Revision Notes: Lectures 6 → 8

16 April 2019 4:37 PM

Horne Logic

- The theoretical properties of 1st order logic are not suited to computation

- As 1st order logic is semi-decidable, we can focus on the decidable subset of 1st order expressions

- We do this by imposing limitations on the order of quantifiers of formulae in Prenex Normal form.

Horne Clauses

- In Horne Logic we restrict ourselves to Horne Clauses which take the form:

$$L_1 \wedge \dots \wedge L_n \rightarrow M, \quad n \geq 0$$

→ elimination

$$\neg L_1 \vee \dots \vee \neg L_n \vee M$$

Where L_i, $i = 1, \dots, n$, & M are Positive (non-negated) Literals

There are 3 types of Horne Clause:

Definite Clauses

If we look at a formula in CNF, we can see it contains a Maximum of 1 Positive literal (M in the case above)

→ can have 0

Facts

Since $n \geq 0$ we can also have the true literal as a Clause eg.

M ← if $n = 0$

Goal Clauses

- Since M can be False we get

$$\neg L_1 \vee \dots \vee \neg L_n$$

In Prolog M is called the head. The rest is the tail.

Variables in Horne Clauses

- Literals $(L_1, \ldots, L_n)$ can be first order Predicates, containing terms (including variables $\in V$)

- However, all variables are considered to be Universally quantified

Restrictions of Horne Logic

- Clearly we cannot express anything that has more than 1 Positive Literal

Disjunction on the LHS

$$A \vee B \rightarrow C \cong \neg(A \vee B) \vee C \cong (\neg A \vee C) \wedge (\neg B \vee C)$$

- Thus, we can express this as 2 Horne Clauses:

$$A \rightarrow C \text{ and } B \rightarrow C$$

Conjunction on the RHS

$$A \rightarrow B \wedge C \cong \neg A \vee (B \wedge C) \cong (\neg A \vee B) \wedge (\neg A \vee C)$$

Thus, we can express this as 2 Horne clauses

$$A \rightarrow B \text{ and } A \rightarrow C$$

Disjunction on the RHS

$$A \rightarrow B \vee C \cong \neg A \vee B \vee C$$

- Clearly, NOT a Horne Clause

- We cannot express disjunctive goals

Existential Quantification

- We cannot deal with all existentially quantified formulae, we want to at least be able to deal with formulae that express true facts about an individual

$\exists x. P(x) \wedge Q(x)$ negates to $\forall x. P(x) \wedge Q(x) \longrightarrow \bot \cong \neg P(x) \vee \neg Q(x)$

This is a ground Clause as no tve literals

Prolog -

- Programming language based on the concepts of resolution & Unification.

- Uses SDL resolution

- Language is restricted to Horne Logic

Syntax

- Very simple syntax

Horne Clauses

- Basis of Prolog Syntax

- Uses a reversed notation

- Take a Horne Clause:

$$L_1 \wedge \ldots \wedge L_n \longrightarrow M$$

reverse

$$M \longleftarrow L_1 \wedge \ldots \wedge L_n$$

Now we can directly translate this into Prolog Syntax

```
M :- L_1, ..., L_n.   <- end of clause
```

`:-` : Implication `,` : Logical Conjunction

Terms

- Notation for terms is different.

- Predicates, Functions & Constants are denoted by names Matching the expression (start lowercase)

$$[a\text{-}z\_][a\text{-}z\ A\text{-}Z\ 0\text{-}9]^*$$

Variables are denoted with letters starting with capitals

Variables are denoted with letters starting with capitals
Matching the expressions

$$[A\text{-}Z\_][a\text{-}z\ A\text{-}Z\ 0\text{-}9]^*$$

Socrates is human, all human are mortal, therefore
Socrates is mortal.

This can be written in Prolog =>

```
mortal(X) :- human(X).
human(socrates).
```

- Predicates are usually also denoted with their arity

e.g. Mortal/1

Conditionals

- Use Pattern Matching & Horn Clauses
- eg. defining Max of 2 given elements:

```
max(X, Y, Z) :- X > Y, Z = X.
max(X, Y, Z) :- X <= Y, Z = Y.
```

Loops

- Necessary in order for Prolog to be a Turing Complete language
- Done using recursion

```
loop(0).
loop(N) :- N > 0,
        : body
        M is N - 1,
        loop(M)
```

↑ recursion

Iteration over lists

```
list_length([], 0).
list_length([H|T], N) :- list_length(T, N1), N is N1 + 1
```

↑ could be replaced

/could be replaced with wildcard (_)

Negation as failure

- Whenever we cannot derive a fact from the knowledge base, we assume it is false (its negation holds)

- Forced in Prolog using the predicate: \+/1

Backwards Chaining

- Programs build the knowledge base
- Queries query the knowledge base & can be written / executed interactively

Evaluation

Prolog Programs are evaluated via backwards chaining
↑ A general inference method

