1 Project: Car Diagnosis Backward Chaining with Uncertainty

1.1 Certainty factors

As we have seen, backward chaining systems are good for solving structured selection types of problems. The Birds system was a good example; however it made the assumption that all information was either absolutely true, or absolutely false. In the real world, there is often uncertainty associated with the rules of thumb an expert uses, as well as the data supplied by the user.

For example, in the Birds system the user might have spotted an albatross at dark and not been able to clearly tell if it was white or dark colored. An Knowledge-Based system should be able to handle this situation and report that the bird might have been either a laysan or black footed albatross.

The rules too might have uncertainty associated with them. For example a mottled brown duck might only identify a mallard with 80% certainty. This chapter will describe an expert system shell called Clam which supports backward chaining with uncertainty. The use of uncertainty changes the inference process from that provided by pure Prolog, so Clam has its own rule format and inference engine.

The most common scheme for dealing with uncertainty is to assign a certainty factor to each piece of information in the system. The inference engine automatically updates and maintains the certainty factors as the inference proceeds.

1.1.1 An example

Let's first look at an example using Clam The certainty factors (preceded by cf) are integers from -100 for definitely false, to +100 for definitely true.

The following is a small knowledge base in Clam which is designed to diagnose a car which will not start. It illustrates some of the behavior of one scheme for handling uncertainty

goal problem.

rule 1
if not turn over and
battery bad
then problem is battery.

rule 2
if lights weak
then battery bad cf 50.

rule 3
if radio weak
then battery bad cf 60.

rule 4

if turn over and
smell gas
then problem is flooded cf 80.

rule 5

if turn over and
gas_gauge is empty
then problem is out of gas cf 90.

rule 6

if turn_over and
gas_gauge is low
then problem is out_of_gas cf 30.

ask turn over
menu (yes no)
prompt 'Does the engine turn over?'

ask lights weak
menu (yes no)
prompt'Are the lights weak?'.

ask radio weak
menu (yes no)
prompt'is the radio weak?'.

ask smell gas
menu (yes no)
prompt'Do you smell gas?'.

ask gas gauge
menu (empty low full)
prompt'What does the gas gauge say?'.

The inference uses backward chaining similar to pure Prolog. The goal states that a value for the attribute problem is to be found. Rule 1 will cause the subgoal of bad battery to be pursued, just as in Prolog. The rule format also allows for the addition of certainty factors.

For example rules 5 and 6 reflect the varying degrees of certainty with which one can conclude that the car is out of gas. The uncertainty arises from the inherent uncertainty in gas gauges. Rules 2 and 3 both provide evidence that the battery is bad, but neither one is conclusive.

1.1.2 Rule Uncertainty

What follows is a sample dialog of a consultation with the Car expert system.

consult, restart, load, list, trace, how, exit

consult
Does the engine turn over?
: yes
Do you smell gas?
: yes

What does the gas gauge say?
empty
low
full
: empty

problem-out of gas-cf-90
problem-flooded-cf-80
done with problem

Notice, that unlike Prolog, the inference does not stop after having found one possible value for problem. It finds all of the reasonable problems and reports the certainty to which they are known. As can be seen, these certainty factors are not probability values, but simply give some degree of weight to each answer.

1.1.3 user Uncertainty

The following dialog shows how the user's uncertainty might be entered into the system.
consult
Does the engine turn over?
: yes
Do you smell gas?
: yes cf 50
What does the gas gauge say?
empty
low
full
: empty

problem-out of gas-cf-90
problem-flooded-cf-40
done with problem

Notice in this case that the user was only certain to a degree of 50 that there was a gas smell. This results in the system only being half as sure that the problem is flooded.

1.1.4 Combining Certainties

Finally consider the following consultation which shows how the system combines evidence for a bad battery. Remember that there were two rules which both concluded the battery was weak with a certainty factor of 50.

