COMP4418: Knowledge Representation and Reasoning

Horn Logic

Maurice Pagnucco
School of Computer Science and Engineering

COMP4418, Week 3

Horn clauses

Clauses are used two ways:

- as disjunctions: (rain $\vee$ sleet)
- as implications: ($\neg$child $\vee$ $\neg$male $\vee$ boy)

Here focus on 2nd use
Horn clause = at most one +ve literal in clause

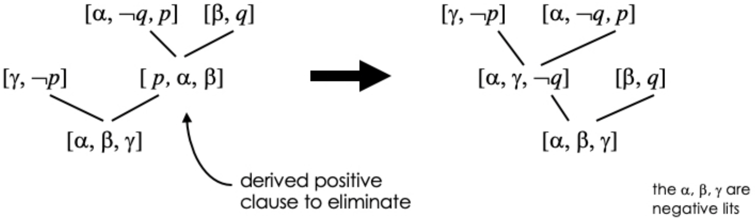
- positive / definite clause = exactly one +ve literal
  $[\neg p_1, \neg p_2, \ldots, \neg p_n, q]$
- negative clause = no +ve literals
  $[\neg p_1, \neg p_2, \ldots, \neg p_n]$

Note:
$[\neg p_1, \neg p_2, \ldots, \neg p_n, q]$ is a representation for
$(\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_n \vee q)$ or
$[(p_1 \wedge p_2 \wedge \ldots \wedge p_n) \rightarrow q]$
So can read as
If p_1 and p_2 and . . . and p_n then q
and write sometimes as
$p_1 \wedge p_2 \wedge \ldots \wedge p_n \rightarrow q$

Resolution with Horn clauses

Only two possibilities:

It is possible to rearrange derivations (of negative clauses) so that all new derived clauses are negative clauses



Can also change derivations such that each derived clause is a resolvent of the previous derived one (-ve) and some +ve clause in the original set of clauses

- Since each derived clause is negative, one parent must be positive (and so from original set) and one negative
- Continue working backwards until both parents of derived clause are from the original set of clauses
- Eliminate all other clauses not on direct path

SLD Resolution

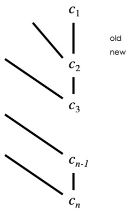
Recurring pattern in derivations

See previously:

- Example 1
- Example 3
- Arithmetic example

But not:

- Example 2
- 3 block example



An *SLD-derivation* of a clause c from a set of clauses S is a sequence of clause $c_1, c_2, \ldots c_n$ such that $c_n = c$, and

1. $c_1 \in S$
2. c_{i+1} is a resolvent of c_i and a clause in S

Write: $S \vdash_{SLD} c$

Note: SLD derivation is just a special form of derivation and where we leave out the elements of S (except c_1)

SLD means S(elected) literals, L(inear) form, D(efinite) clauses

B&L (2005)

Completeness of SLD

In general, cannot restrict Resolution steps to always use a clause that is in the original set

Proof:

$S = \{[p, q], [p, \neg q], [\neg p, q], [\neg p, \neg q]\}$

then $S \vdash []$.

Need to resolve some $[l]$ and $[\neg l]$ to get [].

But S does not contain any unit clauses.

So will need to derive both $[l]$ and $[\neg l]$ and then resolve them together.

But can do so for Horn clauses ...

Theorem: for Horn clauses, $H \vdash []$ iff $H \vdash_{SLD} []$

So: H is unsatisfiable iff $H \vdash_{SLD} []$

This will considerably simplify the search for derivations

Note: in Horn version of SLD-Resolution, each clause $c_1, c_2, \ldots c_n$ will be negative

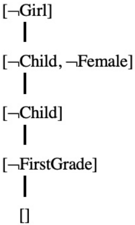
So clauses H must always contain at least one negative clause, c_1.

Example 1 (again)

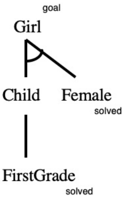
KB:

FirstGrade
FirstGrade $\rightarrow$ Child
Child $\wedge$ Male $\rightarrow$ Boy
Kindergarten $\rightarrow$ Child
Child $\wedge$ Female $\rightarrow$ Girl
Female

Show KB $\cup$ $\{\neg$Girl$\}$ unsatisfiable



or



A goal tree whose nodes are atoms, whose root is the atom to prove, and whose leaves are in the KB

UNSW SYDNEY

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))

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

$H \vdash \exists x \alpha(x)$ iff for some term t, $H \vdash \alpha(t)$

Different answers can be found by finding other derivations

Back-chaining procedure

Satisfiability of a set of Horn clauses with exactly one negative clause

Solve $[q_1, q_2, \ldots, q_n] =$ /* to establish conjunction of q_i */
  If $n = 0$ then return **YES**; /* empty clause detected */
  For each $d \in KB$ do
    If $d = [q_1, \neg p_1, \neg p_2, \ldots, \neg p_m]$ /* match first q */
      and /* replace q by -ve lits */
     Solve $[p_1, p_2, \ldots, p_m, q_2, \ldots, q_n]$ /* 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.

B&L (2005)

Problems with back-chaining

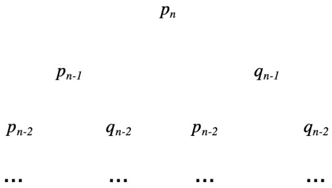
Can go into infinite loop

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

Previous back-chaining algorithm is inefficient

Example:

- consider $2n$ atoms: $p_1, \ldots, p_n, q_1, \ldots, q_n$,
- and $4n - 4$ clauses:
  - $(p_i \Rightarrow p_{i+1}), (q_i \Rightarrow q_{i+1})$,
  - $(p_i \Rightarrow q_{i+1}), (q_i \Rightarrow q_{i+1})$.
- with goal p_n has execution tree like this:



- search eventually fails after 2^n steps!

Is this inherent in Horn clauses?

Forward-chaining

Simple procedure to determine if Horn KB $\vdash q$.

main idea: mark atoms as solved

1. If q is marked as solved, then return **YES**
2. Is there a $\{p_1, \neg p_2, \ldots, \neg p_n\} \in$ KB such that $p_2, \ldots, p_n$ are marked as solved, but the positive literal 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!

Observe:

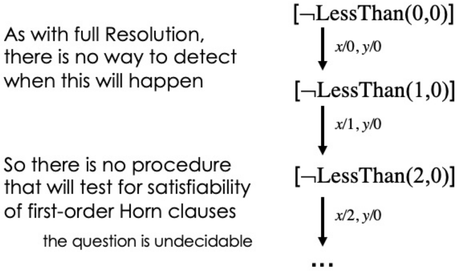
- only letters in KB can be marked, so at most a linear number of iterations
- not goal-directed, so not always desirable

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: LessThan(succ(x),y) $\rightarrow$ LessThan(x,y)
Q: LessThan(zero,zero)



As with full clauses, the best that can be expected 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”