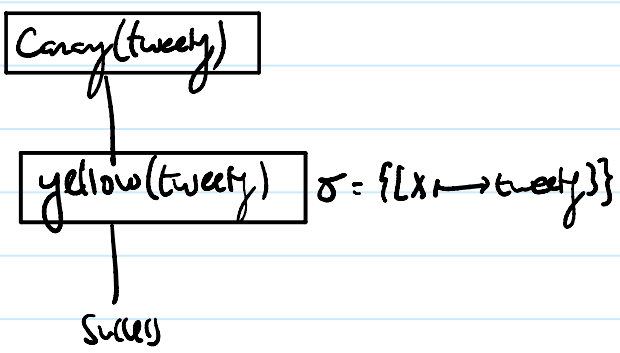
- Queries are matched against heads of clauses, possibly unifying variables

Eg.

```
canary(X) :- yellow(X).

yellow(tweety).
```

The query canary(tweety) is resolved as:



Finite Model Generation

- Reducing the complexity of our language & Calculus by restricting ourselves to finite models.

Quantifies over finite domains

Quantifiers over finite domains

$\mathcal{I}_\phi \models \exists x.\phi$ if $\mathcal{I}_\phi \models \phi$ for some ϕ that maps x onto some value in $\mathcal{U}$

$\mathcal{I}_\phi \models \forall x.\phi$ if $\mathcal{I}_\phi \models \phi$ for assignments ϕ that map x to all possible values in $\mathcal{U}$

- Assume our universe is finite, $\mathcal{U} = \{a_1, \ldots, a_n\}$
- We want to eliminate the need for our ϕ, thus restricting ourselves to predicates, functions and constants.

- As our universe is finite we can actually represent each element in it with a distinct constant in our syntax. This allowing us to remove quantifiers without skolemisation.

$\exists$ & $\vee$

our $\exists$ quantifier can be re-written as:

$$\mathcal{I} \models \exists x.\phi \text{ if } \mathcal{I} \models \phi[x \mapsto c_1] \vee \phi[x \mapsto c_2] \vee \ldots \vee \phi[x \mapsto c_n]$$

- We can represent existential quantifiers with disjunction.

eg interpret $\exists x.P(x)$ over a domain of 3 elements $\{c, d, e\}$
we can write:
$$P(c) \vee P(d) \vee P(e)$$

$\forall$ & $\wedge$

$\forall$ can be re-written as:

$$\mathcal{I} \models \forall x.\phi \text{ if } \mathcal{I} \models \phi[x \mapsto c_1] \wedge \phi[x \mapsto c_2] \wedge \ldots \wedge \phi[x \mapsto c_n]$$

- We can represent universal quantification using conjunction.

Clause Normal Forms

Using the process seen above, clause normal form can be obtained. However, pay close attention to the order you eliminate quantifiers as different forms can be achieved. You want the minimal set. To make reasoning with them easier

Equality

Properties

- Equality is just an equivalence relation
- Therefore, the following Properties hold:
  - Reflexivity - $a \approx a$, for every a
  - Symmetry - if $a \approx b$, then $b \approx a$
  - Transitivity - if $a \approx b$ and $b \approx c$ then $a \approx c$
- Substitution is another important property of equality
  - essentially we can replace like for like
    - if $a \approx b$, we can replace every occurrence of a with b and vice versa

Formal Equality

- formal semantics

$$\mathcal{I} \models a \approx b \text{ iff } \mathcal{I}(a) = \mathcal{I}(b)$$

- Basic Axioms

$$\frac{}{a \approx a}\ \text{Reflexivity} \qquad \frac{P(a) \quad a \approx b}{P(b)}\ \text{Substitution}$$

- Derived Axioms

$$\frac{a \approx b}{b \approx a}\ \text{Symmetry} \qquad \frac{a \approx b \quad b \approx c}{a \approx c}\ \text{Transitivity}$$

- These are derived as they can be obtained from Reflexivity & Substitution

Finite Interpretations

- Whenever we introduce new (new) constants we need to assert that they are distinct from all other constants already in use
- We add 'all different Constants'

- Eg. We want to interpret the following formula over a domain of 3 elements

$$\forall x. P(x, c)$$

- c is already a constant in the syntax. We have to choose 3 new constants $\{d_0, d_1, d_2\}$ & rewrite as:

$$P(d_0, c) \wedge P(d_1, c) \wedge P(d_2, c) \wedge d_0 \not\approx d_1 \wedge d_0 \not\approx d_2 \wedge (c \approx d_0 \vee c \approx d_1 \vee c \approx d_2)$$

As there are only 3 elements in the domain & 4 constants mentioned, one pair must be equal

Finite Model Generation

- Goal is to generate finite models for our formulae

- Equality will now only indicate syntactic equality/in-equality

The Domain

- The domain into which we will interpret will be represented as a set of natural numbers

- Allows an element in domain to be different to other elements in the syntax & implicitly named. (satisfying the 'all different constraint')

- eg. a domain of size 3 would be $D = \{0, 1, 2\}$

Constants

- Will be assigned different elements of D during the generation proc.

