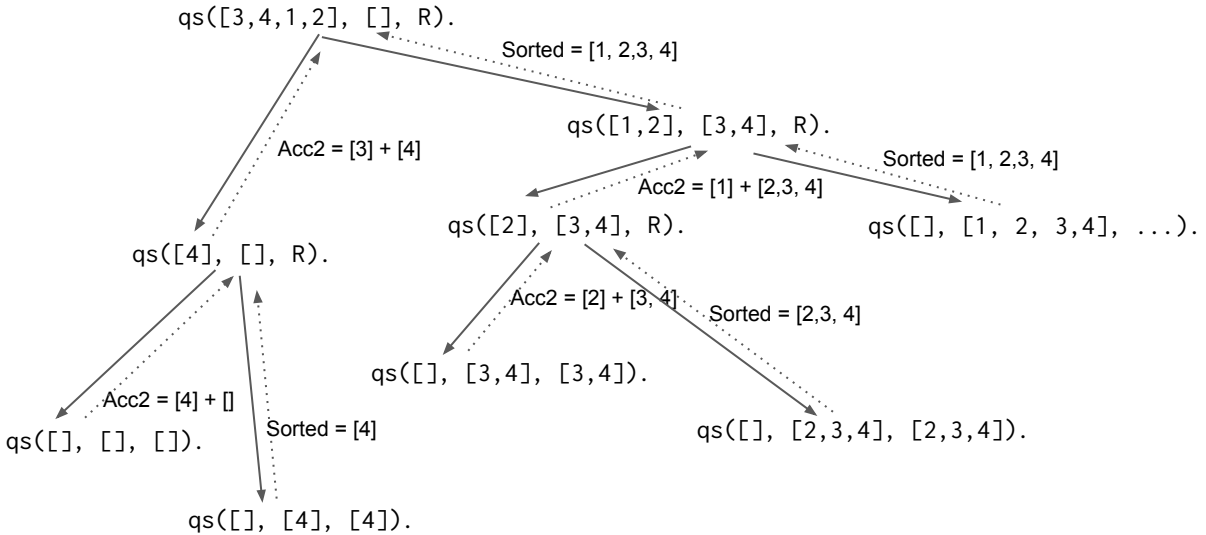
Prolog (4)

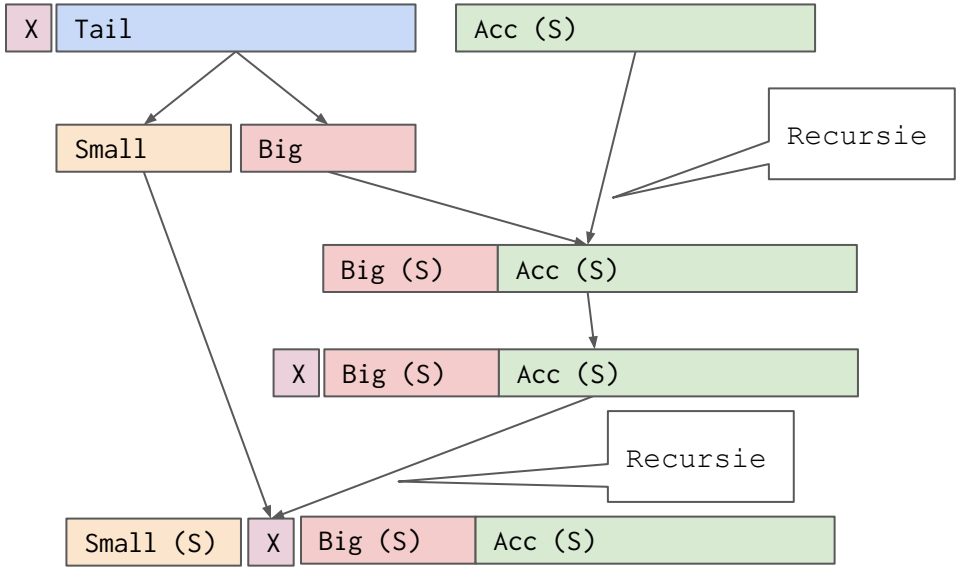
Quicksort

```
qs([], Acc,Acc).
qs([X|Tail], Acc, SortedList):-
     split(X,Tail, Small, Big),
     qs(Big, Acc, Acc1),
     Acc2 = [ X | Acc1],
     qs(Small, Acc2, SortedList).
```



