Symbolic Artificial Intelligence (COMP3008)
Lecture 8: Horn Clauses
17 November 2025

Daniel Karapetyan
daniel.karapetyan@nottingham.ac.uk

Horn clauses

Recap

- The resolution algorithm can tell if a CNF sentence is satisfiable or unsatisfiable
- It has major performance/termination issues
  - In the propositional case, the resolution algorithm is guaranteed to converge but may take time exponential in the number of propositional variables
  - In the FOL case, the resolution algorithm is guaranteed to converge only for unsatisfiable sentences
- These issues are inherent:
  - Propositional entailment is NP-complete
  - FOL entailment is undecidable
- We will consider a subset of FOL for which resolution is more manageable

Horn clauses

- Our restricted language will be based on *Horn clauses*
  - After logician Alfred Horn
- Horn clause is a special case of a CNF clause
- Plan of the lecture:
  1. Horn clauses in propositional logic
  2. Resolution for Horn clauses (propositional case)
  3. Horn clauses in FOL
  4. Prolog

Horn clauses – propositional case

- A *Horn clause* is a clause with at most one positive (not negated) literal, e.g.
  $[\overline{NightObject}, \overline{Bright}, \overline{Moves}, InternationalSpaceStation]$
- This can be read as "If *NightObject* and *Bright* and *Moves* then *InternationalSpaceStation*"

Positive and negative Horn clauses

Positive Horn clause	Negative Horn clause
One positive literal	No positive literals
Positive clause $[\overline{P_1}, \overline{P_2}, \ldots, \overline{P_n}, Q]$ can be read as "If P_1 and P_2 and … and P_n then also Q"	Negative clause $[\overline{P_1}, \overline{P_2}, \ldots, \overline{P_n}]$ is also called a *goal*; to prove that all of P_1, P_2, …, P_n are entailed by the KB, we add $\neg(P_1 \wedge P_2 \wedge \ldots \wedge P_n)$, equivalent to $\neg P_1 \vee \neg P_2 \vee \ldots \vee \neg P_n$ i.e. $[\overline{P_1}, \overline{P_2}, \ldots, \overline{P_n}]$ clause

Note: empty clause [ ] is a negative clause

Resolution with Horn clauses

Resolution with Horn clauses (propositional case)

Recall that

- Positive Horn clause – one positive literal
- Negative Horn clause – no positive literals

The resolution rule cannot work for two negative Horn clauses as all the literals are negative, e.g. one cannot use resolution on the following two clauses:

$$[\overline{P}, \overline{Q}] \qquad \text{and} \qquad [\overline{P}, \overline{R}, \overline{T}]$$

We are left with two options:

- negative + positive clauses can only give a negative clause,
- positive + positive clauses can only give a positive clause

Resolution with Horn clauses

Theorem: if a negative clause c can be derived from a set S of Horn clauses then there exists a derivation of the following form:

1. Select a negative $c_1 \in S$;
2. For $i = 2, 3, 4, \ldots$, let c_i be a resolvent of c_{i-1} and some positive clause from S;
3. For some n, it is true that $c_n = c$.

The proof is non-trivial and we will skip it (you are not expected to know it)

SLD derivation

The derivation described in the previous slide is called *SLD* (for Selected literals, Linear pattern, over Definite clauses)

It is just the usual derivation, with some constraints on what clauses are selected in each iteration

Recall that an empty clause is a negative Horn clause

Observe that the SLD derivation is refutation-complete:

- If SLD derivation gives an empty clause, S is unsatisfiable
- If S is unsatisfiable, SLD is guaranteed to arrive to an empty clause earlier or later

SLD-based algorithms

We will discuss specific algorithms for SLD derivation:

- Backward Chaining
- Forward Chaining

Both algorithms are designed to prove a set of given facts:

- Given a set S of positive Horn clauses
- Given a set $\{Q_1, Q_2, \ldots, Q_n\}$ of propositional variables
- Prove or disprove that all of $Q_1, Q_2, \ldots, Q_n$ are entailed by S
- Note that proving such an entailment is equivalent to adding a negative Horn clause $[\overline{Q_1}, \overline{Q_2}, \ldots, \overline{Q_n}]$ to S and showing derivation of $[\,]$

Backward Chaining

A resolution algorithm that starts from the goal $Q_1 \wedge Q_2 \wedge \ldots \wedge Q_n$ and moves back in the inference chain is called *Backward Chaining*

To prove entailment of Q_1, we need to find a clause that includes non-negated Q_1

Say, we found a clause $c = [Q_1, \overline{P_1}, \overline{P_2}, \ldots, \overline{P_k}]$; if all of $P_1, P_2, \ldots, P_k$ are entailed then this proves that Q_1 is also entailed

Proving entailment of one propositional variable is reduced to proving entailment of other propositional variables; this is a recursive algorithm

