University of Malta
BSc(Hons.) IT Year II

Problem Solving Techniques in Artificial Intelligence

Lecture 11

Rule Based Inference

- General Forward Chaining Rules
- Code for generalised adding and deleting
- Rule Ordering
- Non Termination
- Rules with Variables
- Optimisation

Department of Computer Science and AI
1998/9

General Rules

So far we have only considered *positive* forward chaining rules which refer to the *presence* of items in the database.

Now we consider conditions that are fulfilled by the *absence* of facts, e.g.

- If you believe p then believe q
- If you believe r and do not believe s then believe p
- If you believe q then do not believe t

In order to allow for negative conditions we need to change the form of the rules by associating with each item a *polarity* + or -.

```
(p,+)          -:  (q,+).
(r,+),(s,-)    -:  (p,+).
(q,+)          -:  (t,-)
```

Logic versus Computation

It is tempting to equate such rules with similar looking sentences of logic:

```
     P   -> Q
R & -S   -> P
     Q   -> -T
```

The similarity is rather superficial, however. Forward chaining rules area computational devices, whilst logical sentences simply state relationships.

- (p,-) should not be viewed as the negation of p. Rather, it represents *lack of evidence* for p.
- the -: operation should not be viewed as logical implication. It is rather in instruction to modify the database in certain ways.

Logic versus Computation Example

Consider the two logical rules:

```
-Q -> T
-Q -> -T
```

In logic these rules are said to *contrapose*: Since they are together equivalent to `(Q v T) / (Q v -T)`. In the case of logic, we can infer Q (this is the use of resolution).

Compare the following two forward chaining rules operating in an empty database.

```
(q,-) -: (t,+)
(q,-) -: (t,-)
```

Clearly, q is not added, and the absence of q is perfectly consistent. What does in fact happen is completely implementation dependent.

Example with intial database

`s. t.`

Rules

```
r1:  (p,+)          -:  (q,+).
r2:  (r,+),(s,-)    -:  (p,+).
r3:  (q,+)          -:  (t,-)
```

Execution Trace

`s. t.`

add(r).

`s. t. r.`

delete(s)

`t. r.`

r2: add(p) - r1: add(q) - r3: delete(t)

`q. p. r.`

Inefficient Implementation

```
add_gen_slow(X)
  :- clause(X,true), !.
add_gen_slow(X)
  :- assert(X),
     propagate_gen_slow(X,+).

del_gen_slow(X) :- not(clause(X,true)), !.
del_gen_slow(X)
  :- retract(X),
     propagate_gen_slow(X,-).

propagate_gen_slow(X,S) :-
  (If -: Then),
  amember((X,S),If),
  not((amember((Y,+),If), not(clause(Y,true)))),
  not((amember((Y,-),If), clause(Y,true))),
  update_gen_slow(Then),
  fail.

propagate_gen_slow(_,_).

update_gen_slow((P,+)) :- add_gen_slow(P).
update_gen_slow((P,-)) :- del_gen_slow(P).
```

Inefficiency

- This representation of the rules is inefficient because each time a single assertion is added or deleted from the database, *all* the preconditions of *all* the rules must be checked. Thus each add operation requires a time proportional to the size of the entire rule database.

- One way round this problem is to change the representation of the rules.

- Rather than represent the rules directly we give each rule a unique identifier.

- We then create pointers from each condition in the if part to this identifier, and from the identifier to the then part.

- The former are implemented by if(Ifcond,+/-,Id), and the latter by then(Id,N,Then,+/-) where N is a counter representing the number of unsatisfied conditions in the if part.

Optimised Representation

```
r1:  (p,+)           -:  (q,+).
r2:  (r,+),(s,-)     -:  (p,+).
r3:  (q,+)           -:  (t,-)
```

```
r1:
if(p,+,r1).
then(r1,1,q,+).

r2:
if(r,+,r2).
if(s,-,r2).
then(r2,2,p,+).

r3:
if(q,+,r3).
then(r3,1,t,-).
```

Execution Trace

	`r. s. t.`

add(r)	`if(r,+,r2).`
	`if(s,-,r2).`
	`then(r2,1,p,+).`

	`r. t.`

delete(s)	`if(r,+,r2).`
	`if(s,-,r2).`
	`then(r2,0,p,+).`

	`p. r. t.`
add(p)	`if(p,+,r1).` `then(r1,0,q,+).`
	`q. p. r. t.`
add(q)	`if(q,+,r3).` `then(r3,0,t,-).`
delete(t)	`q. p. r.`

