LabJournaal

Wumpus Agent Robot

1e-jaars AI project ‘Zoeken, Sturen en Bewegen’

24 juni 2004

David de Bos
Universiteit van Amsterdam
debos@climbers.co.uk
studentnummer: 0382930

Sebastiaan de Stoppelaar
Universiteit van Amsterdam
sebastiaan.destoppelaar@planet.nl
studentnummer: 0331937

Opdracht

Het doel is om binnen 4 dagen (!) een robot te ontwerpen die het wumpus spel kan spelen[1]. De robot moet bewegen door het doolhof en waarnemingen doen, zoals een briesje of stank. Het path-planning onderdeel en het uitrekenen van de bijbehorende bewegingen kan gebeuren door een externe computer, die in verbinding staat met de robot.

De opdracht is redelijk ingewikkeled en daarom hebben we deze verdeeld in een aantal deelproblemen:

- **A) de robot**: welke robot kan voldoen aan de gestelde eisen? naast actuatoren moeten ook sensoren aanwezig zijn;
- **B) de wumpuswereld**: hoe ziet de wumpuswereld eruit? is dat een echt doolhof met kamers, briesjes en stank? of wordt gekozen voor een eenvoudiger representatie? deze keuzes hangen sterk af van de gemaakte keuzes bij onderdeel A;
- **C) planner**: er moet een programma geschreven worden dat kan redeneren waar de robot heen moet, op basis van de huidige positie en de reeds waargenomen percepties;
- **D) pilot**: er moet een programma geschreven worden dat de robot op basis van een gegeven plan kan aansturen; dit onderdeel hangt ook weer af van de gemaakte keuzes bij A;

In de volgende secties zullen we beschrijven hoe we elk onderdeel opgelost hebben, welke keuzes we daarbij gemaakt hebben en welke aannamen gedaan zijn.

Onderdeel A: de robot

De keuze was beperkt tot de drie beschikbare robots op de UvA: de Aibo van Sony, UMI-RTX (een robot-arm met grijper), en een LegoMindstorms wagentje.

In eerste instantie dachten we aan de Aibo. Deze robot leek ideaal voor het probleem: hij is goed in staat om de basisbewegingen uit te voeren (naar voren lopen, naar links/rechts draaien) en is uitgerust met een webcam om waarnemingen te doen. De robot kan door middel van een Wireless LAN communiceren met een externe computer. Ook achteraf lijkt een Aibo ons nog steeds een zeer goede optie om dit probleem aan te pakken.
De reden dat we niet voor de Aibo gekozen hebben, had meer te maken met het aantal beschikbare Aibo's op de universiteit en de beschikbare tijd. De eerste projectdag lukte het namelijk niet om de Aibo's aan te sluiten op het WirelessLAN van het RobotLab. Verder was de vraag naar Aibo's ook groter dan het beschikbare aantal.

---

[1] zie 'Artificial Intelligence – A Modern Approach', Russell and Norvig, voor een beschrijving van het spel.

De UMI-RTX was niet geschikt wegens het gebrek aan sensoren en omdat we als WumpusAgent graag een vrij-bewegende robot wilden hebben.

De keus is dus gevallen op het LegoMindstorms wagentje. Deze robot was uitgerust met drie lichtsensoren, en twee kleine motoren.

De nadelen van het wagentje:

- heeft geen mogelijkheid om real-time te communiceren met een computer; kan slechts uitgerust worden met een vijftal programma's die standalone draaien op de microprocessor van de robot;
- het beschikbare geheugen van de robot is zeer gering; een programma kan slechts 32 16-bits variabelen bevatten en mag niet groter zijn dan 5k;
- verder was de programmeertaal waarin geprogrammeerd moest worden, NQC (Not Quite C), zeer beperkt;

Omdat deze robot toch onze enige keuze was en de nadelen met zeer veel denkwerk nog wel op te lossen waren, zijn we dus hiermee aan de slag gegaan.

Zie appendix C voor de technische gegevens van de gebruikte hardware en software.

P	P		

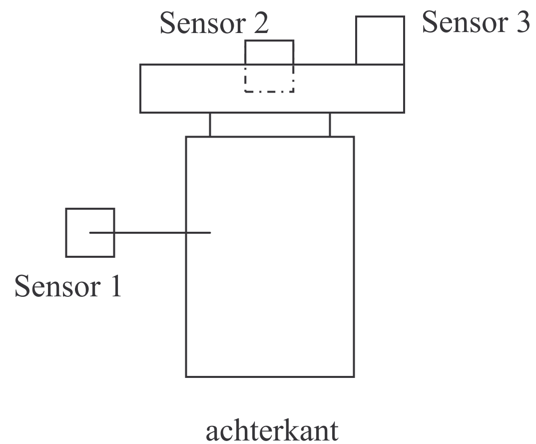
Onderdeel B: de wumpuswereld

Omdat de keuzes in onderdeel A een aantal keer gewijzigd zijn, is ons idee voor de wumpuswereld ook een aantal keer veranderd.
Voor de Aibo hadden we de volgende representatie bedacht:

