1

Knowledge Based Systems

(CM0377)

Lecture 11

(Last modified 26th March 2001)

2

Implementation of expert systems

- In this lecture we shall see:
  - a simple backward-chaining interpreter
  - a simple way of handling *uncertainty* (using certainty factors)
  - a simple forward-chaining interpreter

3

A simple backward chaining interpreter (***Simpl.pl***)

- This one is so simple it doesn't even remember the answers to the questions it asks you!
- Based around rules of the form:
- if_then([<list of conditions>], <conclusion>)
- Where there is an implicit *and* between the conditions. E.g.

```
if_then(['it is mammal','it is tawny',
'it is carnivore','it has black stripes'],
'it is a tiger').
```

4

Example session

- Remember to use *load_files* with *compilation_mode(assert_all).*

```
| ?- run.
Select a knowledge base ...'Simpl2'.
{consulting e:/lect99~1/kbs/lect11~1/simpl2.pl...}
The procedure if_then/2 is being redefined.
    Old file: e:/lect99~1/kbs/lect11~1/simpl1.pl
    New file: e:/lect99~1/kbs/lect11~1/simpl2.pl
Do you really want to redefine it? (y, n, p, or ?) y
{e:/lect99~1/kbs/lect11~1/simpl2.pl consulted, 10 msec
   720 bytes}
```

- Note that I had previously used simpl1.pl.

5

Example session (ctd.)

```
Type in a goal ...'it is a tiger'.
Is it true that it has hair?
|: no.
Is it true that it gives milk?
|: yes.
Is it true that it is tawny?
|: yes.
Is it true that it is carnivore?
|: yes.
Is it true that it has black stripes?
|: yes.
Proved!

yes
| ?-
```

6

The program ...

- The **run** predicate simply clears everything from previous runs, determines file of Prolog **if_then** facts to consult, consults it and then tries to **prove** the user-specified hypothesis:

```
run:-
   retractall(if_then(_,_)),
   write('Select a knowledge base ...'),
   read(KB),
   load_files([KB], [compilation_mode(assert_all)]),
   write('Type in a goal ...'),
   read(Goal),
   prove(Goal),
   write('Proved!'), nl, !.

run:-
   write('Not proved!'), nl.
```

Proving ...

7

- It's true if either you can find an if-then rule, and all the conditions are satisfied, or the result of asking about it is 'yes'.

```
prove(Fact):-
   if_then(Conditions,Fact),
   infer_each(Conditions).
prove(Fact):-
   ask(Fact).
```

- And all conditions are satisfied if the first one is and all the others are:

```
infer_each([]).
infer_each([Fact|Rest]):-
   prove(Fact),
   infer_each(Rest).
```

Asking ...

8

- This predicate succeeds only if the user types 'yes.':

```
ask(Fact):-
   write('Is it true that '), write(Fact), write('?'),
   nl,
   read(Ans),
   Ans=yes.
```

Uncertainty

9

- There are various ways of handling uncertainty, including:
  - textually
  - using Bayesian inference (based on the idea of conditional probabilities)
  - using certainty factors
- We'll only look at certainty factors

10

Combining uncertain knowledge

- We can assign a *certainty factor* between 0 and 1 to each of the pieces of knowledge supplied to the system, and assume that:
  - cf(a and b) = min(cf(a), cf(b))
  - cf(a or b) = max(cf(a), cf(b))
  - cf(not a) = 1 - cf(a)
- Note that only the cf for 'not' is actually probabilistically sound.
- This is a limited application of 'fuzzy logic'

11

Example session with the new program (*cf.pl*)

```
| ?- run.
Select a knowledge base ...'Simpl2'.
{consulting e:/lect99~1/kbs/lect11~1/simpl2.pl...}
{e:/lect99~1/kbs/lect11~1/simpl2.pl consulted, 0 msec
   368 bytes}
Type in a goal ...'it is a tiger'.
Please give certainty for it has hair...
|: 0.3.
Please give certainty for it is tawny...
|: 0.5.
Please give certainty for it is carnivore...
|: 0.2.
Please give certainty for it has black stripes...
|: 0.7.
Certainty is 0.2

yes
| ?-
```

12

The revised program ...

- Now we *always* obtain a certainty, so only one **run** clause:

```
run:-
   retractall(if_then(_,_)),
   write('Select a knowledge base ...'),
   read(KB),
   load_files([KB], [compilation_mode(assert_all)]),
   write('Type in a goal ...'),
   read(Goal),
   prove(Goal, Cert),
   write('Certainty is '), write(Cert), nl, !.
```

13

Proving ...

