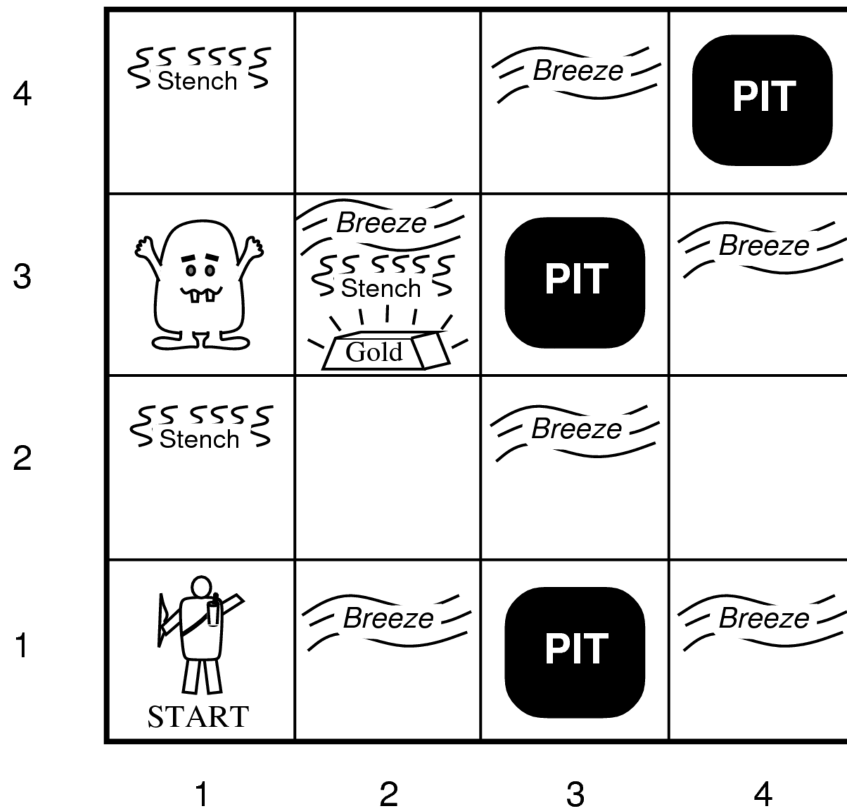
ALPs with Sensing: The Wumpus Agent



The agent can

- turn 90° (counter-) clockwise
- go forward to adjacent cell
- grab gold
- shoot its one arrow in any direction
- exit through cell (1,1)

The agent had better not

- enter a cell with a pit
- meet the Wumpus, unless shot dead

Percepts: stench, breeze, glitter, scream

Task: Explore Environment and Collect Gold

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2 **OK**	2,2	3,2	4,2
1,1 **A** **OK**	2,1 **OK**	3,1	4,1

(a)

- **A** = *Agent*
- **B** = *Breeze*
- **G** = *Glitter, Gold*
- **OK** = *Safe square*
- **P** = *Pit*
- **S** = *Stench*
- **V** = *Visited*
- **W** = *Wumpus*

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2 **OK**	2,2 **P?**	3,2	4,2
1,1 **V** **OK**	2,1 **A** **B** **OK**	3,1 **P?**	4,1

(b)

Combining Sensor Information

1,4	2,4	3,4	4,4
1,3 **W!**	2,3	3,3	4,3
1,2 **A** **S** **OK**	2,2 **OK**	3,2	4,2
1,1 **V** **OK**	2,1 **B** **V** **OK**	3,1 **P!**	4,1

A = *Agent*
B = *Breeze*
G = *Glitter, Gold*
OK = *Safe square*
P = *Pit*
S = *Stench*
V = *Visited*
W = *Wumpus*

Encoding Incomplete State Information

Incomplete states can be encoded with the help of constraints

```
init(Z0) :- Z0 = [at(1,1),facing(north),has(arrow) | Z],
            not_holds(dead,Z),
            not_holds(has(gold),Z),
            not_holds(wumpus(1,1),Z),
            not_holds(pit(1,1),Z),

            holds(gold(GX,GY),Z), [GX,GY] :: 1..4,

            holds(wumpus(WX,WY),Z,Z1), [WX,WY] :: 1..4,
            not_holds_all(wumpus(WX1,WX2),Z1),

            duplicate_free(Z0).
```