- een plat bord van 4 x 4 hokjes;
- elk hokje heeft een eigen kleur: wit voor niets, geel voor een briesje, blauw voor een stank en het roze bot van de Aibo is het goud;

De sensoren op de Lego robot kunnen geen onderscheid in kleur maken en slechts een zeer gering onderscheid tussen verschillende grijswaarden. Eigenlijk kan die robot dus alleen het verschil tussen wit en zwart waarnemen.
Hierdoor hebben we besloten om het probleem te versimpelen; er bestaat geen wumpus in onze wumpuswereld (!).
Daardoor hoeft de robot slechts 1

waarneming te doen, zwart voor een briesje of wit voor niets.



De bedoeling is als volgt:

- de middelste sensor van de robot volgt steeds de zwarte lijn;
- wanneer de linker sensor een zwarte lijn detecteert (dan staat de robot dus in het midden van een hokje) stop met bewegen;
- vervolgens kan de derde sensor (aan de linker kant) een meting doen; wordt er zwart gemeten dan staat dat voor een briesje;

Onderdeel C: planner

De planner krijgt als invoer de huidige positie, de bezochte hokjes, en de gedane waarnemingen. Als uitvoer komt een reeks acties die achtereenvolgens uitgevoerd moeten worden.

De planner hadden we al voor een groot deel geschreven in Prolog. Als het kon wilden we die oplossing graag gebruiken voor deze opdracht. Met de Aibo zou dat wel kunnen door middel van een koppeling met Java, maar onze Lego robot kon dat niet. Aangezien de robot niet real-time kon communiceren met de computer moest ook de planner in NQC geprogrammeerd worden. Dat leek ons een zeer moeilijke opgave, maar niet onmogelijk.
Het probleem zat hem vooral in het beschikbare geheugen, namelijk 32 variabelen, elk van het type integer (16-bits). Met dit kleine aantal variabelen moesten we alle informatie die de robot nodig had (percepties, bezochte hokjes, positie, orientatie, gevonden pad en het agenda bij path planning) opslaan.

Alle percepties hebben we in één integer opgeslagen. Er zijn namelijk zestien hokjes, en elk hokje heeft een briesje of niet. Dat zijn dus precies 2^{15} gegevens, en dat is precies wat er in een integer past. De hokjes hebben we hiervoor genummerd van 0 tot 15, beginnend bij linksonder. Als er dus een briesje is waargenomen in de hokjes 1 en 4, dan is de waarde van perceptions in binaire notatie 10010, en dat staat gelijk aan 18.
Deze methode hebben we ook gebruikt voor de bezochte hokjes (weer 16 hokjes, met elk een ja of nee). De positie en orientatie zijn ieder één variabele.

Een pad bestaat ook maximaal uit 16 hokjes (eigenlijk zelfs 14), maar nu moet je niet alleen onthouden welke hokje, maar ook in welke volgorde. En daar hebben we lang over nagedacht maar nog geen oplossing voor gevonden. Ons laatste idee was dat je dat met 4 16-bits variabelen misschien zou kunnen opslaan. Elk hokje is namelijk te representeren

met 4 bits, dus er kunnen vier hokjes in één variabele opgeslagen worden. Het pad {0,1,2,6} (dat is van linksonder twee naar rechts en één naar boven) is dan weer te geven door: 0000 0001 0010 0110 en dat staat gelijk aan 294.

Uiteindelijk hebben we de planner niet afgekregen. De planner kan op dit moment slechts bepalen naar welk hokje gegaan moet worden, maar hoe je daar naar toe komt moet nog geprogrammeerd worden.

Onderdeel D: pilot

Voor de Aibo hebben we nooit nagedacht over een pilot omdat dat nog niet in beeld was toen we besloten van de Aibo af te stappen.
De pilot voor de Lego Robot moest ook in NQC geschreven worden. Nu was dat niet heel erg moeilijk, want dat is precies waar NQC uitermate geschikt voor is.
Ons eerste idee was dat de robot steeds een lijn moest blijven volgen tot het volgende hokje. Dit lijn-volgen hebben we als volgt geimplementeerd:

- zolang je op de lijn zit, rechtdoor rijden.
- als je van de lijn af bent, wisselend naar links en rechts draaien met een steeds grotere hoek, totdat je weer de lijn detecteerd;

Het maken van een draai is niet meer dan de ene motor vooruit zetten en de andere achteruit. Dit moet de robot net zolang doen totdat de middelste sensor weer een lijn tegenkomt.

Dit idee bleek te werken en hebben we dus ook niet aangepast.

