Training Logical Mouses

A logical approach to agent design

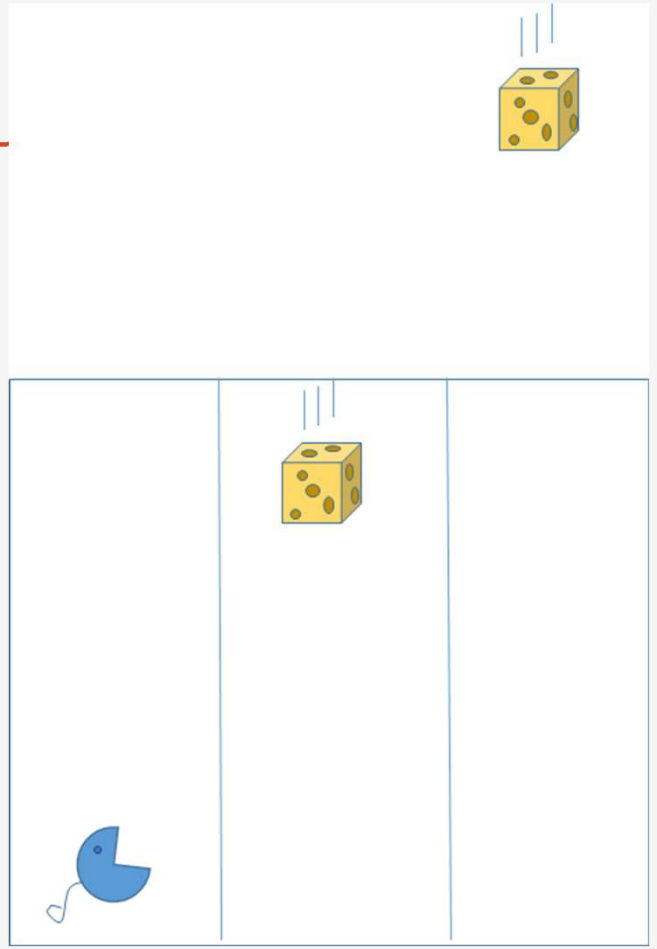
R. Basili (Univ. Roma, Tor Vergata), a.a. 2020-21

Overview

- Progettiamo un agente logico in Prolog
- Progettiamo la sua base di conoscenza, con l'intento di ottimizzarne la capacità di raggiungere l'obbiettivo
- Osserviamo il comportamento autonomo dell'agente
- Ci porremo il problema: *può la conoscenza dell'agente essere appresa e non codificata manualmente*?
- Vedremo come collezionare esempi (positivi) del comportamento dell'agente (i.e. scelte della azione)
- Introdurremo un'ambiente per acquisire modelli di classificazione mediante Machine Learning: Weka
- Applicando Weka al caso del nostro agente, acquisiremo un modello decisionale (regole) attraverso l'apprendimento di una funzione di classificazione mediante Decision trees.
- Vedremo le analogie tra la base di conoscenza (Prolog) progettata top-down e il modello di azioni (albero delle decisioni) indotto dai dati dal DTL.

Mouse And Cheese

Nel mondo proposto, è necessario animare un piccolo topolino a raccogliere i pezzi di formaggio in caduta dall'alto in un ambienet 3x3 (vedi figura a lato).



Si progetti in Prolog il mondo del topolino (ambiente, sensori e mondi possibili) in modo da poter programmare le azioni del topolino che massimizzano l'utilità del goal:

mangiare il numero massimo di pezzetti di formaggio.

Definire la strategia di scelta delle azioni separatamente dalla applicazione delle azioni, in modo da favorire lo sviluppo di stategie diverse.

Si usi il predicato random/1 per generare a caso le posiizoni del topolino e del formaggio di volta in volta.

Describing the state

The world corresponds to a 3x3 Map, where coordinates X (row) and Y (column) define an individual position.

