Logic and Knowledge Representation

Problem Types, and Problem solving methods
27 April 2018

Giovanni Sileno gsileno@enst.fr
Télécom ParisTech, Paris-Dauphine University
Problem solving
Well-defined problems

Problems are *well-defined* when there is a simple test to conclude whether a solution is a solution.

Well-defined problems & problem spaces

Problems are *well-defined* when there is a simple test to conclude whether a solution is a solution.


People solve problems by *searching* through a problem space, consisting of the *initial state*, the *goal state*, and *all possible states in between*.

Problem and solution spaces

solution(s, p) can be interpreted as s satisfies p
Problem and solution spaces

Solution(s, p) can be interpreted as s satisfies p

Problem: how to generate solutions?
Problem and solution spaces

solution(s, p) can be interpreted as s satisfies p

problem: how to generate solutions?
Defining the problem...

An old lady wants to visit her friend in a neighbouring village. She takes her car, but halfway the engine stops after some hesitations. On the side of the road she tries to restart the engine, but to no avail.

Defining the problem...

An old lady wants to visit her friend in a neighbouring village. She takes her car, but halfway the engine stops after some hesitations. On the side of the road she tries to restart the engine, but to no avail.

Which is the problem here?

from ill-defined to well-defined problems...

Suite of problem types

Suite of problem types

Suite of problem types

*behavioural view: system + environment*

Suite of problem types

Suite of problem types

**behavioural view: system + environment**

- planning
- assessment

- modelling
- assignment
- design

**structural view: system**

Suite of problem types

behavioural view: system + environment

structural view: system

Suite of problem types

*behavioural view: system + environment*

- planning
- assessment

- modelling
- assignment

- design
- monitoring
- diagnosis

*structural view: system*

Agent problem-solving cycle

- interpret model
- plan design
- execute implement
- monitor
- diagnose

sensing → acting → sensing

starting in-house failure response

outsourcing failure response
Agent problem-solving cycle

where ill-defined problems come up

starting in-house failure response

outsourcing failure response
Agent problem-solving cycle

where well-defined problems are set up

interpret model → plan design → execute implement

acting

where ill-defined problems come up

monitor → diagnose

sensing

starting in-house failure response

outsourcing failure response
Agent problem-solving cycle

where well-defined problems are set up

where tasks are assigned and scheduled

where ill-defined problems come up

interpreting model

sensing

plan design

acting

execute implement

monitor

sensing

starting in-house failure response

diagnose

outsourcing failure response
Forward and backward reasoning
How do we approach a problem?

Q.9. A shell is fired vertically upward with a velocity of 98 m/s. Find,

a) The time taken by it to reach the highest point.
b) How long it will stay in the air. [http://hometuitionsinkarachi.over-blog.com]
c) The maximum height reached.
d) The velocity with which it will hit the ground.

\[ a = \frac{g}{h} = \frac{9.8 \text{ m/s}^2}{98} \]
\[ t = \frac{v}{g} = \frac{98}{9.8} \]
\[ h = \frac{1}{2} \cdot g \cdot t^2 = \frac{1}{2} \cdot 98 \cdot \left(\frac{98}{9.8}\right)^2 = 980 + 490 = 1470 \text{ m} \]

How do we approach a problem?

Novice students start from the goal.
They look for a formula returning the goal, and then for formulas returning what it is needed by the previous one, up they formulas satisfied with the given data.

How do we approach a problem?

Novice students start from the goal. They look for a formula returning the goal, and then for formulas returning what it is needed by the previous one, up they formulas satisfied with the given data.

Experts start from the data. They apply formulas directly to data, according to some heuristics.

How do we approach a problem?

Novice students start from the goal.
They look for a formula returning the goal, and then for formulas returning what it is needed by the previous one, up they formulas satisfied with the given data.

Experts start from the data.
They apply formulas directly to data, according to some heuristics.


and Prolog?
Example of diagnosis task

"expert" knowledge
Example of diagnosis task

```
leak_in_bathroom :-
    hall_wet,
    kitchen_dry.

problem_in_kitchen :-
    hall_wet,
    bathroom_dry.

no_water_from_outside :-
    window_closed ;
    no_rain.

leak_in_kitchen :-
    problem_in_kitchen,
    no_water_from_outside.

hall_wet.
bathroom_dry.
window_closed.
```

"expert" knowledge  prolog program
Some technical detail

```prolog
:- dynamic(kitchen_dry/0, no_rain/0).

leak_in_bathroom :-
    hall_wet,
    kitchen_dry.

problem_in_kitchen :-
    hall_wet,
    bathroom_dry.

no_water_from_outside :-
    window_closed ;
    no_rain.

leak_in_kitchen :-
    problem_in_kitchen,
    no_water_from_outside.

hall_wet.
bathroom_dry.
window_closed.
```

necessary to define **fluents**, i.e. facts that might change along the execution

?- leak_in_bathroom.  
   FALSE  
?- leak_in_kitchen.  
   TRUE  

prolog program
Some technical detail

:- dynamic(kitchen_dry/0, no_rain/0).