Appendix A: dagboek

Zaterdag 19 / zondag 20 juni	
Var	Nadenken over project “Go, where no one has gone before”. Idee: Whumpus wereld nabouwen met een Aibo als agent. We hebben een representatie voor de werelde bedacht die de Aibo kan herkennen: geel vlak is een briesje uit een put; blauw vlak is stank van de wumpus; de roze Aibo kluif is het 'goud'.
Maandag 21 juni	
10:15	Introductiepraatje Arnoud Visser
11:00	Introductie van de Aibo's in het robotlab in het Sciencepark. Door wat technische storingen met de Aibo's konden we die niet gebruiken. Bovendien waren vrij gecompliceerd voor ons probleem qua beeldherkenning. Arnoud droeg de robot van Ben Bruidegom als alternatief aan.
13:00	Lunch
14:00	Afspraak met Ben Bruidegom. Hij had een Lego RCX robotje voor ons die werkte met drie lichtsensoren. Die lichtsensoren kunnen goed het verschil tussen zwart en wit waarnemen. We hebben ons Wumpus probleem opnieuw gedefinieerd. Nu is het onze bedoeling de robot een zwarte lijn te laten volgen om hem naar het volgende vakje te laten navigeren.
15:30	Thuis experimenteren met de Lego robot. Het lukte niet de juiste software te vinden om hem aan te sturen.
16:30	Einde van de dag.
Dinsdag 22 juni	
10:00	Thuis verder gezocht naar software om te communiceren met de robot. Dit lukte uiteindelijk met de Bricx C Commander.
11:00	Onderzoek naar de op C lijkende programmeertaal voor Lego RCX. We hebben verschillende tutorials doorgenomen.
12:00	Onderzoek naar code voltooid, start met programmeren van de wumpus. Prolog is niet te gebruiken, de RCX is namelijk standalone en er kan alleen een klein C programma naar verzonden worden. Dus we beginnen helemaal opnieuw met programmeren.

	We hebben ontdekt dat we maar 31 variabelen mogen gebruiken, het programma mag maar 5 kb zijn, dus we moeten heel efficiënt programmeren. Idee voor tegelrepresentatie gevonden: Twee sensors. De middelste volgt een lijn, dus linker zoekt naar een punt als stopsignaal. Uit de grijstint moet hij dan meteen de perceptie opmaken. Sensoren getest. Niet voldoende nauwkeurig voor gevoelig verschil in grijstinten.
13:00	Nieuw idee: Nog een sensor aan de rechterkant voor het perceptie gedeelte. De afstand tussen de rechtersensor en de middelste sensor is groter dan de afstand tussen de linker sensor en de middelste sensor. Anders krijg je problemen als hij de zelfde route de andere kant op neemt. We hebben de robot ook omgebouwd zodat de sensoren dichter bij de grond staan. Dan nemen ze nauwkeuriger waar.
15:00	De robot kan nu lijnen volgen. We hebben ook een testbord in elkaar geklust om de robot te kunnen testen. Alleen wanneer de robot een kwartslag draait, brengt ons stopteken de robot in de war en weet hij niet meer waar hij is. We hebben de stopsensor op een andere plek geplaatst om dit probleem op te lossen. De stop sensor zit nu in het midden van de robot. Zo herkent hij een kruispunt als punt op te stoppen.
16:00	De sensoren leveren nog wat problemen op: de perceptievlakken worden bij het draaien af en toe als de lijn aangezien. We hebben de perceptiesensor nu wat naar voren gezet. Zo zit dat vlak niet meer in de draaicirkel van de robot.
17:00	Hard bezig geweest met de wumpus software. We slaan nu gegevens binair in de variabelen op, zodat we er veel meer informatie in kwijt kunnen. Elke binaire 1 of 0 staat nu voor een boolean. In de RCX zijn namelijk alleen integers op te slaan.
18:00	Einde van de dag.
Woensdag 23 juni	
10:00	Gesprekje met Arnoud. Hij gaf aan dat het labreport uitgebreider moet. Hij rade een Wiki Site aan, wij zijn eigenwijs.
11:00	Verder programmeren aan de wumpuswereld. Het path plannen leverde wat problemen op gezien de beperkte

	mogelijkheden van de RCX programmeer taal; het gebrek aan variabelen, het gebrek aan ruimte en het gebrek aan functionaliteit. Een functie kan bijvoorbeeld geen waarde retouneren. Wij denken hier een oplossing voor gevonden hebben. We hebben het 'bewegen' van de robot nog wat verbeterd. Af en toe raakte bij het draaien van de robot de sensoren in de war van de lijnen. We hebben alle plaatsen subtiel gewijzigd om die problemen te verhelpen.
13:00	Lunch.
14:00	David is verder gaan programmeren op de UvA. Sebastiaan is gaan werken aan het verslag om aan de wensen van Arnoud te voldoen.
	Einde van de dag.

Het dagboek hebben we na woensdag niet meer strikt bijgehouden.

