Knowledge Representation & Reasoning

Shyamanta M Hazarika
CSE, SoE
Tezpur University

Prolog

Horn clauses form the basis of Prolog

Append(nil,*y*,*y*)

Append(*x*,*y*,*z*) $\Rightarrow$ Append(cons(*w*,*x*),*y*,cons(*w*,*z*))

What is the result of appending [c] to the list [a,b] ?

Append(cons(a,cons(b,nil)), cons(c,nil), *u*) goal

| *u* / cons(a,*u*′)

Append(cons(b,nil), cons(c,nil), *u*′)

| *u*′ / cons(b,*u*″)

Append(nil, cons(c,nil), *u*″)

solved: *u*″ / cons(c,nil)

So goal succeeds with *u* = cons(a,cons(b,cons(c,nil)))
that is: Append([a b],[c],[a b c])

With SLD derivation, can always extract answer from proof

$H \models \exists x\, \alpha(x)$

iff

for some term *t*, $H \models \alpha(t)$

Different answers can be found by finding other derivations

Back-chaining procedure

Solve[q_1, q_2, ..., q_n] = /* to establish conjunction of q_i */

If n=0 then return **YES**; /* empty clause detected */

For each $d \in$ KB do

If d = [q_1, $\neg p_1$, $\neg p_2$, ..., $\neg p_m$] /* match first q */

and /* replace q by -ve lits */

Solve[p_1, p_2, ..., p_m, q_2, ..., q_n] /* recursively */

then return **YES**

end for; /* can't find a clause to eliminate q */

Return **NO**

Depth-first, left-right, back-chaining

- depth-first because attempt p_i before trying q_i
- left-right because try q_i in order, 1,2, 3, ...
- back-chaining because search from goal q to facts in KB p

This is the execution strategy of Prolog

First-order case requires unification *etc.*

Problems with back-chaining

Can go into infinite loop

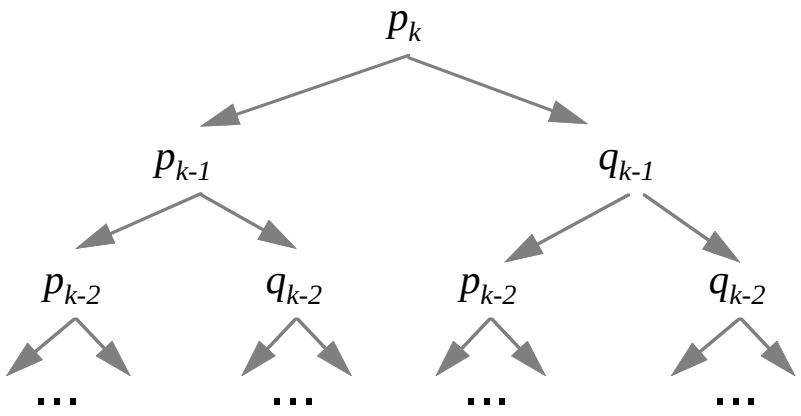
tautologous clause: $[p , \neg p]$ (corresponds to Prolog program with `p :- p`).

Previous back-chaining algorithm is inefficient

Example: Consider $2n$ atoms, $p_0, ..., p_{n-1}, q_0, ..., q_{n-1}$ and $4n-4$ clauses

$$(p_{i-1} \Rightarrow p_i),\ (q_{i-1} \Rightarrow p_i),\ (p_{i-1} \Rightarrow q_i),\ (q_{i-1} \Rightarrow q_i).$$

With goal p_k the execution tree is like this



Solve[p_k] eventually fails after 2^k steps!

Is this problem inherent in Horn clauses?

Forward-chaining

Simple procedure to determine if Horn KB |= q.

main idea: mark atoms as solved

1. If q is marked as solved, then return **YES**
2. Is there a $\{p_1, \neg p_2, ..., \neg p_n\} \in$ KB such that $p_2, ..., p_n$ are marked as solved, but the positive lit p_1 is not marked as solved?
   - no: return **NO**
   - yes: mark p_1 as solved, and go to 1.

FirstGrade example:

Marks: FirstGrade, Child, Female, Girl then done!