Arithmetic Constraints – Standard `lib(fd)`

- arithmetic functions `+, -, *`
- equality, inequality, ordering constraints `#=, #\=, #>=, ...`
- range constraints `X :: m..n, X :: [a,b,c,...]`
- logical connectives `#\/, #/\`

```
?- X :: 1..4, X #> 2*Y, Y #>=1,
   W :: [c1,c2,c3], W#\=c1 #/\ W#\=c3.

X = X{3,4}
Y = 1
W = c2
```

State Constraints

- negation `not_holds(F,Z), not_holds_all(F,Z)`
- disjunction `or_holds([F1,...,Fn],Z)`
- single fluent occurrence `duplicate_free(Z)`

(Other constraints may be defined and used.)

Constraint Handling Rules

Constraint Handling Rules (CHRs)

simplification rule: $H \Leftrightarrow B$
propagation rule: $H \Rightarrow B$

(where head H constraint and body B sequence of atoms)

Operational Semantics: States

Let G goal (sequence of atoms), C constraint (store).

- $\langle G; C \rangle$ state
- $\langle G; \mathit{true} \rangle$ initial state
- $\langle \square; C \rangle$ successful final state
- $\langle G; \mathit{false} \rangle$ failed final state

CHR State Transitions

Simplify

If $H \Leftrightarrow B$ is a fresh variant of a CHR with variables $\vec{x}$

and $\models (\forall)(C \supset (\exists \vec{x})\ F = H)$

then $\langle F, G; C \rangle \mapsto \langle B, G; (F{=}H) \wedge C \rangle$

Propagate

If $H \Rightarrow B$ is a fresh variant of a CHR with variables $\vec{x}$

and $\models (\forall)(C \supset (\exists \vec{x})\ F = H)$

then $\langle F, G; C \rangle \mapsto \langle F, B, G; (F{=}H) \wedge C \rangle$

CHRs for the State Constraints

```
not_holds(_,[])      <=> true.
not_holds(F,[F1|Z]) <=> neq(F,F1), not_holds(F,Z).

not_holds_all(_,[])      <=> true.
not_holds_all(F,[F1|Z]) <=> neq_all(F,F1), not_holds_all(F,Z).

duplicate_free([])     <=> true.
duplicate_free([F|Z]) <=> not_holds(F,Z), duplicate_free(Z).
```

where `neq(F,F1)` and `neq_all(F,F1)` are auxiliary predicates

Example 1

```
?- not_holds(pit(3,X), [pit(Y,3), pit(3,4) | Z],
   X :: 3..4, Y :: 2..3.

X = 3
Y = 2

Constraints:
not_holds(pit(3,3), Z)
```

```
?- not_holds_all(pit(X,X), [pit(Y,4),gold(3,3) | Z],
   Y :: 1..4.

Y = Y{1..3}

Constraints:
not_holds_all(pit(X,X), Z)
```

Example 2 (cf Slide 4)

```
?- init(Z0).

Z0 = [at(1,1),facing(north),has(arrow),
      gold(GX,GY),wumpus(WX,WY) | Z],
GX = GX{1..4}
GY = GY{1..4}
WX = WX{1..4}
WY = WY{1..4}

Constraints:
not_holds(dead,Z),
...
not_holds(pit(1,1),Z),
not_holds_all(wumpus(WX1,WY1),Z),
not_holds(at(1,1),Z),
...
not_holds(gold(GX,GY),Z),
duplicate_free(Z)
```

Recap: The Need to Reason Logically

1,4	2,4	3,4	4,4
1,3 W!	2,3	3,3	4,3
1,2 A S OK	2,2 OK	3,2	4,2
1,1 V OK	2,1 B V OK	3,1 P!	4,1

A = *Agent*
B = *Breeze*
G = *Glitter, Gold*
OK = *Safe square*
P = *Pit*
S = *Stench*
V = *Visited*
W = *Wumpus*