Backward Chaining pseudocode

Algorithm 1: *BackwardChaining*$(S, \{Q_1, Q_2, \ldots, Q_n\})$

input : A set S of positive Horn clauses
input : A set $\{Q_1, Q_2, \ldots, Q_n\}$ of propositional variables
output: TRUE if all of $Q_1, Q_2, \ldots, Q_n$ are entailed by S and FALSE otherwise

```
1 if n = 0 (i.e. there are no propositional variable to prove) then
2     return TRUE;
3 for c ∈ S such that c = [Q_1, ¬P_1, ¬P_2, ..., ¬P_k] do
4     if BackwardChaining(S, {Q_2, Q_3, ..., Q_n, P_1, P_2, ..., P_k}) then
5         return TRUE;
6 return FALSE;
```

Backward Chaining Properties

- The algorithm is sound (any conclusion it makes is correct)
- It is not guaranteed to terminate even in the propositional case
- It may be very inefficient in relatively simple cases

Forward Chaining

Forward Chaining is a completely different procedure to solve the same problem (to determine if a set of propositional variables is entailed by a KB consisting of positive Horn clauses)

Forward Chaining is much more efficient than Backward Chaining

The idea is to mark propositional variables as 'solved' when we prove that they are entailed by the KB, and then propagate them

Every time we find a propositional variable that is entailed by the KB, we mark it as *solved*

If there is a clause $[A, \overline{B_1}, \overline{B_2}, \ldots, \overline{B_k}]$, where A is unsolved but all the other constants are solved (i.e. known to be entailed), A is then entailed and can be marked solved

Forward Chaining pseudocode

Algorithm 2: Forward Chaining

input : A set S of positive Horn clauses
input : A set $\{Q_1, Q_2, \ldots, Q_n\}$ of propositional variables
output: TRUE if all of Q_1, Q_2, …, Q_n are entailed by S and FALSE otherwise

```
1 Mark every propositional variable P used in S as unsolved;
2 Loop
3     Select c ∈ S such that c = [A, ¬B_1, ¬B_2, ..., ¬B_k], where A is not
        solved and all of B_1, B_2, ..., B_k are solved;
4     if c exists then
5         Mark A as solved;
6         if all of Q_1, Q_2, ..., Q_n are solved then
7             return TRUE;
8     else
9         return FALSE;
```

Forward Chaining Properties

- Forward Chaining is sound and refutation-complete for propositional case
- Forward Chaining is a polynomial time algorithm for propositional case
- In fact, it can be implemented to have $O(|S|)$ number of iterations, i.e. the number of iterations is linear in the size of the KB (in the CNF form)!

Propositional case – summary

- Resolution is much simpler with Horn clauses
- We can prove that for any unsatisfiable set of Horn clauses, there exists an SLD derivation of an empty clause [ ]
- We studied two SLD-based resolution algorithms
  - Backward Chaining is a straightforward recursive procedure that is not very efficient and may not even terminate
  - Forward Chaining is a much more efficient procedure that remembers all the propositional variables that it proved to be entailed

Horn clauses and FOL

- We can extend Horn clauses with quantifiers, etc., e.g.
  $[P(x), \overline{Q_1(x)}, \overline{Q_2(y)}]$
  means
  $\forall x.\forall y.\big(P(x) \vee \neg Q_1(x) \vee \neg Q_2(y)\big)$
- Satisfiability of FOL Horn clauses is undecidable; there is no entailment procedure that is guaranteed to terminate
- FOL version of Horn clauses is the basis of the *Prolog language*

Prolog

Prolog

- Prolog is a logical programming language
  - Declarative – you describe what you want but not how to achieve it
  - Based on FOL Horn clauses
- Prolog was first implemented in 1972
- There have been other logical programming languages but Prolog is the most widely used
- There are several modern implementations of the language, e.g. SWI-Prolog, a free open-source implementation

Data types in Prolog

Prolog supports only one data type: *term*; terms can be

- Atoms – abstract named entities, e.g.: **`a`**, **`daniel`**, **`planet`**
  - Begin with a lower-case letter
- Numbers – Prolog natively supports floats and integers
  - In many implementations including SWI-Prolog, integers are unbounded by default
  - SWI-Prolog also supports rational numbers (n/m) for precise fractional arithmetic
  - For details on numeric types in SWI-Prolog, see `https://www.swi-prolog.org/pldoc/man?section=arith`
- Variables – which can take arbitrary values, e.g.: **`X`**, **`Good`**
  - Begin with an upper-case letter or underscore
- Compound term (see next slide)

Compound terms

Compound terms represent more complex objects

An example of a compound term:
`is_sunny(july, spain)`
Here `is_sunny` is a *functor* and `july` and `spain` are *arguments*

Each argument is a term itself

Special cases of compound terms:

- Atom is a compound term with zero arguments
- A list is a compound term, e.g.: `[]`, `[uk, spain]`, `[1, sunny, [uk, spain]]`
- String (a list of characters), e.g.: `""`, `"Hello World!"`

Program structure

A Prolog program is a set of *clauses* which can be of two types:

- *Rules* are Horn clauses defining relations between terms, e.g. “If c_1 is a cat and c_1 is a parent of c_2 then c_2 is also a cat”
- *Facts* state what is known to be true, e.g. “kitty is a cat”

Execution

To execute a program, the user needs to specify a query, i.e. a formula to be verified, e.g. “Is tom a cat?”