Quicksort

```
qs([], Acc,Acc).
qs([X|Tail], Acc, SortedList):-
      split(X,Tail, Small, Big),
      qs(Big, Acc, Acc1),
      Acc2 = [ X | Acc1],
      qs(Small, Acc2, SortedList).
```

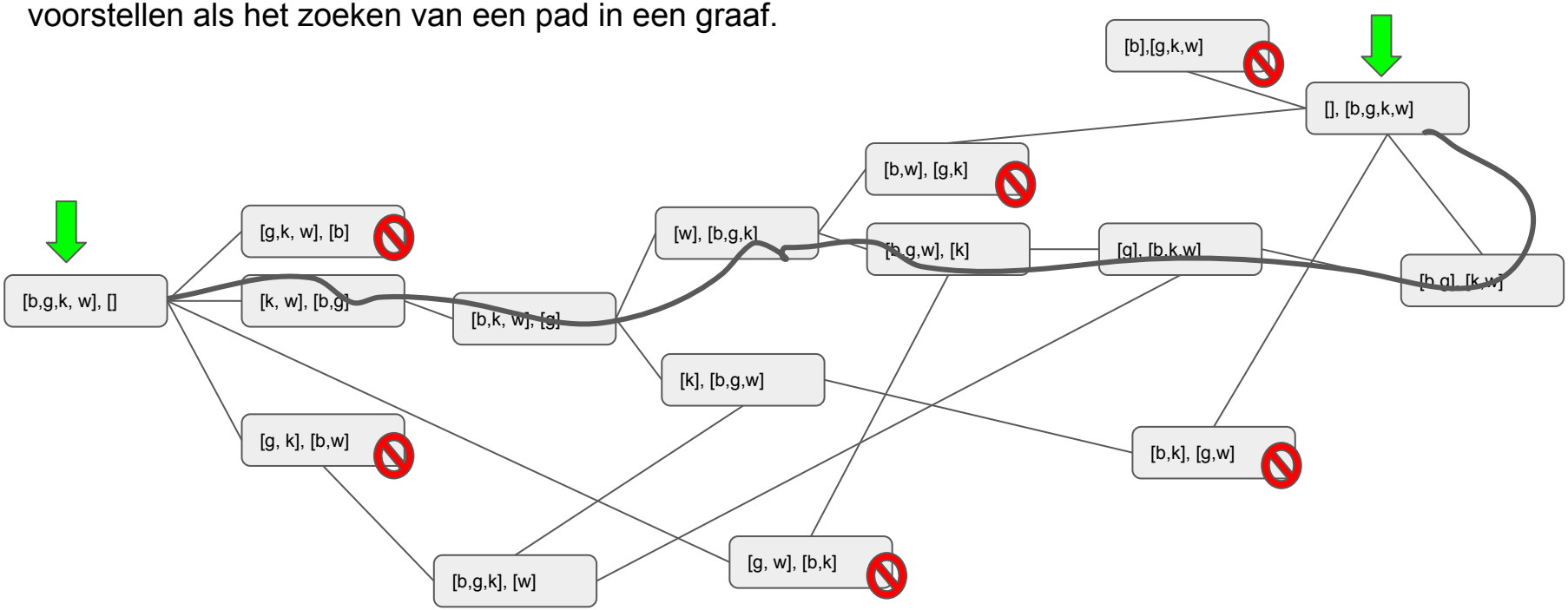


Inductief denken:

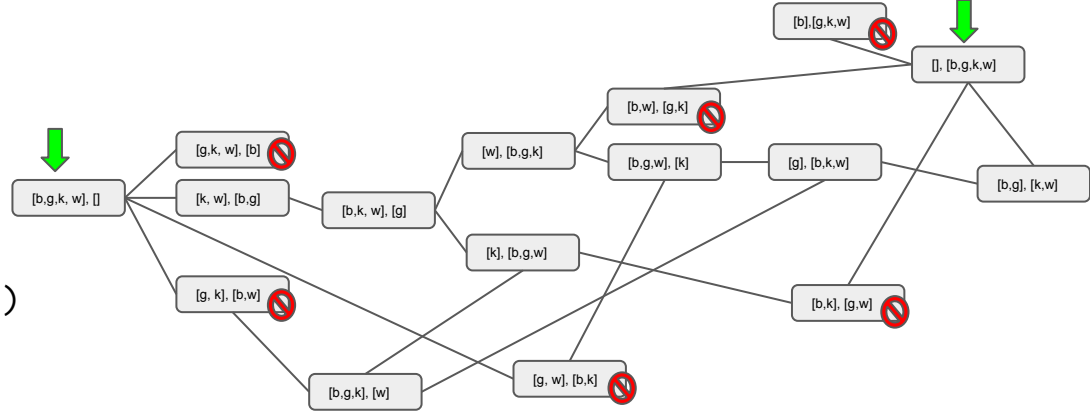
1. Base case opstellen (`k = 0`)
2. Aannemen dat het predicaat correct is voor `k`
3. Schrijf het predicaat voor `k + 1`