:consult

Does the engine turn over?
: no
Are the lights weak?
: yes
Is the radio weak?
: yes

problem-battery-cf-75
done with problem

In this case the system combined the two rules to determine that the battery was weak with certainty factor 75. This propagated straight through rule 1 and became the certainty factor for problem battery.

1.1.5 Properties of Certainty Factors

There are various ways in which the certainty factors can be implemented, and how they are propagated through the system, but they all have to deal with the same basic situations:

1) rules whose conclusions are uncertain;
2) rules whose premises are uncertain;
3) user entered data which is uncertain;
4) combining uncertain premises with uncertain conclusions;
5) updating uncertain working storage data with new, also uncertain information;

Clam uses the certainty factor scheme which was developed for MYCIN, one of the earliest expert systems used to diagnose bacterial infections. Many commercial expert system shells today use this same scheme.

1.2 MYCIN's Certainty Factors

The basic MYCIN certainty factors (CFs) were designed to produce results that seemed intuitively correct to the experts. Others have argued for factors that are based more on probability theory and still others have experimented with more complex schemes designed to better model the real world. The MYCIN factors, however, do a reasonable job of modeling for many applications with uncertain information .
We have seen from the example how certainty information is added to the rules in the then clause. We have also seen how the user can specify CFs with input data. These are the only two ways uncertainty gets into the system.
Uncertainty associated with a particular run of the system is kept in working storage. Every time a value for an attribute is deter- mined by a rule or a user interaction, the system saves that attribute value pair and associated CF in working storage.
The CFs in the conclusion of the rule are based on the assumption that the premise is known with a CF of 100. That is, if the conclusion has a CF of 80 and the premise is known to CF 100, then the fact which is stored in working storage has a CF of 80. For example, if working storage contained:

turn over cf 100
smell gas cf 100

then a firing of rule 4

rule 4
if turn over and
smell gas
then problem is flooded cf 80
would result in the following fact being added to working storage: problem flooded cf 80

1.2.1 Determining Premise CF

However, it is unlikely that a premise is perfectly known. The system needs a means for determining the CF of the premise. The algorithm used is a simple one. The CF for the premise is equal to the minimum CF of the individual sub goals in the premise. If working storage contained:
turn over cf 80
smellgas cf 50
then the premise of rule 4 would be known with CF 50, the minimum of the two.

1.2.2 Combining Premise CF and Conclusion CF

When the premise of a rule is uncertain due to uncertain facts, and the conclusion is uncertain due to the specification in the rule, then the following formula is used to compute the adjusted certainty factor of the conclusion:

CF = RuleCF * PremiseCF /100.

Given the above working storage and this formula, the result of a firing of rule 4 would be:
problem is flooded cf 40

The resulting CF has been appropriately reduced by the uncertain premise. The premise had a certainty factor of 50, and the conclusion a certainty factor of 80, thus yielding an adjusted conclusion CF of 40.

1.2.3 Combining Cfs

Next consider the case where there is more than one rule which supports a given conclusion. In this case each of the rules might fire and contribute to the CF of the resulting fact. If a rule fires supporting a conclusion, and that conclusion is already represented in working memory by a fact, then the following formulae are used to compute the new CF associated with the fact. X and Y are the CFs of the existing fact and rule conclusion.

$$CF(X, Y) = X + Y(100 - X)/100. \quad X, Y \text{ both} > 0$$
$$CF(X, Y) = X + Y/(1- \min(|X|, |Y|)). \quad \text{one of } X, Y < 0$$
$$CF(X, Y) = -CF(-X, -Y). \quad X, Y \text{ both} < 0$$

For example, both rules 2 and 3 provide evidence for battery bad.
rule 2
if lights weak

then battery_bad cf 50.

rule 3
if radio weak
then battery bad cf 50.