Appendix B: Broncode

Planner geschreven in Prolog

```
:- dynamic visited/2.   % alle hokjes die je bezocht hebt: visited(X,Y)
:- dynamic fringe/2.    % alle hokjes die je nog kan bezoeken: fringe(X,Y)
:- dynamic percepts/3.  % alle percepties die je al gedaan hebt:
percepts(X,Y,P)

:- dynamic settings/3.
% hierin wordt opgeslagen:
% - de agents world_size: settings(my_world_size,X,Y)
% - positie:  settings(position,X,Y)
% - orientatie:  settings(orientation,OX,OY)
% de orientatie werkt als volgt: naar rechts is 1,0; naar links is -1,0;
% naar boven is 0,1 en naar beneden is 0,-1

% ook object is dynamic want de wumpus kan gedood worden
:- dynamic object/4.

% of je nog een pijl hebt of niet;
:- dynamic arrow.

% de wereld
world_size(4,4).
world([[2,3,p],[2,2,a],[2,2,w],[1,2,p]]).
% er kunnen meerdere objecten op 1 positie zijn (zie voorbeeld hierboven);

% alle objecten, met hoe je ze kan waarnemen en in welke kamers
% stereo betekent waarneembaar in aanliggende kamers
% mono is alleen in de eigen kamer waarneembaar
```

```
% laatste argument geeft aan hoeveel er maximaal in de wereld mogelijk zijn
% 0 voor onbeperkt
% elk gevaarlijk object heeft de perception death in dezelfde kamer (mono)
object(w,s,stereo,1).
object(p,b,stereo,0).
object(w,death,mono,_).
object(p,death,mono,_).
object(a,g,mono,1).

% wat is het doel van het spelletje? goud grabben!
goal_action(grab).

% restart het spel
restart :-
	X=1, Y=1, OX=1, OY=0,
	retractall(visited(_,_)),
	retractall(fringe(_,_)),
	retractall(percepts(_,_,_)),
	insert(arrow),
	insert(object(w,s,stereo,1)),
	insert(object(w,death,mono,_)),
	update(my_world_size,999,999),
	update(position,X,Y),
	update(orientation,OX,OY),
	insert_percepts(X,Y),
	update_fringes(X,Y),
	insert(visited(X,Y)).

% find-gold: restart, roept de agent aan
% en geeft de positie van het goud terug+de gemaakte acties
find-gold([X,Y],Actions) :-
	restart,
	agent([],RevActions), !,
	reverse(RevActions,Actions),
	settings(position,X,Y).

%--------------------------------------
% Er is een zevental Agent predicaten:
% - goal-actie heb je uitgevoerd, klaar is kees;
% - je ziet goud, pak het dan;
% - je bent doodgegaan, gebruik een nieuw leven;
% - je hebt een plan, voer dat uit;
% - je weet waar de wumpus is, schiet 'm neer;
% - er is nog een provable safe square, ga daar heen;
% - er is nog een possible safe square, ga daar heen;

% wrapper
agent(Plan,Actions) :-
	agent(Plan,[],Actions).

% als de laatste actie de goal-actie was dan is ie klaar
agent(_,[Goal|Actions],[Goal|Actions]) :-
	goal_action(Goal), !.

% als je plan verschilt van grab en je ziet iets glinsteren, pak het dan!
agent(Plan,AccActions,Actions) :-
	Plan \== [grab],
	settings(position,X,Y),
	percepts(X,Y,g), !,
	agent([grab],AccActions,Actions).

% als je plan verschilt van extralife en je gaat dood, gebruik dan een extra
leven!
```

```
agent(Plan,AccActions,Actions) :-
       Plan \== [extralife],
       settings(position,X,Y),
       percepts(X,Y,death), !,
       agent([extralife],AccActions,Actions).

% als de agent een plan heeft, voer dan het plan uit
agent([P|Lan],AccActions,Actions) :-
       action(P), !,
       agent(Lan,[P|AccActions],Actions).

% is er een fringe-square waarvan je zeker weet dat
% de wumpus er is en je hebt nog een pijl, ga dan
% op moord expeditie en kill the wumpus!
agent([],AccActions,Actions) :-
       fringe(NX,NY),
       ask(NX,NY,w),
       arrow,
       prepare_to_shoot(NX,NY,RevPlan), !,
       reverse(RevPlan,Plan),
       agent(Plan,AccActions,Actions).

% is er een fringe-square waarvan je zeker weet
% dat er zowel geen wumpus als geen put is?
% ga dan naar dat hokje toe
agent([],AccActions,Actions) :-
       fringe(NX,NY),
       ask(NX,NY,and(niet(p),niet(w))), !,
       find_route(NX,NY,RevPlan),
       reverse(RevPlan,Plan),
       agent(Plan,AccActions,Actions).

% is er anders een fringe-square waarvan je niet zeker weet
% dat er een wumpus of een put is?
% ga dan naar dat hokje toe (een gokje wagen);
agent([],AccActions,Actions) :-
       fringe(NX,NY),
       \+ ask(NX,NY,or(p,w)), !,
       find_route(NX,NY,RevPlan),
       reverse(RevPlan,Plan),
       agent(Plan,AccActions,Actions).

%-----------------------------------

% er zijn verschillende forwards:

% je gaat dood....neem dan niets anders waar dan de dood
action(forward) :-
       settings(orientation,OX,OY),
       settings(position,X,Y),
       NewX is X + OX,
       NewY is Y + OY,
       get_neighbour(X,Y,NewX,NewY),
       percept(NewX,NewY,death),
       insert(percepts(NewX,NewY,death)),
       update(position,NewX,NewY),
       retractall(fringe(NewX,NewY)), !.

% je ging naar voren en je kon naar voren
action(forward) :-
       settings(orientation,OX,OY),
       settings(position,X,Y),
       NewX is X + OX,
```

```
        NewY is Y + OY,
        get_neighbour(X,Y,NewX,NewY), !,
        update(position,NewX,NewY),
        insert_percepts(NewX,NewY),
        update_fringes(NewX,NewY),
        insert(visited(NewX,NewY)).

% je ging naar boven maar je bumpte: update je world_size
action(forward) :-
