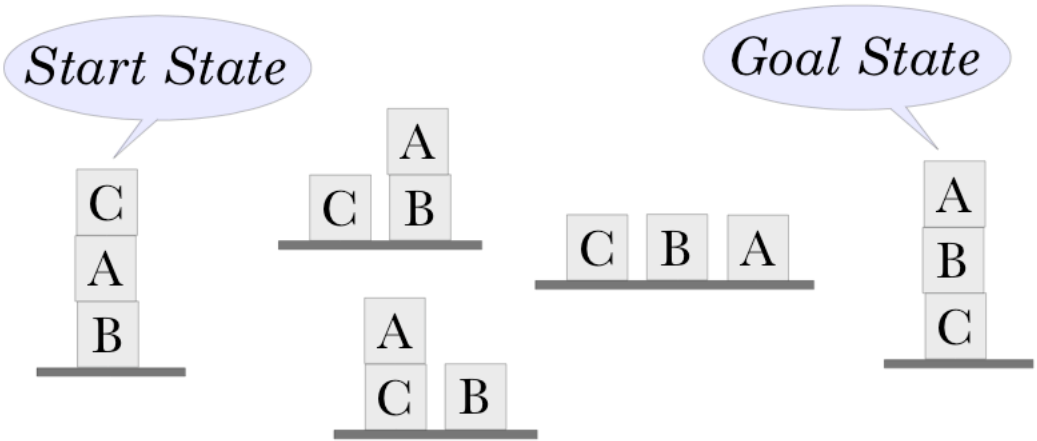
Voorbeeld: blokken wereld



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

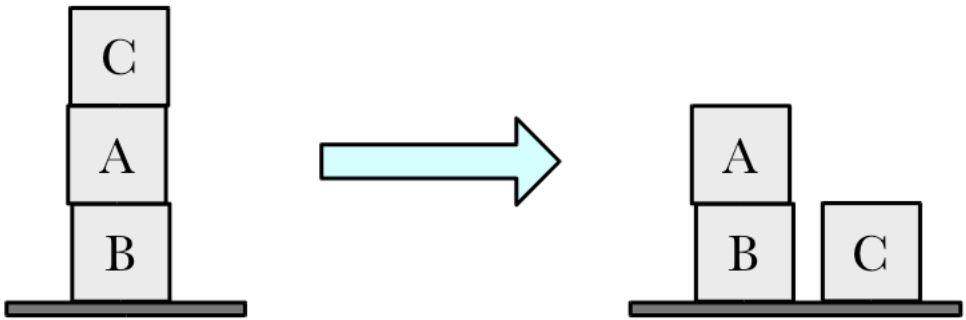
Toestanden

De toestanden zijn de verschillende legitieme bloksituaties:



Overgangen (transformaties, zetten)

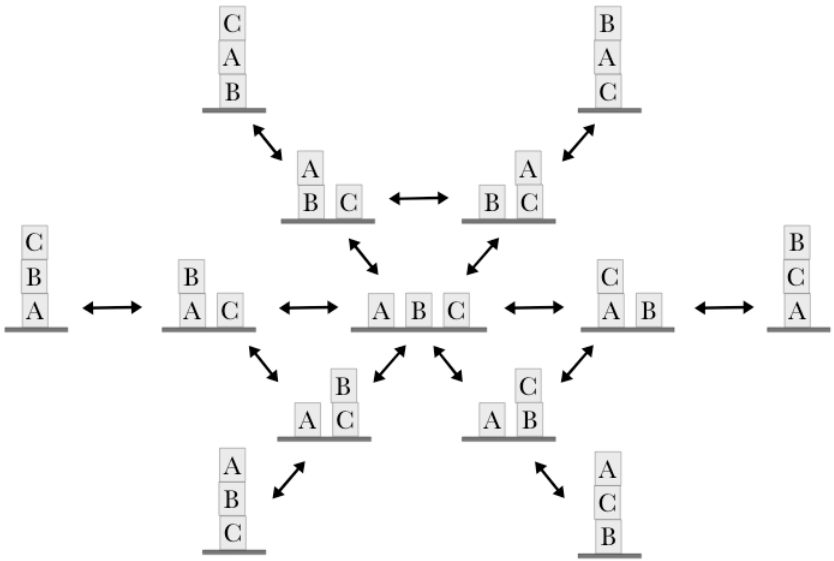
We beschikken over een stel zetten (transformaties) om van een gegeven toestand in een andere over te gaan:



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:



Toestandsruimte 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

df, bf, best

We belichten de overeenkomsten en de verschillen tussen de drie zoekalgoritmen. Alle drie maken ze een onderscheid tussen twee sets van knopen:

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

De algoritmen verschillen in de manier waarop de Open set wordt beheerd:

- `df`: een stapelgeheugen (stack). Nieuw te exploreren knopen worden vooraan toegevoegd.
- `bf`: een wachtrij (queue). Nieuw te exploreren knopen worden achteraan toegevoegd.
- `best`: een priority queue. De plaats in de rij hangt af van de prioriteit van een knoop.

df

Hier de wrapper. Toestanden worden gerepresenteerd als [Knoop,Ouder] paren (ouder is nil in het geval van de startknoop).

```
df(Start,Goal) :-
    retractall(solution(_)),
    assert(solution(Goal)),
    empty_stack(Empty_open), % lege lijst
    stack([Start,nil], Empty_open, Open_list), % ttz append
    empty_set(Closed_set), % lege lijst
    search(df,Open_list, Closed_set, Goal).

empty_stack([]).
empty_set([]).
stack(Element,Stack,[Element|Stack]).
```

Vergelijk: bf

Alleen de datastructuur voor Open aanpassen: stack wordt queue.

```
bf(Start,Goal) :-
    retractall(solution(_)),
    assert(solution(Goal)),
    empty_queue(Empty_open),
    enqueue([Start,nil], Empty_open, Open_queue),
    empty_set(Closed_set),
    search(bf,Open_queue, Closed_set, Goal).

enqueue(Element,Queue,BigQ) :- queue_cons(Element,Queue,BigQ).
dequeue(Element,Queue,BigQ) :- queue_cons(Element,Queue,BigQ).
```

Zoeken: df

Hieronder de niet-recursieve gevallen.

```
search(df,Open_list,_,_) :-
  empty_stack(Open_list), % geen knopen meer te verkennen
  write('No solution found.').

search(df,Open_list, Closed_set, Goal) :-
  stack([State,Parent],_,Open_list), State=Goal, % doel gevonden
  write('The solution path is:'), nl,
  get_path([State,Parent],Closed_set,Path,[]),
  write(Path).
```

We volgen het spoor State, Parent, Grandparent, ... om het afgelegde pad uit de set van bezochten knopen te halen.

```
get_path([State,nil|_],_,[State|R],R).
get_path([State,Parent|_],Closed,P,P1) :-
    member([Parent,GrandParent|_],Closed),
    get_path([Parent,GrandParent|_],Closed,P,[State|P1]).
```

Zoeken: df, recursie

Het recursieve geval. Een nieuw te exploreren toestand wordt van de stapel gehaald. De kinderen van deze toestand worden vooraan aan de stapel toegevoegd, de toestand zelf gaat naar de verkende knopen.

```
search(df,Open_list, Closed_set, Goal) :-
  stack([State,Parent],Rest_open, Open_list),
  get_children(State, Rest_open, Closed_set, Children),
  append(Children, Rest_open, New_open), % vooraan toevoegen
  union([[State,Parent]],Closed_set,New_closed_set),
  search(df,New_open, New_closed_set, Goal),
  !.

get_children(State, _, _, Children) :-
   findall([NewState,State],s(State,NewState),Children).
```

Je kan ook `Rest_open,Closed_set` bij het aanmaken van kinderen in de beschouwing betrekken (bijvoorbeeld voor lusdetectie . . .). Dat laten we hier buiten beschouwing.

