Ch. 5 Controlling Backtracking

Backtracking

- *Backtracking* is the attempt to (re)satisfy a goal by exploring alternative ways to satisfy it.
- *Chronological backtracking* is backtracking in which we always go back to the most recent goal which still has unexplored possible alternative solutions.

Backtracking

- Prolog will automatically backtrack if this is necessary to satisfy a goal.
- After a query returns an answer, we can ask for additional answers by typing a semi-colon (;). This causes backtracking to look for alternative solutions.

Example

```
parent(bill, sally).
parent(sue, sally).

?- parent(X, sally).
X = bill ;
X = sue ;
No
```

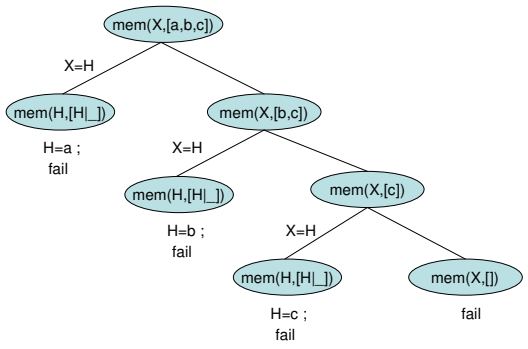
Example

- Given this query, the system tries the first fact, which succeeds giving the answer *bill*.
- The semi-colon initiates backtracking, so the system tries the second fact, which also succeeds giving the answer *sue*.
- The next semi-colon initiates backtracking again, but there are no more relevant facts or rules to use, so the goal fails.

Backtracking Example

```
member(Head, [Head|_]).
member(X, [_|Tail]) :- member(X, Tail).
?- member(X, [a,b,c]).
X = a ;
X = b ;
X = c ;
No
```

And-Or Tree for member Procedure



Example

(ex) a double-step function:

f(X, 0) :- X<3.

f(X, 2) :- 3=<X, X<6.

f(X, 4) :- 6=<X.

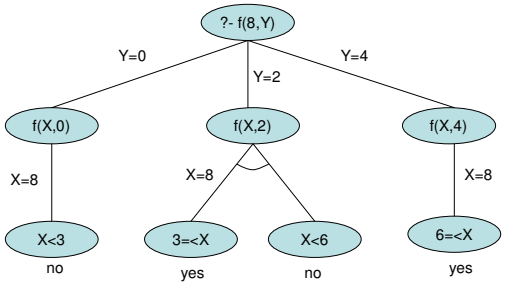
?- f(8, Y).

Y = 4

Example

- Given this query, the system tries the first rule, but this fails because 8 is not less than 3.
- It then tries the second rule, but this also fails because 8 is not less than 6.
- Finally, it tries the third rule and this one succeeds giving an answer of 4.

And-Or Tree for Double-Step Function



Pointless Backtracking

?- f(1,Y).

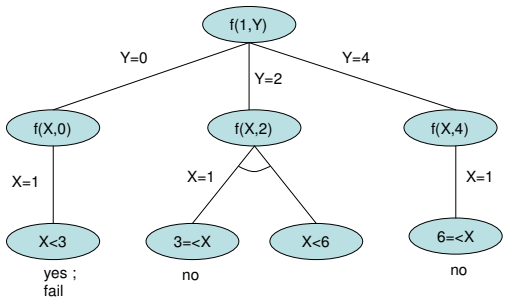
Y=0 ;

No

Pointless Backtracking

- Given this query, the first clause of the f procedure returns the answer Y=0.
- Entering the semi-colon triggers a search for alternative solutions.
- But this backtracking is *pointless* since we already know it cannot succeed.
- For any given input, the f procedure can produce only one possible output.

And-Or Tree for Pointless Backtracking



Preventing Pointless Backtracking

- We can prevent pointless backtracking by using the *cut* command.
- Cuts are used to make code more efficient.
- Cut is symbolized by the exclamation point (!).
- A cut is a goal that always succeeds.
- A cut is like a one-way door that lets you out, but doesn't let you back in.

Example with Cuts

(ex) the double-step function with cuts:

f(X, 0) :- X<3, !.

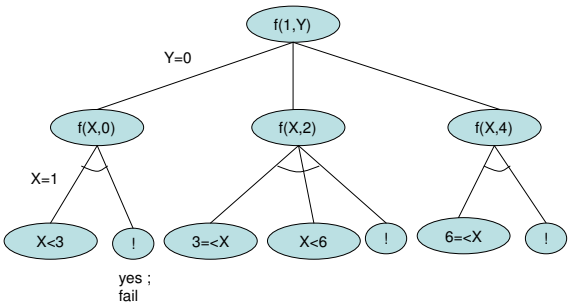
f(X, 2) :- 3=<X, X<6, !.

f(X, 4) :- 6=<X.

Example with Cuts

- This version of the f procedure returns the same values as the previous version, but now, once an answer has been found, pointless backtracking is prevented by the use of cuts.
- It is unnecessary to add a cut to the third clause, because there are no alternatives beyond it.

