Artificial Intelligence with Logic Programming - Notes

Dom Hutchinson

February 4, 2020

Contents

1 Introduction

Definition 0.1 - *Types of AI*

1. *Weak AI* - Can solve a specific task.
2. *Strong AI* - Can solve general problems.
3. *Ultra Strong AI* - Can solve general problems & explain why/what it is doing.

2 Logic Programming

Definition 0.1 - *Logic Programming*
Logic Programming is a *declarative paradigm* where programs are concieved as a logical theory, rather than a set-by-step description of an algorithm. A Procedure call is viewed as a theorem which the truth needs to be established about. (*i.e.* executing a programming is analogous to searching for truth in a system).

Remark 0.1 - *Variables*
In *Logic Programming* a *Variable* is a variable in the mathematical sense, that is they are are placeholders that can take on any value.

Remark 0.2 - *Machine Model*
A *Machine Model* is an abstraction of the computer on which programs are executed. In *Imperative Programming* we assume a dynamic, state-based machine model where the state of the computer is given by the contents of its memory & a program statement is a transition from one statement to another. In *Logic Programming* we do not assume such a dynamic model.

2.1 Clausal Logic

Notation 1.1 - *Variables & Values*
Variables are denoted by having a capitalised first letter, whereas *values* are completly lowercase.

Definition 1.1 - *Clausal Logic*
Clausal Logic is a formalism for representing & reasoning with knowledge.

Keyword	Description
`S:-C.`	**If** condition `C` holds **then** statement `S` is true.
`S:-\+C.`	**If** condition `C` does **not** hold **then** statement `S` is true.
`S:-C1,C2.`	**If** conditions `C1` **and** `C2` both hold **then** statement `S` is true.
`S:-C1;C2.`	**If** at least one of `C1` **or** `C2` hold **then** statement `S` is true.
`S:-C,!.`	**Cut** the program after finding **first** `S` where `C` holds.

Definition 1.2 - *Facts & Rules*
Facts are logical formulae which are defined for explicit values **only**. *Facts* denoted unconiditional truth.

```
nearby(bond_street,oxford_circus).
```

Rules are logical formula which are defined in termms of variables (and explicit values). *Rules* denote conditional truth.

```
nearby(X,Y):-connected(X,Z,L),connected(Z,Y,L)
```

Definition 1.3 - *Query,* `?-`
A *Query* asks a question about the knowledgebase we have defined. If we just pass *values* to a *Query* then it shall simply return whether the statement is true or not. If we pass *unbound variables* as well then it shall return values for the variable which make the statement true, if any exist.

Example 1.1 - *Query*

```
1 ?-nearby(bond_street,oxford_circus)
2 ?-nearby(bond_street,X)
```

(1) will return *true* if we have defined `bond_street` to be near to `oxford_circus`.
(2) will return all the values of `X` (*i.e.* stations) which are near to `bond_street`.

Definition 1.4 - *Resolution*
In order to answer a query `?-Q1,Q2,...` find a rule `A:-B1,...,Bn` such that `A` matches with `Q1` then proceed to answer `?-B1,...,Bn,Q2,...`.
This is a *procedural interpretation* of logical formulae & is what allows *Logic* to be a programming language.

Definition 1.5 - *Functor*
*Functor*s provide a way to name a complex object composed of simpler objects & are never evaluated to determine a value.

```
reachable(X,Y,noroute):-connected(X,Y,L)
reachable(X,Y,route(Z,R)):-connected(X,Z,L),connected(Z,Y,R)
```

Querying `?-reachable(oxford_circus,tottenham_court_road,R)` will return a route `R` which connects the two stations, on a single line.
The above definition can be read as `X` is reachable from `Y` if they are connected **or** if there exists a station `Z` which is connected.

Definition 1.6 - *List Functor,* `.`
The *List Functor* takes two arguments, one on each side, and has terminator `[]`.

[a,b,c]≡`.(a,.(b,.(c,[])))`

Alternatively we can use a pipe to distinguish between a value and the rest of the list

`[X,Y|R]`

Remark 1.1 - *Logical Formulae*
Logical Formulae determine what is being modelled & the set of formulae that can be derived by applying inference rules.
i.e. *Logical Formulae* have both declarative (what is true) & procedural (how the truth is reached) meaning.
Changing the order of statements in a clause does not change the declarative meaning ($3 = 2 + 1 \equiv 3 = 1 + 2$) but does change the procedural meaning & thus chanes what loops we may get stuck in, what solutions are found first & how long execution takes.
N.B. The procedural meaning of *Logical Formulae* is what allows them to be used as a programming language.

2.2 SLD-Resolution

Definition 2.1 - *SLD-Resolution*
SLD-Resolution is the process `Prolog` uses to resolve a query.

1. ***S**election Rule* - Left-to-right (Always take the left branch if it is there).
   This is equivalent to reading clauses top to bottom in a `Prolog` program.

