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.

These slides are mainly based on Ronald J. Brachman, Hector J. Levesque. Knowledge representation and reasoning, Morgan Kaufmann, 2004.

Facts and rules

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

- Facts – are ground terms (without variables)
- Rules – are conditionals that express new relations – they are universally quantified.

Mather(jane,john)

Father(john,bill)

…

Parent(x,y) ⇐ Mother(x,y)

Parent(x,y) ⇐ 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

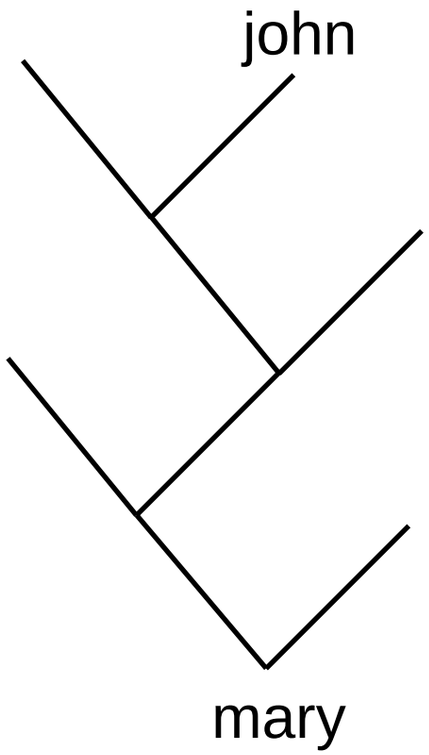
We can express the Ancestor relation in three logically equivalent ways:

1. Ancestor(x,y) ⇐ Parent(x,y)
   Ancestor(x,y) ⇐ Parent(x,z) ∧ Ancestor(z,y)

2. Ancestor(x,y) ⇐ Parent(x,y)
   Ancestor(x,y) ⇐ Parent(z,y) ∧ Ancestor(x,z)

3. Ancestor(x,y) ⇐ Parent(x,y)
   Ancestor(x,y) ⇐ Ancestor(x,z) ∧ Ancestor(z,y)

Rule formation and search strategies



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

Fib(0,1)

Fib(1,1)

Fib(s(s(n)),v) ⇐ Fib(n,y) ∧ Fib(s(n),z) ∧ Plus(y,z,v)

Plus(0,z,z)

Plus(s(x),y,s(z)) ⇐ Plus(x,y,z)

Note: 0 is shortcut for zero; 1 for s(zero); 2 for s(s(zero)) and so on.

Algorithm design

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

Fib(0,1)

Fib(1,1)

Fib(s(s(n)),v) ⇐ Fib(n,y) ∧ Fib(s(n),z) ∧ Plus(y,z,v)

Plus(0,z,z)

Plus(s(x),y,s(z)) ⇐ Plus(x,y,z)

Note: 0 is shortcut for zero; 1 for s(zero); 2 for s(s(zero)) and so on.

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.

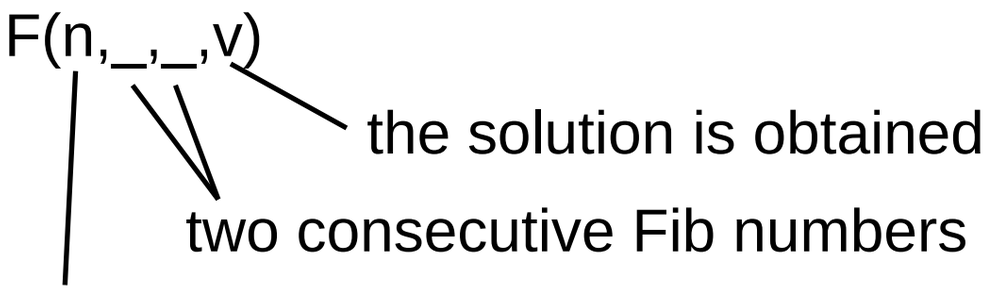
Algorithm design

An alternative is

$$Fib(n,v) \Leftarrow F(n,1,0,v)$$

$$F(0,y,z,z)$$

$$F(s(n),y,z,v) \Leftarrow Plus(y,z,s) \wedge F(n,s,y,v)$$



Goal order

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

For example

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

We have two options:

Find an American and see if he is a cousin.

Find a cousin and see if he is American.

