EECS 3401 — AI and Logic Prog. — Lecture 8

Adapted from slides of Brachman & Levesque (2005)

Vitaliy Batusov
vbatusov@cse.yorku.ca

York University

October 7, 2020

Reasoning with Horn Clauses

- Today: **Reasoning with Horn Clauses**
- Also today: **Procedural Control of Reasoning**
- Required reading: Russell & Norvig, Chapter 9; Clocksin & Mellish Chapters 4 and 10

Reasoning with Horn Clauses

Recall:

- A clause is a disjunction of literals: $(p, q, r, \neg s)$
- A Horn clause: same but at most one positive literal is allowed: $(p, \neg q, \neg r, \neg s)$
- Think of Horn clauses as **implications**

$\neg q_1 \vee \neg q_2 \vee \ldots \vee \neg q_n \vee p$	Horn clause
$(q_1 \wedge q_2 \wedge \ldots \wedge q_n) \rightarrow p$	same, as an implication
`p :- q_1, q_2, ... , q_n.`	same, as a Prolog rule

Horn Clauses

Some more terminology:

- **Positive** (definite) clause: has exactly one positive literal

$$(\neg q_1, \neg q_2, \ldots, \neg q_n, p)$$

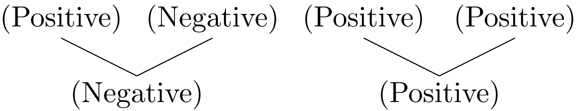
- **Negative** clause: no positive literals

$$(\neg q_1, \neg q_2, \ldots, \neg q_n)$$

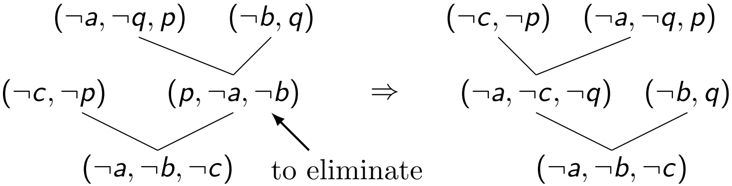
The empty clause $\{\}$ is *negative*

Resolution with Horn Clauses

- When resolving Horn clauses, there are only two possibilities:



- It is possible to rearrange a resolution proof of a **negative clause** so that all new derived clauses are negative:



Further Restricting Resolution

It is also possible to perform derivations in such a way that each derived clause is a resolvent of a previously-derived negative clause and some positive clause from the knowledge base

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

SLD Resolution

- S Selected literals
- L Linear form
- D Definite clauses

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

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

Notation: $KB \vdash_{\text{SLD}} c$

An SLD derivation is just a special form of a resolution derivation where we also leave out the KB clauses (except c_1)

SLD Resolution

- In general, SLD Resolution is less powerful than regular resolution
- Consider KB:

$$(p, q)$$
$$(p, \neg q)$$
$$(\neg p, q)$$
$$(\neg p, \neg q)$$

  States "p and q are identical and mutex at the same time"
- Fact: $KB \vdash ()$, but $KB \nvdash_{\text{SLD}} ()$
  Because to get $()$ we need to resolve (p) with $(\neg p)$ or (q) with $(\neg q)$, but the KB itself doesn't contain unit clauses. Thus, a unit clause with a positive literal needs to be derived, which is not allowed by SLD.

Completeness of SLD

For **Horn clauses**, SLD Resolution is sufficient.

Theorem

SLD Resolution is refutation-complete for Horn clauses.
Let KB be a set of Horn clauses.

$$KB \vdash () \quad \textit{iff} \quad KB \vdash_{SLD} ()$$

So, KB is unsatisfiable iff $KB \vdash_{\text{SLD}} ()$. This considerably simplifies the search for derivations.

- Note: in an SLD derivation with a Horn KB, each clause in $c_1, c_2, \ldots, c_n$ will be negative.
- Thus, KB must contain at least one negative clause c_1, and this will be the only negative clause from KB used.
- Typically, KB is a collection of positive Horn clauses, and the negation of the query is the negative clause

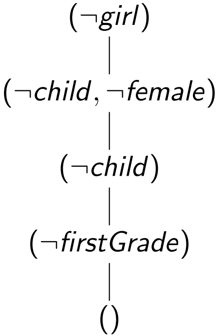
Example

Show that $KB \cup \{\neg girl\}$ is unsatisfiable:

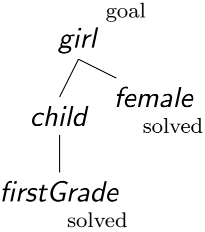
KB

$(firstGrade)$
$(\neg firstGrade, child)$
$(\neg child, \neg male, boy)$
$(\neg kindergarten, child)$
$(\neg child, \neg female, girl)$
$(female)$