leak_in_bathroom :-
    hall_wet,
    kitchen_dry.

problem_in_kitchen :-
    hall_wet,
    bathroom_dry.

no_water_from_outside :-
    window_closed ;
    no_rain.

leak_in_kitchen :-
    problem_in_kitchen,
    no_water_from_outside.

hall_wet.
bathroom_dry.
window_closed.

necessary to define fluents, i.e. facts that might change along the execution

How? ... using assert and retract

?- leak_in_bathroom.  
   FALSE 
?- leak_in_kitchen.  
   TRUE
Generalization

• Using Prolog's own syntax for rules may gave certain disadvantages however:
  – this syntax may not be the most suitable for a user unfamiliar with Prolog; e.g. experts
  – the knowledge base is not syntactically distinguishable from the rest of the program

• Let us create a small DSL (*domain specific language*)!
A simple interpreter for rules

% symbols of DSL and priority

:- op(800, fx, if).
:- op(700, xfx, then).
:- op(300, xfy, or).
:- op(200, xfy, and).
A simple interpreter for rules

% symbols of DSL and priority

:- op(800, fx, if).
:- op(700, xfx, then).
:- op(300, xfy, or).
:- op(200, xfy, and).

% knowledge base

if
  hall_wet and kitchen_dry
then
  leak_in_bathroom.

if
  hall_wet and bathroom_dry
then
  problem_in_kitchen.

if
  window_closed or no_rain
then
  no_water_from_outside.

if
  problem_in_kitchen and
  no_water_from_outside
then
  leak_in_kitchen.

fact(hall_wet).
fact(bathroom_dry).
fact(window_closed).
Backward chaining

% backward chaining rule interpreter

is_true(P) :-
    fact(P).

is_true(P) :-
    if Condition then P,
    is_true(Condition).

is_true(PI and P2) :-
    is_true(PI), is_true( P2).

is_true(PI or P2) :-
    is_true(PI) ; is_true( P2).
for forward chaining we need to materialize the (partial) conclusions!

% necessary with SWI-Prolog.
:- dynamic(sunshine/0, raining/0, fog/0).

nice :-
sunshine, not(raining).

funny :-
sunshine, raining.

disgusting :-
raining, fog.

raining.
fog.

?- nice.
?- disgusting.
?- retract(fog).
?- disgusting.
?- assert(sunshine).
?- funny.
Forward chaining

% forward chaining rule interpreter
forward :-
        new-derived-fact(P),
        !,
        write('Derived:'), write(P), nl,
        assert(fact(P)),
        forward
        ;
        write('No more facts').

new-derived-fact(Concl) :-
        if Cond then Concl,
        not(fact( Concl)),
        composed-fact( Cond).

composed-fact(Cond) :-
        fact(Cond).

composed-fact(Cond1 and Cond2) :-
        composed-fact(Cond1),
        composed-fact( Cond2).

composed-fact(Cond1 or Cond2) :-
        composed-fact(Cond1)
        ;
        composed-fact( Cond2).
Backward vs Forward chaining

- Going from an initial state to a goal state can be unsuitable for problems with a large number of rules (all facts are derived, even the not useful ones).
Backward vs Forward chaining

- Going from an initial state to a goal state can be unsuitable for problems with a large number of rules (all facts are derived, even the not useful ones).

- Searching backwards from the goal state usually eliminates spurious paths.
Backward vs Forward chaining

- Going from an initial state to a goal state can be unsuitable for problems with a large number of rules (all facts are derived, even the not useful ones).

- Searching backwards from the goal state usually eliminates spurious paths. 

  *but it does not enjoy caching abilities!*
Types of reasoning (with rules)
Types of reasoning (Pierce) - 1

Deduction

Rule: All the beans from this bag are white.
Fact: These beans are from this bag.
⇒ Result: These beans are white.

This conclusion is *certainly* true, if the premises are true.
Types of reasoning (Pierce) - 2

Induction

Fact: These beans are from this bag.
Fact: These beans are white.

⇒ Hyp. rule: All the beans from this bag are white.

This conclusion is true until proved otherwise.
Abduction

Rule: All the beans from this bag are white.

Observed fact: These beans are white.

⇒ Hyp. fact: These beans are from this bag.

This conclusion is *plausibly* true.
Resuming...

**Deduction**
- Asserted Rule
- + Asserted Fact
- = Asserted Fact

**Induction**
- Observed Fact
- + Observed Fact
- = Hypothetical Rule

**Abduction**
- Observed Fact
- + Known Rule
- = Plausible Fact
Planning
Monkeys and bananas

• A hungry monkey is in a room. Suspended from the roof, just out of his reach, is a bunch of bananas. In the corner of the room is a box. The monkey desperately wants the bananas but he can’t reach them. What shall he do?
Monkeys and bananas

• After several unsuccessful attempts to reach the bananas, the monkey *walks to the box*, *pushes* it under the bananas, *climbs* on the box, *picks* the bananas and eats them.
Planning

• To solve this problem the monkey needed to devise a plan, a sequence of actions that would allow him to reach the desired goal.

• Planning is a topic of traditional interest in AI.