Logical Reasoning with Constraints

```
sf(sense_stench,Percept,Z) :-
   holds(at(X,Y),Z),
   XE#=X+1, XW#=X-1, YN#=Y+1, YS#=Y-1,
   ( Percept=false -> not_holds(wumpus(XE,Y),Z),
                      not_holds(wumpus(XW,Y),Z),
                      not_holds(wumpus(X,YN),Z),
                      not_holds(wumpus(X,YS),Z) ;
     Percept=true ->
       or_holds([wumpus(XE,Y),wumpus(X,YN),
                 wumpus(XW,Y),wumpus(X,YS)],Z) ).
```

```
?- Z3 = [at(2,1),wumpus(WX,WY) | Z],
   (WX#=1 #/\ WY#=3) #\/ (WX#=2 #/\ WY#=2),
   sf(sense_stench,false),Z3).

Z3 = [at(2,1),wumpus(1,3) | Z]
```

Limitations

Conjunctions and disjunctions in `lib(fd)` only evaluated if one of the atomic constraints has been decided, for the sake of efficiency (linear complexity).

```
?- WX,WY :: 3..4, Z = [wumpus(WX,WY)],
   not_holds(wumpus(3,4),Z),
   not_holds(wumpus(4,3),Z),
   not_holds(wumpus(4,4),Z).

Z = [wumpus(WX,WY)]

Constraints:
WX#\=3 #\/ WY#\=4
WX#\=4 #\/ WY#\=3
WX#\=4 #\/ WY#\=4
```

Remark: Using a 1-dimensional encoding of the cave helps.

Updating Incomplete States

- Consider the incomplete state knowledge

$$Holds(F(y),z) \wedge [Holds(F(A),z) \vee Holds(F(B),z)]$$

What can the agent know about the updated state $z' = z - F(A)$?

- Consider the incomplete state knowledge

$$[Holds(F(x),z) \vee Holds(F(A),z)] \wedge \neg Holds(F(B),z)$$

What can the agent know about the updated state $z' = z + F(x)$?

Updating Incomplete States in Prolog

```
minus(Z, [], Z).

minus(Z, [F|Fs], Z2) :- ( not not_holds(F,Z) -> holds(F,Z,Z1) ;
                            not     holds(F,Z) -> Z1=Z ;
                            cancel(F,Z,Z1), not_holds(F,Z1)
                          ),
                          minus(Z1, Fs, Z2).


plus(Z, [], Z).

plus(Z, [F|Fs], Z2) :- ( not     holds(F,Z) -> Z1=[F|Z] ;
                           not not_holds(F,Z) -> Z1=Z ;
                           cancel(F,Z,Zp), Z1=[F|Zp], not_holds(F,Z1)
                         ),
                         plus(Z1, Fs, Z2) ).
```

where `cancel(F,Z,Z1)` is an auxiliary predicate

State Update Axioms for the Wumpus Agent

```
state_update(Z1,turn_left,Z2) :-
   holds(facing(D),Z1),
   D#=north #/\ D1#=west #\/ ...,
   update(Z1,[facing(D1)],[facing(D)],Z2).

state_update(Z1,turn_right,Z2) :-
   holds(facing(D),Z1),
   (D#=north #/\ D1#=east) #\/ ...,
   update(Z1,[facing(D1)],[facing(D)],Z2).

state_update(Z1,go,Z2) :-
   holds(at(X,Y),Z1),
   holds(facing(D),Z1),
   adjacent(X,Y,D,X1,Y1),
   update(Z1,[at(X1,Y1)],[at(X,Y)],Z2).
```

where `adjacent(X,Y,D,X1,Y1)` is an auxiliary predicate

State Update Axioms (cont'd)

```
state_update(Z1,grab,Z2) :-
    holds(at(X,Y),Z1),
    update(Z1,[has(gold)],[gold(X,Y)],Z2).

state_update(Z1,shoot,Z2) :-
    update(Z1,[],[has(arrow)],Z),
    cancel(dead,Z,Z2).

state_update(Z1,exit,Z2) :-
    update(Z1,[],[at(1,1)],Z2).


% Sensing actions

state_update(Z,sense_stench,Z).

state_update(Z,sense_breeze,Z).

state_update(Z,sense_glitter,Z).

state_update(Z,sense_scream,Z).
```