SLD derivation:



Goal tree:



Prolog

Horn clauses form the basis of Prolog. Consider:

```
append([], Z, Z).
append([E1|R1], Y, [E1|Rest]) :- append(R1, Y, Rest).
```

Observe/recall:

- `[]` is a constant
- `[a, b, c]` is really the term $cons(a, cons(b, cons(c,$ `[]` $)))$[1]
- The second rule is actually the clause
  ($\neg$ `append(R1, Y, Rest)`, `append([E1|R1], Y, [E1|Rest])`),
  or, expressing lists as terms,
  $(\neg append(R1, Y, Rest), append(cons(E1, R1), Y, cons(E1, Rest)))$

[1] Here, *cons* stands in for the functor `'[|]'` for clarity

Prolog

$$(append(\texttt{[]}, Z, Z))$$
$$(\neg append(R1, Y, Rest), append(cons(E1, R1), Y, cons(E1, Rest)))$$

What is the result of `append([a,b],[c], X)` ?

$$append(cons(a, cons(b, \texttt{[]})), cons(c, \texttt{[]}), X)$$

$E1 = a, R1 = cons(b, \texttt{[]})$ | $Y = cons(c, \texttt{[]}), X = cons(E1, Rest)$

$$append(cons(b, \texttt{[]}), cons(c, \texttt{[]}), Rest)$$

$E1' = b, R1' = \texttt{[]}$ | $Y' = cons(c, \texttt{[]}), Rest = cons(E1', Rest')$

$$append(\texttt{[]}, cons(c, \texttt{[]}), Rest')$$

$$Rest' = cons(c, \texttt{[]})$$

Goal succeeds with $X = cons(a, cons(b, cons(c, \texttt{[]})))$, i.e., `[a, b, c]`.

Back-chaining Procedure

Prolog uses the following back-chaining procedure to decide whether a sequence of goals is true.

solve($q_1, q_2, \ldots, q_n$):
if $n = 0$ **then**
  return Yes
for all $d \in KB$ **do**
  if d is $(q_1, \neg p_1, \neg p_2, \ldots, \neg p_m)$ **then**
    if solve($p_1, p_2, \ldots, p_m, q_2, \ldots, q_n$) = Yes **then**
      return Yes
return No

This is depth-first, left-right back-chaining.

- Depth-first because attempt to prove p_i before trying q_i
- Left-right because proves q_i in order, $i = 1, 2, 3, \ldots$
- Back-chaining because search from goal q to KB facts p

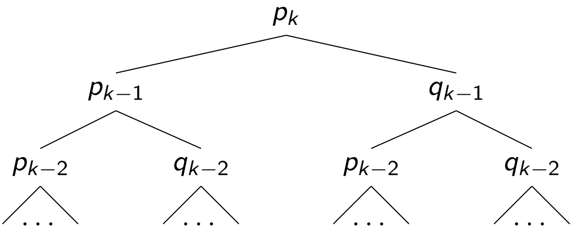
Problems with Back-chaining

- Can enter an infinite loop
  Clause $(p, \neg p)$ says nothing of use (it's a tautology), but corresponds to a prolog program `p :- p.`

- Inefficient
  Consider $2n$ atoms $p_0, \ldots, p_{n-1}, q_0, \ldots, q_{n-1}$ and $4(n-1)$ clauses

$$(\neg p_{i-1}, p_i), \quad (\neg q_{i-1}, p_i), \quad (\neg p_{i-1}, q_i), \quad (\neg q_{i-1}, q_i)$$

  The proof of goal p_k eventually fails after 2^k steps.



Forward-chaining

Forward-chaining is a simple procedure to determine if Horn KB $\models q$

- Main idea: mark atoms as solved

  loop
    if q is marked as solved **then**
      return Yes
    for all $(p_1, \neg p_2, \ldots, \neg p_m) \in KB$ **do**
      if $p_2, \ldots, p_m$ are marked as solved, but p_1 is not **then**
        mark p_1 as solved
      else
        return No

- Not goal-oriented, so not always desirable
- Can, in principle, run in linear time

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: $(\neg lessThan(succ(X), Y), lessThan(X, Y))$

Query: $lessThan(0, 0)$

$$(\neg lessThan(0,0))$$
$$\Big|\ X = 0,\ Y = 0$$
$$(\neg lessThan(1,0))$$
$$\Big|\ X' = 1,\ Y' = 0$$
$$(\neg lessThan(2,0))$$
$$\Big|\ X'' = 2,\ Y'' = 0$$
$$\dots$$

- As with regular resolution, there isn't and cannot be a general way to detect when this will happen.
- Satisfiability of FOL Horn theories is **undecidable**
- Best we can do is to give control of the deduction to the user