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

Backtracking control and negation as failure

- predicate ! (“cut”) in PROLOG[1]

! is always true; it prevents backtracking in the place it occurs 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:

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

1. Ivan Bratko. Prolog Programming for Artificial Intelligence, Pearson Education Canada, 4th Edition, 2011.

Backtracking control and negation as failure

- predicate ! (“cut”) in PROLOG

! is always true; it prevents backtracking in the place it occurs 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:

```
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, 1=<3,2<0 false

X=1,Y=2 false

X=1, Y=4 false

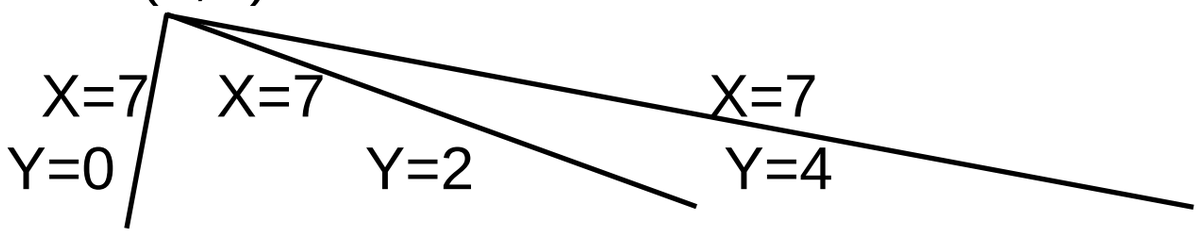
The program should have stopped after the first check.

Backtracking control and negation as failure

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

green !

?-f(7,Y).



7=<3, !	7>3, 7=<6,!	7>6	redundant tests
no	no	yes	

```
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

Backtracking control and negation as failure

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

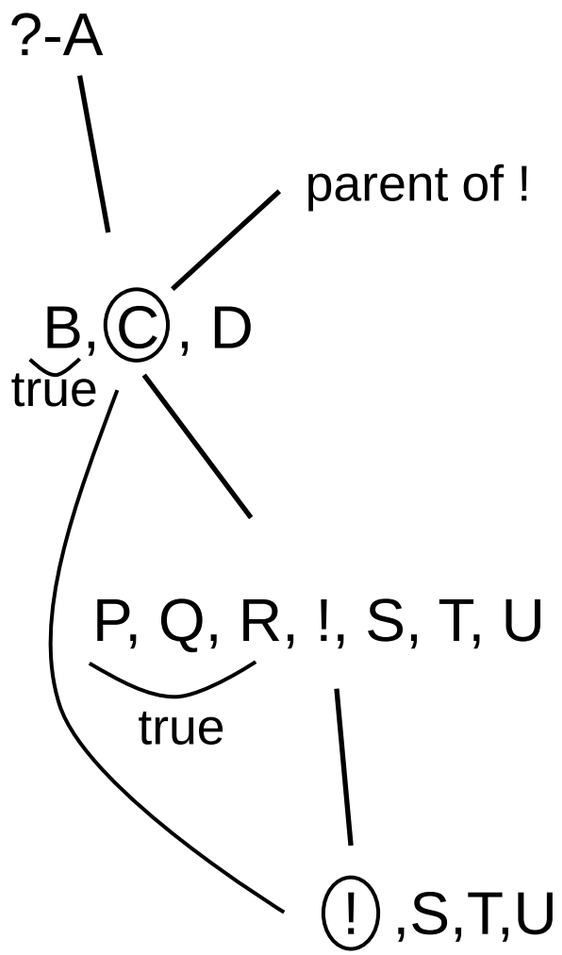
```
C:-P,Q,R,!,S,T,U.
C:-V.
A:-B,C,D.
```

?-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.

Backtracking control and negation as failure



in the goal tree, backtracking is prevented

between ! and its parent

! affects only the execution of C

Backtracking control and negation as failure

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

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

Given the following 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).
```

Backtracking control and negation as failure

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).
```

Now we can write

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

Backtracking control and negation as failure

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

- being able to solve ¬G
- being unable to solve G – this happens 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.

Backtracking control and negation as failure

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 information in the program to prove that Mary is a human being”

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

Backtracking control and negation as failure

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.

Rules in production system

Production systems formalize knowledge in a certain form called production rules and use forward-chaining reasoning to derive new things. They are used in many practical applications (e.g., experts systems).

A production system maintains a working memory (WM) which contains assertions that are changing during the operation of the system.

A production rule consists of an antecedent set of conditions and a consequent set of actions:

IF conditions THEN actions

The antecedent conditions are tests applied to the current state of the WM; the consequent actions are a set of actions that modify the WM.

Rules in production system

The production system operates in a three step cycle that repeats until no more rules are applicable to the WM:

1. Recognize – find the applicable rules, i.e. those rules whose antecedent conditions are satisfied by the current WM
2. Resolve conflicts – among the rules found at the first step (called conflict set), choose those to fire
3. Act – change the WM by performing the consequent actions of all the rules selected at step 2

Rules in production system

Working Memory

WM consists of a set of working memory elements (WMEs).

A WME has the from:

(type $attribute_1$:$value_1$…$attribute_n$:$value_n$)

where type, $attribute_i$ and $value_i$ are atoms.