Graaf problemen (1)

Heel vaak in prolog kan men een probleem voorstellen als het zoeken van een pad in een graaf.



Graaf problemen (1)



Dus:

We kunnen dit oplossen met `path/4 :)`

Probleem:

Grafe structuur niet gegeven? `:(`

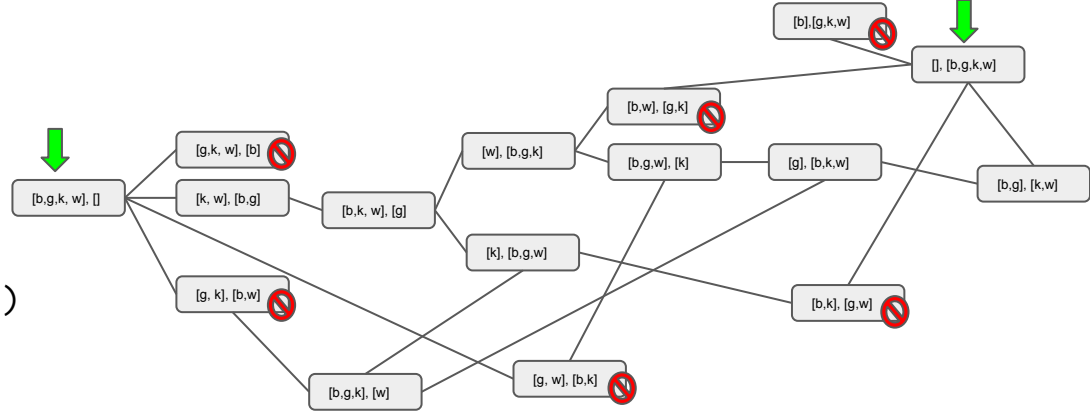
→ Finish ?

→ neighbours ?

→ unsafe nodes ?

```
beginState(s(l, [g,k,w],[])).
endState(s(r, [], [g,k,w])).

solve(Solution) :-
      beginState(S),
      endState(F),
      path(S, F, [], Solution).
```

Graaf problemen (1)



Dus:

We kunnen dit oplossen met `path/4 :)`

Probleem:

Grafe structuur niet gegeven? `:(`

→ Finish ?

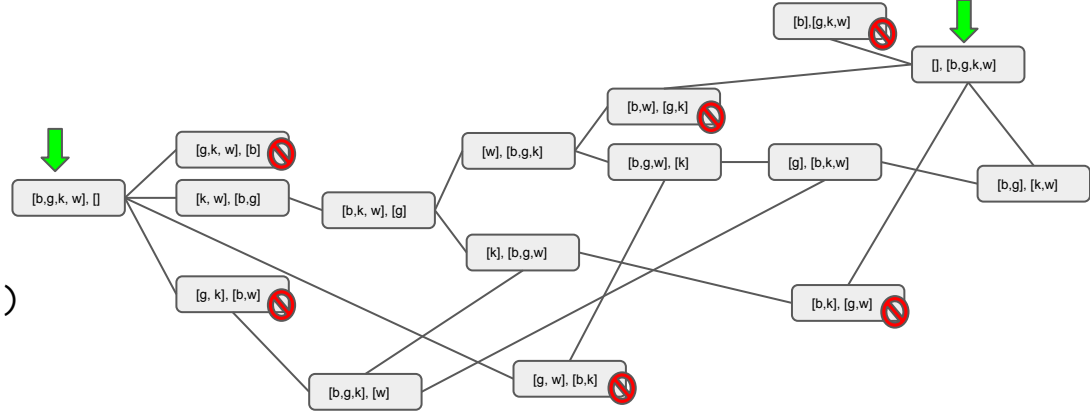
→ neighbours ?