        settings(orientation,_,1), !,
        settings(position,_,Y),
        settings(my_world_size,X,_),
        FY is Y + 1,
        retractall(fringe(_,FY)),
        update(my_world_size,X,Y).

% je ging naar opzij maar je bumpte: update je world_size
action(forward) :-
        settings(orientation,1,_),
        settings(position,X,_),
        settings(my_world_size,_,Y),
        FX is X + 1,
        retractall(fringe(FX,_)),
        update(my_world_size,X,Y).

action(forward).

% draait links
action(turnleft) :-
        settings(orientation,X,Y),
        NewX is -Y,
        update(orientation,NewX,X).

% draait rechts
action(turnright) :-
        settings(orientation,X,Y),
        NewY is -X,
        update(orientation,Y,NewY).

% bij grab is het spelletje afgelopen
action(grab).

% een extra leven houdt in:
% - een stap terugdoen
action(extralife) :-
        settings(position,X,Y),
        settings(orientation,OX,OY),
        NewX is X - OX,
        NewY is Y - OY,
        update(position,NewX,NewY).

% shoot:
% heb je nog een pijl, en je kijkt in de richting van de wumpus
% verwijder dan alle wumpus-dingen (de wumpus zelf en alle stenches)
action(shoot) :-
        arrow,
        settings(position,X,Y),
        settings(orientation,OX,OY),
        world(W),
        member([WX,WY,w],W),
        same_direction(X,Y,OX,OY,WX,WY), !,
        retractall(object(w,_,_,_)),
        retractall(percepts(_,_,s)),
```

```
        retractall(percepts(WX,WY,death)),
        retractall(arrow).

% anders was het mis, ben je alleen je pijl kwijt
action(shoot) :-
        retractall(arrow).

% update verwijdert een bepaalde setting en voegt de nieuwe in
update(Pred,X,Y) :-
        retractall(settings(Pred,_,_)),
        assert(settings(Pred,X,Y)).

% insert voegt iets toe aan de database als het er nog niet in staat
insert(X) :-
        \+ X,
        asserta(X).

insert(_).

% prepare_to_shoot = find_route maar dan de eerste niet een forward, maar een
shoot!
prepare_to_shoot(GoalX,GoalY,[shoot|Actions]) :-
        find_route(GoalX,GoalY,[forward|Actions]).

% find_route zoekt dmv breadth-first een route met alleen kamers
% daarna worden met get_actions de daaraan gekoppelde acties
% gegenereerd (zoals turnleft, forward, etc).
find_route(GoalX,GoalY,Actions) :-
        settings(position,X,Y),
        solve_bf(X,Y,[[[GoalX,GoalY]]],Path), !,
        settings(orientation,OX,OY),
        get_actions(Path,OX,OY,Actions), !.

%----------------------------------------
% standaard breadth-first zoek algoritme

solve_bf(X,Y,[[[X,Y]|Path]|_],[[X,Y]|Path]).

solve_bf(GoalX,GoalY,[Path|Paths],Solution) :-
        extend(Path,NewPaths),
        append(Paths,NewPaths,Paths1),
        solve_bf(GoalX,GoalY,Paths1,Solution).

extend([[X,Y]|Path],NewPaths) :-
        findall(
                [[NX,NY],[X,Y]|Path],
                (
                        get_visited_neighbour(X,Y,NX,NY),
                        \+ member([NX,NY],Path)
                ),
                NewPaths
        ).
%----------------------------------------

% get_actions geeft alle acties nodig om het pad
% te doorlopen;
get_actions(Path,OX,OY,Actions) :-
        get_actions(Path,OX,OY,[],Actions).

get_actions([_],_,_,Actions,Actions).

get_actions([[X,Y],[X1,Y1]|T],OX,OY,AccActions,Actions) :-
        do_step(X,Y,OX,OY,X1,Y1,NewOX,NewOY,NewActions),
```

```
	append(NewActions,AccActions,NewActions1),
	get_actions([[X1,Y1]|T],NewOX,NewOY,NewActions1,Actions).

% simuleert 1 stap (van ene hokje naar andere hokje):
% - draai eerst je gezicht goed
% - ga daarna naar voren
do_step(X,Y,_,_,X,Y,_,_,[]) :- !.
do_step(X,Y,OX,OY,NewX,NewY,NewOX,NewOY,[forward|Actions]) :-
	NewOX is NewX - X,
	NewOY is NewY - Y,
	change_orientation(OX,OY,NewOX,NewOY,Actions).

% bepaalt een actie om je orientatie te wijzigen
change_orientation(X,Y,X,Y,[]).
change_orientation(X,Y,NewX,X,[turnleft]) :-   NewX is -Y.
change_orientation(X,Y,Y,NewY,[turnright]) :-  NewY is -X.
change_orientation(X,Y,NewX,NewY,[turnleft,turnleft]) :- X = NewX; Y = NewY.

% bepaalt of je in dezelfde richting kijkt
same_direction(X,Y,OX,OY,X1,Y1) :-
	same_dir(X,Y,OY,X1,Y1);
	same_dir(Y,X,OX,Y1,X1).

same_dir(X,Y,OR,X1,Y1) :-
	0 is X1-X,
	DifY is Y1 - Y,
	DifY =\= 0,
	Temp is OR / DifY,
	Temp > 0.

% als je death waarneemt mag je niets anders waarnemen;
insert_percepts(X,Y) :-
	\+ visited(X,Y),
	percept(X,Y,death), !,
	insert(percepts(X,Y,death)).

% anders alle percepts opvragen dmv findall en toevoegen;
insert_percepts(X,Y) :-
	\+ visited(X,Y), !,
	findall(
		[X,Y,Percept],
		(
			percept(X,Y,Percept),
			insert(percepts(X,Y,Percept))
		),
		_
	).

insert_percepts(_,_).

% als je aankomt op positie X,Y wordt die kamer
% verwijdert uit de 'lijst' fringes;
% alle aanliggende kamers die je nog niet bezocht hebt
% worden toegevoegd aan de fringes (bij insert/1 wordt
% er rekening gehouden of die fringe niet al bestaat);

update_fringes(X,Y) :-
	retractall(fringe(X,Y)),
	findall(
		[X,Y],
		(
			get_my_neighbour(X,Y,NewX,NewY),
			\+ visited(NewX,NewY),
```

```
                    insert(fringe(NewX,NewY))
              ),