Examples

(person age:21 home:bucharest)

(student name:john dept:computerScience)

Rules in production system

Declaratively, a WME is an existential sentence:

$$\exists x[type(x) \wedge attribute_1(x)=value_1 \wedge \ldots \wedge attribute_n(x)=value_n]$$

WMEs represent objects and relationships between them can be handled by reification (i.e. transforming a sentence into an object)

Purchases(john,bike,nov11)

Purchases(p3) ^ agent(p3)=john ^ object(p3)=bike…

For example,

(basicFact relation:olderThan firstArg:john secondArg:mary)

may be used to express the fact that John is older than Mary.

Rules in production system

Production rules

The conditions of a rule are connected by conjunctions. A condition can be positive or negative (written as —cond) and has the form:

$$(\text{type attribute}_1\text{:specification}_1 \ldots \text{attribute}_k\text{: specification}_k)$$

where each specification is one of the following:

- an atom
- a variable
- an evaluable expression within []
- a test within {}
- conjunction, disjunction or negation of a specification

Rules in production system

For instance,

(person age:[n+1] height:x)

is satisfied if there is a WME whose type is person and whose age attribute is n+1, where n is specified elsewhere.

If the variable x is not bound, then x will be bound with the value of the attribute height; otherwise, the value of the attribute height in the WME has to be the same as the value of x.

—(person age:{<20 ∧ >6}) is satisfied if there is no WME in WM whose type is person and age is between 6 and 20 (negation as failure)

Rules in production system

A WME matches a condition if the types are identical and for each attribute/specification pair in the condition, there is a corresponding attribute/value pair in the WME and the value matches the specification.

The matching WME may have other attributes that are not mentioned in the condition:

condition (control phase:5)

matching WME (control phase:5 position:7…)

Rules in production system

The consequent set of actions of production rules have a procedural interpretation. All actions are executed in sequence and they can be of the following types:

1. ADD – adds a new WME to WM
2. REMOVE i – removes from WM the WME that matches the i^{th} condition in the antecedent of the rule; not applicable if the condition is negative
3. MODIFY i (attribute specification) – modifies the WME that matches the i^{th} condition in the antecedent by replacing its current value for attribute by specification; not applicable if the condition is negative.

Obs. In ADD and MODIFY, the variables that appear refer to the values obtained when matching the antecedent of the rule.

Rules in production system

Example 1

IF (student name:x) THEN ADD (person name:x)

(the equivalent of ∀x.Student(x) ⊃ Person(x))

Example 2 – assume that a rule added a WME of type birthday

IF (person age:x name:n) (birthday who:n)

THEN MODIFY 1 (age [x+1])

REMOVE 2

Rules in production system

Example 1

IF (student name:x) THEN ADD (person name:x)

(the equivalent of ∀x.Student(x) ⊃ Person(x))

Example 2 – assume that a rule added a WME of type birthday

IF (person age:x name:n) (birthday who:n)

THEN MODIFY 1 (age [x+1])

REMOVE 2

/

to prevent the rule from firing again

Rules in production system

Example 3 – to indicate the phase of computation

IF (starting) THEN REMOVE 1

ADD (control phase:1)

…

IF (control phase:x) …other conditions

THEN MODIFY 1 (phase [x+1])

IF (control phase:5) THEN REMOVE 1

Rules in production system

Example 4 – we have three cubes in a heap, each of different size. With a robotic hand, we want to move the cubes in the positions named 1, 2 and 3. The goal is to place the largest cube in position 1, the middle one in 2 and the smallest one in position 3.

WM
1. (counter value:1)
2. (cube name:A size:10 position:heap)
3. (cube name:B size:30 position:heap)
4. (cube name:C size:20 position:heap)

Rules
1. IF (cube position:heap name:n size:s)
   —(cube position:heap size:{>s})
   —(cube position:hand)
   THEN MODIFY 1 (position hand)
2. IF (cube position:hand) (counter value:i)
   THEN MODIFY 1 (position i)
        MODIFY 2 (value [i+1])

there are no conflicts because only one rule can fire at a time

Rules in production system

System operation:

Rule 1 → n=B, s=30 →WME3 changes to (cube name:B size:30 position:hand)

Rule 2 → i=1 → WME3 changes to (cube name:B size:30 position:1)
WME1 changes to (counter value:2)

Rule 1 → n=C, s=20 →WME4 changes to (cube name:C size:20 position:hand)

Rule 2 → i=2 → WME4 changes to (cube name:C size:20 position:2)
WME1 changes to (counter value:3)

Rule 1 → n=A, s=10 →WME2 changes to (cube name:A size:10 position:hand)

Rule 2 → i=3 → WME2 changes to (cube name:A size:10 position:3)
WME1 changes to (counter value:4)

No rule is applicable now, so the production system halts.

Final WM
1. (counter value:4)
2. (cube name :A size:10 position:3)
3. (cube name :B size:30 position:1)
4. (cube name :C size:20 position:2)

Rules in production system

To do:
Compute the greatest common factor of two integers.
Represent the initial WM, the rules of the production system and the final WM after the system operates.

Example: represent the input values 6 and 9 in the initial WM; after the system runs, the value 3 should be in the final WM.