Code for Generalised Adding and Deleting

```
%% Code from Shoham's Book

add(X) :- clause(X,true), !.
add(X) :- assert(X), propagate(X,+).

del(X) :- not(clause(X,true)), !.
del(X) :- retract(X), propagate(X,-).
```

Propagate

Find all rules whose lhs contains X, update their counters, and attempt to propagate the effect.

```
% X is the item, S is the polarity (+ or -)
propagate(X,S) :-
 if(X, S1,Id),
 once((retract(then(Id,N,Then,S2)),
       update_counter(N,S,S1,M),
       assert(then(Id,M,Then,S2)))),
 cond_propagate(M,Then,S2),
 fail.

propagate(_,_).
```

Once

once(+Goal) executes a conjunctive goal. If it succeeds it does not backtrack into the conjunctive goal.

```
once(Goal) :- Goal, !.
```

once/1 can in many cases be replaced with `->/2`. The only difference is how the cut behaves (see !/0). The following two clauses are identical:

1) `a :- once((b, c)), d.`
2) `a :- b, c -> d.`

Updating Counter

```
% If polarity is as expected,
% decrement counter.
% Otherwise, increment it.

update_counter(N,S,S,M) :- !, M is N-1.
update_counter(N,_,_,M) :- M is N+1.

% If counter is 0,
% propagate conclusion;
% otherwise do nothing

cond_propagate(0,Then,+) :- !, add(Then).
cond_propagate(0,Then,-) :- !, del(Then).
cond_propagate(_,_,_).
```

Rule Ordering

```
(p,+),(q,-) -:  (r,+).
(p,+),(r,-) -:  (q,+).
```

Assume that the database is initially empty, and we add p.

Either

- r will be added and q will not be added, or
- q will be added and r will not be added

Result of propagation can depend upon the order in which rules are examined.

Non Termination

```
(p,+),(q,+) -:  (p,-).
(p,-),(q,+) -:  (p,+).
```

suppose initial database contains neither p nor q, and then q is added to the database.

First p is added, then deleted, and the propagation continues infinitely.

- In general one cannot guarantee termination.
- However, there are restrictions on the rules that can guarantee termination, in particular
- If all *then* parts of rules have the same polarity' then forward chaining is guaranteed to terminate.
- Clearly, updates of only positive rules are guranteed to terminate.

Variables in Forward Chaining Rules

Assume that conjuncts appearing in rules will be of the form

$$p(X_1, X_2, \ldots, X_n)$$

for $n > 0$, and that as before, each conjunct is associated with a polarity, e.g.

```
(male(X),1),(married(X),0) -:  (bachelor(X),1)
```

- It is simple to write an inefficient forward-chaining program for rules that include variables.
- However, the optimisations that we applied to ground rules (rules without variables) cannot be obviously applied here.
- The basic problem is that the optimisations we have already introduced requires a unique counter to be associated with each rule.

Optimisation

The basic problem is that a rule with variables is not really a single rule.

Instead, it is equivalent to a collection of rules without variables – the set of all its possible instances.

```
(male(X),1),(married(X),0) -:  (bachelor(X),1)
```

is equivalent to

```
(male(jack),1),(married(jack),0)
                    -: (bachelor(jack),1)
(male(john),1),(married(john),0)
                    -: (bachelor(john),1)
etc.
```

In general, one cannot simply replace the rule with its instances because the number may be infinite.

Approaches to Optimisation

Basic problem is to avoid searching every condition of every rule whenever a new fact is added to the database.

```
r1:  (p(X),+),(q(X),+),(r(X),-) -:  s(X).
```

- **Rule indexing:** link each possible condition appearing on the lhs of rules with a list of rules that mention that condition. Whenever adding (or deleting) a new fact, only look at rules mentioned under that fact's index. Thus p(X), q(X), r(X) indexes r1 above.

- **Instance lists**: maintain, for each rule and for each positive conjunct on the rule's lhs, the instances of the conjunct that appear in the database. If the database contains facts p(a) and p(b), we would associate the list [p(a), p(b)] with the first conjunct.