

Chapter 15

Expert Systems

An *expert system* is a program that behaves like an expert in a certain field that is closely associated with a collection of domain dependent knowledge.

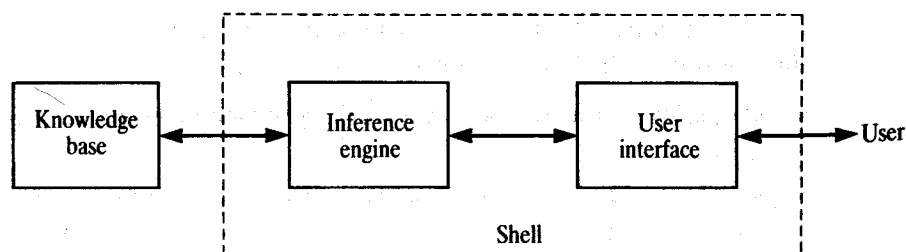
They are also called *knowledge-based systems*, since many of them deal with such issues as fixing cars and/or people, circuit construction, computer configuration, etc.. Hence, we also require such a system to *explain* its behavior and its decision making process.

An additional feature of such system is that it can deal with *uncertainty* and *incompleteness*. For example, a patient does not have all the symptoms, a doctor can still diagnose his illness.

The process

To build an expert system, we usually have to provide the following components: some *problem-solving* functions that can use domain-specific knowledge to solve problems, which might have to deal with uncertainty; and some *user-interaction* features that works like an interface between the system and the user, which should also be able to explain to the user.

Thus, an expert system consists of three pieces: a knowledge base, an inference engine, and a user interface.



Knowledge representation

Although anything goes as far as KR is concerned, the *if-then* turns out to be the most popular formalism for representing knowledge.

Those rules are just general conditional statements: If P then C , meaning if preconditions P hold, so does the conclusion C .

They are popular since each group of rules define a small, and relatively independent piece of knowledge (Modularity); new rules can be added relatively independent of other rules (Incrementability); old rules can be deleted (Modifiability); and they support the system's transparency, since they can be used to *explain* things in term of *how* and *why*.

Examples

A general format of an *if-then* rule can be the following:

IF condition A THEN conclusion B follows
with certainty F.

For example,

IF

1. the infection is primary bacteraemia,
2. the site of the culture is some sterile site,
3. the suspected portal of the organism entry
is gastrointestinal tract

THEN

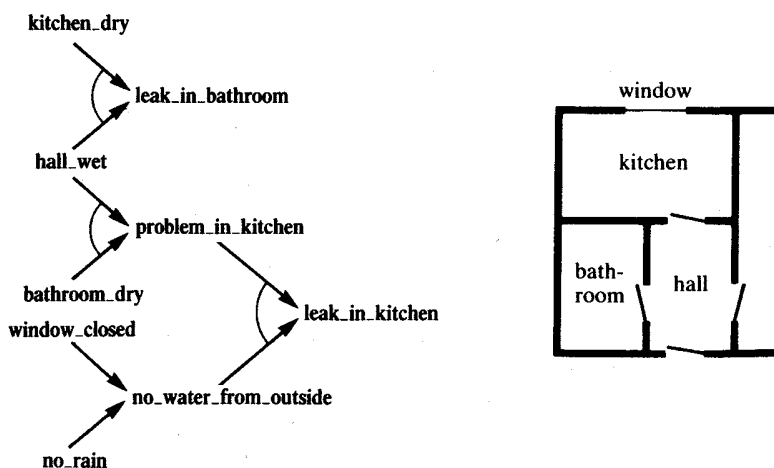
there is suggestive evidence (0.7) that the
identity of the organism is bacteroides.

Knowledge acquisition

If you want to set up a serious expert system, you have to consult some experts in that field.

This complicated process of extracting certain knowledge from experts and/or literature is referred to as *knowledge acquisition*.

Assume that we have obtained the following information,



we could have the following rule:

if hallway wet and bathroom dry
then the problem is in the kitchen.

“Think” about them

Once all the knowledge are in place, we need to have an *inference engine* that will draw conclusions based on the knowledge.

There are essentially two ways to do it: *backward chaining*, and *forward chaining*. Both of them can be easily implemented in Prolog.

Since we have to provide some explanation to the user as how the conclusion is reached, the *way* to search for the right conclusion is important.

The other approach

In backward chaining, we start with some hypothesis, e.g., `leak_in_kitchen`, then we reason backwards to find support.

For example, in this case, we have to confirm both `problem_in_kitchen`, and `no_water_from_outside`.

The former is confirmed if we can establish that the hallway is wet, but (really means and) the bathroom is dry; and the latter can be confirmed if we find out that there is no rain outside, or the windows are closed.

In terms of Prolog

We can express our knowledge in terms of the following facts:

```
leak_in_kitchen:-  
    problem_in_kitchen,  
    no_water_from_outside.
```

```
problem_in_kitchen:-  
    hall_wet, bathroom_dry.
```

```
no_water_from_outside:-  
    window_closed  
    ;  
    no_rain.
```

```
hall_wet. bathroom_dry.  
window_closed.
```

Then, if we pose the following question
`?-leak_in_kitchen`, the answer would be “yes”.

But, a general user might not like this syntax too much. Moreover, in this format, the knowledge base cannot be easily distinguishable from the inference engine. It will be more natural to use such operators as `if`, `then`, `or`, and `and`.

We also represent all the facts in terms of a predicate `fact`, e.g., `fact(hall_wet)`. When we ask a question, `P`, we adopt another predicate `is_true(P)`.