→ unsafe nodes ?

```
% base case (= edge(S,F))
path(S, F, P, [F|P]) :-
     transit(S, F).
path(S, F, P, Result) :-
     transit(S, I),
     \+ member(I, P),
     path(I, F, [S|P], Result).
```

Merk dat `transit/2` exact dezelfde functie heeft als `edge/2`

Graaf problemen (1)



Dus:

We kunnen dit oplossen met `path/4 :)`

Probleem:

Grafe structuur niet gegeven? `:(`

→ Finish ?

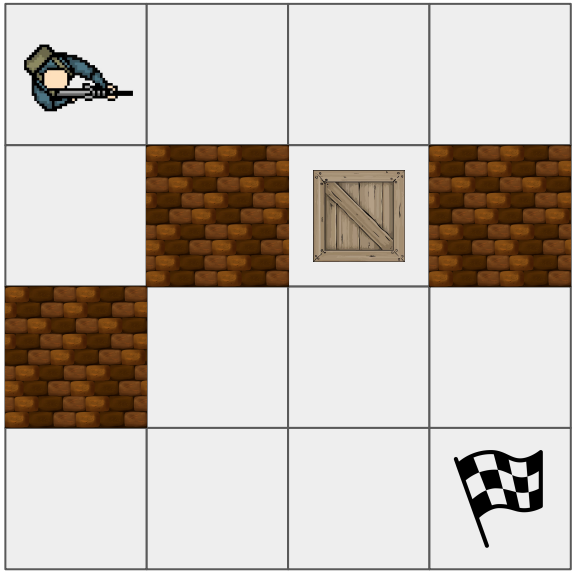
→ neighbours ?

→ unsafe nodes ?

```
% only farmer crosses
transit(s(l, L, R), s(r, L, R)) :- safe(L).
transit(s(r, L, R), s(l, L, R)) :- safe(R).

% farmer takes an object l -> r
transit(s(l, L, R), s(r, L1, R1)) :-
      select(X, L, L1),
      sort([X|R], R1),
      safe(L1).
% farmer takes an object r -> l
transit(s(r, L, R), s(l, L1, R1)) :-
      select(X, R, R1),
      sort([X|L], L1),
      safe(R1).
```

De `transit/2` genereert **dynamisch** de edges.

Graaf problemen (2)



Regels:

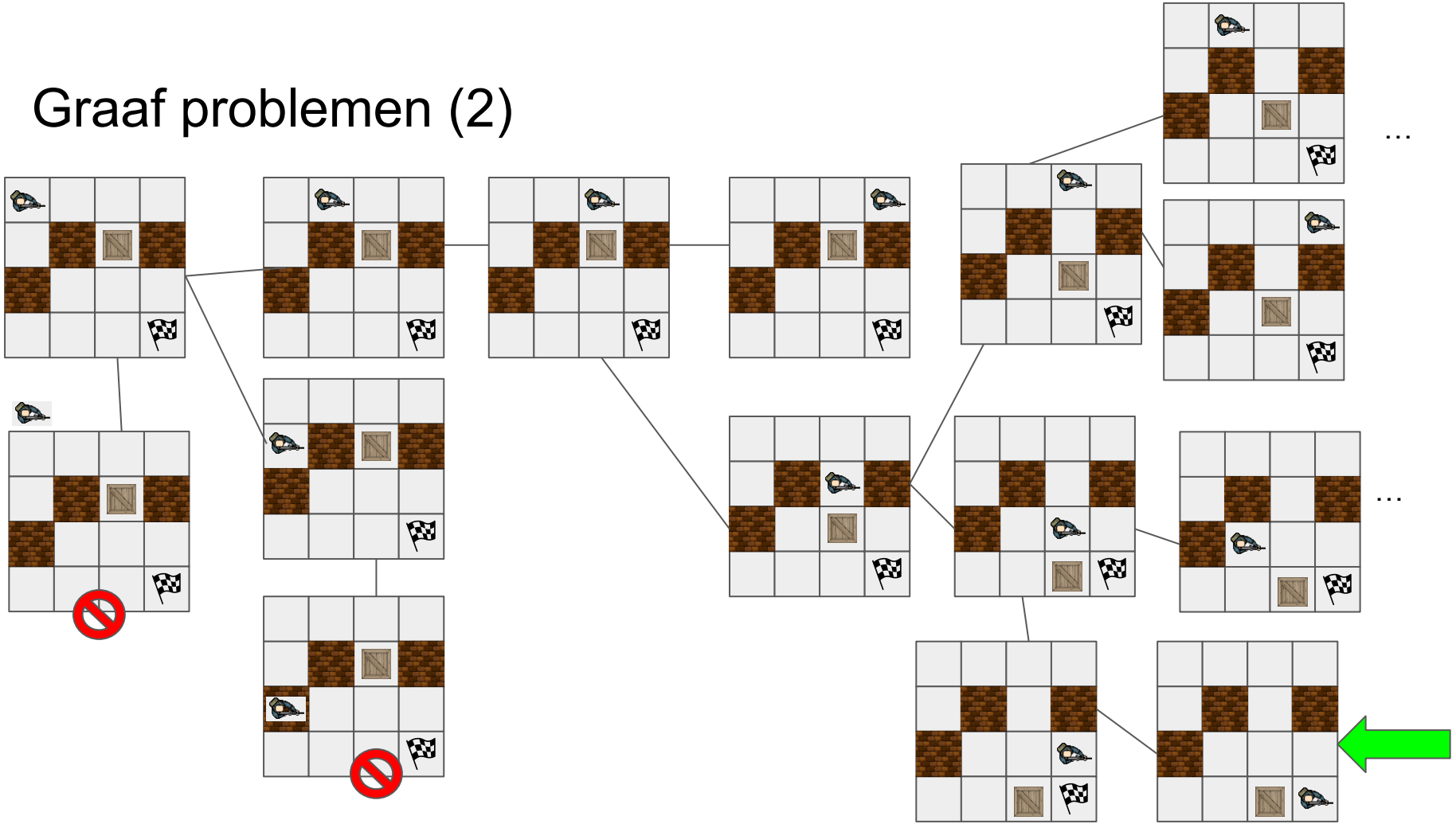
1. Je kan niet door muren
2. Je kan niet uit het speelveld
3. Je kan de doos duwen als je er naast staat.
4. Je kan de doos niet in een muur duwen of uit het speelveld duwen.

Doel:

bereik het vlaggetje

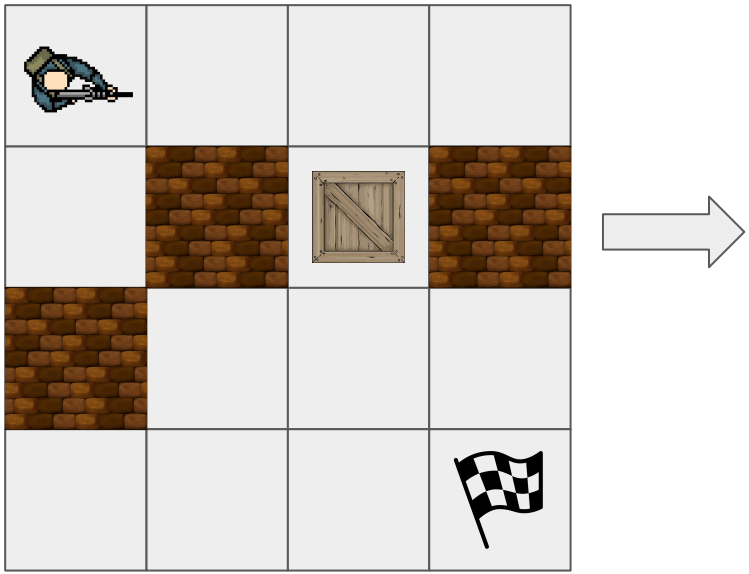
Een mogelijke oplossing is om hierover na te denken als een graaf probleem!