Prolog then tries to determine if this formula is entailed, subject to the clauses specified in the program

- If the query contains variables, Prolog will also report the values of the variables that make it entailed
- If there are several such values (combinations of values), it will report all of them one by one

Rules

Each rule is written in the following form:

`Head :- Body.`

which reads as “Head if Body” and is equivalent to the following Horn clause:

$Head \vee \neg Body$ or, equivalently, $Body \rightarrow Head$

The body (the right side of a rule) can contain a conjunction and/or disjunction of several terms:

- **`modern, expensive, highquality`**
  stands for $modern \wedge expensive \wedge highquality$
- **`expensive; highquality`**
  stands for $expensive \vee highquality$
- **`modern, expensive; highquality`**
  stands for $(modern \wedge expensive) \vee highquality$

Facts

A clause with empty body is called a *fact*, e.g.:

```
expensive.
```

Facts state true facts, i.e. *expensive* $\equiv$ TRUE

An example of a program with compound terms:

```
animal(X) :- cat(X).
cat(tom).
```

Another example; function `length` links lists to their length:

```
length(List, L).
```

- For a known list `List`, this fact will 'assign' the length of the list to `L`
- For a known number `L`, the same fact 'creates' a list `List` of that length

Execution example

Consider the following program in file ‘test.pl’:

```
cat(kitty).
cat(X) :- parent(X, Y), cat(Y).
parent(tom, kitty).
```

To load the program, we write the file name (without extension) in square brackets:

```
?- [test].
```

The response is `true.` – to indicate the success of the operation

We can now query

```
?- cat(X).
```

The response is

```
X = kitty ;
X = tom ;
false.
```

Working of Prolog

When the user enters the query, Prolog attempts to prove it (i.e. prove that it is entailed)

- If the query contains variables, Prolog also needs to find the assignments of variables that make it satisfiable

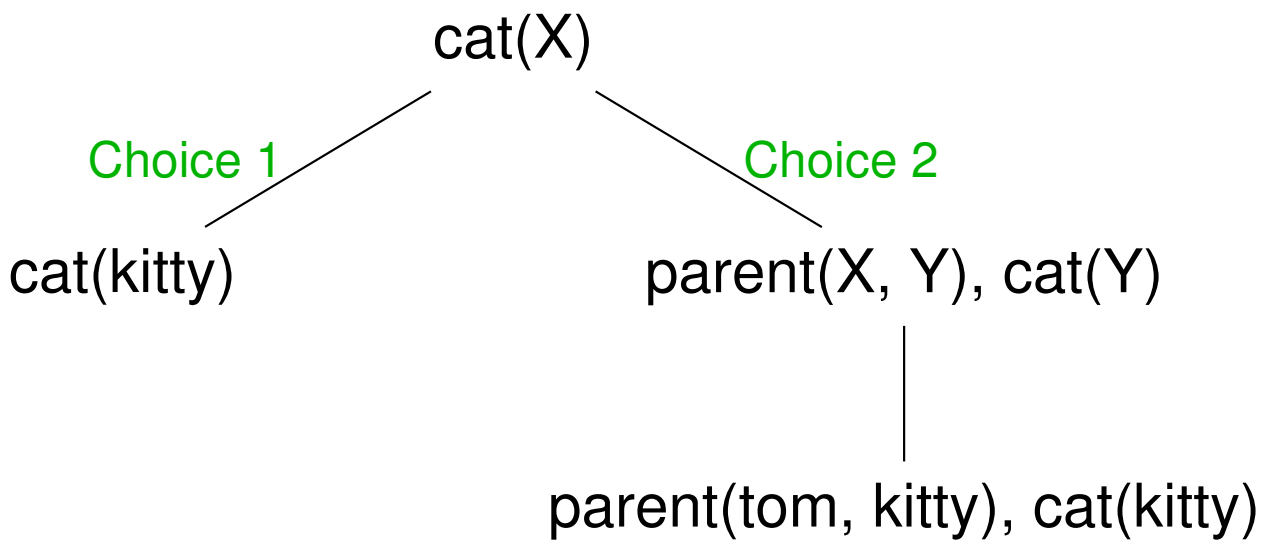
It uses the SLD resolution for the negated query, attempting to infer [ ]

Execution resembles function calls, however multiple clause heads can match the query

Backtracking

```
cat(kitty).
cat(X) :- parent(X, Y), cat(Y).
parent(tom, kitty).

?- cat(X).
```



Choice points

Backtracking works as follows

- Every time there is more than one clause with the head matching the query, Prolog creates a *choice point*
- Prolog uses depth-first search: it picks the first choice and proceeds with it
- If a choice does not lead to success, or if next answer is requested, Prolog backtracks to the last choice point and tries a different choice

Scrolling through the answers in the Prolog command line:

- To request the next answer, press ';'
- To terminate the enumeration, press '.'