Procedural control of reasoning

Automated proving methods answer a question by trying all logically permissible options in the knowledge base. These reasoning methods are domain-independent. But in some situations it is not feasible to search all logically possible ways to find a solution.
We often have an idea about how to use knowledge and we can "guide" an automated procedure based on properties of the domain.
We will see how knowledge can be expressed to control the backward-chaining reasoning procedure.

Facts and rules

The clauses in a KB can be divided in two categories:

- facts – are ground atoms (without variables).
- rules – are conditionals that express new relations – they are universally quantified.

$Mother(jane, john)$
$Father(john, bill)$
...
$Parent(x, y) \Leftarrow Mother(x, y)$
$Parent(x, y) \Leftarrow Father(x, y)$

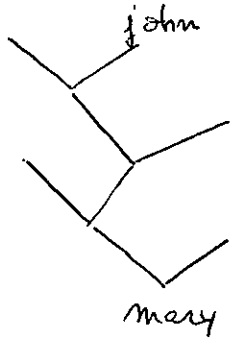
Rules involve chaining and the control issue regards the use of the rules to make it most effective.

Rule formation and search strategies

We can express the Ancestor relation in three logically equivalent ways:

1. $Ancestor(x, y) \Leftarrow Parent(x, y)$
   $Ancestor(x, y) \Leftarrow Parent(x, z) \wedge Ancestor(z, y)$
2. $Ancestor(x, y) \Leftarrow Parent(x, y)$
   $Ancestor(x, y) \Leftarrow Parent(z, y) \wedge Ancestor(x, z)$

3. $Ancestor(x,y) \Leftarrow Parent(x,y)$
$Ancestor(x,y) \Leftarrow Ancestor(x,z) \wedge Ancestor(z,y)$



1. We search top-down in the family tree.
2. We search down-top.
3. We search in both directions in the same time.

If people had on average one child, then 1) would be of order d and 2) of order 2^d, where d is the depth of search. if people had more than 2 children, 2) would be a better option.

Algorithm design

The Fibonacci series $\begin{cases} x_0 = 0 \\ x_1 = 1 \\ x_{n+2} = x_{n+1} + x_n, \; n \geq 0 \end{cases}$

KB $\begin{cases} Fib(0,1) \\ Fib(1,1) \\ Fib(s(s(n)),v) \Leftarrow Fib(n,y) \wedge Fib(s(n),z) \wedge Plus(y,z,v) \\ Plus(0,z,z) \\ Plus(s(x),y,s(z)) \Leftarrow Plus(x,y,z) \end{cases}$

Most of the computation is redundant

$Fib(10,\_)$ calls $Fib(9,\_)$ and $Fib(8,\_)$

$Fib(11,\_)$ calls $Fib(10,\_)$ and $Fib(9,\_)$

Each application of Fib calls Fib twice and it generates an exponential number of Plus subgoals.

An alternative is $\begin{cases} Fib(n,v) \Leftarrow F(n,1,0,v) \\ F(0,y,z,y) \\ F(s(n),y,z,v) \Leftarrow Plus(y,z,s) \wedge F(n,s,y,v) \end{cases}$

$F(n,\underbrace{-,-}_{\text{2 consecutive Fib numbers}},v)$ — the solution obtained when n is 0

starts from n towards 0

Goal order

From logical point of view, all ordering of subgoals are equivalent, but the computational differences can be significant.

For example

$$AmericanCousin(x,y) \Leftarrow American(x) \wedge Cousin(x,y)$$

we have two options:

- find an American and see if he/she is a cousin
- find a cousin and see if he/she is American

In this case, solving first $Cousin(x,y)$ and then $American(x)$ is better than the other way around.

Predicate ! ("cut") in PROLOG – backtracking control and negation as failure

! is always true; it prevents backtracking in the place of occurrence in the program.
If ! doesn't change the declarative meaning of the program, then it is called green; otherwise it is red.