Graaf problemen (2)



Graaf problemen (2)

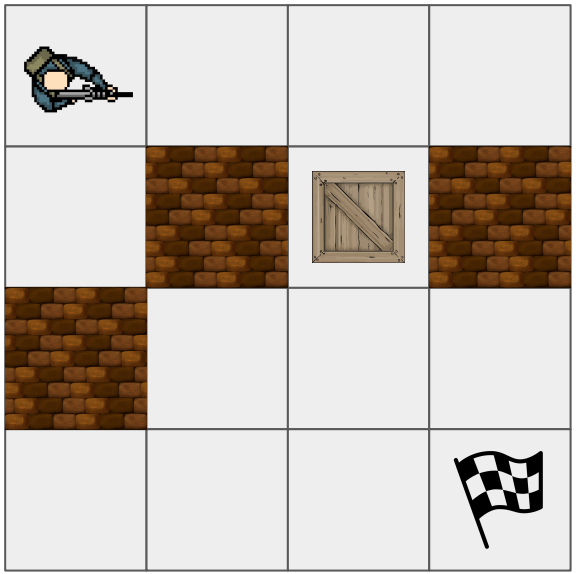
Vraag: Schrijf predicaat solve/1 dat een series states terug geeft.




```
grid(4, 4).

wall(p(1, 2)).
wall(p(2, 3)).
wall(p(4, 3)).

goal(p(4, 1)).
```

Graaf problemen (2)

Vraag: Schrijf predicaat solve/1 dat een series states terug geeft.



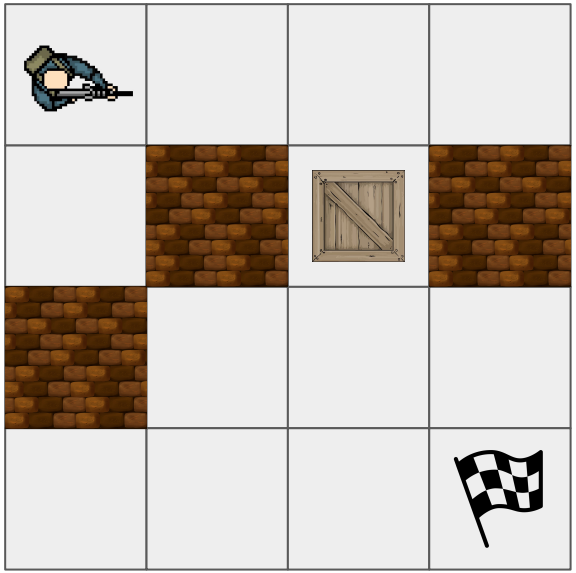
```
grid(4, 4).

wall(p(1, 2)).
wall(p(2, 3)).
wall(p(4, 3)).

goal(p(4, 1)).
```

1. Een state voorstellen:
   → 2 bewegende voorwerpen
   → Voor dit klein voorbeeldje

```
s(PlayerPos, BoxPos).
```

Graaf problemen (2)

Vraag: Schrijf predicaat solve/1 dat een series states terug geeft.



```
grid(4, 4).

wall(p(1, 2)).
wall(p(2, 3)).
wall(p(4, 3)).

goal(p(4, 1)).
```

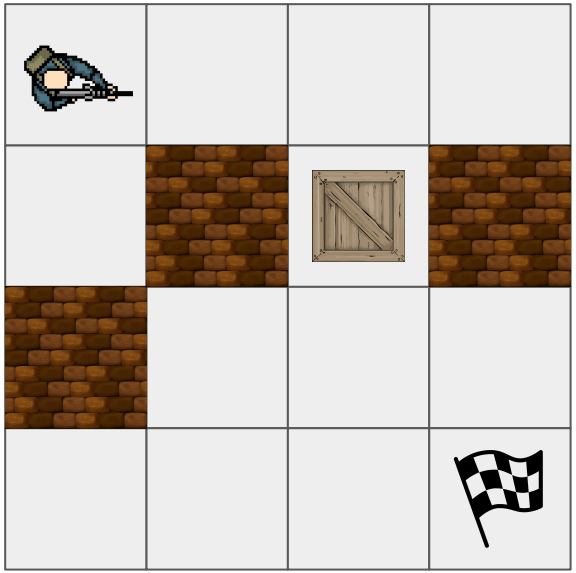
2. Begin en eind toestand

```
beginState(s(Player, Box)) :-
     Player = p(1, 4),
     Box = p(3, 3).

endState(s(P, _)) :-
     goal(P).
```

Doos mag overal zijn

Graaf problemen (2)

Vraag: Schrijf predicaat solve/1 dat een series states terug geeft.