And-Or Tree for f with Cuts



Green Cuts

(ex) f with green cuts:

f(X, 0) :- X<3, !.

f(X, 2) :- 3=<X, X<6, !.

f(X, 4) :- 6=<X.

- The types of cuts used in this example are called *green cuts*.

Green Cuts

- The defining feature of green cuts is that if they are removed then the procedure will still produce correct answers, although maybe less efficiently.

Further Inefficiencies

(ex) f with green cuts:

f(X, 0) :- X<3, !.

f(X, 2) :- 3=<X, X<6, !.

f(X, 4) :- 6=<X.

Further Inefficiencies

- Although the cuts have removed some inefficiencies from this code, there are still other sources of inefficiency:
  - 3=<X is a redundant test since we already know that X<3 is false.
  - 6=<X is a redundant test since we already know that X<6 is false.

Removing more inefficiencies

- Getting rid of the redundant tests, we get the following definition for f:

f(X, 0) :- X<3, !.

f(X, 2) :- X<6, !.

f(X, 4).

Removing more inefficiencies

- This code can be read as:

  if X<3 then Y=0;

  else if X<6 then Y=2;

  else Y=4;
- This is the most efficient version of this procedure.

Red Cuts

(ex) f with red cuts:

f(X, 0) :- X<3, !.

f(X, 2) :- X<6, !.

f(X, 4).

- The types of cuts used in this code are called *red cuts*.

Red Cuts

- The defining feature of red cuts is that if they are removed from the procedure then the procedure may produce incorrect answers.

Removing Red Cuts

(ex) If we remove the cuts, we get:

```
f(X, 0) :- X<3.
f(X, 2) :- X<6.
f(X, 4).
?- f(2, X).
X = 0;      % right answer
X = 2;      % wrong answer
X = 4;      % wrong answer
No
```

Another Example

- the *max* procedure:

```
max(X, Y, X) :- X>=Y.
max(X, Y, Y) :- X<Y.
```

- can be rewritten with a red cut as:

```
max1(X, Y, X) :- X>=Y, !.
max1(X, Y, Y).
```

What Cut Does

1. Cannot backtrack through a cut.
2. Cannot try alternative rules for the parent goal of the cut.

cut example

```
b.
d.
e.
f.
v.
a :- b, c, d.
c :- e, !, f, fail.
c :- v.
```

cut example trace

```
[trace] 3 ?- a.
   Call: (7) a ? creep
   Call: (8) b ? creep
   Exit: (8) b ? creep
   Call: (8) c ? creep
   Call: (9) e ? creep
   Exit: (9) e ? creep
   Call: (9) f ? creep
   Exit: (9) f ? creep
   Call: (9) fail ? creep
   Fail: (9) fail ? creep
   Fail: (8) c ? creep
   Fail: (7) a ? creep
No
```

An Effect of Cut

1 ?- member(X, [a, b, c]).
X = a ;
X = b ;
X = c ;
No

An Effect of Cut

- member procedure with cuts

```
mem1(H, [H|_]) :- !.
mem1(E, [_|T]) :- mem1(E, T).
```

2 ?- mem1(X, [a, b, c]).
X = a ;
No

Prolog Negation

(ex) Mary likes all animals except snakes.
likes(mary, X) :- animal(X), \+ snake(X).

Prolog Negation

- Negation in Prolog can be written as:

\+ P
\+(P)
not P
not(P)

Prolog Negation

- Negation in Prolog is not *logical negation* but instead is *negation as failure.*

Prolog Negation

logical not:

P	¬P
T	F
F	T

negation as failure:

P	\+P
succeeds	fails
fails	succeeds

The Closed-World Assumption

?- human(mary).

No

?- \+ human(mary).

Yes

The Closed-World Assumption

- These answers have their usual meanings only under *the closed-world assumption* – the knowledge-base contains all relevant information about the given domain.
- Therefore if something is not provable using the available facts and rules, then it must be false.

A definition of *not* in Prolog

not can be defined in Prolog as:

not P ≡ (P, !, fail) ; true

A Problem with *not*

good_standard(jeanLuis).

expensive(jeanLuis).

good_standard(francesco).

reasonable(Restaurant) :- \+ expensive(Restaurant).

?- good_standard(X), reasonable(X).

X = francesco

?- reasonable(X), good_standard(X).

No – because there is an expensive restaurant, the first goal fails!

Explanation of a Problem with *not*

reasonable(Restaurant) :-

\+ expensive(Restaurant).

- The effect of this rule differs depending on whether or not *Restaurant* is bound.
- If *Restaurant* is bound, then that restaurant is assumed to be reasonable if it is not provable that it is expensive. This is what happened in the first query.

Explanation of a Problem with *not*

reasonable(Restaurant) :- \+ expensive(Restaurant).

- If *Restaurant* is unbound, then the system tries to find an expensive restaurant. If it doesn't find one then *reasonable(Restaurant)* succeeds, but if it does find one (any one) then *reasonable(Restaurant)* fails! Therefore this rule cannot be used to find a reasonable restaurant. This is what happened in the second query.
- Therefore, the ordering of goals in a query can matter.