- Needs an auxiliary **min** clause ...

```
prove(Fact, Cert):-
   if_then(Conditions,Fact),
   infer_each(Conditions, Cert).
prove(Fact, Cert):-
   ask(Fact, Cert).

infer_each([], 1).
infer_each([Fact|Rest], Cert):-
   prove(Fact, C1),
   infer_each(Rest, C2),
   min(C1, C2, Cert).

min(X, Y, X):- X < Y, !.
min(X, Y, Y).
```

14

Asking ...

- Ask for a certainty value, not for whether something is true or false:

```
ask(Fact, Cert):-
   write('Please give certainty for '),
   write(Fact), write('...'), nl,
   read(Cert).
```

15

A forward chaining system (*fwd.pl*)

- The scenario: want to do a variation on bubble sort on a list of numbers.
- The rules in English:
  - If X is to the left of something that is to the left of Y and X > Y then swap them
  - If X is to the left of Y and X > Y then swap them
  - If there isn't any X to the left of any Y where X > Y then stop
- We always try the rules in the order they are given. E.g.

```
| ?- go.
Initial state:  [5,3,2,4,1]
Swap performed: [2,3,5,4,1]
Swap performed: [2,3,1,4,5]
Swap performed: [1,3,2,4,5]
Swap performed: [1,2,3,4,5]

yes
| ?-
```

16

Rules

- Rules of the following form, where *any* antecedent/consequent goals can be specified:

```
rule(<Rule number>,
   <List of antecedent goals>, <consequent goal>)
```

- So for our example:

```
rule(1,[left_of(A,B),left_of(B,C), bigger_than(A,C)], swap(A,C)).
rule(2,[left_of(A,B),bigger_than(A,B)], swap(A,B)).
rule(3,[no_left_of_and_bigger],stop).
```

- Rely on a current_list fact:

```
go:-
  retractall(current_list(_)), retractall(stopped),
  assert(current_list([5,3,2,4,1])),
  current_list(X), write('Initial state:  '), write(X), nl,
  run_rules.
```

17

Recognise-act procedure

- If still running, get list of rules that could fire (CList), choose rule, fire it, repeat:

```
run_rules:-
  \+(stopped),!,
  conflict_set(CList),
  choose_rule(CList,Rule),
  fire_rule(Rule),
  run_rules.

run_rules.
```

18

Determining conflict set

- Conflict set is all rules belonging to the conflict set:

```
conflict_set(CList):-
  findall(R, belongs_to_c_set(R), CList).
```

- Those which belong to the conflict set are those whose condition is satisfied:

```
belongs_to_c_set(rule(N, Cond, Act)):-
  rule(N, Cond, Act), satisfy(Cond).
```

- Conditions satisfied if each is satisfied individually:

```
satisfy(Conditions):- satisfy_each(Conditions),!.

satisfy_each([]).
satisfy_each([H|T]):- call(H), satisfy_each(T).
```

19

Choosing and firing

- The world's simplest conflict resolver!

```
choose_rule([R|Ignore], R).
```

- Fire a rule by **call**ing:

```
fire_rule(rule(N, Cond, Act)):-
  call(Act),!.
```

20

Predicates used in rules

- (The predicates can be *anything*)

```
left_of(A,B):-
  current_list(L),
  split(L,A,[B|_]).
split([H|T],H,T).
split([H|T],A,List):-
  split(T,A,List).

bigger_than(A,B):-
  A>B.
no_left_of_and_bigger:-
  \+(some_left_of_and_bigger).
some_left_of_and_bigger:-
  left_of(A,B),
  bigger_than(A,B).
```

21

Predicates used in rules (ctd.)

```
swap(A,B):-
  retract(current_list(L)),
  list_with_swap(L,A,B,New),
  assert(current_list(New)),
  write('Swap performed: '),
  write(New),nl.

stop:-
  assert(stopped).

list_with_swap([],_,_,[]).
list_with_swap([A|T],A,B,[B|T1]):-
  list_with_swap(T,A,B,T1).
list_with_swap([B|T],A,B,[A|T1]):-
  list_with_swap(T,A,B,T1).
list_with_swap([H|T],A,B,[H|T1]):-
  \+(H=A),\+(H=B),
  list_with_swap(T,A,B,T1).
```