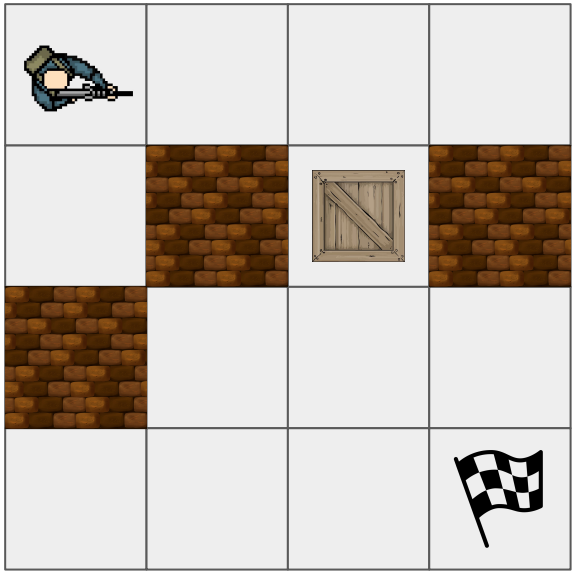
```
grid(4, 4).

wall(p(1, 2)).
wall(p(2, 3)).
wall(p(4, 3)).

goal(p(4, 1)).
```

3. Solve/1 implementeren (copy/paste)

```
solve(Path) :-
     beginState(S),
     endState(G),
     path(S, G, [S], Path).
```

Graaf problemen (2)

Vraag: Schrijf predicaat solve/1 dat een series states terug geeft.



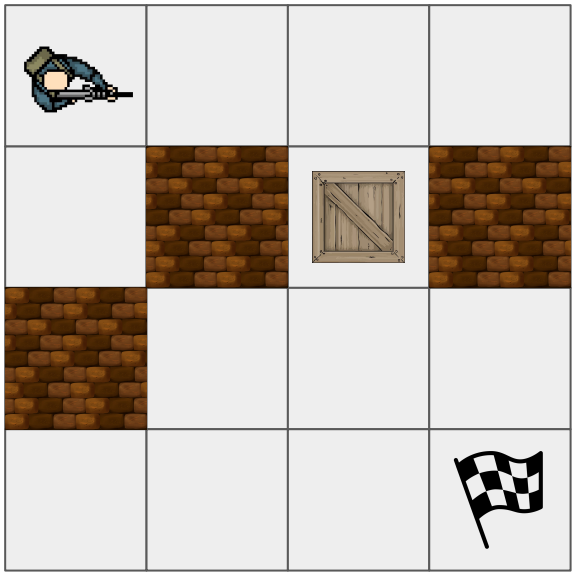
```
grid(4, 4).

wall(p(1, 2)).
wall(p(2, 3)).
wall(p(4, 3)).

goal(p(4, 1)).
```

4. path/4 implementeren (copy/paste)

```
path(F, F, Path, Path) :-
path(S, F, P, Result) :-
      transit(S, I),
      \+ member(I, P),
      path(I, F, [S|P], Result).
```

Graaf problemen (2)

Vraag: Schrijf predicaat solve/1 dat een series states terug geeft.



```
grid(4, 4).

wall(p(1, 2)).
wall(p(2, 3)).
wall(p(4, 3)).

goal(p(4, 1)).
```

5. `transit/2`
   → Player beweegt
   → Doos duwen

```
transit(S, S1) :- push(S,S1).
transit(S, S1) :- move(S,S1).

push(s(P, B), s(P, B1)) :-
     % 1. Check if player is
     %    next to box.
     % 2. Perform box move.
     % 3. Check if the box is still
     %    in a valid position.

move(s(P, B), s(P1, B)) :-
     % 1. Select left, right, up
     %    or down as possible moves.
     % 2. Perform player move.
     % 3. Check if player is still
     %    in a valid position.
```

Cut operator (1)

De cut !/0 operator zal backtracken vermijden

```
H :- G1, …, Gn, !, Gm, … .
H :- K1, …, Ks.
H :- write(“end reached”).
```

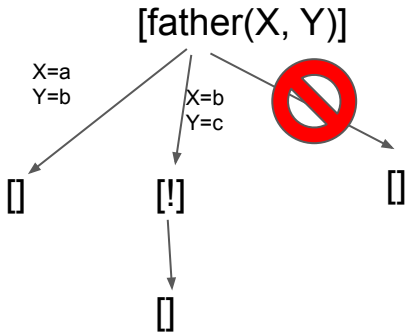
1. We backtracken **niet** meer over de goals G1, …, Gn
2. We backtracken **niet** meer over de andere regels voor H