Percepts

```
sf(sense_stench,Percept,Z) :- ...

sf(sense_breeze,Percept,Z) :- ...

sf(sense_glitter,Percept,Z) :-
   holds(at(X,Y),Z),
   ( Percept=false -> not_holds(gold(X,Y),Z) ;
     Percept=true  -> holds(gold(X,Y),Z) ).

sf(sense_scream,Percept,Z) :-
   ( Percept=false -> not_holds(dead,Z) ;
     Percept=true  -> holds(dead,Z) ).


% Non-sensing actions

sf(turn_left,true,_).

...
```

An Example ALP for the Wumpus Agent (1)

```
main :-
   Choicepoints = [[north,west]], Backtrack = [],
   do(sense_stench), do(sense_breeze), do(sense_glitter),
   main_loop(Choicepoints, Backtrack).


main_loop(_, Backtrack) :-
   ?(at(X,Y) and gold(X,Y)), do(grab), go_home(Backtrack).

main_loop([[]|Choicepoints],Backtrack) :-
   go_back(Choicepoints,Backtrack).

main_loop([Choices|Choicepoints], Backtrack) :-
   Choices = [Direction|Directions],
   explore(Direction,[Directions|Choicepoints],Backtrack).
```

An Example ALP for the Wumpus Agent (2)

```
explore(Direction,Choicepoints,Backtrack) :-
   ?(at(X,Y)), adjacent(X,Y,Direction,X1,Y1),
   ?(not wumpus(X1,Y1) and not pit(X1,Y1)),
   turn_to(Direction),
   do(go),
   do(sense_stench), do(sense_breeze), do(sense_glitter),
   main_loop([[north,east,south,west]|Choicepoints],
             [Direction|Backtrack]).

explore(Direction,Choicepoints,Backtrack):-
   main_loop(Choicepoints,Backtrack).

go_back(Choicepoints,[Direction|Backtrack]) :-
   turn_to_reverse_direction(Direction),
   do(go), main_loop(Choicepoints,Backtrack).

go_home([]) :- do(exit).

go_home([Direction|Backtrack]) :-
   turn_to_reverse_direction(Direction),
   do(go), main_loop(Choicepoints,Backtrack).
```

An Example ALP for the Wumpus Agent (3)

```
turn_to(Direction) :- ?(facing(Direction)).

turn_to(Direction) :- ?(not facing(Direction)),
                      do(turn_left), turn_to(Direction).

turn_to_reverse_direction(Dir) :-
   (Dir #= north #/\ RevDir #= south) #\/ ...,
   turn_to(RevDir).
```

Recap: Histories

A history h is a sequence of (action, sensing result)-pairs

$$(a_1, v_1), (a_2, v_2), \ldots, (a_n, v_n) \qquad (n \geq 0)$$

- $End[h, s] \stackrel{\text{def}}{=} Do(a_n, \ldots, Do(a_2, Do(a_1, s)) \ldots)$
- $Sensed[h, s] \stackrel{\text{def}}{=} \{\neg\}SF(a_1, Do(a_1, s)) \wedge \ldots \wedge \{\neg\}SF(a_n, End[h, s])$

  where "¬" is placed wrt. a_i iff $v_i = false$

Generalized Derivation Rules

- Actions:

$$\frac{Poss(a, s_1, s_2) \wedge Q_2 \wedge \ldots \wedge Q_n}{(Q_2 \wedge \ldots \wedge Q_n)\theta}$$

where $\Sigma \cup \{Sensed[h, s]\} \models Poss(a, s_1, s_2)\theta$ with substitution θ on the variables in $Poss(a, s_1, s_2)$

- Tests:

$$\frac{\phi[s] \wedge Q_2 \wedge \ldots \wedge Q_n}{(Q_2 \wedge \ldots \wedge Q_n)\theta}$$

where $\Sigma \cup \{Sensed[h, s]\} \models \phi[s]\theta$ with substitution θ on the variables in ϕ

Online Execution of ALP with Sensing

Recap: For online execution of an ALP, the agent actually performs each action and updates its state

```
do(A,Z1,Z2) :- perform(A,SF),
               state_update(Z1,A,Z2),
               sf(A,SF,Z2).
```

(where `perform(A,SF)` is assumed to return the sensing result in `SF`)