              _
       ).

% percept wordt door 'god' teruggegeven aan de hand van het wereldmodel
percept(X,Y,P) :-
       world(W),
       object(O,P,stereo,_),
       get_neighbour(X,Y,X1,Y1),
       member([X1,Y1,O],W).

percept(X,Y,P) :-
       world(W),
       object(O,P,mono,_),
       member([X,Y,O],W).

% je weet zeker dat er niets is als je er geweest bent
model(niet,X,Y,_) :-
       visited(X,Y), !.

% je weet zeker dat het object er niet is als het object niet (meer) bestaat
model(niet,_,_,O) :-
       \+ object(O,_,_,_).

% je weet zeker dat het object er niet is als het mono is en je niet in
% dezelfde kamer de bijbehorende percept hebt gedaan
model(niet,X,Y,O) :-
       object(O,P,mono,_),
       visited(X,Y),
       \+ percepts(X,Y,P).

% tenslotte weet je zeker dat het object er niet is als het stereo is en je
niet in
% alle aanliggende kamers dezelfde bijbehorende percept hebt gedaan
model(niet,X,Y,O) :-
       object(O,P,stereo,_),
       findall([NX,NY],get_visited_neighbour(X,Y,NX,NY),N),
       \+ shag_rooms_for_p(N,P).

% object O is er als je de perceptie daar waarneemt (mono)
model(wel,X,Y,O) :-
       object(O,P,mono,_),
       P \== death,
       percepts(X,Y,P).

% object O is er als alle omringende kamers de juiste perceptie hebben
% en die alleen voor X,Y kunnen zijn; (stereo en oneindig veel objecten)
model(wel,X,Y,O) :-
       object(O,P,stereo,0),
       bagof([NX,NY],get_visited_neighbour(X,Y,NX,NY),N),
       shag_rooms_for_p_xp(N,X,Y,P,O).

% maximaal 1 stereo object (zoals de wumpus): vind alle percepties en
% zoek de dubbele buren; verwijder alle buren waarvan je zeker weet dat
% object O er niet is; hou je 1 buur over, dan weet je dat daar object O is
model(wel,X,Y,O) :-
       object(O,P,stereo,1),
       bagof([SX,SY],percepts(SX,SY,P),S),
       get_doubles(S,D),
       shag_doubles(D,O,[[X,Y]]).

% get_doubles geeft alle 'overlappende' buren van een aantal kamers;
```

```
get_doubles(S,D) :-
	get_doubles(S,[],D).

get_doubles([],D,D).

get_doubles([[X,Y]|T],[],D) :-
	bagof([NX,NY],get_my_neighbour(X,Y,NX,NY),N),
	get_doubles(T,N,D).

get_doubles([[X,Y]|T],AccD,D) :-
	bagof([NX,NY],(get_my_neighbour(X,Y,NX,NY),member([NX,NY],AccD)),N),
	get_doubles(T,N,D).

% shag_doubles verwijdert uit alle 'dubbele buren', de kamers waarvan je zeker
% weet dat object O er niet is;
shag_doubles(D,O,NewD) :-
	shag_doubles(D,O,[],NewD).

shag_doubles([],_,D,D).

shag_doubles([[X,Y]|T],O,AccD,D) :-
	ask(X,Y,niet(O)), !,
	shag_doubles(T,O,AccD,D).

shag_doubles([[X,Y]|T],O,AccD,D) :-
	shag_doubles(T,O,[[X,Y]|AccD],D).

% shag_rooms_for_p slaagt als alle kamers dezelfde percept P hebben
shag_rooms_for_p([],_).
shag_rooms_for_p([[X,Y]|T],P) :-
	percepts(X,Y,P),
	shag_rooms_for_p(T,P).

% shag_rooms_for_p_xp slaagt als alle kamers dezelfde percept P hebben
% en ze alleen maar voor het betreffende hokje (OldX,OldY) kunnen zijn
shag_rooms_for_p_xp([],_,_,_,_).
shag_rooms_for_p_xp([[X,Y]|T],OldX,OldY,P,O) :-
	percepts(X,Y,P),
	bagof(
		[NX,NY],
		(
			get_my_neighbour(X,Y,NX,NY),
			[NX,NY] \== [OldX,OldY]
		),
		N
	),
	shag_rooms_for_o(N,O),
	shag_rooms_for_p_xp(T,OldX,OldY,P,O).

% slaagt als alle kamers NIET object O hebben
shag_rooms_for_o([],_).
shag_rooms_for_o([[X,Y]|T],O) :-
	object(O,_,stereo,_),
	model(niet,X,Y,O),
	shag_rooms_for_o(T,O).

%----------------------------
% neighbour-predicaten

% geeft dmv backtracken alle buren van X,Y
% dit predicaat wordt niet door de agent gebruikt
% maar alleen door 'god' die bijvoorbeeld aan de
% agent een perceptie moet doorgeven;
```

```
get_neighbour(X,Y,X1,Y1) :-
       world_size(W,H),
       (
       X > 1, X1 is X-1, Y1=Y;
       X < W, X1 is X+1, Y1=Y;
       X1=X, Y > 1, Y1 is Y-1;
       X1=X, Y < H, Y1 is Y+1
       ).

% geeft dmv backtracken alle buren van X,Y
% voor zover de agent kan weten; hij gaat
% namelijk uit van zijn wereld-grootte en
% die kan afwijken van de werkelijke grootte
get_my_neighbour(X,Y,X1,Y1) :-
       settings(my_world_size,W,H),
       (
       X > 1, X1 is X-1, Y1=Y;
       X < W, X1 is X+1, Y1=Y;
       X1=X, Y > 1, Y1 is Y-1;
       X1=X, Y < H, Y1 is Y+1
       ).

% geeft dmv backtracken alle buren van X,Y
% die je al eerder bezocht hebt;
get_visited_neighbour(X,Y,X1,Y1) :-
       (
       X1 is X-1, Y1=Y;
       X1 is X+1, Y1=Y;
       X1=X, Y1 is Y-1;
       X1=X, Y1 is Y+1
       ),
       visited(X1,Y1).

%------------------------------

% ask predicaten spreken voor zich
% je kan alleen iets opvragen over objecten en een niet-operator
% werkt alleen als het direct voor een object staat;
% dat is namelijk voldoende functioneel in de wumpus-wereld;

% LET OP: er wordt expliciet onderscheid gemaakt tussen 'wel' en 'niet'
% model(wel,X,Y,P) slaagt als je 100% zeker weet dat object P op X,Y zich
bevindt
% model(niet,X,Y,P) slaagt als je 100% zeker weet dat object P zich NIET op X,Y
bevindt

ask(X,Y,niet(P)) :-
       model(niet,X,Y,P).

ask(X,Y,P) :-
       model(wel,X,Y,P).

ask(X,Y,or(P,Q)) :-
       ask(X,Y,P);
       ask(X,Y,Q).

ask(X,Y,and(P,Q)) :-
       ask(X,Y,P),
       ask(X,Y,Q).
```