• To be able to plan, a system needs to be able to reason about the *individual* and *cumulative effects* of a series of actions.

• This is a skill that is only observed in a few animal species and only mastered by humans.
Ingredients

- Actions, with conditions and consequences:  
  \[ \text{action(InitialState, Action, ObtainedState)} \]
Ingredients

- Actions, with conditions and consequences:
  \[ \text{action}(\text{InitialState}, \text{Action}, \text{ObtainedState}) \]

- States of the world
  \[ \text{state}(\text{middle}, \text{onbox}, \text{middle}, \text{not\_holding}) \]
Ingredients

- Actions, with conditions and consequences:

  \[\text{action}(\text{state}(P, \text{floor}, P, T), \text{climb}, \text{state}(P, \text{onbox}, P, T)).\]

- States of the world

  \[\text{state}(\text{middle}, \text{onbox}, \text{middle}, \text{not\_holding})\]
Cooking everything

action(state(middle, onbox, middle, not_holding),
    grab,
    state(middle, onbox, middle, holding)).

action(state(P, floor, P, T),
    climb,
    state(P, onbox, P, T)).

action(state(P1, floor, P1, T),
    push(P1, P2),
    state(P2, floor, P2, T)).

action(state(P1, floor, B, T),
    walk(P1, P2),
    state(P2, floor, B, T)).

success(state(_, _, _, holding)).

success(State1) :-
    action(State1, A, State2),
    write("Action : "), write(A), nl,
    write(" --> "), write(State2), nl,
    success( State2).

?- success(state(door, floor, window, not_holding)).
Another exercise: the tower of Hanoi
Recursion
Recursion

- Recursion is a concept widely used in computer science and linguistic.
  - an object defined in terms of itself
  - a procedure invoking itself
Recursion

- Hypothesis: natural language is recursive, as (some) syntaxic categories are recursive.

- John thinks Emily plays well.

  statement

  statement
Recursion

- Three phases: descent, stop at bottom, ascent.

```
even([ ]).  
even([_,_|L]) :- even(L).
```
Recursion

• Three phases: *descent, stop at bottom, ascent.*

even([ ]).
even([_, _|L]) :- even(L).

?- even([3, 5, 3]).
Recursion

- Three phases: *descent*, *stop at bottom*, *ascent*.

```prolog
even([ ]).
even([_,_|L]) :- even(L).
```

```prolog
?- even([3, 5, 3]).
```

*Seek for an entrance*
Recursion

- Three phases: descent, stop at bottom, ascent.

```
even([]).
even([_,_|L]) :- even(L).
?- even([3, 5, 3]).
```

*Propagate recursively*
Recursion

- Three phases: descent, stop at bottom, ascent.

```prolog
even([ ]).
even([_,_]|L)) :- even(L).
?- even([3, 5, 3]).
```

Reach the bottom of the recursion
Recursion

- Three phases: descent, stop at bottom, ascent.

\[
even([ \_ ]) \rightarrow even([\_ , \_ | L]) \rightarrow even(L).
\]

?- even([3, 5, 3]).

Retrace back
Recursion

- Three phases: \textit{descent}, \textit{stop at bottom}, \textit{ascent}.

\texttt{even([ ])}.

\texttt{even([_, _, L]) :- even(L).}

\texttt{?- even([3, 5, 3]).}

\textit{Propagate the result back}
Recursion

- Three phases: *descent, stop at bottom, ascent.*

```prolog
mirror(Left, Right) :-
    invert(Left, [], Right).

invert([X|L1], L2, L3) :-
    invert(L1, [X|L2], L3).
invert([], L, L).
```
Conclusions
Conclusions

- Symbolic AI presents relevant techniques to solve problems that can be described in symbolic terms, that in many cases outperform humans.

Conclusions

• Symbolic AI presents relevant techniques to solve problems that can be described in symbolic terms, that in many cases outperform humans.

• AI methods are today implemented in new generations of expert systems and all IT infrastructures of organizations.
Conclusions

● Symbolic AI presents relevant techniques to solve problems that can be described in symbolic terms, that in many cases outperform humans.

● AI methods are today implemented in new generations of expert systems and all IT infrastructures of organizations.

● However, many problems cannot be adequately handled by symbolic techniques, as e.g. those faced by sensory-motor modules:
  - vision, action
Conclusions

• In robotics, starting from the 80s, a radically different paradigm started to be considered, renouncing to symbolic representations.

• As Rodney Brooks famously put it: “Elephants don't play chess”
  - overlap with *machine learning*
Conclusions

• In robotics, starting from the 80s, a radically different paradigm started to be considered, renouncing to symbolic representations.

• As Rodney Brooks famously put it: “Elephants don't play chess”
  - overlap with machine learning

• Similarly, failures of symbolic AI explains today interest for deep learning techniques.
Conclusions

• However, we should not forget, that a good deal of our interactions with other people is not too far from playing chess.
  – expressing how and why is fundamental for humans, and for this we need symbols.

• Symbolic AI and Statistical AI occupy different sides of the spectrum of intelligent behaviour.