Note: FirstGrade gets marked since all the negative atoms in the clause (none) are marked

Observe:

- only letters in KB can be marked, so at most a linear number of iterations
- not goal-directed, so not always desirable
- a similar procedure with better data structures will run in *linear* time overall

First-order undecidability

Even with just Horn clauses, in the first-order case we still have the possibility of generating an infinite branch of resolvents.

KB:

$\text{LessThan}(\text{succ}(x),y) \Rightarrow \text{LessThan}(x,y)$

Query:

$\text{LessThan}(\text{zero},\text{zero})$

As with full Resolution, there is no way to detect when this will happen

$[\neg\text{LessThan}(0,0)]$
↓ $x/0, y/0$
$[\neg\text{LessThan}(1,0)]$
↓ $x/1, y/0$
$[\neg\text{LessThan}(2,0)]$
↓ $x/2, y/0$
...

There is no procedure that will test for the satisfiability of first-order Horn clauses

the question is undecidable

As with non-Horn clauses, the best that we can do is to give control of the deduction to the *user*

to some extent this is what is done in Prolog, but we will see more in “Procedural Control”

Declarative / procedural

Theorem proving (like resolution) is a general domain-independent method of reasoning

Does not require the user to know how knowledge will be used

> will try all logically permissible uses

Sometimes we have ideas about how to use knowledge, how to search for derivations

> do not want to use arbitrary or stupid order

Want to communicate to theorem-proving procedure some *guidance* based on properties of the domain

- perhaps specific method to use
- perhaps merely method to avoid

Example: directional connectives

In general: control of reasoning

DB + rules

Can often separate (Horn) clauses into two components:

Example:

MotherOf(jane,billy)
FatherOf(john,billy)
FatherOf(sam, john)
...
ParentOf(x,y) $\Leftarrow$ MotherOf(x,y)
ParentOf(x,y) $\Leftarrow$ FatherOf(x,y)
ChildOf(x,y) $\Leftarrow$ ParentOf(y,x)
AncestorOf(x,y) $\Leftarrow$ ...
...

a database of facts

- basic facts of the domain
- usually ground atomic wffs

collection of rules

- extends the predicate vocabulary
- usually universally quantified conditionals

Both retrieved by unification matching

Control issue: how to use the rules

Rule formulation

Consider AncestorOf in terms of ParentOf

Three logically equivalent versions:

1. $\text{AncestorOf}(x,y) \Leftarrow \text{ParentOf}(x,y)$
   $\text{AncestorOf}(x,y) \Leftarrow \text{ParentOf}(x,z) \wedge \text{AncestorOf}(z,y)$
2. $\text{AncestorOf}(x,y) \Leftarrow \text{ParentOf}(x,y)$
   $\text{AncestorOf}(x,y) \Leftarrow \text{ParentOf}(z,y) \wedge \text{AncestorOf}(x,z)$
3. $\text{AncestorOf}(x,y) \Leftarrow \text{ParentOf}(x,y)$
   $\text{AncestorOf}(x,y) \Leftarrow \text{AncestorOf}(x,z) \wedge \text{AncestorOf}(z,y)$

Back-chaining goal of AncestorOf(sam,sue) will ultimately reduce to set of ParentOf(–,–) goals

1. get ParentOf(sam,*z*): find child of Sam searching *downwards*
2. get ParentOf(*z*,sue): find parent of Sue searching *upwards*
3. get ParentOf(–,–): find parent relations searching *in both directions*

Search strategies are not equivalent

if more than 2 children per parent, (2) is best

Algorithm design

Example: Fibonacci numbers

1, 1, 2, 3, 5, 8, 13, 21, ...

Version 1:

$\text{Fibo}(0, 1)$

$\text{Fibo}(1, 1)$

$\text{Fibo}(\text{s}(\text{s}(n)), x) \Leftarrow \text{Fibo}(n, y) \wedge \text{Fibo}(\text{s}(n), z) \wedge \text{Plus}(y, z, x)$

Requires *exponential* number of Plus subgoals

Version 2:

$\text{Fibo}(n, x) \Leftarrow \text{F}(n, 1, 0, x)$

$\text{F}(0, c, p, c)$

$\text{F}(\text{s}(n), c, p, x) \Leftarrow \text{Plus}(p, c, s) \wedge \text{F}(n, s, c, x)$

Requires only *linear* number of Plus subgoals