Assume the following facts are in working storage:

lights weak cf 100
radio weak cf 100

A firing of rule 2 would then add the following fact:

battery_bad cf 50

Next rule 3 would fire, also concluding battery bad cf 50. However there already is a battery bad fact in working storage so rule 3 updates the existing fact with the new conclusion using the formulae above. This results in working storage being changed to:
battery bad cf 75

This case most clearly shows why a new inference engine was needed for Clam. When trying to prove a conclusion for which the CF is less than 100, we want to gather all of the evidence both for and against that conclusion and combine it. Prolog's inference engine will only look at a single rule at a time, and succeed or fail based on it.

1.3 Rule Format

Since we are writing our own inference engine, we can design our own internal rule format as well. (We will use something easier to read for the user.) It has at least two arguments, one for the IF or left hand side (LHS) which contains the premises, and one for the THEN or right hand side (RHS) which contains the conclusion. It is also useful to keep a third argument for a rule number or name. The overall structure looks like:
rule(Name, LHS, RHS).

The name will be a simple atom identifying the rule. The LHS and RHS must hold the rest of the rule. Typically in expert systems, a rule is read LHS implies RHS. This is backwards from a Prolog rule which can be thought of as being written RHS :- LHS, or RHS is implied by LHS. That is the RHS (conclusion) is written on the left of the rule, and the LHS (premises) is written on the right.
Since we will be backward chaining, and each rule will be used to prove or disprove some bit of information, the RHS contains one goal pattern, and its associated CF. This is:

rhs(Goal, CF)

The LHS can have many sub-goals which are used to prove or disprove the RHS:

lhs(Goalist)

where Goalist is a list of goals.
The next bit of design has to do with the actual format of the goals themselves. Various levels of sophistication can be added to these goals, but for now we will use the simplest form,

which is attribute value pairs. For example, gas gauge is an attribute, and low is a value. Other attributes have simple yes-no values, such as smell gas. An attribute-value pair will look like:

av(Attribute, Value)

where Attribute and Value are simple atoms. The entire rule structure looks like:

```
rule(Name,
        lhs( [av(Al, Vl), av(A2, V2), ....] ),
        rhs( av(Attr, Val), CF) ).
```

Internally, rule 5 looks like:

```
rule(5,
        lhs( [av(turns over, yes), av(gas gauge, empty)] ),
        rhs( av(problem, flooded), 80) ).
```

This rule format is certainly not easy to read, but it makes the structure clear for programming the inference engine. There are two ways to generate more readable rules for the user. One is to use operator definitions. The other is to use Prolog's language handling ability to parse our own rule format. The built-in definite clause grammar (DCG) of most Prologs is excellent for this purpose. Later in this chapter we will use DCG to create a clean user interface to the rules. The forward chaining system in a later chapter uses the operator definition approach.

1.4 The Inference Engine

Now that we have a format for rules, we can write our own inference engine to deal with those rules. Let's summarize the desired

1) combine certainty factors as indicated previously;
2) maintain working storage information that is updated as new evidence is acquired;
3) find all information about a particular attribute when it is asked for, and put that information in working storage.

The major predicates of the inference engine are:

```
findgoal
        %already know       fact
        %ask user
                askable
                query_user
        %derived from rules
                fg
                        rule
                        prove   findgoal
                adjust
                update
                        fact
                        combine
```

1.4.1 Working Storage

Let's first decide on the working storage format. It will simply contain the known facts about attribute-value pairs. We will use the Prolog database for them and store them as: fact (av(A, V), CF).

1.4.2 Find a value for an Attribute

We want to start the inference by asking for the value of a goal. In the case of the Car expert system we want to find the value of the attribute problem. The main predicate that does inferencing will be findgoal. In the Car expert system it could be called from an interpreter with the following query:

?- findgoal(av(problems X), CF).

The findgoal predicate has to deal with three distinct cases:

1) the attribute -value is already known;
2) there are rules to deduce the attribute-value;
3) we must ask the user.

The system can be designed to automatically ask the user if there are no rules, or it can be designed to force the knowledge engineer to declare which attribute values will be supplied by the user. The latter approach makes the knowledge base for the expert system more explicit, and also provides the opportunity to add more information controlling the dialog with the user. This might be in the form of clearer prompts, and/or input validation criteria.
We can define a new predicate askable that tells which attributes should be retrieved from the user, and the prompt to use. For example:

askable(live, 'Where does it live?').
With this new information we can now write findgoal.