Planner geschreven in NQC

```
/* wumpus.nqc

Program for a lego robot to play the wumpus game

Representation for the maze:
-----------------
| 12| 13| 14| 15|
|---|---|---|---|
| 8 | 9 | 10| 11|
|---|---|---|---|
| 4 | 5 | 6 | 7 |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
-----------------
*/

int pos;                 // your position
int dir;                 // your direction
int perceptions;         // all the perceptions done at a square
int visited;             // if you have visited the square
int fringes;             // not visited squares adjacent to visited squares

int buf1, buf2, buf3;    // buffers for anything you want
int i, j, k;             // counting variables
int bool1, bool2;                // two boolean variables
int result;

task main()
{
  pos = 0;           // start square is (0)
  dir = 1;           // you look to the right
  perceptions = 0;   // there are no perceptions on start-square;
  visited = 1;       // you have visited (0) --> 1 == 1;
  fringes = 18;      // the fringes are (4) and (1) --> 10010 = 18;

  do_move();          // do a move;
  get_perceptions(); // get perceptions after the move has been done;
  update_fringes();  // update the fringe squares
}

sub do_move()
{
  // Select a fringe from the list
  i = 0;
  buf1 = fringes;
  CreateDatalog(15);
  bool1 = 0;
  while (buf1 > 0) {
    if ((buf1 % 2) == 1)   // fringe found!
    {
      // Try to determine if the fringe-square is pit-free:
      // that is: one of the visited neighbours is breeze-free!

      // So: get a visited square!
      j = 0;
      buf2 = visited;
      while (buf2 > 0) {
        if ((buf2 % 2) == 1)   // visited square found!
        {
          if ( (j-4==i) ||
```

```
                (j+4==i) ||
                ((j-1==i) && (i % 4 < 3)) ||
                ((j+1==i) && (j % 4 < 3))) // if it is a neighbour
          {
            // Find perceptions
            k = 0;
            buf3 = perceptions;
            bool2 = 0;
            while (buf3 > 0) {
              // if percetion on square k == current square j
              if ((buf3 % 2 == 1) && (k == j)) {  // perception on square
found!
                bool2 = 1;
                break;
              }
              buf3 /= 2;
              k++;
            }
            if (bool2 == 0) {
              // if no perception was found then plan a path to fringe square i

              // HERE COMES PATH PLANNING FROM pos TO i


              // END PATH PLANNING

              // CONVERT PATH TO MOVES

              // UPDATE DIRECTION AND POSITION



              bool1 = 1;  // you can break out of the fringe loop
              break; // you can break out of the neighbours loop
            }
          }
        }
        buf2 /= 2;
        j++;
      }
      // no 100% safe square was found!
    }
    if (bool1 == 1) break;
    buf1 /= 2;
    i++;
  }
}
```