- Allowing us to eliminate the disjunct of equalities seen above

Initialisation

1) Generate ground clauses from input formulae using elements in D

- We can often remove clauses already after this step if they are trivially satisfied or contain simple equalities, (domain elements only)

2) Setting up tables for the interpretation of constants, functions & Predicates

Constants & functions - interpreted using tables corresponding to their

Constants & functions - interpreted using tables corresponding to their arity containing elements from D

- Constant, c, is interpreted by a single cell
- A n-ary function is interpreted by an n+1 array.

Predicates - interpreted by tables corresponding to their arity containing elements from {T, F}

Cell assignment

- In each step a cell from a table is assigned a value
- Cells from constants or functions are assigned values from D
- Cells from predicates are assigned truth values

- Constants normally assigned first
- Predicate cells are assigned last.

Clause Removal

- Similar to DPLL
- Clauses can be removed if a literal evaluates to T
  - A literal can be removed from a clause if it evaluates to F
- If a clause is empty, we have to backtrack
- If the set of clauses is empty we have a valid (partial) model
  - If some cells are not yet assigned, they can be assigned freely with elements from the domain

Propagation

- 2 types:

- Positive Propagation
  - Replace all occurrences of a term with the domain value it is assigned to
  - Recursive Process.

Negative Propagation

- Derives negated equalities.
- Helps eliminate possible values

- eg. if we know $f(2,3) = 4$ and $f(2, g(5)) \neq 4$ we can propagate that $g(5) \neq 3$

Constraint Satisfaction Problems

- A CSP is a triple, consisting of:
  - A finite set V of variables
  - A finite set D, called the domain,
  - A finite set of constraints C, where a constraint $c \in C$ is a pair (t, R). Such that $t \subseteq V$ and $R \subseteq \underbrace{D \times \dots \times D}_{n}$ with $|t| = n$

- The elements of C are essentially relations on subsets of variables from V

Initial Variable assignment

- We assign initial values to each $v \in V$ in the domain D in a domain of the variable D_v. Thereby, v can only be assigned an subset of the elements in D.
- To enforce this algorithmically are implemented unary constraints to the CSP.
- Effectively we have $D_v \subseteq D$
- We will look at ways to do this

Constraint Networks

CSP can be represented as a constraint network

- depending on the type of constraints, we choose different types of network

Unary Constraints - Restrict the value of a single variable.

- Can easily be satisfied by simply Pre-Processing the domain of the corresponding variables to remove any value or variables with the constraint.

- Does not require an explicit representation in the network.

- **Binary Constraints** - Relate 2 variables together.

- Can only be represented on a constraint graph.

- Can only be represented on a Constraint graph
  - vertices represent variables & their domain interpretations
  - edges represent constraints.

Higher order / n-ary constraints - relate multiple variables to eachother

- represented in Constraint hyper-graphs
  - 2 types of vertices
    - Variable
      - Represent domain interpretation of the variable
      - Have an edge for each Constraint that effects them
    - Constraint
      - Represent constraints

- As binary Constraints are easier to solve, often we attempt to decompose higher order Constraints into binary ones.

Consistency Algorithms

- Many ways to enforce consistency in CSP.
- 3 Main algorithms:
  - Node Consistency -
    - Requires that every unary constraint on a variable is satisfied by all values in the domain of the variable & vice versa.
    - Can be achieved by reducing the domain of each variable to the values that satisfy any unary constraint on that variable
    - ∴ Unary Constraints can be neglected & considered incorporated into the domain.

Arc Consistency

- Suppose Constraint, c, has scope $\{X, Y_1, \ldots, Y_k\}$. Arc $\langle X, C\rangle$ is consistent if, for each value $x \in D_X$, there are values $y_1, \ldots, y_k$, where $y_i \in D_{Y_i}$ such that $c\langle X = x, Y_1 = y_1, \ldots, Y_k = y_k\rangle$ is satisfied

$y_i = v_{y_i}$ such that $C(x = v_x, Y_1 = y_1, \ldots, Y_k = y_k)$ is satisfied

- A network is Arc Consistent, if all its arcs are arc consistent.
  - If there is no assignment for one of the variables Y_i, then an arc is inconsistent.
- Generally arc consistency is not enough on its own to make a CSP satisfiable

Path Consistency

- Higher-order method
- Allows consideration of multiple constraints at the same time

Solving CSP

- Basic idea is a backtrack search
  1. enforce node consistency
  2. Enforce arc consistency (as much as possible)
  3. Make choice of instantiation for a variable
  4.
  5. If one of the above steps results in an inconsistent CSP, backtrack

Examples of CSP-solving algorithms:

- A simplistic algorithm
  - Cycle over pairs of variables, enforcing arc consistency
    Repeat until no domain changes
- The AC-3 algorithm
  - Ignores constraints that have not been modified since they were last analysed

Begin looking at a set containing all constraints.

- Begins looking at a set containing all constraints.
  - enforces arc consistency in arc constraint
  - Removed when its domain of values changes.