University of Malta
BSc(Hons.) IT Year II

Problem Solving Techniques in Artificial Intelligence

Lecture 3

Reasoning

- Forward Chaining
- Representing Forward Chaining Rules
- Adding and Deleting Facts
- Optimisation

Department of Computer Science and AI

Forward Chaining

Consider an if-then rule such as

R1: If you believe both p and q then believe r

One way to use this rule is to wait until a query about r is asked, and replace it with a query about p and q.

This is a goal driven, backward chaining use of the rule as seen in Prolog, for example

```
r(X) :- p(X), q(X)
```

There is, however, another way to use the rule, namely, as soon as p and q are known to be true in the knowledge base, then add r.

This is a data-driven, forward-chaining use of the rules which may have ramifications if there are other rules present in the database such as

R2: If you believe both r then believe s

Forward versus Backward Chaining

- **Backward Chaining**
  - Is conservative, reducing goals to subgoals only to the extent needed to obtain a solution.
  - Does not modify the database
  - Is used to model reasoning
- **Forward Chaining**
  - Does not terminate until all ramifications have been computed
  - Does change the database
  - Is used as an automatic database routine rather than as a reasoning mechanism.

Forward Chaining as a Special Case

As described so far, FC involves addition of a fact as a result of addition of another fact. This is a special case of one knowledge base event triggering another. For instance, we could also envisage:

- A deletion event triggering an addition event
- A deletion event triggering a deletion event
- An addition event triggering a deletion event.

We can also envisage recording the depedency of the conclusion upon the premise, so that if the premise is later removed, so is then conclusion. This is the basis of *reason maintenance systems*

Representing Forward Chaining Rules in Prolog

First we must represent FC rules and label them as such. We define a symmetric operator* '`-:`':

```
op(1150,xfx,-:)
```

For the moment we will restrict ourselves to *positive rules* which only mention facts present in the database on the lhs and facts to be added on the rhs.

Examples of positive rules are

```
r   -: q.
r,s -: p.
q   -: t.
```


*See Bratko pp 83-87 for further information about operator declarations

Addition of Database Items

Suppose we wish to model

1. Addition of a new idem to the database.
2. Propagation of its effects

Assuming that rules have at most a single condition on the lhs, one definition would be as follows:

if the fact is already there, do nothing.

```
add_single(X) :- clause(X,true), !.
```

otherwise assert it, find all triggered rules, and propagate

```
add_single(X) :- assert(X),
                 (X -:B),
                 add_single(B),
                 fail.
add_single(_).
```

Complete Program

```
op(1150,xfx,-:).

p   -: q.
r,s -: p.
q   -: t.

add_single(X)  :- clause(X,true), !.
add_single(X)  :- assert(X),
                  (X -:B),
                  add_single(B),
                  fail.
add_single(_).

PROGRAM TRACE
2 ?- t.
[WARNING: Undefined predicate: `t/0']
No
3 ?- add_single(p).
Yes
4 ?- t.
Yes
```

Rules with Several L.H. Conditions

```
R1. p   -: q.
R2. r,s -: p.
R3. q   -: t.
```

`add_single` is unable to handle rules such as R2 that contain more than one condition. The following program addresses the problem:

```
add_positive_slow(X) :- clause(X,true), !.
```

find all rules whose lhs contains X, make sure all other lh items are in the database, and apply add_positive_slow to their rh sides

```
add_positive_slow(X) :-
  assert(X),
  (If -: Then),
  amember(X,If),
  not((amember(Y,If), not(clause(Y,true)))),
  add_positive_slow(Then),
  fail.

add_positive_slow(_).
```

Optimisation

- add_positive_slow is highly inefficient since each add operation requires *all* conditions in *all* rules to be examined (i.e time proportional to the entire forward chaining database).

- The complexity should be independent of the total number of rules or the number of conditios in the rules.

- One route towards optimisation is to improve the representation of the rules as follows:

  - Give each rule a unique identifier

  - Create pointers from each condition to this identifier: if(Cond,Id).

  - Create pointers from the identifier to the then part: then(Id,N,Then).

Optimising The Rules

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

For example, the above rules would be represented as follows. The counter N represents the number of unsatisfied rhs conditions.

if(p,r1).
then(r1,1,q).
	if(r,r2).
	if(s,r2).
	then(r2,2,p).
		if(q,r3).
		then(r3,1,t).

When **add**ing a new fact,

- all rules in whose if part that fact appears are accessed, and their counter is decremented.

- If the counter ever reaches zero, then the then part is **add**ed to the database.

Database Snapshots

```
if(p,r1).          if(r,r2).          if(q,r3).
then(r1,1,q).      if(s,r2).          then(r3,1,t).
                   then(r2,2,p).
```

Adding the fact r yields

```
r.                 if(r,r2).          if(q,r3).
if(p,r1).          if(s,r2).          then(r3,1,t).
then(r1,1,q).      then(r2,1,p).
```

Then adding the fact q yields

```
r. q. t.           if(r,r2).          if(q,r3).
if(p,r1).          if(s,r2).          then(r3,0,t).
then(r1,1,q).      then(r2,1,p).
```

Deleting Items

We shoud also note that with this representation, we have to take account of the effect of deleting database items. Thus if we delete r at this point, we might expect the counter on rule 2 to be incremented.

r. q. t.
if(p,r1).
then(r1,1,q).
	if(r,r2).
	if(s,r2).
	then(r2,1,p).
		if(q,r3).
		then(r3,0,t).

q. t.
if(p,r1).
then(r1,1,q).
	if(r,r2).
	if(s,r2).
	then(r2,2,p).
		if(q,r3).
		then(r3,0,t).

Definition of add_positive

```
add_positive(X) :- clause(X, true), !.

add_positive(X) :-
  assert(X),
  if(X,Id),
    once((retract( then(Id,N,Then)),
          M is N-1,
          assert(then(Id,M,Then)))),
  cond_propagate_positive(M, Then),
  fail.

add_positive(_).


cond_propagate_positive(0, Then) :-
  !,
  add_positive(Then),

cond_propagate_positive(_,_).
```

Definition of del_positive

```
del_positive(X) :- not(clause(X, true)), !.

del_positive(X) :-
  retract(X),
  if(X,Id),
    once((retract( then(Id,N,Then)),
          M is N+1,
          assert(then(Id,M,Then)))),
  fail.

del_positive(_).
```

Remarks

- The complexity of this implementation is proportional to the number of pointers from the asserted or deleted fact to rule identifiers

- The implementation of del_positive does not attempt to propagate the retraction of facts. Suppose in the above example, we were to delete q. Strictly speaking, t ought to be deleted as well, but this does not happen. This is the domain of *reason maintenance systems*.