Pilot geschreven in NQC

```
#define THRESHOLD 750

int time;
int ready;

task main()
{
  SetPower(OUT_A+OUT_C,2);
  SetSensor(SENSOR_1,SENSOR_LIGHT);
  SetSensorMode(SENSOR_1,SENSOR_MODE_RAW);
```

```
  SetSensor(SENSOR_2,SENSOR_LIGHT);
  SetSensorMode(SENSOR_2,SENSOR_MODE_RAW);

  forward();
  turn_left();
  forward();
  forward();
  turn_right();
  forward();
  turn_right();
  forward();
}

sub forward() {
  start follow_line;
  until (ready > 0) {}
  OnFwd(OUT_A+OUT_C); Wait(20); Off(OUT_A+OUT_C);
}

task turn_to_line()
{
  stop follow_line;
  time = 0;
  OnFwd(OUT_C); OnRev(OUT_A);
  while (true)
  {
    repeat(time)
    {
      Wait(1);
      if (SENSOR_1 >= THRESHOLD) // stop sign!
      {
        Off(OUT_C+OUT_A);
        stop follow_line;
      }
      else if (SENSOR_2 >= THRESHOLD)
       start follow_line;
    }
    SetDirection(OUT_A+OUT_C,OUT_TOGGLE);
    time++;
  }
}

task follow_line()
{
  ready = 0;
  stop turn_to_line;
  OnFwd(OUT_A+OUT_C);
  until (SENSOR_1 >= THRESHOLD && SENSOR_2 >= THRESHOLD) // stop sign!
    if (SENSOR_2 <= THRESHOLD)        // check if you're on the track!
      start turn_to_line;
  Off(OUT_C+OUT_A);
  ready = 1;
}

sub turn_left()
{
  ready = 0;
  SetPower(OUT_A+OUT_C,4);
  OnRev(OUT_A); OnFwd(OUT_C); Wait(50);
  while (SENSOR_2 < THRESHOLD) {}
  Off(OUT_A+OUT_C);
  SetPower(OUT_A+OUT_C,2);
  ready = 1;
```

```
}

sub turn_right()
{
  SetPower(OUT_A+OUT_C,4);
  OnFwd(OUT_A); OnRev(OUT_C); Wait(50);
  while (SENSOR_2 < THRESHOLD) {}
  Off(OUT_A+OUT_C);
  SetPower(OUT_A+OUT_C,2);
}
```

Appendix C: hardware en software

Gebruikte hardware:

- Lego steentjes
- Lego RCX 1.0

Gebruikte software:

- SWI-PROLOG, versie 5.2.10
- Bricx Command Center, versie 3.3