The knowledge base

We have the following knowledge base:

```
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).
```

The BC code

Taking into account everything we just mentioned, we have the following program:

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

is_true(P):- fact(P).
is_true(P):-
    if Condition then P,
    is_true(Condition).
is_true( P1 and P2):-
    is_true(P1),is_true(P2).
is_true(P1 or P2):-
    is_true(P1)
    ;
    is_true( P2).
```

Now, we can pose such question as
`is_true(leak_in_kitchen).`

Forward chaining

In contrast to backward chaining, Forward chaining, in no surprise, starts from some confirmed finding, tries to derive the desired conclusion.

For example, given the above facts, if we start the process by saying `?-forward`, we will see the following session:

```
Derivee: problem_in_kitchen  
Derivee: no_water_from_outside  
Derivee: leak_in_kitchen  
No more facts.
```

The code is in the next transparency.

The FC code

```
forward:-new_derived_fact(P),!,  
  write('Derivee: '),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

Both of them are involved with some sort of search, but they differ in direction. Backward chaining goes from goals to supportive data (proof by contradiction); while the forward chaining goes from facts to conclusion (proof by induction).

There is no clear cut as which one is better. It really depends on the issue at hand. If we want to establish certain hypothesis, then backward chaining seems to be more natural: the only thing involved will be the relevant ones. On the other hand, forward chaining is handy when the system has to monitor whether a special occasion has just arisen.

In reality, we might even combine both chaining in our system. For example, some symptoms may trigger a doctor to make some diagnosis in forward chaining, then come backwards for more evidence to support such a conjecture.

Why? Why? Why?

There are standard ways to generate explanation in rule-based system. Two usual types of explanation are called “how” and “why”.

When the system comes up with an answer, the user may ask *how* does the system find this answer. The typical explanation provides a trace of the process to derive this answer. For example, the trace of reaching the conclusion that there is a leak in the kitchen is because 1) there is a problem in the kitchen, since the hall is wet, but the bathroom is dry; and 2) no water came from outside, since the window is closed.

The *how* type deals with the whole process, thus is answered with the support of a *proof tree*. In contrast, the *why* question occurs during the process, thus makes the reasoning process into an interactive one.

Proof tree construction

Let the operator \leq be the one-step answer to *how*, then the following code provides a proof tree for the question P.

```
:-op(800,xfx,<=).
```

```
is_true(P,P):-fact(P).
```

```
is_true(P,P<=CondProof):-
```

```
    if Cond then P,
```

```
    is_true(Cond,CondProof).
```

```
is_true(P1 and P2,Proof1 and Proof2):-
```

```
    is_true(P1,Proof1),
```

```
    is_true(P2,Proof2).
```

```
is_true(P1 or P2,Proof):-
```

```
    is_true(P1,Proof)
```

```
    ;
```

```
    is_true(P2,Proof).
```

Uncertainty

In the above discussion, everything is either white or black. Many real life situation does not work like this. Typical expertise is by and large educated guess: it is usually true, but with exceptions.

We can model this type of uncertainty with some qualification, other than just true or false. For example, *true, highly likely, unlikely, impossible*.

We can also assign a *degree of belief*, e.g., between 0 and 1. Ideally, we would like to make use of the solid probability theory when assigning such *certainty factors*, but it would be much more difficult than some simpler, *ad hoc*, uncertainty schemes.

A simple thing

We simply add a certain factor to each proposition, e.g., `Proposition:Factor`. Thus, the following defines a rule and provides a degree of belief for this rule:

If Cond. then Conc.: Certainty

It is also easy to combine certainty for compound propositions:

$$\begin{aligned}c(P_1 \text{ and } P_2) &= \min\{c(P_1), c(P_2)\} \\c(P_1 \text{ or } P_2) &= \max\{c(P_1), c(P_2)\}\end{aligned}$$

For a rule if P_1 then $P_2 : c$, we have that

$$c(P_2) = c \times c(P_1).$$

Certainty of a conclusion is the product of that of the rule and that of the condition.

In terms of Prolog

We can loosen some of the rules, e.g.,

```
if hall_wet and bathroom_dry
then problem_in_kitchen:0.9
```

We can also have facts like `given(hall_wet,1)`.
The inference engine will look like the following:

```
certainty(P,Cert):- given(P,Cert).
```

```
certainty(Cond1 and Cond2,Cert):-
  certainty(Cond1,Cert1),
  certainty(Cond2,Cert2),
  min(Cert1,Cert2,Cert).
```

```
certainty(Cond1 or Cond2, Cert):-  
    certainty(Cond1, Cert1),  
    certainty(Cond2, Cert2),  
    max(Cert1, Cert2, Cert).
```

```
certainty(P, Cert):-  
    if Cond then P:C1,  
    certainty(Cond, C2),  
    Cert is C1*C2.
```

This schema is obviously too simple. For example, if the certainty of a and b are 0.5 and 0, respectively, by our rule, that of the a or b is 0.5. Now, if the certainty of b goes up to 0.5 later, the certainty of a or b is still 0.5, which does not make lots of sense.

Going from here

Many more sophisticated schemas have been designed, used, and investigated, to handle the problem of uncertainty in expert system.

One of the common issues is how to handle the dependency between propositions. For example, in *if a or b then c*, the certainty of *c* should not only depend on that of *a* and *b*, but also on the relationship between *a* and *b*: they tend to occur together, or not, which one occurs more often, etc..

Since it is really difficult to figure this out, people often assume that *a* and *b* are independent of each other.

If you are interested in this piece, read on the rest of this chapter on belief network.