Zoeken: bf

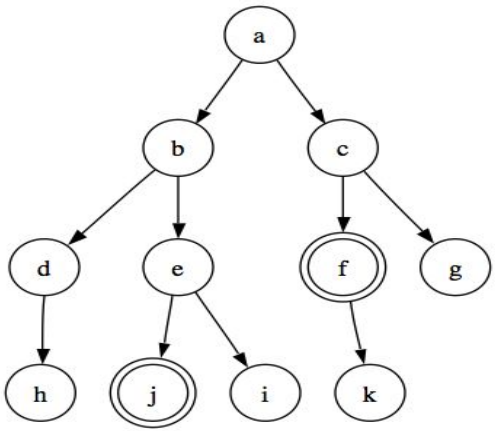
Voor breadth first zoeken moet alleen de behandeling van Open aangepast worden: een queue ipv een stack.

```
search(bf,Open_queue,_,_) :-
  empty_queue(Open_queue),  % geen knopen meer om te verkennen
  write('No solution found.').

search(bf,Open_queue, Closed_set, Goal) :-
  dequeue([State,Parent],_,Open_queue), State=Goal, % succes
  write('The solution path is:'), nl,
  get_path([State,Parent],Closed_set,Path,[]),
  write(Path).

search(bf,Open_queue, Closed_set, Goal) :-
  dequeue([State,Parent],Rest_open, Open_queue),
  get_children(State, Rest_open, Closed_set, Children),
  queue_append(Rest_open, Children, New_open),
  union([[State,Parent]],Closed_set,New_closed_set),
  search(bf,New_open, New_closed_set, Goal),
  !.
```

Voorbeeld



$\rightsquigarrow$ df vindt eerst oplossing j

$\rightsquigarrow$ bf vindt eerst oplossing f, dichter bij het startpunt

Queues

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).
```

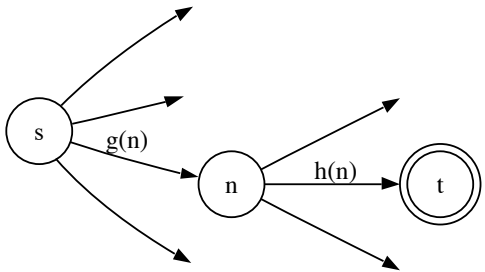
Definities vind je in de startcode voor Week 8.

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

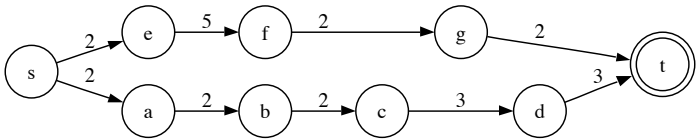
$$0 \leq h(n) \leq h^*(n)$$

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!

(Zie Bratko, Fig 12.2 voor de stappen.)

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 in Prolog

Toestanden worden gerepresenteerd als [State,Parent,G,H]

De wrapper, en de niet-recursieve gevallen:

```
best(Start,Goal) :-
    retractall(solution(_)),
    assert(solution(Goal)),
    empty_set(Closed_set),
    empty_heap(Empty_open),
    h(Start, Goal, H), % heuristiek
    add_to_heap(Empty_open,H,[Start,nil,0,H], Open_heap),
    search(best,Open_heap, Closed_set, Goal).

search(best,Open_heap,_,_) :-
    empty_heap(Open_heap),
    write('No solution found with these rules.').

search(best,Open_heap, Closed_set, Goal) :-
    get_from_heap(Open_heap,_,[State,Parent,_,_],_), State=Goal,
    write('The solution path is:'), nl,
    get_path([State,Parent,_,_],Closed_set,Path,[]),
    write(Path).
```

best first: recursief geval

```
search(best,Open_heap, Closed_set, Goal) :-
  get_from_heap(Open_heap,_,[State,Parent,G,H],Rest_open),
  get_best_children([State,Parent,G,H], Goal, Children),
  write('Trying: '),write((State,G,H)),nl,
  list_to_heap(Children,ChildrenHeap),
  merge_heaps(ChildrenHeap,Rest_open,New_open),
  union([[State,Parent,G,H]],Closed_set,New_closed_set),
  search(best,New_open, New_closed_set, Goal),!.
```

Van een lijst Prioriry-Item maakt `list_to_heap` een priority queue.

```
get_best_children([State,_,G0,_], Goal,Children) :-
   findall(F-[NewState,State,G,H],
       (
       s_cost(State,NewState,Cost),
       G is G0+Cost,
       h(NewState,Goal,H),
       F is G+H
       ),
       Children).
```