There are two main objects in a map:

Cheese Position: predicate *cheese(X,Y)*

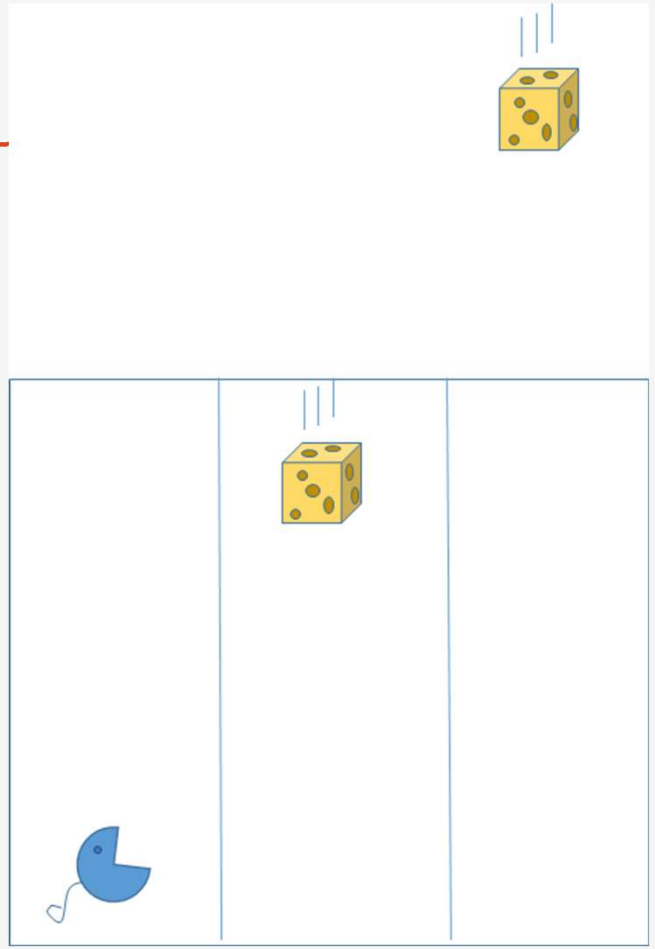
Mouse Position: predicate *mouse(X,Y)*

The State is thus defined as a pair (Prolog list)

[cheese(XC,YC), mouse(XM, YM)]

At every time stamp, the State list (i.e. the pair) represents the current state of the environment.

"*There is only one cheese at a time*" is implied by the fact that there is only ONE cheese(X,Y) predicate in the State declaration.



Goal & Utility

Goal: The mouse stops when **there is no further cheese** that is falling down

As *[cheese(XC,YC), mouse(XM, YM)]*

describes the State of the World, any value YC outside the *[1,3]* range is a valid indicator of the fact that there is no bit of cheese falling.

Default: When *YC == 0*, assume the Goal has been satisfied and the mouse stop acting.

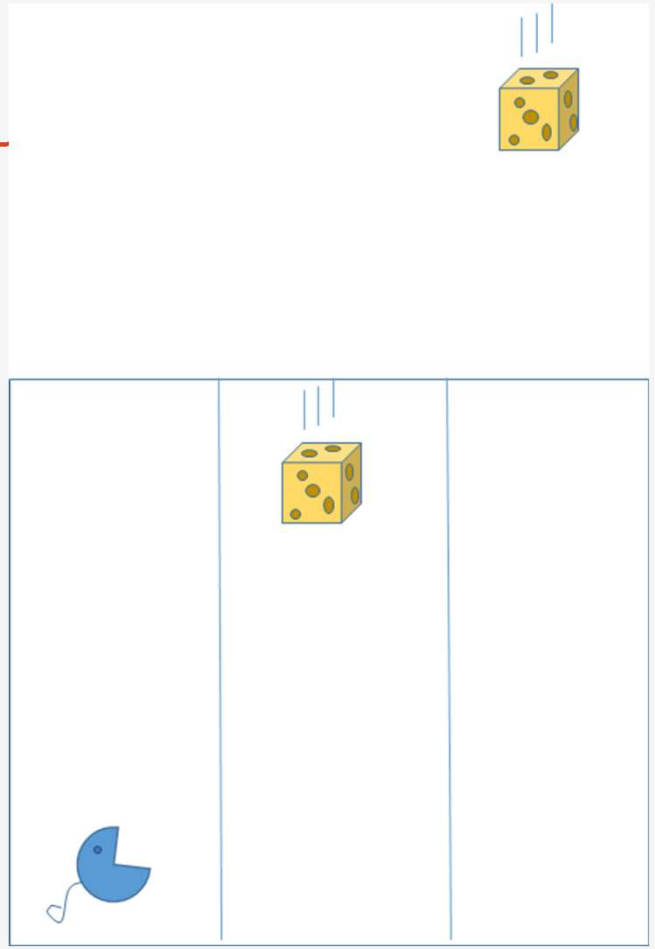
Moreover, as **the mouse does not jump**, at each time stamp ***XM == 1***.