Cut operator (2)

De cut !/0 operator zal backtracken vermijden

```
father(a, b).
father(b, c) :- !.
father(c, d).
```

```
?- father(X, Y).
X = a, Y = b
X = b, Y = c
```



Cut operator (3)

De cut !/0 operator zal backtracken vermijden:

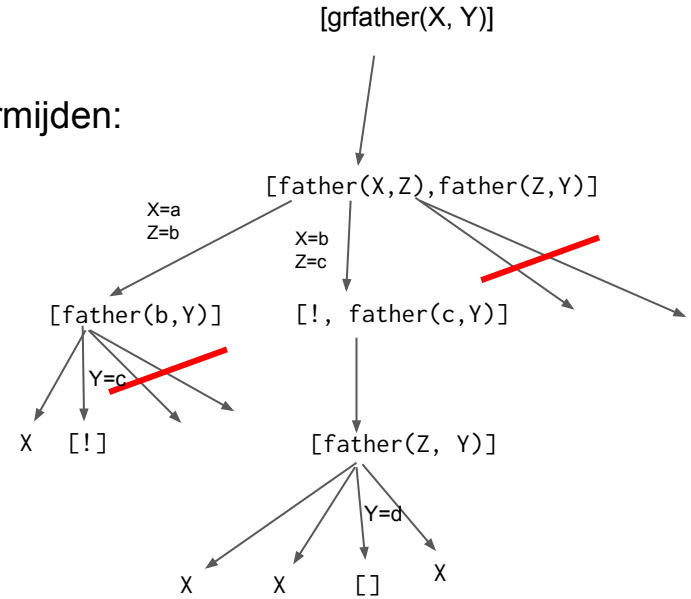
```
?- member(X, [1,2]),!,member(Y, [3,4]).
X = 1, Y = 3
X = 1, Y = 4
```

Cut operator (4)

De cut !/0 operator zal backtracken vermijden:

```
father(a, b).
father(b, c) :- !.
father(c, d).
father(d, e).

grfather(X, Y) :-
     father(X, Z),
     father(Z, Y).
```

```
?- grfather(X, Y).
X = a, Y = c
X = b, Y = d
```



Negation as failure

De not in prolog:

```
\+(P) :- P, !, fail.
\+(P) :- true.
```

Pas op met de not operator:

```
parent(jan, an).
male(jan).
male(jef).

childless(M):- male(M), \+ parent(M, _C).
childless2(M):- \+ parent(M, _C), male(M).
```

`parent(M, _)` is true als er een parent bestaat. Dus `\+ parent(M, _)` is true als er **geen** parents bestaan.

`parent(jef, _)` is true als er een object bestaat zodat jef de parent is. Dus `\+ parent(jef, _)` is true als er **geen** object bestaat zodat jef de parent is.

Core difference: initialisatie

Oefeningen

- Oefening 2 & 3 zijn **ex-examenvragen:**
  → Goed idee van echte examenvragen
  → Vrij complex, maar maak ze stapje voor stapje

- Het is niet de bedoeling om beide in 2,5 uur op te lossen

- **Volgende oefenzitting**: een individuele opgave, hier help ik niet mee.

Oefening 1.

- Gebruik `is/2` en `>/2` built-ins.
- Kijk naar de template en volg +- wat zij doen.
- Probeer te begrijpen waarom !/0 nodig is.

Oefening 2.

zoekt alle X zodat p voldaan is.

`b.v. findall(X, neighbour(N, X), L).`

Voor **check/0**:

- `findall(X, p(X), L)`
- `length(L, N)`
- Schrijf hulp predicaten: `node/1, neighbour/2, degree/2, …`
- De ‘color’ conditie kan je formuleren als …
- De ‘color’ conditie **moet niet gelden** voor node 1
- Denk na over gebruik van `\+`

∀ Node: even(Node), color(Node) ⇔ ¬∃ Node: ¬(even(Node), color(Node))

Voor **tour**:

```
tour(T) :-
     check,
     findall(X, path(..., X), L),
     sort(L, [T|_]).
```

Dit is gewoon een `path/n` probleem met extra condities (kleuren, alle edges moeten exact 1 maal worden bezocht, …)