1.4.3 Attribute value Already known

The first rule covers the case where the information is in working storage. It was asserted so we know all known values of the attribute have been found. Therefor we cut so no other clauses are tried.

```
findgoal( av(Attr, Val), CF) :-  ( av(Attr, Val), CF), !.
```

1.4.4 Ask User for attribute Value

The next rule covers the case where there is no known information, and the attribute is askable. In this case we simply ask.

```
findgoal(av(Attr, Val), CF) :-
                                not fact(av(Attr, ), ),
```

```
                askable(Attr, Prompt),
                query user(Attr, Prompt),
                !,
                findgoal(av(Attr, Val), CF).
```

The query user predicate prompts the user for a value and CF and then asserts it as a fact. The recursive call to findgoal will now pick up this fact.

```
query user(Attr, Prompt) :-
                write(Prompt),
                read(Val),
                read(CF),
                asserta( fact(av(Attr, Val), CF)).
```

1.4.5 Deduce Attribute Value from Rules

The final rule of findgoal covers the interesting case of using other rules. Remember that the inferencing is going to require looking for all rules which provide support for values for the sought attribute, and combining the CFs from them. This is done by calling fg, which uses a repeat fail loop to continue to find rules whose RHS conclude a value for the attribute. The process stops when the attribute is known with a CF of 100, or all the rules have been tried.

```
findgoal(Goal, CurCF) :- fg(Goal, CurCF).

fg(Goal, CurCF) :-    rule(N, lhs(IfList), rhs(Goal, CF)),
                      prove(IfList, Tally),
                      adjust(CF, Tally, NewCF),
                      update(Goal, NewCF, CurCF),
                      CurCF== 100, !.
fg(Goal, CF) :-             fact(Goal, CF).
```

The three new predicates called in fg are as follows:

- prove: prove the LHS premise and find its CF;
- adjust: combine the LHS CF with the RHS CF;
- update: update existing working storage values with the new conclusion

This is the guts of the inferencing process for the new inference engine. First it finds a rule whose RHS matches the pattern of the goal. It then feeds the LHS of that rule into prove which succeeds if the LHS can be proved. The prove predicate returns the combined CF from the LHS. If prove fails, backtracking is initiated causing the next rule which might match the goal pattern to be tried.

```
prove(IfList, Tally) :- prov(IfList, 100, Tally).
prov([], Tally, Tally).
prov([HlT], CurTal, Tally) :-
                findgoal(H, CF),
                min(CurTal, CF, Tal),
                Tal >= 20,
                prov(T, Tal, Tally).
min(X, Y, X) :- X =< Y, !.
min(X, Y, Y) :- Y =< X.
```

After prove succeeds, adjust computes the combined CF based on the RHS CF and the Tally from the LHS.

```
adjust(CF1, CF2, CF) :-        X is CF1*CF2 /100,
                               int_round(X, CF).

int_round(X, I) :-             X s= 0,
                               I is integer(X + 0.5).
int_round(X, I) :-             X <0,
                               I is integer(X - 0.5).
```

Then update combines the new evidence for this attribute-value pair with any existing known evidence. The first argument is the attribute - value pair just deduced, and the second is its CF. The third argument is the returned value for the CF when it is combined with existing facts for the attribute-value pair.

```
update(Goal, NewCF, CF) :-
                               fact(Goal, OldCF),
                               combine(NewCF, OldCF, CF),
                               retract( fact(Goal, OldCF) ),
                               asserta( fact(Goal, CF) ),

update(Goal, CF, CF) :-        asserta( fact(Goal, CF) ).

combine(CFl, CF2, CF) :-       CF1 >= 0,
                               CF2 >= 0,
                               X is CF1 + CF2*(100 - CF1)/100,
                               int_round(X, CF).
combine(CF1, CF2, CF) :-CF1 < 0,
                               CF2 < 0,
                               X is - ( -CF1 -CF2 * (100 + CF1)/100),
                               int_round(X, CF).
combine(CF1, CF2, CF) :- (CFl < 0; CF2 < 0),
                               (CFl > 0; CF2 > 0),
                               abs_minimum(CFl, CF2, MCF),
                               X is 100 * (CF1 + CF2) / (100 - MCF),
                               int_round(X, CF).
```