The function $f(x) = \begin{cases} 0, & x \leq 3 \\ 2, & x \in (3,6] \\ 4, & x > 6 \end{cases}$ can be implemented as:

KB
```
f(X, 0) :- X =< 3.
f(X, 2) :- 3 < X, X =< 6.
f(X, 4) :- 6 < X.
```

```
?- f(1,Y), 2 < Y.
```

$X=1, Y=0$ $\quad 1 =< 3, 2 < 0$ false

$X=1, Y=2$ false

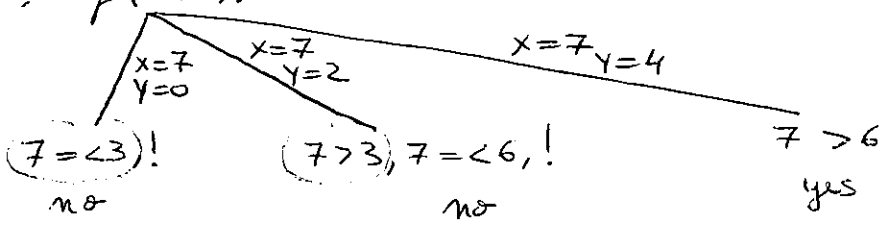
$X=1, Y=4$ false

The program should have stopped after the first check.

```
f(X,0) :- X =< 3, !.
f(X,2) :- 3 < X, X =< 6, !.
f(X,4) :- 6 < X.
```

green!

?- f(7,Y).



redundante tests

```
f(X,0) :- X =< 3, !.
f(X,2) :- X =< 6, !.
f(X,4).
```

red! - if we remove ! and ask:

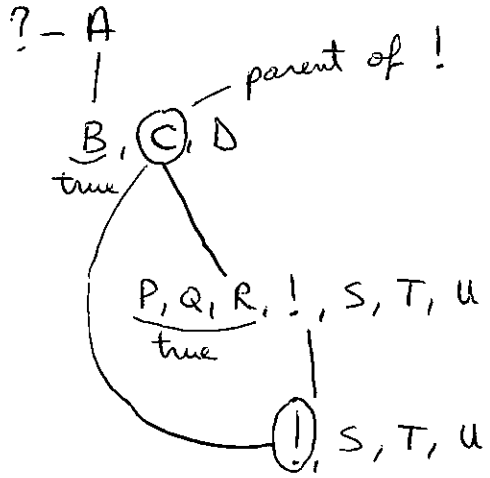
?- f(1,Y).
Y=0;
Y=2;
Y=4;
false

The parent of a "cut" is that PROLOG goal that matches the head of the rule that contains that "cut".

```
C :- P, Q, R, !, S, T, U.
C :- V.
A :- B, C, D.
```

?- A.

Backtracking is possible for P, Q, R, but as soon as ! is executed, all of the alternative solutions are suppressed. Also, the alternative C :- V will be suppressed.



- in the goal tree, backtracking is prevented between ! and its parent.
- ! affects only the execution of C.

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

```
member(X, [X|L]) :- !.
member(X, [_|L]) :- member(X, L).
```

Given KB
```
p(1).
p(2) :- !.
p(3).
```
what are PROLOG answers to the following questions:

```
?- p(X).
?- p(X), p(Y).
?- p(X), !, p(Y).
```

Negation as failure - predicate "fail" is always false.

John likes all animals, with the exception of snakes.

```
likes(john, X) :- snake(X), !, fail.
likes(john, X) :- animal(X).
```

We define the unary predicate "not" as following: not(G) fails if G succeeds; otherwise not(G) succeeds.

```
not(G) :- G, !, fail.
not(G).
```

```
likes(john, X) :- animal(X), not(snake(X)).
```

Procedurally, we distinguish between two types of negative situations with respect to a goal G:

- being able to solve $\neg G$
- being unable to solve G - this can happen when we run out of options when trying to prove that G is true.

"Not" in PROLOG doesn't correspond exactly to the mathematical negation. When PROLOG processes a "not" goal, it doesn't try to solve it directly, but to solve the opposite.

If the opposite cannot be demonstrated, then PROLOG assumes that the "not" goal is solved.

Such a reasoning is based on the Closed-World Assumption. That is to say that if something is not in the KB or it cannot be derived from the KB, then it is not true and consequently, its negation is true.

For example, if we ask:

```
?- not(human(mary)).
```

the answer is "yes" if human(mary) is not in KB. But it should not be understood as "Mary is not a human being", but rather "there is not enough information in the program to prove that Mary is a human being".

Usually, we do not assume the "Closed-World" – if we do not explicitly say "human(mary)", we do not implicitly understand that Mary is not a human being.

Other examples:

1)
```
composite(N) :- N>1, not(primeNumber(N)).
```

The failure to prove that a number greater than 1 is prime is sufficient to conclude that the number is composite.

2)
```
good(renault).
good(audi).
expensive(audi).
reasonable(Car) :- not(expensive(Car)).
```

```
?- good(x), reasonable(x).
```

```
?- reasonable(x), good(x).
```

! is useful and, in many situations, necessary, but it must be used with special attention.