2. ***L**inear Resolution* - The shape of the proof trees obtained. ***D**efinite Clauses*

N.B. These rules vary for different languages.

Definition 2.2 - *SLD-Tree*
SLD-Trees are graphical representations of *SLD-Resolution*.
They are not the same as a proof tree as only the resolvents are shown (no input clauses or unifiers) andit contains every possible resolution step, leading to every leaf on an *SLD-Tree* being an empty clause $\square$.
The left-most leaf in an *SLD-Tree* will be returned as the first solution, if we end up in an infinite sub-tree then no solution will ever be returned.
Non-leaf nodes in *SLD-Trees* are called *choice points*. *N.B.* If a leaf is a failure then we underline it.

Proposition 2.1 - *Traversing an SLD-Tree*
`Prolog` traverses *SLD-Trees* in a *depth-first* fashion, backtracking whenever it reaches a success or a failure. This can be visualised as following the left branch of the tree whenever it is avaiable.
This leads to `Prolog` being incomplete (does not always return all solutions) as once it is in an infinite sub-tree it has no way of escaping.

Remark 2.1 - *Backtracking*
Since we need to be able to *Backtrack* after finding successes or failures, we need to store all previous resolvents which have not been fully explored yet as well as a pointer to the most recent program clause that has been tried.
Due to the depth-first nature of `Prolog` this can be stored as a *stack*.

Remark 2.2 - *Order of clauses*
Due to the *Selection Rule* being left-to-right you should *put non-recursive clauses before recursive clauses* to ensure that solutions are found before infinite sub-trees.

Definition 2.3 - *Transitivity*
Transitivity is a property of predicate that states

$$X = Y \ \& \ Y = Z \implies X = Z$$

Transitivity clauses can cause infinite loops is not implemented carefully.

Remark 2.3 - *Why not use Breadth-First Search?*
Breadth-First Search requires for each branch at a given level to be tracked (rather than just one in *Depth-First Search*), this requires more memory and was deemed memory-inefficient by `Prolog` developers.

Remark 2.4 - *Queries with no solution*
Some queries do not have a solution as the *SLD-Tree* they produce is infinite, no matter how it is read & thus an answer can never be resolved.
Consider

```
brother_of(paul,peter).
brother_of(X,Y):-brother_of(Y,X).
```

The query `brother_of(peter,maria)` has no answer since its *SLD-Tree* is infinite.

Definition 2.4 - *Cut,* `!`
A *Cut* says that once it has been reached, stick to all the variable substitutions found after entering its clause. (*i.e.* Do not backtrack past it?).
Cuts have the effect of pruning the *SLD-Tree*.
Cuts which don't remove any successful branches are called *Green Cuts* and are harmless. *Cuts* which do remove successful branches are caleld *Red Cuts* and affects the procedular meaning of the process.
N.B. *Cuts* make the left sub-tree deterministic.

Notation 2.1 - *In SDL-Trees we put a box around subtrees which are pruned by* `!`*.*

2.3 Negation as Failure

Definition 3.1 - *Netgation as Faliure*
Negation as Failure is an interpretation of the semantics for failure.
It states if we cannot prove `q` to be true then we take `not(q)` to be true.
i.e. If we don't have enough information to state where `q` is true then we assume it to be false.
e.g. Negation as failure would resolve that *"I cannot prove God exists, therefore God does not exist."*.

Proposition 3.1 - *Altering Negation as Failure*

Proposition 3.2 - *Cut as Failure*
Suppose we have a program of the following form

```
p:-q,!,r.
p:-s.
```

We can interpret it as

```
p:-q,r.
p:-not_q,s.
```

where `not_q:=q,fail.`.

Definition 3.2 - *Negation Function,* `not()`
Let `Goal` be a literal we wish to negate

```
not(Goal):-Goal,!,fail.
not(Goal).
```

Definition 3.3 - *Call Function,* `call()`
TODO

Remark 3.1 - `not()` *&* `call()` *are called meta-predicates as they take formulas as arguments.*

Remark 3.2 - *Negation as Failure is* ***not*** *Logical Negation*
If we cannot prove predicate `q` we know that `q` is not a logical consequence of the program, but that does not mean that its negation `:-q` is a logical consequence of the program.
This is acceptable reasoning in some scenarios, but not others.

Definition 3.4 - *Logical Negation*
Logical Negation can only be expressed by indefinite clauses, as below

```
p;q,r.
p;q:-s.
s.
```

Definition 3.5 - *Conditionals, `if_then_else()`*
We give the following definition where if `S` holds then `T` is executed, else `U` is executed.

```
if_then_else(S,T,U):-S,!,T/
if_then_else(S,T,U):-U.
```

Notation 3.1 - *Implication, `->`*
We can nest `if_then_else()` to add more *elif* clauses but is clumsy notation.
Instead we simplify `if_then_else(S,T,U).` to `S->T;U.`.
This can be nested as `P->Q;(R->S;T)`