1.4.6 Negation

One last point is to deal with negation. The premises might also be of the form not goal. In this case we call findgoal for the goal, and complement the CF to find the degree of certainty of its negation. For example if a fact has a CF of 70, then not fact has a certainty of -70.

```
findgoal(not Goal, NCF) :-
                                    findgoal(Goal, CF),
                                    NCF is - CF, !.
```

1.5 Making the Shell

Now that the inference engine is built, it can become part of a shell. The code to build this first version of the Clam shell is the same as that used to build the Native shell. It consists of a command loop with the commands load, consult, and exit. Figure 3.2 shows the architecture of Clam.

```
super :-            repeat,
                    write('consult, load, exit'), nl,
                    write(' :' ) ,
                    read line(X),
                    doit(X),
                    X == exit.

doit(consult) :- top goals, !.
doit(load) :- load rules, !.
doit(exit) .
```

1.5.1 Starting the Inference

```
top_goals :-    top_goal(Attr),
                top(Attr),
                print_goal(Attr),
                fail.
                top_goals.

top(Attr) :-
                findgoal(av(Attr, Val), CF), !.

top(_) :-       true.

print_goal(Attr) :-nl,
                fact(av(Attr, X), CF),
                CF >= 20,
                outp(av(Attr, X), CF), nl,
                fail.
                print_goal(Attr) :-write('done with '), write(Attr), nl,
                nl.

outp(av(A, V), CF) :-
                output(A, V, PrintList),
                write(A-'cf'-CF),
                printlist(PrintList), !.
outp(av(A, V), CF) :- write(A-V-'cf'-CF) .

printlist([]).
printlist([H | T]) :-write(H), printlist(T) .
```

1.6 User Manuel for CLAM

CLAM is an EMYCIN-type expert system shell supporting backward chaining with uncertainty and explanations. Unlike NATIVE, which uses native Prolog syntax for the knowledge, CLAM uses Definite Clause Grammar (DCG) rules to parse a rule syntax of its own design.

```
c:\> clam
```

To start up CLAM from the listener, type:

```
?- [clam].      % equivalent to consult('clam.pro').
?- main.
```

An additional file is BLDRULES.PRO that uses DCG to convert a file of rules in Prolog syntax into CLAM syntax.
The commands are: consult, restart, load, list, trace, how, exit

They are entered without the ending period after the > prompt except for the load command which does a normal Prolog read(X) to get the file name.

consult Starts the consultation

restart Clears the knowledge base of known facts for rerunning a consultation.

load Loads the knowledge base. Other commands do not require Prolog syntax but this one does. Use 'car.ckb'. with the ending period.

list Lists the known facts in a consultation

trace Sets trace on or off, as in trace on or trace off

how For example, to get an explanation type:
how problem is battery

exit Exits.

1.6.1 Review and Discution questions

1 Add attribute object value triples to the knowledge representation of Clam.

2 There are other ways of dealing with uncertainty in the literature which could be used with Clam. A simple one would just use a few text modifiers such as weak, very weak, or strong and have rules for combining them. Implement this or some other scheme in Clam .

3 Isolate the predicates which are used for calculating certainty factors so it is easy to add additional methods. Implement them so the calling predicates do not need to know the syntax of the certainty factor, since they might be text, numbers, or more complex data structures.

4 Allow rules to have optional threshold values associated with them which override the default of 20. This would be an addition to the rule syntax as well as the code.

5 Visit http://vip.pdc.dk/vipexamples/esta/pdcindex.htm