Procedural Control of Reasoning

- Theorem proving (e.g. resolution) is a general domain-independent method of reasoning
- Not tailored to a specific domain or application — treats all knowledge the same
- With some applications, though, there are glaringly obvious shortcuts to be exploited
- Want to be able to guide the theorem-proving procedure

Facts & Rules

(Recall from Prolog) When working with Horn clauses, we can separate them into **facts** and **rules**

```
motherOf(jane, billy).                /* specific facts */
fatherOf(john, billy).
fatherOf(sam, john).

parentOf(X, Y) :- motherOf(X, Y). /* universal rules */
parentOf(X, Y) :- fatherOf(X, Y).
childOf(X, Y) :- parentOf(Y, X).
ancestorOf(X, Y) :- /* and so on */
```

Both are retrieved by unification matching.
Same rules can be formulated in different ways to guide the proving procedure.

Rule Formulation: Example

Consider `ancestorOf` defined in terms of `parentOf`.

Base case:

```
ancestorOf(X, Y) :- parentOf(X, Y).
```

Recursive case variants:

```
ancestorOf(X,Y) :- parentOf(X,Z), ancestorOf(Z,Y).    % V1
ancestorOf(X,Y) :- parentOf(Z,Y), ancestorOf(X,Z).    % V2
ancestorOf(X,Y) :- ancestorOf(X,Z), ancestorOf(Z,Y). % V3
```

The back-chaining goal of `ancestorOf(sam, sue)` will ultimately reduce to a set of `parentOf(_,_)` goals

- V1 `parentOf(sam,Z)` — find child of `sam` searching *downwards*
- V2 `parentOf(Z,sue)` — find parent of `sue` searching *upwards*
- V3 `parentOf(_,_)` — find a parent relation searching in both directions

Algorithm Design

Consider Fibonacci numbers: 1, 1, 2, 3, 5, 8, 13, 21, ...

Version 1:

```
fibo(0, 1).
fibo(1, 1).
fibo(M,X) :- N1 is M-1, fibo(N1,Y),
             N2 is M-2, fibo(N2,Z), X is Y+Z.
```

This requires an *exponential* number of `+` subgoals.

Version 2:

```
fibo(N, X) :- f(N, 1, 0, X).
f(0, C, P, C).
f(M, C, P, X) :- N is M-1, S is P+C, f(N, S, C, X).
```

This requires a *linear* number of `+` subgoals.

Ordering Goals

Consider:

```
americanCousinOf(X,Y) :- american(X), cousinOf(X,Y).
```

Logically, rearranging the subgoals on RHS makes no difference.
Pragmatically, though...

- If asking `americanCousinOf(fred, sally)`, we are fine either way.
- If asking `americanCousinOf(X, sally)`, the difference is between
  - find cousin of `sally`, check if American
  - *an American*, check if `sally`'s cousin
- So: order goals sensibly. **Generate** cousins, **test** for American.

Commit

- Usually need to allow for backtracking in goals, as in `americanCousinOf(X,Y) :- cousinOf(X,Y), american(X).`
- Sometimes, want to prevent backtracking — i.e., *commit*
  In certain application, may need a clause like
  `goal :- test, subgoal.`
  where the first subgoal `test` may serve as a guard for the subgoal `subgoal`, i.e., to prove goal `goal`, check if the rule is even applicable by proving `test`, and if so, then commit to `subgoal` as the *only way* of achieving `goal`.
- The **cut** meta-operator `!` cuts off all backtracking for `goal` at the point where it appears.
  `goal :- test, !, subgoal.`

If–Then–Else

At times, a useful pattern:

```
goal :- condition, !, case1.
goal :- case2.
```

To achieve `goal`: if `condition`, commit to `case1`, else use `case2`.

Example:
Instead of laboriously writing two mutually-exclusive conditions

```
expt(A,N,X) :- even(N), ...
expt(A,N,X) :- odd(N), ...
```

use a single condition with an "else":

```
expt(A,N,X) :- N=0, !, X=1.
expt(A,N,X) :- even(N), !, ... /* for even numbers */
expt(A,N,X) :- ... /* for odd numbers */
```

Controlling Backtracking using !

Consider solving the goal like

```
ancestorOf(jane,billy), male(jane)
```

```
parentOf(jane,billy), male(jane)
```

```
male(jane)
```

Fails

```
parentOf(Z,billy), ancestorOf(jane,Z), male(jane)
```

Eventually fails

The goal here should really be

```
ancestorOf(jane, billy), !, male(jane)
```

End of Lecture

- Next time: Hopefully, only one more lecture on Prolog semantics, features, and tricks, and then **Search**