Eating is possible when the mouse is under the falling column of the cheese, and at the time the cheese touches the ground he is able to reach the same cell.

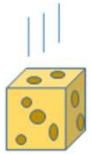
The **EATing** action is possible thus ONLY WHEN for some *YC=YM<>0*,

XC == XM == 1.

Utility: **The mouse has to maximize the bit of cheese he is able to eat when they are on the ground** (i.e. *XC=1*).

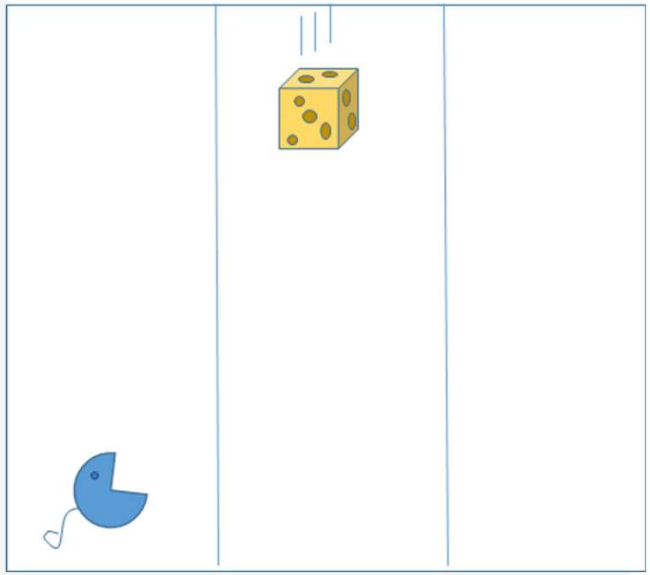


The Mouse as an agent: requirements

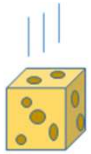


The following notions have to be specified:

- **Initialization** of the world (Use random/1)
- Synchronization of the world changes (i.e. the stepwise falling process of the cheese) with the mouse observations and actions
- The actions is selected by the mouse at each step in the attempt to maximize the utility
- ***Goal-satisfaction check*** at each step



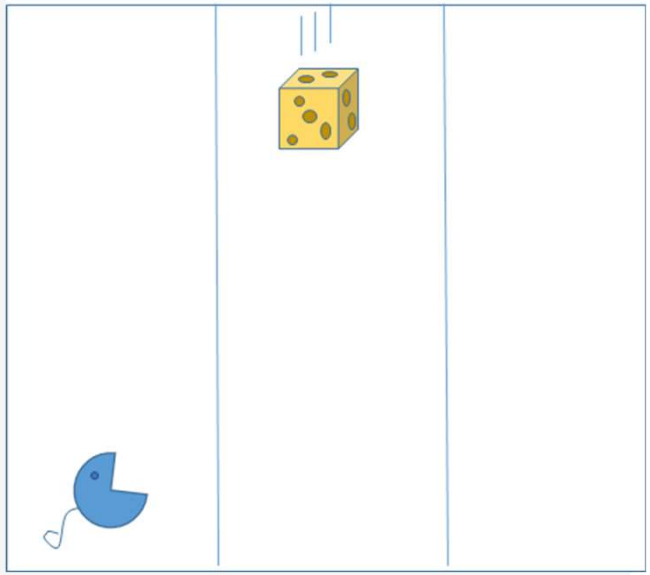
The Mouse as an agent: requirements



Main problem instance cycle:

- Syncronize each requirements above in a step-by-step process. In Prolog:

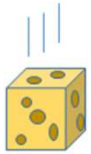
```
run_problem_instance :-
    init_the_world( YC, YM),
    solve([cheese(3,YC), mouse(1,YM)], 0, Feed),
    write("\n Task carried out: !!"),nl, write("The mouse has eaten "),
     write(Feed), write(" cheese bits!!\n"), nl.
```



where:

```
init_the_world(CheeseNextColumn, MouseNextColumn)  :-
    random_pos_the_cheese(CheeseNextColumn),
    random_pose_the_mouse(MouseNextColumn).
```

Solving the task



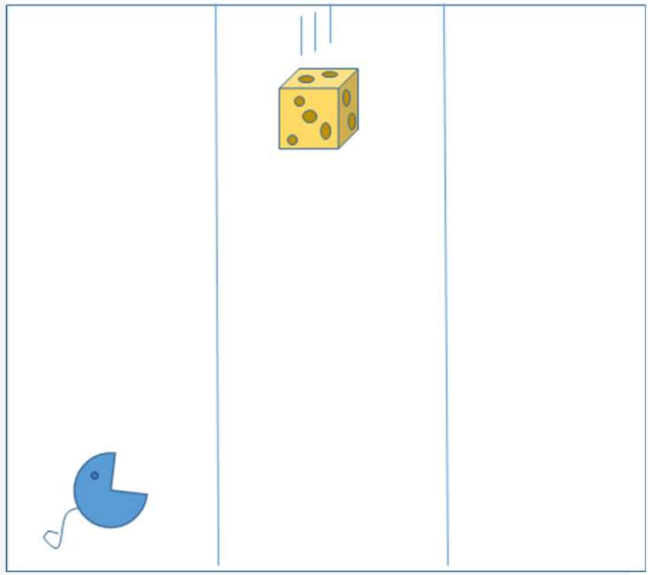
Two major actions:

- Check if the goal has been reached (no more cheese)
- OBSERVE/DECIDE/ACT cycle

Main solve **cycle**:

- In Prolog:

```
solve([cheese(XC,YC), mouse(_,_)], CurrFeed, CurrFeed) :-
        check_goal_satisfaction(cheese(XC,YC)),
         !.
solve([cheese(XC,YC), mouse(XM, YM)], CurrFeed, AllFeeds) :-
          decide_rule( [cheese(XC,YC), mouse(XM, YM)], CurrFeed, Rule),
          apply_action(Rule, [cheese(XC,YC), mouse(XM, YM)], CurrFeed, NewState, NewFeed),
          env_step(NewState, NextState),
          !,
          solve( NextState, NewFeed, AllFeeds).
```



where:

```
env_step([cheese(1, _), mouse(XM, YM)], [cheese(3, NewYC), mouse(XM, YM)]) :-
                                          /* generate a new starting position at column NewYC */
          random_pos_the_cheese(NewYC),
env_step([cheese(XC, YC), mouse(XM, YM)], [cheese(NewXC, YC), mouse(XM, YM)]) :- /* falling down ...*/
          XC > 1, NewXC is XC-1.
```

Decision Rules for acting

```
decide_rule( [cheese(1, Y), mouse(1,Y)], _, eat) :-
        !.
decide_rule( [cheese(_anyXC, Y), mouse(_anyXM, Y)], _, stand) :-
        \+(_anyXC == _anyXM),
        !.
decide_rule( [cheese(_, NewYC), mouse(_, YM)], _, moveright) :-
        NewYC > YM,
        !.
decide_rule( [cheese(_, NewYC), mouse(_, YM)], _, moveleft) :-
        NewYC < YM,
        !.
decide_rule( _anyState, _anyFeed, _, stand).
```

Run the program

The program is accessible in file: Mouse.pl

Use the predicate

?- run_problem_instance.

to run the agent.

The program includes:

- Messages to inspect the Mouse behaviour
- The (hand coded) decision rules that are optimal
- Logging of examples (see Machine Learning approach next slides)